ΑΛΕΞΑΝΔΡΕΙΟ Τ.Ε.Ι ΘΕΣΣΑΛΟΝΙΚΗΣ
ΣΧΟΛΗ ΤΕΧΝΟΛΟΓΙΚΩΝ ΕΦΑΡΜΟΓΩΝ
ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ

Πτυχιακή εργασία

«Τεκμηρίωση Προτύπων Σχεδίασης με τη UML»

Της φοιτήτριας:
Μαργαρίτη Θεοδώρα
Α.Μ 052837

Επιβλέπων καθηγητής:
Απόστολος Αμπατζόγλου

Θεσσαλονίκη 2012

ΠΕΡΙΛΗΨΗ

Από τον ορισμό των προτύπων σχεδίασης γίνεται σαφής ο ρόλος τους σε ότι αφορά την δημιουργία ενός ορθά δομημένου συστήματος με πολλές δυνατότητες επαναχρησιμοποίησης και προσαρμογής τμημάτων κώδικά. Παράλληλα, οι εφαρμογές ανοιχτού λογισμικού κερδίζουν συνεχώς έδαφος, τόσο μεταξύ ιδιωτών, όσο και (μικρομεσαίων, κυρίως) επιχειρήσεων, καθώς αποτελούν μια πολύ σημαντική και διαρκώς εμπλουτιζόμενη πηγή εργαλείων, χαμηλού κόστους και μεγάλης ευελιξίας. Οι εφαρμογές ανοικτού λογισμικού μπορούν να τροποποιηθούν, να εξελιχτούν και να προστεθούν σε αυτές, επιπλέον, επιθυμητές λειτουργίες, προκειμένου να προσαρμοσθούν στις εξειδικευμένες ανάγκες του τελικού κάθε φορά χρήστη. Μέρος της εντυπωσιακής τους διάδοσης, επομένως, οφείλεται στο γεγονός ότι ο κώδικάς τους είναι επεκτάσιμος και παραμετροποιήσιμος. Έτσι, ο προγραμματισμός τέτοιων εφαρμογών, δίνει ακόμη μεγαλύτερη σημασία στην χρήση προτύπων.

Τα πρότυπα σχεδίασης, στον αντικειμενοστρεφή προγραμματισμό θέτουν μια κοινή βάση μεταξύ των σχεδιαστών λογισμικού, σκιαγραφώντας μια συγκεκριμένη δομή στον κώδικα, πάνω στην οποία θα βασιστεί η εξέλιξη και βελτίωση της κάθε εφαρμογής, από οποιονδήποτε θελήσει να ασχοληθεί, στα πλαίσια της κοινότητας του ανοιχτού λογισμικού.

Στα πλαίσια της παρούσας εργασίας, επιδιώκεται να γίνει μια μελέτη της χρήσης από πρότυπα σχεδίασης τόσο σε εργαλεία ανάπτυξης αντικειμενοστρεφούς λογισμικού, όσο και σε εφαρμογές ηλεκτρονικού εμπορίου, προγραμματισμένων σε Java. Πρόκειται για δυο τομείς με έντονο ενδιαφέρον και ραγδαία διάδοση στον χώρο του ανοικτού λογισμικού, που ωστόσο διαφέρει αρκετά από άποψη αντικειμένου, ώστε να είναι δυνατόν να γίνουν όχι μόνο τετριμμένες ερευνητικές υποθέσεις, αλλά και συγκρίσεις σχετικά με την ποιότητα του κώδικα, σε ότι αφορά την χρήση των προτύπων.

Σκοπός της έρευνας είναι να προσδιοριστούν τα πιο συχνά χρησιμοποιημένα πρότυπα σχεδίασης ανά κατηγορία ανοικτού λογισμικού, αν υπάρχει όντως κάποια αξιοσημείωτη διαφορά μεταξύ των δυο κατηγοριών της έρευνας. Γίνεται, επίσης, μια αξιολόγηση άλλων παραγόντων που εικάζεται ότι θα μπορούσαν να σχετίζονται με το βαθμό και το είδος των προτύπων που χρησιμοποιούνται.

ABSTRACT

The definition of design templates make clear the role in terms of creating a properly structured system with many possibilities for reuse and adapt code snippet.
Meanwhile, open source applications are gaining ground both between individuals and (small, mostly) business, including are a very important and constantly enriched source tools Low cost and high flexibility. The open source applications can be modified to be developed and added to them,
moreover, desired functions, to adapt to specialized needs of the end user each time. Part of the impressive spread, therefore, due to the fact that the code is extensible and configurable. Thus, the programming of such applications, giving even greater emphasis on use of standards.
The design patterns, object-oriented programming to make a common basis between software designers, sketching a specific structure in the code, upon which to base the development and improvement of any application by anyone wanting to deal in part of the open source community.
In the present work seeks to make a study of use of design patterns in both object-oriented software development tools, and in e-commerce applications, programmed in Java. These two areas with intense interest and rapid dissemination in the area of open software, but quite different in terms of object that can be done not only trivial affairs research, as well as comparisons on the quality of the code, regarding the use of standards.
The survey aims to identify the most frequently used design templates by category open source, if there is indeed a remarkable difference between the two categories of investigation. There is also an evaluation of other factors believed to could be related to the degree and nature of the standards
are used.

