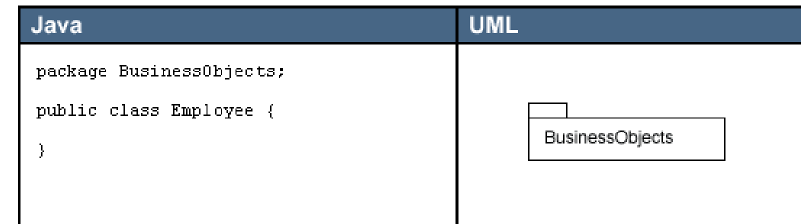
Αρχές Αντικειμενοστρεφούς Σχεδίασης

Object – Oriented Design Principles

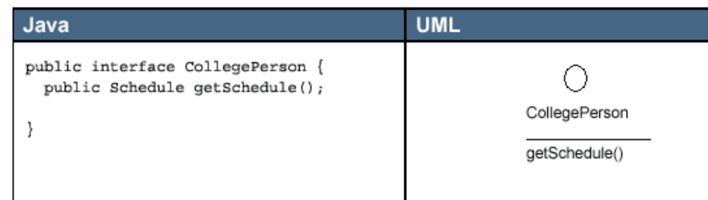
Μεταφορά της UML σε Java



Java	UML

`package BusinessObjects;`
`public class Employee {`
`}`
	BusinessObjects

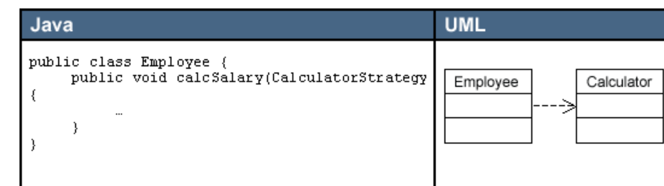
Μεταφορα της UML σε Java



Java	UML

`public interface CollegePerson {`
`  public Schedule getSchedule();`
`}`
	CollegePerson
	getSchedule()

Μεταφορα της UML σε Java

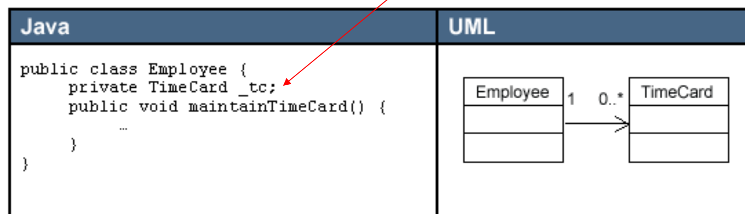


Java	UML

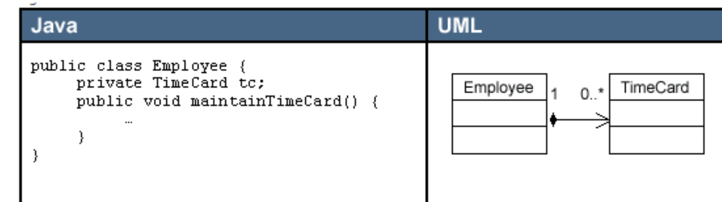
`public class Employee {`
`     public void calcSalary(CalculatorStrategy`
`{`
`          …`
`     }`
`}`
	Employee - - -> Calculator

Μεταφορα της UML σε Java

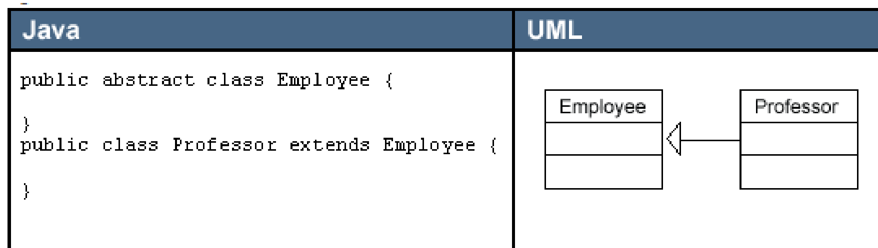
Θα έπρεπε να μπεί πίνακας



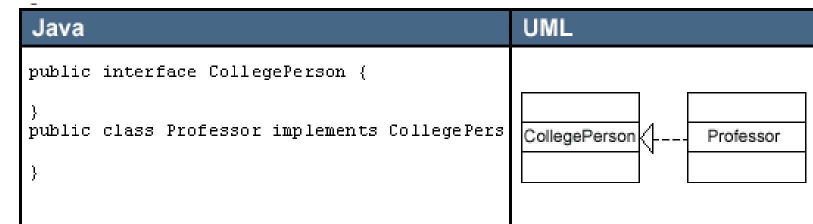
Μεταφορα της UML σε Java



Μεταφορα της UML σε Java

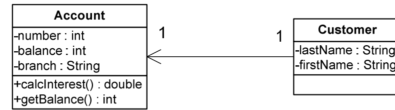


Μεταφορα της UML σε Java



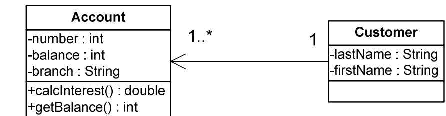
C++ σε UML

```
Class Customer
{
        public:
        Customer();
        Account* getAccount()
            {return theAccount;}
        void setAccount(Account
            *value)
            {theAccount=value;}
        private:
        string lastName;
        string firstName;
        Account* theAccount;
}
```



C++ σε UML

```
class Customer
{
        public:
        Customer();
        Account* getAccount (int index
            ) {return
            theAccounts[index];}
        void setAccount(int index,
            Account
            *value){relatedAccount=valu
            e;}
        private:
        string lastName;
        string firstName;
        Account* theAccounts[];
}
```



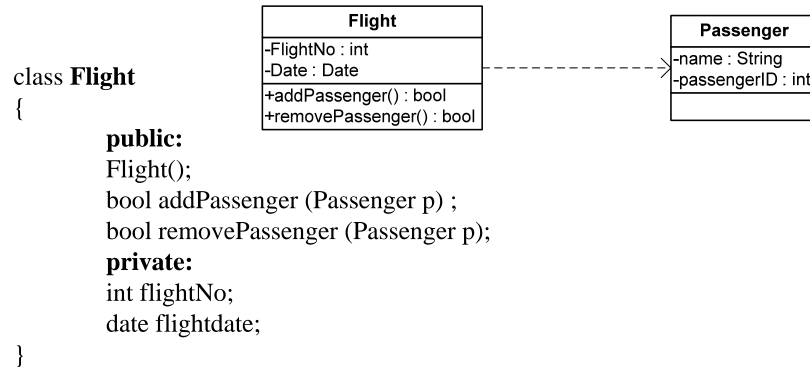
C++ σε UML

```
class Car
{
        public:
        Car();
        Engine* getEngine () ;
        void setEngine(Engine
            *value);
        private:
        string model;
        int serialNo;
        Engine* theEngine;
}
```

C++ σε UML

```
class Car
{
        public:
        Car();
        Engine getEngine () ;
        void setEngine(Engine
            value);
        private:
        string model;
        int serialNo;
        Engine theEngine;
}
```

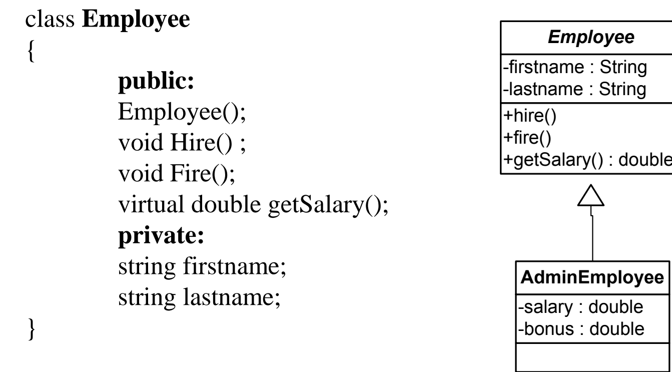
C++ σε UML




```
class Flight
{
        public:
        Flight();
        bool addPassenger (Passenger p) ;
        bool removePassenger (Passenger p);
        private:
        int flightNo;
        date flightdate;
}
```

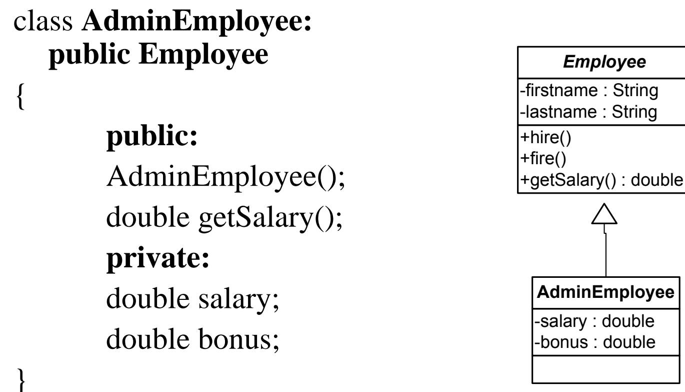

C++ σε UML

```
class Employee
{
        public:
        Employee();
        void Hire() ;
        void Fire();
        virtual double getSalary();
        private:
        string firstname;
        string lastname;
}
```