ΠΕΡΙΕΧΟΜΕΝΑ

ΛΙΣΤΑ ΕΙΚΟΝΩΝ

ΠΡΟΛΟΓΟΣ

Ως πρότυπο σχεδίασης (design pattern) ορίζεται, γενικά, ένας επίσημος τρόπος τεκμηρίωσης μιας λύσης για ένα πρόβλημα σε κάποιο συγκεκριμένο τομέα, μέσα από την εμπειρία. Η ιδέα καθιερώθηκε από τον αρχιτέκτονα Christopher Alexander (1977/79) στον τομέα της αρχιτεκτονικής, και έχει προσαρμοστεί για διάφορους άλλους κλάδους, συμπεριλαμβανομένης της επιστήμης των υπολογιστών. Στη ανάπτυξη λογισμικού, ένα πρότυπο σχεδίασης είναι μια γενική, επαναχρησιμοποιήσιμη λύση σε ένα πρόβλημα που εμφανίζεται, συνήθως, κατά τον σχεδιασμό.
Στον αντικειμενοστρεφή προγραμματισμό, τα πρότυπα σχεδίασης προσέλκυσαν έντονα το ενδιαφέρον της κοινότητας ανάπτυξης λογισμικού και διαδόθηκαν ευρέως, σχετικά σύντομα αφότου προτάθηκαν, το 1995. Η σχεδίαση επαναχρησιμοποιήσιμου αντικειμενοστρεφούς λογισμικού είναι γενικά ένα δύσκολο έργο, καθώς θα πρέπει να αφορά ένα συγκεκριμένο πρόβλημα, αλλά συγχρόνως να γενικεύεται τόσο, ώστε να μπορεί να αντιμετωπίσει παρόμοια μελλοντικά προβλήματα και ανάγκες, ελαχιστοποιώντας τον επανασχεδιασμό.
Οι υποστηρικτές των προτύπων σχεδίασης ισχυρίζονται ότι η χρήση τους οδηγεί στην δημιουργία ορθά δομημένων συστημάτων λογισμικού, τα οποία μπορούν εύκολα να συντηρηθούν ή να επαναχρησιμοποιηθούν. Ένα πρότυπο σχεδίασης αν και δεν μπορεί να μετατραπεί άμεσα σε κώδικα, περιγράφει τη λύση ενός προβλήματος, σε πολλές διαφορετικές καταστάσεις. Ορίζει τις σχέσεις και τις αλληλεπιδράσεις μεταξύ των κλάσεων και των αντικειμένων, χωρίς να προσδιορίζει τις ίδιες τις κλάσεις ή τα αντικείμενα της τελικής εφαρμογής. Βελτιώνει την αναγνωσιμότητα του κώδικα για τους προγραμματιστές που είναι εξοικειωμένοι με τα πρότυπα, κάνοντας ευκολότερη την συντήρηση και την βελτίωση των εφαρμογών. Κυρίως, όμως, συμβάλλει στην «Κάθε πρότυπο περιγράφει ένα πρόβλημα που εμφανίζεται ξανά και ξανά στο περιβάλλον μας, και στη συνέχεια περιγράφει τον πυρήνα της λύσης του προβλήματος αυτού, με τέτοιο τρόπο ώστε να μπορεί κανείς να χρησιμοποιήσει αυτή τη λύση ένα εκατομμύριο φορές, χωρίς ποτέ να το κάνει δύο φορές με τον ίδιο τρόπο», Christopher Alexander πρόληψη ζητημάτων που, ενώ δεν είναι εύκολα ορατά εξαρχής, μπορεί να προκαλέσουν μεγάλα προβλήματα στα επόμενα στάδια ανάπτυξης μιας εφαρμογής. Γενικότερα, η χρήση των προτύπων αναδεικνύεται σημαντική, καθώς με μικρή εμπειρία μπορεί κανείς να αξιοποιήσει τη συσσωρευμένη εμπειρία άλλων, επιταχύνοντας την διαδικασία ανάπτυξης λογισμικού. Όπως έχει ήδη αναφερθεί, ο κατάλογος των προτύπων σχεδίασης είναι μεγάλος. Ορισμένα από αυτά είναι προφανή ή αποτελούν την εξ' ορισμού επιλογή ενός σχεδιαστή λογισμικού. Άλλα πρότυπα αποτελούν λιγότερο προφανείς λύσεις και απαιτείται προσπάθεια για την κατανόηση του προβλήματος που επιλύουν όσο και του τρόπου υλοποίησής τους. Η βασικότερη κατηγοριοποίηση των προτύπων προτάθηκε από τους ίδιους τους συγγραφείς του πρώτου καταλόγου προτύπων και όρισε τρεις κατηγορίες, με βάση τον σκοπό και το πεδίο εφαρμογής. Σύμφωνα με αυτήν, τα πρότυπα σχεδίασης διακρίνονται σε: κατασκευαστικά (creational) τα οποία αφορούν την διεργασία δημιουργίας αντικειμένων, δομικά (structural) τα οποία ασχολούνται με τη σύνθεση κλάσεων/αντικειμένων και πρότυπα συμπεριφοράς (behavioral) που

χαρακτηρίζουν τους τρόπους με τους οποίους οι κλάσεις αλληλεπιδρούν και κατανέμουν τις αρμοδιότητες. Παράλληλα με αυτή την κατηγοριοποίηση, έχουν προταθεί και άλλες, με σκοπό να διευκολύνουν την μελέτη και κατανόηση των

.

ΚΕΦΑΛΑΙΟ 1 - Δημιουργικά πρότυπα σχεδίασης (Creational design patterns)

Τα κατασκευαστικά πρότυπα συμβάλλουν στην δημιουργία ενός συστήματος ανεξάρτητου από τον τρόπο με τον οποίο θα δημιουργηθούν τα αντικείμενά του. Όλες οι αντικειενοστρεφείς γλώσσες προγραμματισμού έχουν κάποιο προγραμματιστικό ιδίωμα για την δημιουργία νέων αντικειμένων (π.χ. τον τελεστή new. Τα πρότυπα σχεδίασης που ανήκουν στην κατηγορία των κατασκευαστικών προτύπων επιτρέπουν την συγγραφή μεθόδων που δημιουργούν νέα αντικείμενα, χωρίς την άμεση χρήση των συγκεκριμένων ιδιωμάτων. Το γεγονός αυτό επιτρέπει να αναπτυχθούν μέθοδοι κλάσεων που παράγουν ομάδες διαφορετικών αντικειμένων καθώς και την επέκτασή τους για νέα αντικείμενα, χωρίς την τροποποίηση του κώδικα μεθόδων. Σε αντίθετη περίπτωση, η δημιουργία ενός διαφορετικού αντικειμένου για κάθε περίπτωση,θα απαιτούσε τη χρήση ελέγχων if/else ή εντολών switch, στοιχεία που υποδηλώνουν κακή εφαρμογή των αρχών του αντικειμενοστρεφούς προγραμματισμού, καθώς δυσχεραίνουν τη συντήρηση λογισμικού.Βασικό χαρακτηριστικό των προτύπων αυτής της κατηγορίας είναι η χρήση της κληρονομικότητας. Τα πρότυπα αυτής της κατηγορίας ονομάζονται: Abstract Factory, Builder, Factory Method, Prototype και Singleton.

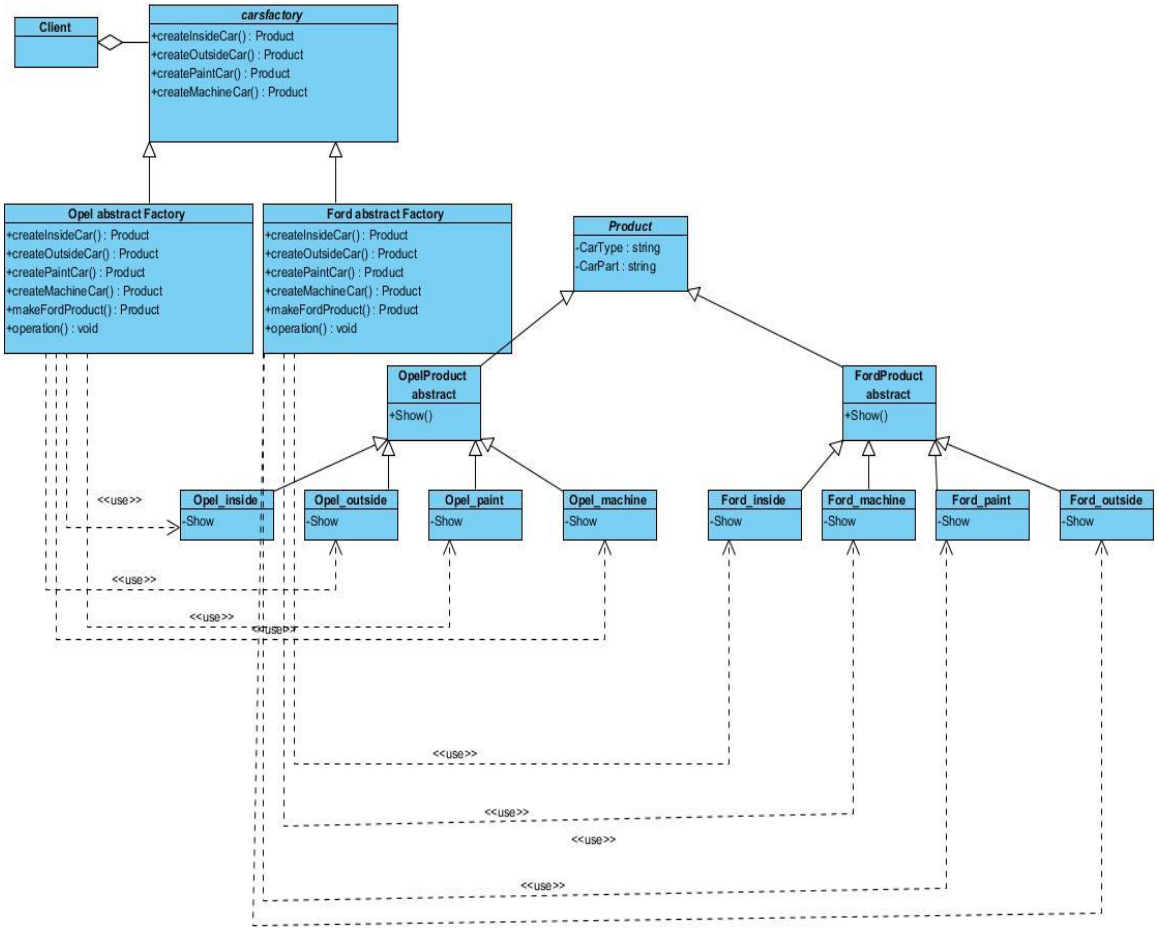
Abstract factory

Το αφηρημένο μοντέλο εργοστάσιο είναι ένα λογισμικό πρότυπο σχεδίασης που παρέχει έναν τρόπο για να δημιουργούμε μια ομάδα μεμονωμένων factory που έχουν ένα κοινό θέμα. Σε κανονικές συνθήκες χρήσης, το λογισμικό πελάτη δημιουργεί μια συγκεκριμένη εφαρμογή με αφηρημένη κλάση του εργοστασίου και στη συνέχεια χρησιμοποιεί τις γενικές διασυνδέσεις για να δημιουργήσει τα συγκεκριμένα αντικείμενα που αποτελούν μέρος του θέματος. Ο πελάτης δεν ξέρει ποια συγκεκριμένα αντικείμενα είναι που λαμβάνει από κάθε μία από αυτές τις εσωτερικές κλάσεις, δεδομένου ότι χρησιμοποιεί μόνο τις γενικές διασυνδέσεις των προϊόντων τους. Αυτό το πρότυπο διαχωρίζει τις λεπτομέρειες της εφαρμογής ενός συνόλου αντικειμένων από τη γενική χρήση τους.Ένα factory είναι η τοποθεσία ή μια συγκεκριμένη κατηγορία στον κώδικα με τον οποίο κατασκευάζονται τα αντικείμενα . Η πρόθεση στην οποία απασχολεί το σχέδιο είναι να απομονώσουμε την δημιουργία αντικειμένων από τη χρήση τους. Χρήση αυτού του προτύπου παρέχει τη δυνατότητα να ανταλλάξετε συγκεκριμένες εφαρμογές χωρίς να αλλάξετε τον κωδικό που τους χρησιμοποιεί, ακόμη και σε χρόνο εκτέλεσης . Ωστόσο, η απασχόληση σε αυτό το πρότυπο, όπως και με παρόμοια πρότυπα σχεδίασης , μπορεί να οδηγήσει σε άσκοπη πολυπλοκότητα και επιπλέον εργασία κατά την αρχική συγγραφή του κώδικα.