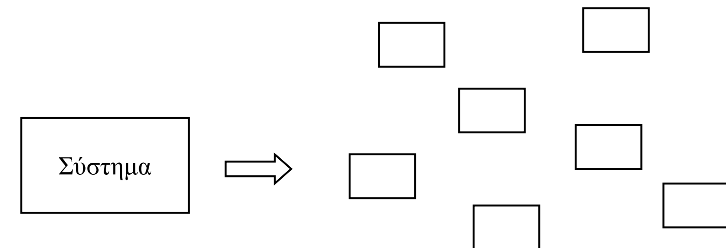
C++ σε UML

```
class AdminEmployee:
   public Employee
{
        public:
        AdminEmployee();
        double getSalary();
        private:
        double salary;
        double bonus;
}
```



Σχεδίαση

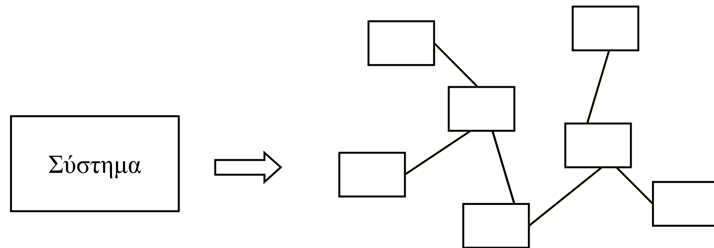
- Σχεδίαση (οποιουδήποτε τεχνικού έργου) είναι:



- Η αποσύνθεση ενός συστήματος σε τμήματα (μονάδες)

Σχεδίαση

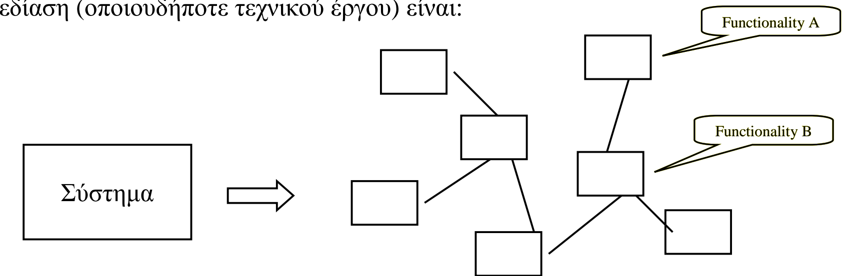
• Σχεδίαση (οποιουδήποτε τεχνικού έργου) είναι:



• Η αποσύνθεση ενός συστήματος σε τμήματα (μονάδες)
• Ο καθορισμός των σχέσεων μεταξύ των τμημάτων

Σχεδίαση

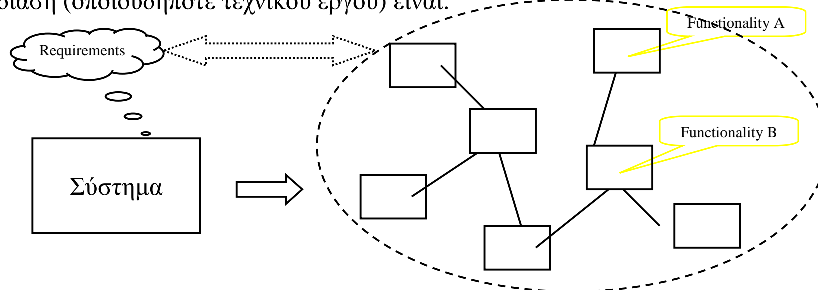
• Σχεδίαση (οποιουδήποτε τεχνικού έργου) είναι:



• Η αποσύνθεση ενός συστήματος σε τμήματα (μονάδες)
• Ο καθορισμός των σχέσεων μεταξύ των τμημάτων
• Η ανάθεση αρμοδιοτήτων σε κάθε τμήμα

Σχεδίαση

• Σχεδίαση (οποιουδήποτε τεχνικού έργου) είναι:



• Η αποσύνθεση ενός συστήματος σε τμήματα (μονάδες)
• Ο καθορισμός των σχέσεων μεταξύ των τμημάτων
• Η ανάθεση αρμοδιοτήτων σε κάθε σχήμα
• Η επικύρωση ότι όλα τα τμήματα μαζί επιτυγχάνουν τους σκοπούς του συστήματος

Συμπτώματα

• Τι συνιστά "καλή" αντικειμενοστρεφή σχεδίαση ?

Συμπτώματα "κακής" σχεδίασης:

• **Δυσκαμψία (Rigidity):** Το σύστημα είναι δύσκολο να τροποποιηθεί διότι κάθε αλλαγή οδηγεί σε πληθώρα αλλαγών σε άλλα τμήματα του συστήματος

• **Ευθραυστότητα (Fragility):** Οι αλλαγές που πραγματοποιούνται στο λογισμικό προκαλούν σφάλματα σε διάφορα σημεία.

• **Ακινησία (Immobility):** Υπάρχει δυσκολία διαχωρισμού του συστήματος σε συστατικά τα οποία μπορούν να επαναχρησιμοποιηθούν σε άλλες εφαρμογές.

• **Έλλειψη ρευστότητας (Viscosity):** Η πραγματοποίηση τροποποιήσεων με λάθος τρόπο είναι ευκολότερη από την πραγματοποίησή τους με τον ορθό τρόπο.

• **Περιττή Πολυπλοκότητα (Needless Complexity):** Το λογισμικό περιλαμβάνει στοιχεία που δεν είναι (ούτε πρόκειται να γίνουν) χρήσιμα.

• **Περιττή Επανάληψη (Needless Repetition):** Η σχεδίαση περιλαμβάνει επαναλαμβανόμενες δομές που θα μπορούσαν να ενοποιηθούν υπό μία κοινή αφαίρεση.

• **Αδιαφάνεια (Opacity):** Δυσκολία κατανόησης μιας μονάδας (σε επίπεδο σχεδίου ή κώδικα).

Αρχές

Αρχές Αντικειμενοστρεφούς Σχεδίασης:

1. Αρχή της Ενσωμάτωσης
2. Αρχή της Χαμηλής Σύζευξης
3. SRP – Single Responsibility Principle (Αρχή της Μοναδικής Αρμοδιότητας)
4. OCP – Open-Closed Principle (Αρχή Ανοικτής-Κλειστής Σχεδίασης)
5. LSP – Liskov Substitution Principle (Αρχή Υποκατάστασης της Liskov)
6. DIP – Dependency Inversion Principle (Αρχή της Αντιστροφής των Εξαρτήσεων)
7. ISP – Interface Segregation Principle (Αρχή του Διαχωρισμού των Διασυνδέσεων)

Παραβίαση μιας ή περισσοτέρων αρχών οδηγεί σε ένα ή περισσότερα από τα συμπτώματα

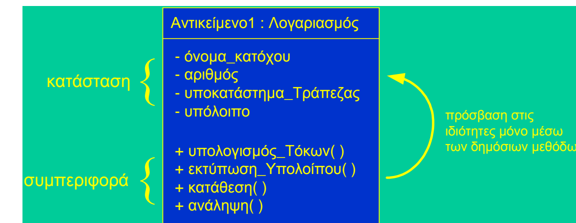
Οι ανωτέρω αρχές δεν εφαρμόζονται διαρκώς και χωρίς λόγο. Μόνο ως θεραπεία σε κάποιο σύμπτωμα που έχει ήδη διαπιστωθεί. --

Αρχή της Ενσωμάτωσης

Θεμελιώδης κανόνας Αντικειμενοστρεφούς Προγραμματισμού (Encapsulation Principle)

Συνήθως εφαρμόζεται ακόμα και αν δεν συνειδητοποιείται η χρησιμότητα

Αρχή της Ενσωμάτωσης: *Η εσωτερική κατάσταση ενός αντικειμένου πρέπει να είναι τροποποιήσιμη μόνο μέσω της δημόσιας διασύνδεσης του*



Πρακτικά εφαρμόζεται θέτοντας την ορατότητα όλων των ιδιοτήτων private

Αρχή της Ενσωμάτωσης

Πλεονέκτημα: Διατήρηση της εγκυρότητας ενός αντικειμένου

Κάθε αντικείμενο περιλαμβάνει κάποιες αναλλοίωτες, συνθήκες που πρέπει να είναι πάντοτε αληθείς

Έστω μια κλάση `TimeStamp`

```
class TimeStamp {
public:
   TimeStamp();
   TimeStamp(int hr, int min, int sec);
   void printTimeStamp();

   int hour;              //μέλη δεδομένων με δημόσια ορατότητα !!!
   int minute;
   int second;
};
```

Κατασκευή αντικειμένου: `TimeStamp T1(23, 45, 17);`

Αρχή της Ενσωμάτωσης

Έστω ότι κάποια κλάση-πελάτης της `TimeStamp` επιχειρεί να αυξήσει κατά δύο ώρες τη χρονική στιγμή ως εξής:

```
//κώδικας κλάσης - πελάτη
   . . .
   T1.hour++;
   T1.hour++;
```

Θα προκύψει ένα **μή-έγκυρο** αντικείμενο (ώρα 25:45:17 !!)