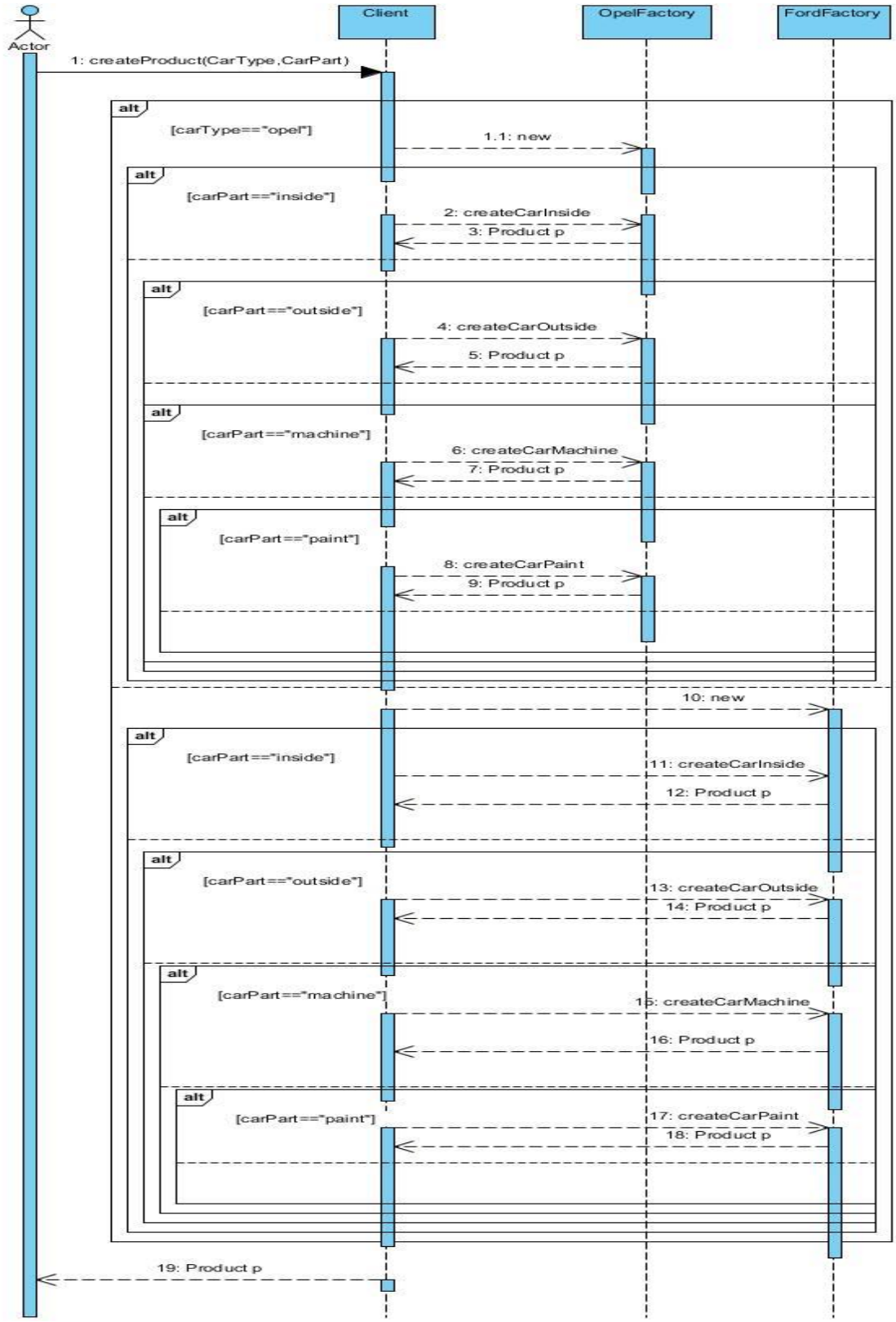
Πρόβλημα:

Με το συγκεκριμένο πρότυπο μπορούμε να δημιουργήσουμε μια εφαρμογή που κατασκευάζει διάφορα τμήματα αυτοκινήτων από δυο εργοστάσια, αδιαφορώντας για τη μάρκα του κάθε αυτοκινήτου. Η συγκεκριμένη εργασία πραγματοποιείται μέσω της αφηρημένης κλάσης εργοστάσιο αυτοκινήτων, ενώ στη συνέχεια δημιουργούνται κλάσεις για κάθε μάρκα αυτοκινήτου, οι μέθοδοι τους οι οποίες υλοποιούνται στις υποκλάσεις της. Για την αναπαράσταση των κάθε προϊόντων (τμήματα αυτοκινήτου) δημιουργούμε μια ακόμα αφηρημένη κλάση στην οποία καταλήγουν οι κλάσεις των διαφορετικών αυτοκινήτων για να μπορέσουμε να καλέσουμε, να αναπαραστίσουμε και να εμφανίσουμε τα αντικείμενα αυτά, τις μεθόδους των αυτοκινήτων, το κάθε μέρος τους.

Class diagram



Sequence diagram



JAVA CODE

```
abstract public class Cars_factory {

abstract OpelProduct createOpelInsideCar();
abstract FordProduct createFordInsideCar();

abstract OpelProduct createOpelOutsideCar();
abstract FordProduct createFordOutsideCar();

abstract OpelProduct createOpelMachineCar();
abstract FordProduct createFordMachineCar();

abstract OpelProduct createOpelPaintCar();
abstract FordProduct createFordPaintCar();

}

public class Ford_Factory extends Cars_factory {

    FordProduct createFordInsideCar(){
                return new Ford_inside("Ford_inside");
        }

    FordProduct createFordOutsideCar(){
                return new Ford_outside("Ford_outsie");
        }

    FordProduct createFordMachineCar(){
                return new Ford_machine("Ford_machine");
        }

    FordProduct createFordPaintCar(){
                return new Ford_paint("Ford_paint");
        }

   @Override
   OpelProduct createOpelInsideCar() {
      throw new UnsupportedOperationException("Not supported yet.");
   }

   @Override
   OpelProduct createOpelOutsideCar() {
      throw new UnsupportedOperationException("Not supported yet.");
   }

   @Override
   OpelProduct createOpelMachineCar() {
      throw new UnsupportedOperationException("Not supported yet.");
   }
```

```
    @Override
    OpelProduct createOpelPaintCar() {
        throw new UnsupportedOperationException("Not supported yet.");
    }

}

public class Ford_machine extends FordProduct {
    Ford_machine(String arg){
        System.out.println("1400 kibika "+arg);
    }
            public void Show() { };
}

public class Ford_paint extends FordProduct {
    Ford_paint(String arg){
        System.out.println("black "+arg);
    }
            public void Show() { };
}

public class Ford_inside extends FordProduct {

    Ford_inside(String arg){
        System.out.println("leather seats "+arg);
    }
            public void Show() { };
}

public class Ford_outside extends FordProduct{
    Ford_outside(String arg){
        System.out.println("sport "+arg);
    }
            public void Show() { };
}

abstract public class FordProduct {

 abstract public void Show();

}

public class Opel_Factory extends Cars_factory {

    OpelProduct createOpelInsideCar(){
                    return new Opel_inside();
            }
            OpelProduct createOpelOutsideCar(){
                    return new Opel_outside("Opel_outside");
```

```
        }

        OpelProduct createOpelMachineCar(){
                return new Opel_machine("Opel_machine");
        }
        OpelProduct createOpelPaintCar(){
                return new Opel_Paint("Opel_paint");
        }

    @Override
    FordProduct createFordInsideCar() {
        throw new UnsupportedOperationException("Not supported yet.");
    }

    @Override
    FordProduct createFordOutsideCar() {
        throw new UnsupportedOperationException("Not supported yet.");
    }

    @Override
    FordProduct createFordMachineCar() {
        throw new UnsupportedOperationException("Not supported yet.");
    }

    @Override
    FordProduct createFordPaintCar() {
        throw new UnsupportedOperationException("Not supported yet.");
    }


}




public class Opel_inside extends OpelProduct {
   Opel_inside(){

    }
    public void Show() { System.out.println("leather seats Opel Inside"); }

}

public class Opel_machine extends OpelProduct {
   Opel_machine(String arg){
        System.out.println("1600 kibika "+arg);
    }
          public void Show() { };

}
```

```
public class Opel_Paint extends OpelProduct {
    Opel_Paint(String arg){
        System.out.println("red "+arg);
    }
          public void Show() { };

}

public class Opel_outside extends OpelProduct {

    Opel_outside(String arg){
        System.out.println("station vagon "+arg);
    }
          public void Show() { };

}

abstract public class OpelProduct {

    public abstract void Show();

}

abstract public class FordProduct {

 abstract public void Show();

}

public class Client {

private static Cars_factory pf=null;
          static Cars_factory getFactory(String string){
                    if(string.equals("a")){
                              pf=new Opel_Factory();
                    }else if(string.equals("b")){
                                        pf=new Ford_Factory();
                              } return pf;
          }

public static void main(String args[]){



                    Cars_factory pfo=Client.getFactory("a");

                    OpelProduct product=pfo.createOpelInsideCar();

                product.Show();
```

```
        OpelProduct product1=pfo.createOpelOutsideCar();

        OpelProduct product2=pfo.createOpelMachineCar();

        OpelProduct product3=pfo.createOpelPaintCar();


    Cars_factory pff=Client.getFactory("b");

        FordProduct productf=pff.createFordInsideCar();

        FordProduct productf1=pff.createFordOutsideCar();

        FordProduct productf2=pff.createFordMachineCar();

        FordProduct productf3=pff.createFordPaintCar();


}
}
```

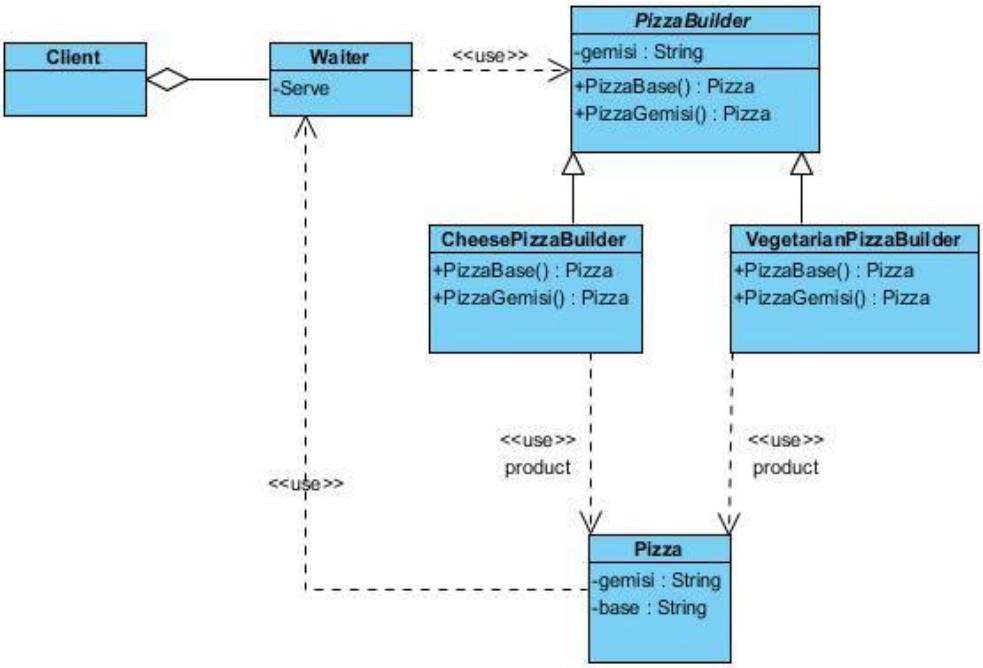
Builder

Το σχέδιο οικοδόμος είναι πρότυπο σχεδιασμού για μια δημιουργία λογισμικού . Η πρόθεση είναι τα αφηρημένα στάδια της κατασκευής των αντικειμένων, έτσι ώστε διαφορετικές εφαρμογές από αυτά τα βήματα μπορούν να κατασκευάσουν διαφορετικές αναπαραστάσεις των αντικειμένων. Συχνά, το πρότυπο οικοδόμος χρησιμοποιείται για την κατασκευή των προϊόντων σύμφωνα με το σύνθετο σχήμα .