Αναλλοίωτη { 0 <= hour <= 23 }

Με την εφαρμογή της αρχής της ενσωμάτωσης ο σχεδιαστής της κλάσης μπορεί να εγγυηθεί την εγκυρότητα των αντικειμένων

- Σχεδίαση του κατασκευαστή <u>ώστε να παράγει μόνο έγκυρα αντικείμενα</u>
- Σχεδίαση των μεθόδων <u>ώστε να μην καταργούν την ισχύ των αναλλοίωτων</u>

Αρχή της Ενσωμάτωσης

Εφαρμογή:

```
class TimeStamp {
. . .
public:
   void incrementHour();

private:
   int hour;
   int minute;
   int second;
. . .
};

void TimeStamp::incrementHour() {
  hour++;
  if(hour == 24)
    hour = 0;
}
```


Αρχή της Ενσωμάτωσης

Συμπερασματικά:

• Η παραβίαση της αρχής μπορεί να προκαλέσει ορισμένα από τα σημαντικότερα προβλήματα

• Επιτρέποντας την τροποποίηση των τιμών των ιδιοτήτων καταργείται η ομαδοποίηση δεδομένων και συμπεριφοράς

• Στη συνέχεια θα είναι δύσκολο να εντοπιστούν ποια τμήματα κώδικα επηρεάζουν ποια δεδομένα

Αρχή της Χαμηλής Σύζευξης

Μονάδες κατά τη σχεδίαση: Ανεξάρτητα συστατικά λογισμικού

• με σαφώς καθορισμένη λειτουργικότητα

• με σαφώς καθορισμένο σύνολο εισόδων και εξόδων

Δύο σημαντικές έννοιες στην Τεχνολογία Λογισμικού είναι:

• η **σύζευξη** (coupling). Αναφέρεται στο βαθμό εξάρτησης μεταξύ δύο συστατικών

• η **συνεκτικότητα** (cohesion). Αναφέρεται στο βαθμό εσωτερικής λειτουργικής συνάφειας μεταξύ των τμημάτων ενός συστατικού

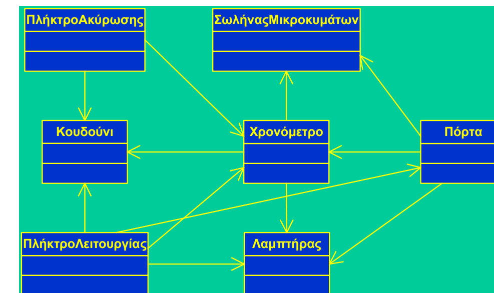
Κύριο μέλημα κατά τη διαδικασία σχεδίασης πρέπει να είναι:

Αρχή της Χαμηλής Σύζευξης: Σε ένα σχέδιο λογισμικού πρέπει να επιδιώκεται η επίτευξη της μικρότερης δυνατής σύζευξης μεταξύ των συστατικών του

Αρχή της Χαμηλής Σύζευξης

Πλεονεκτήματα: Ευκολότερη υλοποίηση, έλεγχος και συντήρηση

Παράδειγμα κακής σχεδίασης (υψηλή σύζευξη):



• Αν τροποποιηθεί η `Χρονόμετρο` απαιτείται (τουλάχιστον) ο έλεγχος και (ενδεχομένως) η μεταγλώττιση των συσχετιζόμενων κλάσεων

• Για να επαναχρησιμοποιηθεί η `Χρονόμετρο` απαιτείται “μεταφορά” των αναφορών της προς τις συσχετιζόμενες κλάσεις

Αρχή της Χαμηλής Σύζευξης

Η μείωση της σύζευξης είναι δύσκολη υπόθεση. Είναι όμως εύκολο να μετρηθεί:

• Μετρική CBO (Coupling between Objects – Chidamber & Kemerer, 1994):

Η τιμή της μετρικής για μια κλάση C ισούται με τον αριθμό των άλλων κλάσεων με τις οποίες υπάρχει σύζευξη. Ως σύζευξη θεώρησαν:

• Χρήση μεθόδων

• Απευθείας πρόσβαση σε μέλη δεδομένων (αν υπάρχει)

• Εξαιρείται η κληρονομικότητα

Επίσης έχει προταθεί να θεωρείται ως σύζευξη:

• Η χρήση τύπων μιας κλάσης ως παραμέτρων σε μεθόδους

• Η δημιουργία αντικειμένων

• Η ύπαρξη φιλικών κλάσεων

• Η αποστολή μηνυμάτων

SRP – Single Responsibility Principle

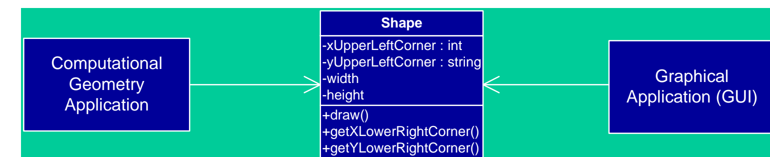
Tom De Marco (1979): Συνεκτικότητα (Cohesion)

Αρχή της Μοναδικής Αρμοδιότητας: *Μία κλάση πρέπει να έχει μόνο ένα λόγο να αλλάξει*

Κάθε κλάση δεν πρέπει να έχει περισσότερες από μία αρμοδιότητες (axis of change)

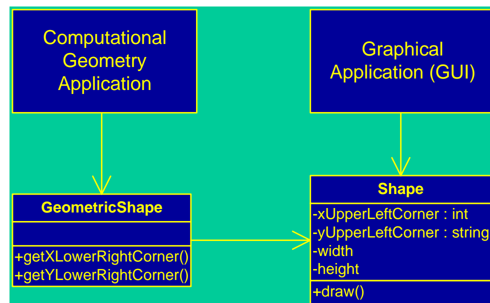
- περισσότεροι από ένας λόγοι αλλαγής
- σύζευξη μεταξύ αρμοδιοτήτων

Έστω η ακόλουθη σχεδίαση



SRP – Single Responsibility Principle

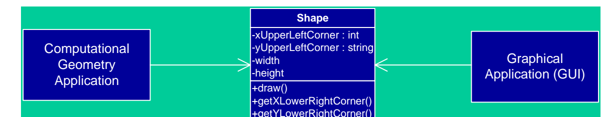
Η εφαρμογή της αρχής SRP επιβάλλει το διαχωρισμό των δύο αρμοδιοτήτων σε δύο τελείως διαφορετικές κλάσεις:



SRP – Single Responsibility Principle

Κακή Σχεδίαση

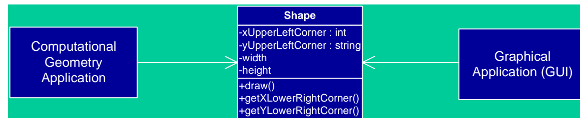
```
public class Shape extends JComponent {
        public void draw()
        {
                rect = new Rectangle(xUpperLeftCorner, yUpperLeftCorner, width, height);
                this.repaint();
        }

        public void paint(Graphics g) {
                super.paint(g);
                Graphics2D g2 = (Graphics2D)g;
                g2.draw(rect);
        }

        public int getXLowerRightCorner() { return width; }
        public int getYLowerRightCorner() { return height; }
        private int xUpperLeftCorner = 0;
        private int yUpperLeftCorner = 0;
        private int width = 200;
        private int height = 100;
        private Rectangle rect;

}
```



SRP – Single Responsibility Principle

Καλή Σχεδίαση

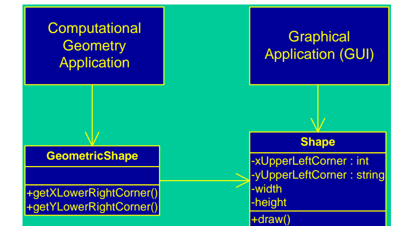
```
public class Main {
            public static void main(String[] args) {
                        Shape myShape = new Shape();
                        JFrame f = new JFrame();
                        f.setSize(600, 600);
                        f.setContentPane(myShape);
                        f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
                        myShape.draw();´//Application 1
                        f.setVisible(true);
                        System.out.println("Lower Right Corner. X: " +
                        myShape.getXLowerRightCorner()
                                                + " Y: " + myShape.getYLowerRightCorner());
// Application 2
            }
}
```



SRP – Single Responsibility Principle

Καλή Σχεδίαση

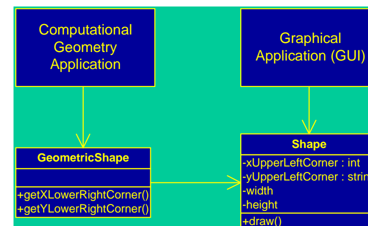
```
public class Shape extends JComponent {

            public void draw(int xTopLeft, int yTopLeft)
            {
                        xUpperLeftCorner = xTopLeft;
                        yUpperLeftCorner = yTopLeft;
                        rect = new Rectangle(xUpperLeftCorner, yUpperLeftCorner, width, height);
                        this.repaint();
            }
            public void paint(Graphics g) {
                        super.paint(g);
                        Graphics2D g2 = (Graphics2D)g;
                        g2.draw(rect);
            }
            public int getShapeX() { return xUpperLeftCorner; }
            public int getShapeY() { return yUpperLeftCorner; }
            public int getShapeWidth() { return width; }
            public int getShapeHeight() { return height; }
            private int xUpperLeftCorner = 0;
            private int yUpperLeftCorner = 0;
            private int width = 200;
            private int height = 100;
            private Rectangle rect;

}
```



SRP – Single Responsibility Principle

Καλή Σχεδίαση

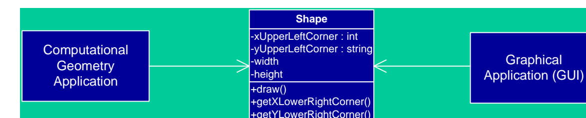
```
public class GeometricShape {

   public GeometricShape(Shape theShape)
            {
                        drawnShape = theShape;
            }
   public int getXLowerRightCorner() {
                        return drawnShape.getShapeX() + drawnShape.getShapeWidth();
            }
   public int getYLowerRightCorner() {
                        return drawnShape.getShapeY() + drawnShape.getShapeHeight();
            }
            private Shape drawnShape;

}
```



SRP – Single Responsibility Principle

Καλή Σχεδίαση

```
public class Main {
            public static void main(String[] args) {

                        Shape myShape = new Shape();
                        JFrame f = new JFrame();
                        f.setSize(600, 600);
                        f.setContentPane(myShape);
                        f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
                        myShape.draw(100, 100);
                        f.setVisible(true);
                        GeometricShape Geom = new GeometricShape(myShape);
                        System.out.println("Lower Right Corner. X: " +
                                    Geom.getXLowerRightCorner()
                                      + " Y: " + Geom.getYLowerRightCorner());

            }
}
```



SRP – Single Responsibility Principle

Ποσοτικοποίηση του βαθμού συνεκτικότητας μιας κλάσης: Μετρική LCOM (Lack of Cohesion between Methods), Chidamber & Kemerer, 1994

$\{I_i\}$: σύνολο μελών δεδομένων που χρησιμοποιούνται από τη μέθοδο M_i

Δύο μέθοδοι M_i και M_j θεωρούνται συνεκτικές εάν $\{I_i\} \cap \{I_j\} \neq \varnothing$.

Καθώς κάθε κλάση περιλαμβάνει n μεθόδους $(M_1, M_2, ..., M_n)$, σχηματίζονται δύο σύνολα από ζεύγη μεθόδων σε κάθε κλάση:

$P = \{(M_i, M_j) \mid \{I_i\} \cap \{I_j\} = \varnothing\}$: το σύνολο των μη συνεκτικών ζευγών μεθόδων

$Q = \{(M_i, M_j) \mid \{I_i\} \cap \{I_j\} \neq \varnothing\}$: το σύνολο των συνεκτικών ζευγών μεθόδων

Η μετρική LCOM ορίζεται ως εξής:

$$LCOM = \begin{cases} |P| - |Q| & , \quad \text{αν } |P| > |Q| \\ 0 & , \quad \text{σε κάθε άλλη περίπτωση} \end{cases}$$

Κατά συνέπεια, όσο μεγαλύτερη η συνεκτικότητα, τόσο μικρότερη η τιμή της LCOM

SRP – Single Responsibility Principle

Τι είναι μία αρμοδιότητα ?

Στα πλαίσια της ανωτέρω αρχής μία αρμοδιότητα ορίζεται ως μία "αιτία αλλαγών".

```
interface Modem
{
  public void dial(String pno);
  public void hangup();
  public void send(char c);
  public char recv();
}
```

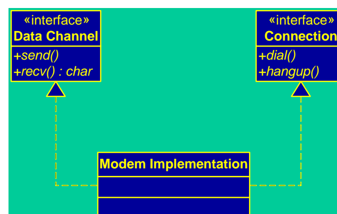
Οι αρμοδιότητες που υπάρχουν εδώ είναι δύο. Θα πρέπει αυτές να διαχωριστούν ?

Η απάντηση εξαρτάται από τον τρόπο με τον οποίο αλλάζει η εφαρμογή.

SRP – Single Responsibility Principle

Αν η εφαρμογή αλλάζει συχνά κατά τέτοιο τρόπο ώστε να τροποποιείται η υπογραφή των μεθόδων σύνδεσης, τότε εμφανίζεται δυσκαμψία καθώς οι κλάσεις που καλούν τις μεθόδους επικοινωνίας `send` και `recv` θα πρέπει κάθε φορά να επαναμεταγλωττίζονται.

Σε αυτή την περίπτωση οι δύο αρμοδιότητες θα πρέπει να διαχωριστούν. Με αυτό τον τρόπο, **οι εφαρμογές που χρησιμοποιούν είτε τη διασύνδεση Data Channel, είτε τη διασύνδεση Connection** αποσυνδέονται μεταξύ τους.



SRP – Single Responsibility Principle

Από την άλλη, αν η εφαρμογή δεν αλλάζει έτσι ώστε να τροποποιούνται οι αρμοδιότητες ανεξάρτητα, τότε δεν υπάρχει λόγος διαχωρισμού τους. Σε αυτή την περίπτωση, ο διαχωρισμός θα προκαλούσε περιττή πολυπλοκότητα.

Κατά συνέπεια, *ένας άξονας αλλαγών είναι άξονας αλλαγών μόνο αν οι αλλαγές συμβαίνουν πράγματι*.

Συμπερασματικά:

Η αρχή της Μοναδικής Αρμοδιότητας είναι ενδεχομένως η απλούστερη αρχή αντικειμενοστρεφούς σχεδίασης αλλά είναι και μία από τις δυσκολότερες να εφαρμοστεί.

Ο συγκερασμός αρμοδιοτήτων είναι κάτι που κάνουμε κατά φυσικό τρόπο. Ο εντοπισμός και ο διαχωρισμός αυτών των αρμοδιοτήτων αποτελεί κατά πολλούς ένα ουσιαστικό κομμάτι της σχεδίασης και ανάπτυξης λογισμικού.

Οι περισσότερες από τις αρχές και τους κανόνες που θα συζητηθούν στη συνέχεια σχετίζονται με τον ένα ή άλλο τρόπο με την αρχή SRP.

OCP – Open-Closed Principle

"*Όλα τα συστήματα λογισμικού αλλάζουν κατά τη διάρκεια ζωής τους*"

Αρχή της Ανοικτής-Κλειστής Σχεδίασης: *Οι οντότητες λογισμικού (κλάσεις, μονάδες, συναρτήσεις κλπ) θα πρέπει να είναι ανοικτές για επέκταση, αλλά κλειστές για τροποποίηση*

Αν η αρχή OCP εφαρμοστεί ορθά, τότε η υλοποίηση περαιτέρω αλλαγών του ιδίου τύπου επιτυγχάνεται ***με την προσθήκη νέου κώδικα, όχι με την τροποποίηση υπάρχοντος κώδικα που ήδη λειτουργεί***.

Ανοικτές για επέκταση: Η συμπεριφορά της μονάδας μπορεί να επεκταθεί

Κλειστές για τροποποίηση: Η επέκταση της συμπεριφοράς δεν οδηγεί σε αλλαγές του πηγαίου ή αντικείμενου κώδικα της μονάδας.

Πώς μπορούμε να ικανοποιήσουμε τις αντικρουόμενες αυτές ιδιότητες ?

OCP – Open-Closed Principle

Abstraction is the key

Σε οποιαδήποτε OOPL είναι δυνατόν να κατασκευαστούν αφαιρέσεις οι οποίες είναι σταθερές και καθορισμένες αναπαριστούν δε ένα απεριόριστο πλήθος συμπεριφορών

Μία μονάδα είναι δυνατό να χειρίζεται μία αφαίρεση

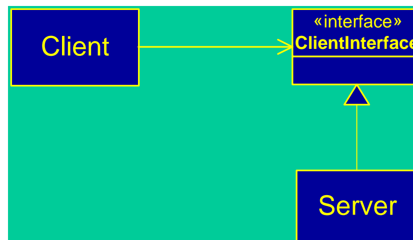
Μία μονάδα που χειρίζεται μία αφαίρεση είναι κλειστή για τροποποιήσεις καθώς εξαρτάται από την αφαίρεση που είναι σταθερή.

Ωστόσο, η συμπεριφορά της μονάδας μπορεί να επεκταθεί δημιουργώντας νέες υποκλάσεις της αφαίρεσης.

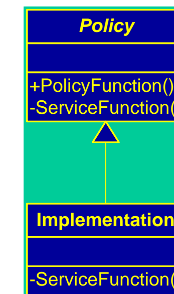
OCP – Open-Closed Principle

Παραβίαση OCP

Συμμόρφωση με OCP (STRATEGY Design Pattern)



OCP – Open-Closed Principle



Τα δύο αυτά πρότυπα είναι οι συνηθέστεροι τρόποι συμμόρφωσης με την αρχή OCP.