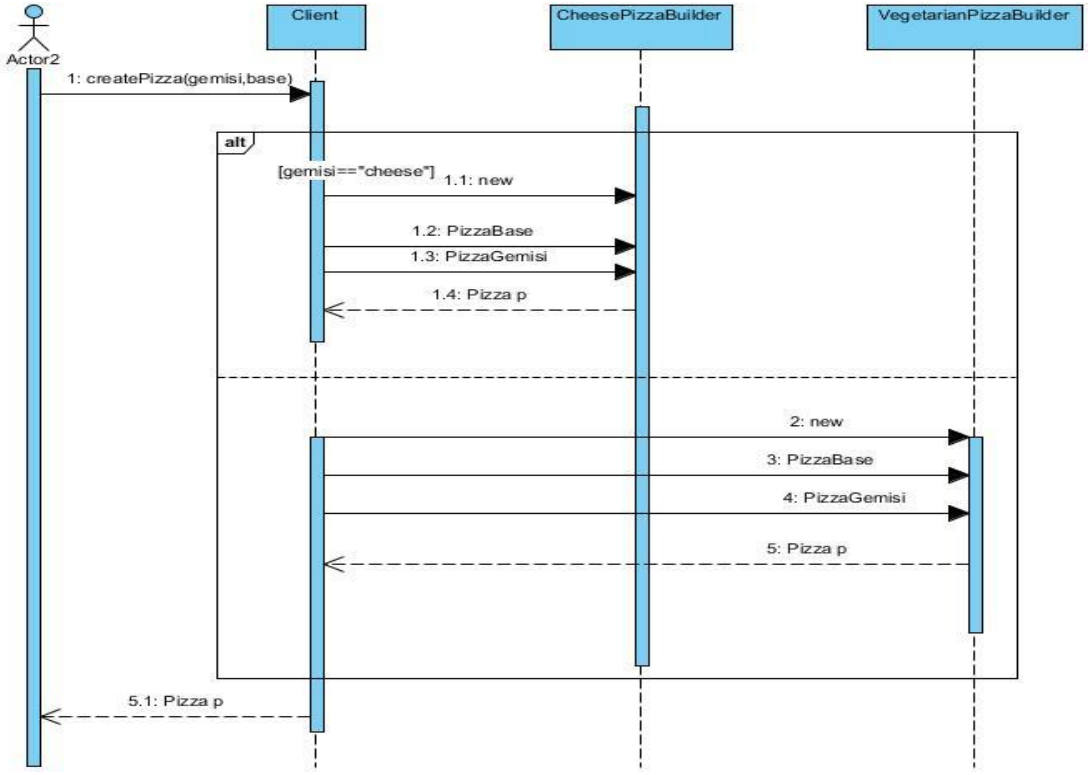
Πρόβλημα:

Με αυτό το πρότυπο χρειάζεται να δημιουργήσουμε αντικείμενο σε διάφορα στάδια. Ένα παράδειγμα από το συγκεκριμένο πρότυπο είναι η λειτουργία ενός καταστήματος fast food πίτσας.Η σερβιτόρα παίρνει την παραγγελία από τον πελάτη και την πηγαίνει την κουζίνα όπου γίνεται η παρασκευή της πίτσας,η παρασκευή είναι μια αφηρημένη κλάση στην οποία υπάρχουν μέθοδοι οι οποίες υλοποιούνται στις υποκλάσεις αναλόγως με τι είδος πίτσας έχει παραγγείλει ο πελάτης.Επιπλέον υπάρχει και μια ακόμα κλάση στην οποία φαίνεται η πίτσα που έχει ολοκληρωθεί.Εκεί βρίσκονται οι έτοιμες πίτσες και η σερβιτόρα τις παίρνει και τις παραδίδει στον πελάτη.

Class diagram



Sequence diagram



JAVA CODE

```
public class Waiter {

    public void servePizza(Client cl){

}
}

abstract public class PizzaBuilder {

    public String base;
    public String gemisi;
    abstract String PizzaBase(String Base);
    abstract String PizzaGemisi(String Gemisi);

}

public class CheesePizzaBuilder extends PizzaBuilder {

    CheesePizzaBuilder(String arg){
        System.out.print("cheese"+arg);
    }
    public String PizzaBase(String base){
       System.out.print("oil");
       return base;
    }
    public String PizzaGemisi(String gemisi){
         System.out.print("cheese");
    return gemisi;
    }




}

public class VegetarianPizzaBuilder extends PizzaBuilder {


    VegetarianPizzaBuilder(String arg){
        System.out.print("vegetarian"+arg);
    }
    public String PizzaBase(String base){
         System.out.print("oil");
      return base;
    }
```

```
    public String PizzaGemisi(String gemisi){
        System.out.print("sogia cheese");
    return gemisi;
    }



}

public class Pizza extends PizzaBuilder{

    public String gemisi;
    public String base;

    @Override
    String PizzaBase (String Base) {
        throw new UnsupportedOperationException("Not supported yet.");
    }

    @Override
    String PizzaGemisi(String Gemisi) {
        throw new UnsupportedOperationException("Not supported yet.");
    }


}

public class Client {


    public static void main(String args[])
    {
        Client cl=new Client();
        Waiter w=new Waiter();

        VegetarianPizzaBuilder vg=new VegetarianPizzaBuilder(" sogia cheese \n");
        CheesePizzaBuilder ch=new CheesePizzaBuilder(" milk cheese \n");

      w.servePizza(cl);
     // System.out.println(ch.PizzaGemisi(null)+"client ordered a ");

       // System.out.println(vg.PizzaBase("")+" is the base");
       // System.out.println(ch.PizzaBase("")+" is the base");

    }
```