OCP – Open-Closed Principle

Shape Application: Μία εφαρμογή `Painter` πρέπει να μπορεί να σχεδιάζει κύκλους και τετράγωνα σε GUI. Κάποιο άλλο πρόγραμμα (π.χ. μια κλάση `ShapeApplication`) δημιουργεί μια λίστα από κύκλους και τετράγωνα και η εφαρμογή πρέπει να διατρέξει τη λίστα και να σχεδιάσει κάθε σχήμα

Υλοποίηση σε Java (download κώδικα από σελίδα) – Κατά τη διάρκεια της εκτέλεσης εξετάζεται ο τύπος κάθε αντικειμένου:

• Συμμορφώνεται η κλάση `Painter` με την αρχή OCP ? (Πέραν των άλλων προβλημάτων η κλάση `Painter` αναλαμβάνει "ξένες" αρμοδιότητες)

• Στην πράξη απαιτείται ο εντοπισμός όλων των σχετικών δομών switch και if/else (μετακίνηση σχημάτων, resizing, διαγραφή, αλλαγή χρωμάτων)

• Οι εντολές switch συνήθως εμπλέκουν λογικούς τελεστές, ή συνδυάζονται (κοινές ενέργειες για ορισμένα σχήματα)

OCP – Open-Closed Principle

Bad Design:

Δυσκαμψία: Όλες οι μονάδες που εξαρτώνται από την `Painter` πρέπει να μεταγλωττιστούν εκ νέου (C++) σε περίπτωση αλλαγών (π.χ. προσθήκη ενός Triangle). Αλυσιδωτά μεταγλωττίζονται εκ νέου όλα τα σχετιζόμενα αρχεία (εκ νέου εγκατάσταση βιβλιοθηκών, DLL κλπ)

Ευθραυστότητα: Η προσθήκη απαιτεί τον εντοπισμό, κατανόηση και τροποποίηση όλων των σχετικών εντολών switch και if/else

Ακινησία: Για μεταφορά της `Painter` απαιτείται και η μεταφορά των αρχείων που αφορούν Square και Circle

OCP – Open-Closed Principle

Improved Design (conformance to OCP):

Στο σύστημα προστίθεται μια νέα αφηρημένη κλάση *Shape* και *ενσωματώνει οτιδήποτε μπορεί να μεταβληθεί*

Η προσθήκη νέου σχήματος *δεν έχει απολύτως καμία επίδραση σε καμία από τις ανωτέρω μονάδες*

Δυσκαμψία: Εξαλείφθηκε διότι δεν απαιτείται να μεταγλωττιστούν άλλα αρχεία αντικείμενου κώδικα που είναι ήδη σε λειτουργία

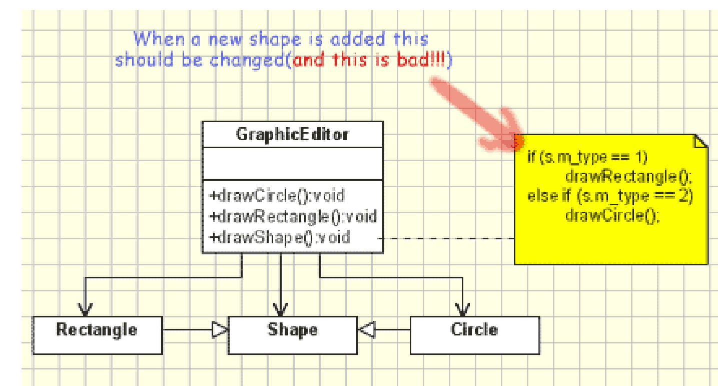
Ευθραυστότητα: Εξαλείφθηκε διότι δεν απαιτείται εντοπισμός σημείων κώδικα if/else και switch

Ακινησία: Εξαλείφθηκε διότι η `Painter` μπορεί να επαναχρησιμοποιηθεί χωρίς να πρέπει να μεταφερθούν οι κλάσεις `Circle` και `Square`

Το βελτιωμένο πρόγραμμα συμμορφώνεται με την αρχή OCP. *Τροποποιείται προσθέτοντας νέο κώδικα και όχι αλλάζοντας υπάρχοντα κώδικα*

OCP – Open-Closed Principle

Παράδειγμα Κακής Σχεδίασης



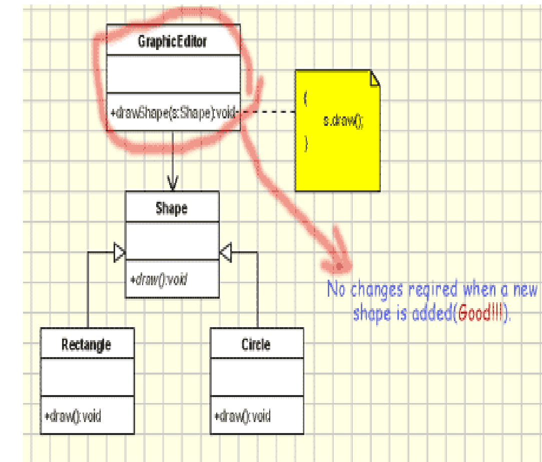
OCP – Open-Closed Principle

Παράδειγμα Καλής Σχεδίασης

```
class GraphicEditor {

    public void drawShape(Shape s) {
    if (s.m_type==1)
    drawRectangle(s);
    else if (s.m_type==2)
    drawCircle(s);
    }
    public void drawCircle(Circle r) {....}
    public void drawRectangle(Rectangle r) {....}
    }

    class Shape {
    int m_type;
    }

    class Rectangle extends Shape {
    Rectangle() {
    super.m_type=1;
    }
    }
    class Circle extends Shape {
    Circle() {
    super.m_type=2;
    }
    }
```


OCP – Open-Closed Principle

Παράδειγμα Καλής Σχεδίασης



OCP – Open-Closed Principle

Παράδειγμα Καλής Σχεδίασης

```
// Open-Close Principle - Good example
    class GraphicEditor {
    public void drawShape(Shape s) {
    s.draw();
    }
    }

    class Shape {
    abstract void draw();
    }

    class Rectangle extends Shape {
    public void draw() {
    // draw the rectangle
    }
    }
```


LSP – Liskov Substitution Principle

Κυριότεροι Μηχανισμοί OOPL: Αφαίρεση, Πολυμορφισμός

Σε statically typed languages μηχανισμός υποστήριξης είναι η κληρονομικότητα

Τι συνιστά "καλή ιεραρχία κλάσεων" ?

Αρχή Υποκατάστασης της Liskov: ***Οι παράγωγοι τύποι πρέπει να μπορούν να υποκαθιστούν τους βασικούς τους τύπους.***

Barbara Liskov (MIT, 1988, **MIT's magnificent seven:** Seven MIT women faculty are among the 60 top scientists cited in the November issues of Popular Science and Discover magazine):

Αυτό που είναι επιθυμητό εδώ είναι κάτι σαν την ακόλουθη αρχή υποκατάστασης: Αν για κάθε αντικείμενο o_1 του τύπου S υπάρχει ένα αντικείμενο o_2 του τύπου T τέτοιο ώστε για όλα τα προγράμματα P που ορίζονται υπό όρους του T, η συμπεριφορά του P παραμένει αναλλοίωτη όταν το o_1 υποκαταστήσει το o_2 τότε ο S είναι παράγωγος τύπος (υποκατηγορία) του T.

π.χ. ένα σύνολο (set) δεν μπορεί να αποτελέσει παράγωγο τύπο μιας λίστας (list)

LSP – Liskov Substitution Principle

Έστω μία συνάρτηση

f (B ref),

if *f (D ref)* συμπεριφέρεται λάθος , D subclass of B

τότε η D παραβιάζει την αρχή LSP.

<u>Λύση</u>:

Έλεγχος μέσα στην *f* αν περνά ως όρισμα αντικείμενο τύπου *D*

→ ΠΑΡΑΒΙΑΣΗ της αρχής OCP

LSP – Liskov Substitution Principle

Παραβίαση

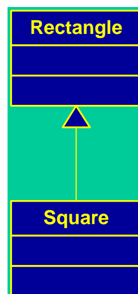
```
class Rectangle {
public:
  void   setWidth(double w)   {itsWidth = w;}
  void   setHeight(double h)  {itsHeight = h;}
  double getArea() const   {return itsHeight * itsWidth;}

private:
  double itsWidth;
  double itsHeight;
};
```


LSP – Liskov Substitution Principle

Αίτημα για αλλαγή: Δυνατότητα σχεδίασης και τετραγώνων πέραν των ορθογωνίων

Λέγεται συχνά, ότι η κληρονομικότητα είναι μία σχέση τύπου "Είναι" (IS-A). Με άλλα λόγια, αν ένα νέο είδος αντικειμένων μπορεί να ειπωθεί ότι ικανοποιεί τη σχέση "Είναι" ως προς ένα παλαιό είδος αντικειμένων, τότε η κλάση του νέου αντικειμένου πρέπει να κληρονομεί την κλάση του παλαιού αντικειμένου.

Για όλες τις λογικές χρήσεις και σκοπούς, ένα τετράγωνο *είναι* ένα ορθογώνιο



LSP – Liskov Substitution Principle

1η ένδειξη: ένα Square δεν χρειάζεται και το ύψος και το πλάτος ως ιδιότητες (σπατάλη μνήμης)

2η ένδειξη: κληρονόμηση των μεθόδων setWidth και setHeight. Λύση:

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

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

Μετά την κλήση οποιασδήποτε μεθόδου το αντικείμενο Square παραμένει γεωμετρικώς ορθό τετράγωνο (invariants άθικτες)

LSP – Liskov Substitution Principle

Έστω ότι σε κάποια εφαρμογή:

```
void f(Rectangle* r)
{
   r->setWidth(32);      //καλείται η Rectangle::setWidth
}
```

Παραβίαση αρχής LSP: Η συνάρτηση f δεν λειτουργεί για παράγωγες κλάσεις της παραμέτρου της (θα κληθεί η setWidth της Rectangle και όχι της Square)

Για τη διόρθωση του προβλήματος πρέπει να τροποποιήσουμε τη βασική κλάση Rectangle. = Παραβίαση αρχής OCP


```
class Rectangle {
public:
  virtual void   setWidth(double w)   {itsWidth = w;}
  virtual void   setHeight(double h)  {itsHeight = h;}
  double         getHeight() const    {return itsHeight;}
  double         getWidth() const     {return itsWidth;}

private:
  Point  itsTopLeft;
  double itsWidth;
  double itsHeight;
};

class Square : public Rectangle {
public:
  virtual void setWidth(double w);
  virtual void setHeight(double h);
};

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

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


LSP – Liskov Substitution Principle

The Real Problem:

Η σχεδίαση φαίνεται συνεπής ως προς τον εαυτό της. Ως προς τους άλλους ?

Έστω η συνάρτηση:

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

έχοντας προσθέσει στην κλάση Rectangle τη μέθοδο getArea

Η συνάρτηση λειτουργεί τέλεια για αντικείμενα Rectangle. Προκαλεί assertion error αν περάσουμε αντικείμενο Square. *Ο συγγραφέας της g θεώρησε ότι αλλάζοντας το πλάτος ενός αντικειμένου Rectangle, το ύψος του παραμένει ανέπαφο*

LSP – Liskov Substitution Principle

Ποιανού είναι το σφάλμα ?

"Το πρόβλημα έγκειται στην g" – ο συγγραφέας της δεν είχε το δικαίωμα να υποθέσει ότι το πλάτος και το ύψος είναι ανεξάρτητα. Ωστόσο, υπάρχουν αναλλοίωτες για ένα ορθογώνιο, και μία από αυτές είναι ότι οι δύο διαστάσεις είναι ανεξάρτητες

Ο συγγραφέας της Square παραβίασε αυτή την αναλλοίωτη, λέγοντας ότι το τετράγωνο "είναι" ένα ορθογώνιο

Αυτό που έχει ενδιαφέρον είναι ότι ο συγγραφέας της Square δεν παραβίασε μία αναλλοίωτη της Square. Κληρονομώντας την Rectangle, ο συγγραφέας της Square παραβίασε μία αναλλοίωτη της Rectangle !

Ένα μοντέλο που εξετάζεται μεμονωμένα δεν μπορεί να επικυρωθεί πλήρως.

Τι συνέβει? Δεν είναι τελικά ένα τετράγωνο και ένα ορθογώνιο ?

Όσον αφορά τη συμπεριφορά ΌΧΙ. Η αρχή LSP επιβάλει ότι σε μία σχέση κληρονομικότητας <u>η συμπεριφορά των παράγωγων κλάσεων πρέπει να μπορεί να υποκαταστήσει τη συμπεριφορά των βασικών κλάσεων</u>

LSP – Liskov Substitution Principle

Design by Contract

Ο συγγραφέας κάθε κλάσης διατυπώνει ρητά τις συμβάσεις υπό τις οποίες λειτουργεί σωστά η κλάση. Οποιοσδήποτε γράφει πρόγραμμα πελάτη αυτής της κλάσης, μπορεί να βασιστεί σε αυτές τις συμβάσεις.

```
{προσυνθήκες}
  μέθοδος
{μετασυνθήκες}
```

ισοδυναμεί με: "εάν ισχύουν οι προσυνθήκες πριν από την εκτέλεση της μεθόδου, μετά το πέρας της εκτέλεσης της μεθόδου, οι μετασυνθήκες θα πρέπει να ισχύουν"

Για το παράδειγμα της μεθόδου setWidth είναι:

```
{old.itsHeight == itsHeight } //η μεταβλητή old.itsHeight  πρέπει
                              //να έχει την τιμή του ύψους
  Rectangle::setWidth(double w);
{(itsWidth == w) && (itsHeight == old.itsHeight) }
```


LSP – Liskov Substitution Principle

Για την κληρονομικότητα ισχύει:

Η επικάλυψη μιας μεθόδου (σε μία υποκλάση) μπορεί μόνο να αντικαταστήσει την αρχική προσυνθήκη με μία ίδια ή ασθενέστερη και την αρχική μετασυνθήκη με μία ίδια ή ισχυρότερη

Όταν ένας πελάτης χρησιμοποιεί ένα αντικείμενο μέσω της διασύνδεσης της βασικής κλάσης, ο πελάτης γνωρίζει μόνο τις προσυνθήκες και μετασυνθήκες για την βασική κλάση. Κατά συνέπεια, οι παράγωγες κλάσεις δεν θα πρέπει να υποθέτουν ότι οι πελάτες θα ικανοποιήσουν προσυνθήκες οι οποίες είναι ισχυρότερες από αυτές που απαιτούνται από τη βασική κλάση. Επιπλέον, πρέπει να ικανοποιούν όλες τις μετασυνθήκες της βασικής κλάσης (ή ακόμα ισχυρότερες).

Η μετασυνθήκη για την `setWidth` της κλάσης Square είναι ασθενέστερη:

```
{none }                     //OK, η προσυνθήκη είναι ασθενέστερη
  Square::setWidth(double w);
{itsWidth == w}  //Πρόβλημα, η μετασυνθήκη είναι ασθενέστερη !!
```


LSP – Liskov Substitution Principle

Συμπερασματικά:

• Θα πρέπει να εξασφαλίζεται (μέσω εκτενών ελέγχων) ότι η χρήση παράγωγων κλάσεων σε όλα τα σημεία όπου χρησιμοποιούνται οι βασικές κλάσεις, δεν αλλοιώνει τη λειτουργικότητα του συστήματος

• Η αρχή της υποκατάστασης παραβιάζεται αν μια συνάρτηση ή πρόγραμμα πελάτης λειτουργεί για μια βασική κλάση αλλά δεν λειτουργεί ορθά για παράγωγες κλάσεις αυτής.

DIP – Dependency Inversion Principle

Αρχή Αντιστροφής των Εξαρτήσεων:

α. ***Οι μονάδες υψηλού επιπέδου δεν θα πρέπει να εξαρτώνται από μονάδες χαμηλού επιπέδου.***

β. ***Οι αφαιρέσεις δεν θα πρέπει να εξαρτώνται από λεπτομέρειες. Οι λεπτομέρειες θα πρέπει να εξαρτώνται από αφαιρέσεις.***

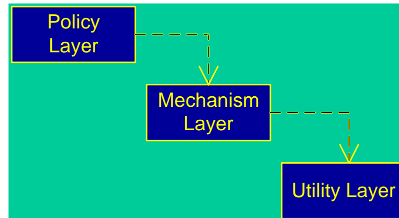
Σε συμβατικές μεθοδολογίες ανάπτυξης λογισμικού, όπως η Δομημένη Ανάλυση και Σχεδίαση (SADT) δημιουργούνται δομές όπου οι υψηλότερες μονάδες στην ιεραρχία καλούν (και άρα εξαρτώνται) τις μονάδες χαμηλότερων επιπέδων. Το αποτέλεσμα είναι η γενική στρατηγική του συστήματος να εξαρτάται από τις λεπτομέρειες υλοποίησης των μονάδων χαμηλού επιπέδου. (Ανάγνωση Roberts κεφ. 11 εισαγωγή)

Μία τέτοια δομή είναι προφανώς παράλογη: Οι μονάδες στα υψηλότερα επίπεδα εμπεριέχουν τους γενικούς κανόνες της εφαρμογής και του πεδίου του προβλήματος (business rules) και συνεπώς πρέπει να έχουν προτεραιότητα και να είναι ανεξάρτητες από τις λεπτομέρειες υλοποίησης. Επιπλέον, αυτοί οι κανόνες είναι συνήθως ένα στοιχείο που θέλουμε να μπορεί να επαναχρησιμοποιηθεί.

DIP – Dependency Inversion Principle

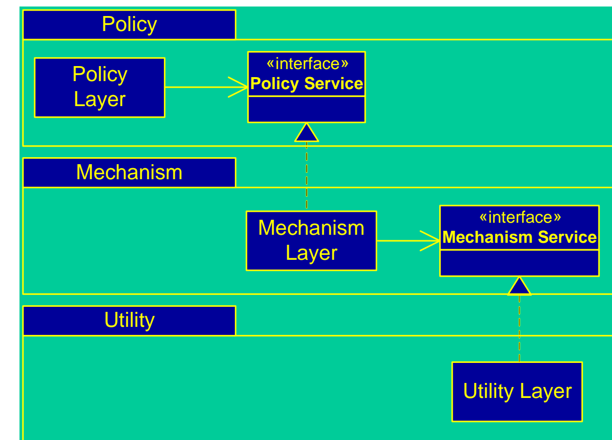
Διαστρωμάτωση (Layering):

Booch: μία καλή αρχιτεκτονική περιέχει σαφώς ορισμένα επίπεδα, καθένα εκ των οποίων παρέχει ένα συνεπές και συμπαγές σύνολο υπηρεσιών δια μέσου μιας ρητά διατυπωμένης διασύνδεσης. Αφελής Μετάφραση:



DIP – Dependency Inversion Principle

Βελτιωμένη Διαστρωμάτωση (Inversion):



Hollywood principle: "Don't call us, we'll call you" You implement the interfaces, you get registered. You get called when the time is right.

DIP – Dependency Inversion Principle

"Να εξαρτάστε από αφαιρέσεις"

Ο κανόνας προτείνει για τον κατασκευαστή κάθε κλάσης να μην εξαρτάται από συγκεκριμένες κλάσεις – ότι δηλαδή όλες οι σχέσεις σε ένα πρόγραμμα θα πρέπει να καταλήγουν σε μία αφηρημένη κλάση ή σε μία διασύνδεση. Σύμφωνα με αυτόν τον κανόνα:

- **Καμία μεταβλητή δεν θα πρέπει να διατηρεί έναν δείκτη ή μία αναφορά προς κάποια συγκεκριμένη κλάση**
- **Καμία κλάση δεν θα πρέπει να κληρονομεί μία συγκεκριμένη κλάση**
- **Καμία μέθοδος δεν θα πρέπει να επικαλύπτει υλοποιημένη μέθοδο οποιασδήποτε από τις γονικές της κλάσεις**

Οι ανωτέρω κανόνες θα παραβιαστούν τουλάχιστον μία φορά, στο σημείο που κατασκευάζουμε στιγμιότυπα

Επιπλέον, δεν υπάρχει λόγος να τους ακολουθήσουμε αν γνωρίζουμε ότι μία συγκεκριμένη κλάση δεν θα αλλάξει

DIP – Dependency Inversion Principle

Παράδειγμα

Ένας ελεγκτής (Controller) αντιλαμβάνεται κάποια αλλαγή στο εξωτερικό περιβάλλον και αποστέλλει μήνυμα ενεργοποίησης/απενεργοποίησης στο σχετιζόμενο αντικείμενο συναγερμού (LampAlarm)

- Η κλάση Controller ελέγχει αντικείμενα VisibleAlarm και μόνον αυτά
- Αν αλλάξει ο πηγαίος κώδικας της LampAlarm (π.χ. πράσινο φως) θα πρέπει να επαναμεταγλωττιστεί (C++) και ο κώδικας της Controller
- Αν θέλουμε να συνδέσουμε τον ελεγκτή με άλλο μηχανισμό (π.χ. έναν κινητήρα), πρέπει να τροποποιήσουμε τον ελεγκτή
- Η πολιτική υψηλού επιπέδου δεν έχει διαχωριστεί από την υλοποίηση

DIP – Dependency Inversion Principle

Εντοπισμός της Υποκείμενης Αφαίρεσης και Αντιστροφή της Εξάρτησης:

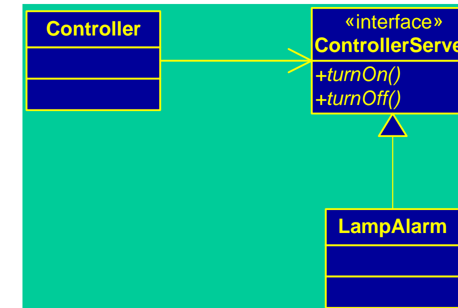
Η υποκείμενη αφαίρεση είναι η ανίχνευση κάποιας αλλαγής και η αποστολή μηνύματος ενεργοποίησης/απενεργοποίησης σε κάποιον παραλήπτη

- Ποιος είναι ο Μηχανισμός Ανίχνευσης; αδιάφορο (λεπτομέρεια)
- Ποιο είναι το αντικείμενο παραλήπτης; αδιάφορο (λεπτομέρεια)

Εφαρμογή της Αρχής DIP: Αντιστροφή της εξάρτησης του ελεγκτή από το αντικείμενο VisibleAlarm

DIP – Dependency Inversion Principle

Εντοπισμός της Υποκείμενης Αφαίρεσης και Αντιστροφή της Εξάρτησης:



Ευελιξία: Τα αντικείμενα Controller μπορούν να ελέγξουν οτιδήποτε συμμορφώνεται με τη διασύνδεση ControllerServer. Ακόμα και αντικείμενα που δεν έχουν ακόμη επινοηθεί.

DIP – Dependency Inversion Principle

Κακή Σχεδίαση

```
// Dependency Inversion Principle - Bad example
class Worker {
        public void work() {}                    }
class Manager {
        Worker m_worker;
        public void setWorker(Worker w) {
                m_worker=w;
        }
        public void manage() {
                m_worker.work();
        }
}
class SuperWorker {
        public void work() {
                //.... working much more
        }
}
```


DIP – Dependency Inversion Principle

Καλή Σχεδίαση

```
interface IWorker {
        public void work();
}
class Worker implements IWorker{
        public void work() {}
}
class SuperWorker  implements IWorker{
        public void work() {} //.... working much more
}
class Manager {
        IWorker m_worker;
        public void setWorker(IWorker w) {
                m_worker=w;
        }
        public void manage() {
                m_worker.work();
        }
}
```

DIP – Dependency Inversion Principle

Συμπερασματικά:

• **Διαδικασιακός προγραμματισμός: Δομές όπου οι μονάδες υψηλού επιπέδου εξαρτώνται από τις μονάδες χαμηλού επιπέδου**

• **Ατυχής Επιλογή: Η πολιτική του συστήματος είναι ευάλωτη σε αλλαγές της υλοποίησης**

•**Αντικειμενοστρεφής Προγραμματισμός: Δομές αντεστραμμένες**

•**Πολλοί θεωρούν τη διαφορά αυτή ως ειδοποιό.**

ISP – Interface-Segregation Principle

Στη συνέχεια, ως διασύνδεση θεωρείται το σύνολο των λειτουργιών μιας οντότητας

Η σχεδίαση με διασυνδέσεις είναι καλή τακτική.

Ωστόσο, θα πρέπει να εξετάζεται προσεκτικά η ίδια η σχεδίαση των διασυνδέσεων

Η αρχή ISP διαπραγματεύεται τα μειονεκτήματα των ογκωδών διασυνδέσεων ("fat" interfaces).

Οι κλάσεις που έχουν μεγάλο αριθμό δημόσιων μεθόδων είναι κλάσεις των οποίων η διασύνδεση δεν είναι συνεκτική και οι μέθοδοι μπορούν να διαχωριστούν σε ένα σύνολο διασυνδέσεων. Κάθε σύνολο εξυπηρετεί μία διαφορετική ομάδα από πελάτες.

Η αρχή ISP αναγνωρίζει ότι υπάρχουν αντικείμενα που χρειάζεται να έχουν μη συνεκτικές διασυνδέσεις. Ωστόσο, προτείνει ότι οι πελάτες δεν θα πρέπει να γνωρίζουν αυτά τα αντικείμενα μέσω μίας μοναδικής κλάσης. Αντιθέτως, οι πελάτες θα πρέπει να γνωρίζουν για πολλές αφηρημένες κλάσεις βάσης με συνεκτικές διασυνδέσεις.

Αρχή Διαχωρισμού των Διασυνδέσεων (Α' ορισμός):

Πολλές εξειδικευμένες διασυνδέσεις είναι προτιμότερες από μια γενική διασύνδεση

ISP – Interface-Segregation Principle

Θεωρούμε ένα σύστημα ασφαλείας. Στο σύστημα υπάρχουν πόρτες (αντικείμενα Door) που μπορούν να κλειδωθούν και να ξεκλειδωθούν.

```
class Door
{
public:
	virtual void lock() = 0;
	virtual void unlock() = 0;
};
```

Η κλάση είναι αφηρημένη έτσι ώστε οι πελάτες να μπορούν να χρησιμοποιούν αντικείμενα που συμμορφώνονται με τη διασύνδεση Door χωρίς να πρέπει να εξαρτώνται από συγκεκριμένες υλοποιήσεις της Door (βλέπε DIP).

ISP – Interface-Segregation Principle

Θεωρούμε τώρα μία τέτοια υλοποίηση ασφαλούς πόρτας (ProtectedDoor) η οποία ενεργοποιεί έναν συναγερμό αν παραβιαστεί. Για το σκοπό αυτό κάθε αντικείμενο ProtectedDoor επικοινωνεί με ένα άλλο αντικείμενο της κλάσης PasswordProtector.

```
class PasswordProtector
{
public:
	void Register(int code, PasswordClient* client);
	void check(int code);
private:
	int safeNumber;
	PasswordClient* myClient;
};
```

ISP – Interface-Segregation Principle

Κάθε πόρτα (αντικείμενο Door) που επιθυμεί να ασφαλιστεί καλεί τη μέθοδο Register του αντικειμένου PasswordProtector καταχωρίζοντας τον εαυτό της. Τα ορίσματα της μεθόδου είναι ένας κωδικός για τον έλεγχο και ένας δείκτης προς ένα αντικείμενο PasswordClient του οποίου το ξεκλείδωμα θα ελέγχεται με κωδικό.

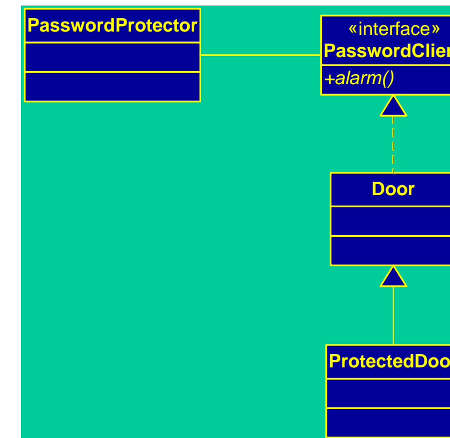
Αν ο κωδικός κατά το άνοιγμα της πόρτας δεν είναι ίδιος με αυτόν που καταχωρήθηκε, θα καλείται η μέθοδος `alarm()` του αντικειμένου PasswordClient.

Για λόγους ανοικτής-κλειστής σχεδίασης, η PasswordClient είναι μία διασύνδεση (αφηρημένη κλάση βάσης) την οποία πρέπει να υλοποιεί οποιοδήποτε αντικείμενο επιθυμεί να ασφαλιστεί.

Πώς μπορεί επομένως η κλάση PasswordProtector να επικοινωνεί με αντικείμενα ProtectedDoor; Υπάρχουν πολλές εναλλακτικές όσον αφορά τη σχεδίαση του συστήματος.

ISP – Interface-Segregation Principle

Αφελής Προσέγγιση:



ISP – Interface-Segregation Principle

Η ανωτέρω πρακτική είναι συνηθισμένη, δημιουργεί όμως προβλήματα:

• Η κλάση Door υλοποιεί πλέον τη διασύνδεση PasswordClient. Δεν χρειάζονται όλες οι πόρτες ασφάλεια μέσω κωδικού.

• Αν απαιτούνται παράγωγοι της Door χωρίς λειτουργικότητα κωδικού, αυτές οι παράγωγοι κλάσεις **θα πρέπει να παράσχουν εκφυλισμένες υλοποιήσεις για τη μέθοδο `alarm()` – μία ενδεχόμενη παραβίαση της αρχής LSP**. Επιπλέον, οι εφαρμογές οι οποίες θα χρησιμοποιήσουν αυτές τις παράγωγες κλάσεις θα πρέπει να εισάγουν και τον ορισμό της διασύνδεσης PasswordClient, παρόλο που αυτή δεν χρησιμοποιείται. Εδώ εμφανίζονται τα συμπτώματα της περιττής πολυπλοκότητας και της περιττής επανάληψης.

• Η διασύνδεση Door έχει "ρυπανθεί" με μία μέθοδο την οποία δεν χρειάζεται. **Εξαναγκάστηκε να συμπεριλάβει αυτή τη μέθοδο μόνο προς όφελος κάποιων από τις υποκλάσεις της**. Αν η πρακτική αυτή επεκταθεί, τότε κάθε φορά που μία παράγωγος χρειάζεται μία νέα μέθοδο, η μέθοδος αυτή θα προστίθεται στη βασική κλάση.

ISP – Interface-Segregation Principle

• Οι Door και PasswordClient αντιπροσωπεύουν διασυνδέσεις που χρησιμοποιούνται από τελείως διαφορετικούς πελάτες. Καθώς οι πελάτες είναι ξεχωριστοί, οι διασυνδέσεις θα πρέπει να παραμείνουν ξεχωριστές επίσης. Ο λόγος είναι ότι οι πελάτες ασκούν επιρροή στις διασυνδέσεις που χρησιμοποιούν.

• Συνήθως, όποτε θεωρούμε δυνάμεις που προκαλούν αλλαγές στο λογισμικό, σκεπτόμαστε για πώς οι διασυνδέσεις επιβάλουν τροποποιήσεις στους χρήστες τους. Ωστόσο, υπάρχει και μία δύναμη που λειτουργεί προς την αντίθετη κατεύθυνση.

• Αν υποθέσουμε για παράδειγμα ότι η ενεργοποίηση του συναγερμού από το αντικείμενο PasswordProtector μπορεί να γίνει σε δύο επίπεδα έντασης (ανάλογα με το αν ο ιδιοκτήτης βρίσκεται μέσα στο σπίτι ή όχι), τότε η αφηρημένη μέθοδος alarm της διασύνδεσης PasswordClient πρέπει να τροποποιηθεί. Η αλλαγή είναι σχετικά απλή (προσθήκη μιας παραμέτρου int στη λίστα των παραμέτρων της alarm) και την αποδεχόμαστε καθώς πρόκειται για μία λογική απαίτηση.

ISP – Interface-Segregation Principle

• Μία τέτοια αλλαγή είναι αναμενόμενο ότι θα επηρεάσει όλους τους χρήστες της PasswordClient. Ωστόσο, με τη σχεδίαση του σχήματος 5.1 **θα πρέπει να τροποποιηθεί και η κλάση Door καθώς και όλοι οι πελάτες της Door**!

• Αντίστοιχα θα πρέπει να επαναμεταγλωττιστούν και όλες οι εφαρμογές που τις χρησιμοποιούν).

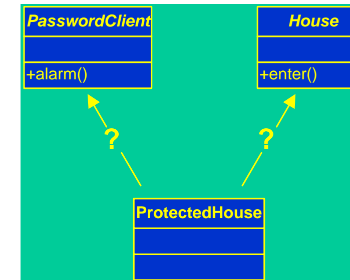
• *Για ποιο λόγο μία μικρή βελτίωση που αφορά την ενεργοποίηση συναγερμού από λάθος κωδικό να έχει οποιαδήποτε επίδραση σε πελάτες παράγωγων κλάσεων της Door οι οποίες δεν έχουν απολύτως καμία ανάγκη από έλεγχο κωδικού;* Εδώ μία αλλαγή σε ένα τμήμα του συστήματος επηρεάζει άλλα, τελείως άσχετα τμήματα, αυξάνοντας το κόστος, τη δυσκολία και το ρίσκο πραγματοποίησης αλλαγών δραματικά.

Αρχή Διαχωρισμού των Διασυνδέσεων (Β' ορισμός):

Οι πελάτες δεν θα πρέπει να εξαναγκάζονται σε εξάρτηση από μεθόδους που δεν χρησιμοποιούν.

ISP – Interface-Segregation Principle

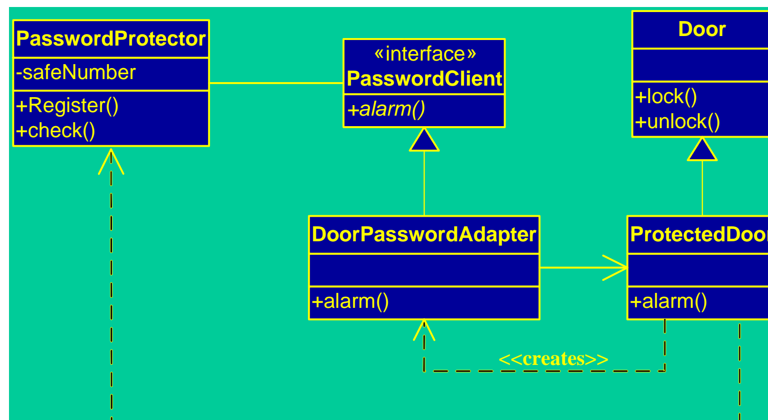
• Ακόμα όμως και αν διαχωριστούν οι διασυνδέσεις, το τελικό ζητούμενο είναι ένα αντικείμενο που θα ενσωματώνει και τα δύο είδη λειτουργικότητας



2 Λύσεις:

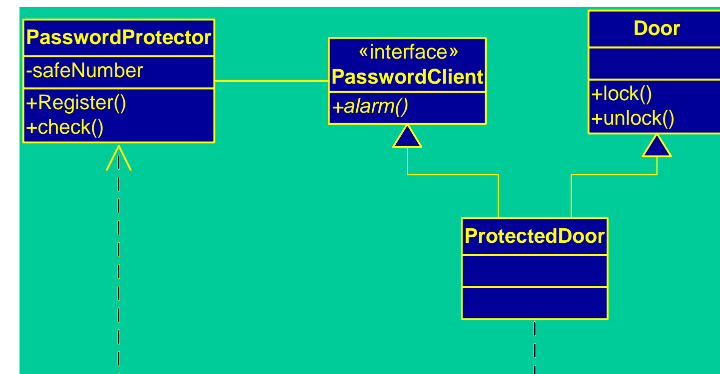
ISP – Interface-Segregation Principle

Διαχωρισμός μέσω Αποστολής Μηνυμάτων (Delegation)



ISP – Interface-Segregation Principle

Διαχωρισμός μέσω Πολλαπλής Κληρονομικότητας



ISP – Interface-Segregation Principle

Συμπερασματικά:

Σε μια καλή σχεδίαση, οι πελάτες θα πρέπει να εξαρτώνται μόνο από τις μεθόδους τις οποίες καλούν

Αυτό επιτυγχάνεται διαχωρίζοντας τη διασύνδεση σε πολλές διασυνδέσεις, μία για κάθε κατηγορία πελατών