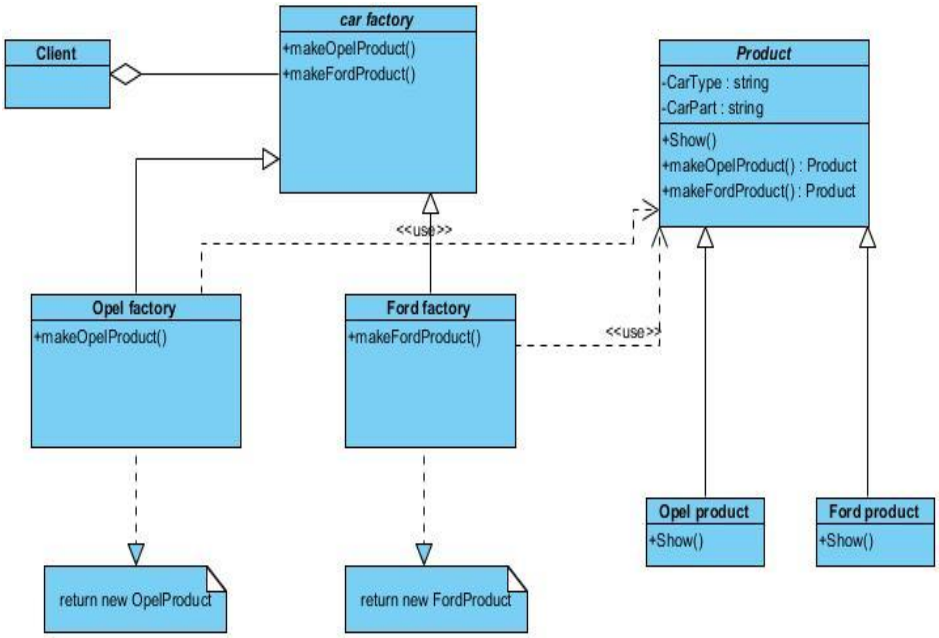
Factory Method

Το πρότυπο μέθοδος εργοστάσιο είναι ένα αντικειμενοστραφής σχεδιαστικό προτύπο για την εφαρμογή της έννοιας των εργοστασίων . Όπως και άλλες δημιουργικές μορφές , ασχολείται με το πρόβλημα της δημιουργίας αντικειμένων (προϊόντων), χωρίς να προσδιορίζει την ακριβή κλάση του αντικειμένου που θα δημιουργηθεί. Η δημιουργία ενός αντικειμένου συχνά απαιτεί πολύπλοκες διαδικασίες και δεν ενδείκνυται να συμπεριληφθούν σε μια σύνθεση αντικείμενο. Η δημιουργία του αντικειμένου μπορεί να οδηγήσει σε σημαντική επικάλυψη του κώδικα, μπορεί να απαιτεί πληροφορίες που δεν έχουν πρόσβαση στην σύνθεση αντικειμένου, δεν μπορεί να παρέχει επαρκές επίπεδο αφαίρεσης.Η μέθοδος αυτή χειρίζεται τα προβλήματα αυτά, καθορίζοντας μια ξεχωριστή μέθοδο για τη δημιουργία των αντικειμένων, τα οποία υποκατηγορίες στη συνέχεια μπορεί να υπερισχύσει για να καθορίσετε το παράγωγο τύπο του προϊόντος που θα δημιουργηθεί. Μερικές από τις διαδικασίες που απαιτούνται για τη δημιουργία ενός αντικειμένου περιλαμβάνουν τον καθορισμό του αντικειμένου που θα δημιουργηθεί, τη διαχείριση της ζωής του αντικειμένου, καθώς και τη διαχείριση των εξειδικευμένων συσσωρευμένων του αντικειμένου. Εκτός του πεδίου εφαρμογής των προτύπων σχεδιασμού, ο όρος *μέθοδος εργοστάσιο* μπορεί επίσης να παραπέμπει σε μια μέθοδο ενός εργοστασίου της οποίας κύριος σκοπός είναι η δημιουργία αντικειμένων.

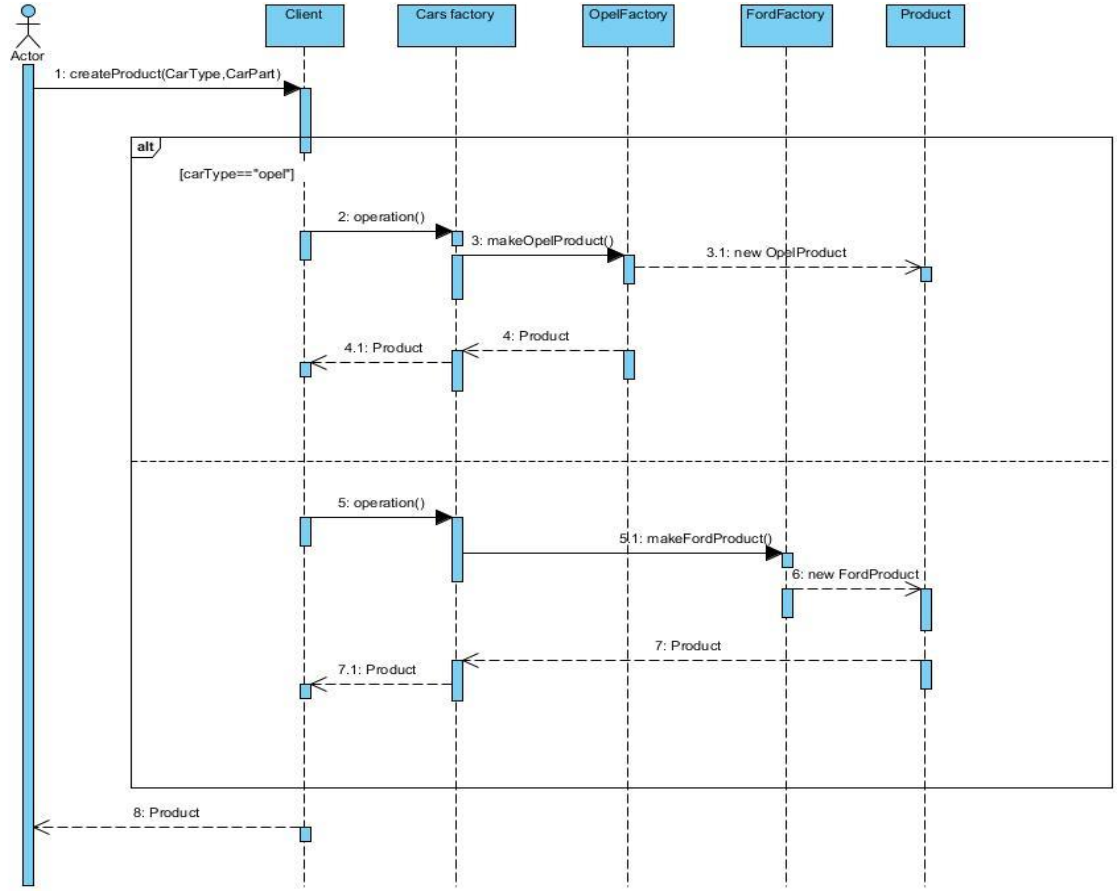
Πρόβλημα:

Με το συγκεκριμένο πρότυπο μπορούμε να δημιουργήσουμε μια εφαρμογή που κατασκευάζει διάφορα τμήματα αυτοκινήτων από δυο εργοστάσια, αδιαφορώντας για τη μάρκα του κάθε αυτοκινήτου. Η συγκεκριμένη εργασία πραγματοποιείται μέσω της αφηρημένης κλάσης εργοστάσιο αυτοκινήτων, ενώ στη συνέχεια δημιουργούνται κλάσεις για κάθε μάρκα αυτοκινήτου, οι μέθοδοι τους οι οποίες υλοποιούνται στις υποκλάσεις της και με τη σειρά τους επιστρέφουν αντικείμενο τύπου προϊόντος στις μεθόδους.Για την αναπαράσταση των κάθε προϊόντων (τμήματα αυτοκινήτου) δημιουργούμε μια ακόμα αφηρημένη κλάση στην οποία καταλήγουν οι κλάσεις των διαφορετικών αυτοκινήτων για να μπορέσουμε να καλέσουμε, να αναπαραστίσουμε και να εμφανίσουμε τα αντικείμενα αυτά, τις μεθόδους των αυτοκινήτων, το κάθε μέρος τους.

Class diagram



Sequence diagram



JAVA CODE

```
public class Product {

}

abstract public class Cars_factory {

    abstract FordProduct makeFordProduct(String msg);
    abstract OpelProduct makeOpelProduct(String msg);

}

public class FordFactory extends Cars_factory {

     FordProduct makeFordProduct(String msg)
  {
      return new FordProduct("Ford product "+msg);
  }

    @Override
    OpelProduct makeOpelProduct(String msg) {
        throw new UnsupportedOperationException("Not supported yet.");
    }


}

public class OpelFactory extends Cars_factory{

   OpelProduct makeOpelProduct(String msg)
  {
      return new OpelProduct("Opel product "+msg);
  }

    @Override
    FordProduct makeFordProduct(String msg) {
        throw new UnsupportedOperationException("Not supported yet.");
    }




}

public class FordProduct {

   public FordProduct(String msg)
   {
```

```
      System.out.println(msg);
   }

    public void Show(String arg)
        {
           System.out.println(arg);
        }
}

public class OpelProduct {

     public OpelProduct(String msg)
     {
        System.out.println(msg);
     }



          public void Show(String arg)
        {
           System.out.println(arg);
        }
}

public class Client {

      private static Cars_factory pf=null;
          static Cars_factory getFactory(String string){
                  if(string.equals("a")){
                           pf=new OpelFactory();
                  }else if(string.equals("b")){
                                   pf=new FordFactory();
                           } return pf;
          }



    public static void main(String args[])
       {
                  Cars_factory pfo=Client.getFactory("a");

                      OpelProduct product=pfo.makeOpelProduct("astra");
                 product.Show("opel astra");

                 Cars_factory pff=Client.getFactory("b");

                      FordProduct productf=pff.makeFordProduct("focus");
                  product.Show("ford focus");
```

```
}

}
```

Prototype

Το πρότυπο είναι ένα πρωτότυπο δημιουργικό σχέδιο που χρησιμοποιείται στην ανάπτυξη λογισμικού , όταν ο τύπος του αντικείμενου που θα δημιουργήσετε καθορίζεται από ένα πρωτότυπο παράδειγμα , το οποίο κλωνοποιήθηκε για την παραγωγή νέων αντικειμένων. Αυτό το πρότυπο χρησιμοποιείται για:

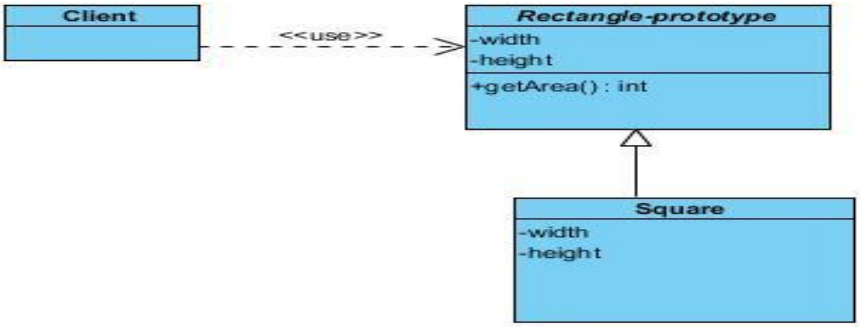
- αποφευχθούν υποκατηγορίες της δημιουργού αντικείμενο στην εφαρμογή-πελάτη.
- αποφύγετε το κόστος της δημιουργίας ενός νέου αντικειμένου στο τυποποιημένο τρόπο (π.χ., χρησιμοποιώντας το « νέο », λέξη-κλειδί), όταν είναι απαγορευτικά ακριβό για μια συγκεκριμένη εφαρμογή.

Για την εφαρμογή του προτύπου, δηλώνει μια αφηρημένη βασική κλάση που καθορίζει μια καθαρά εικονικό *κλώνο* μέθοδο. Κάθε τάξη που χρειάζεται ένα " πολυμορφικό κατασκευαστή " δίνει τη δυνατότητα η ίδια που προέρχεται από την αφηρημένη κλάση βάσης, και υλοποιεί την *κλώνος ()* λειτουργία.Ο πελάτης, αντί να γράψει κώδικα που επικαλείται η "νέα" επιχείρηση σε ένα κωδικοποιημένο όνομα της κλάσης, καλεί τη μέθοδο *κλώνος ()* για το πρωτότυπο, καλεί μια μέθοδο εργοστάσιο με μια παράμετρο που ορίζει το συγκεκριμένο concrete που προέρχεται από κατηγορία που θέλετε, ή να επικαλείται η *κλώνος ()* μέθοδος μέσω κάποιου μηχανισμού που προβλέπεται από άλλο πρότυπο σχεδιασμού.

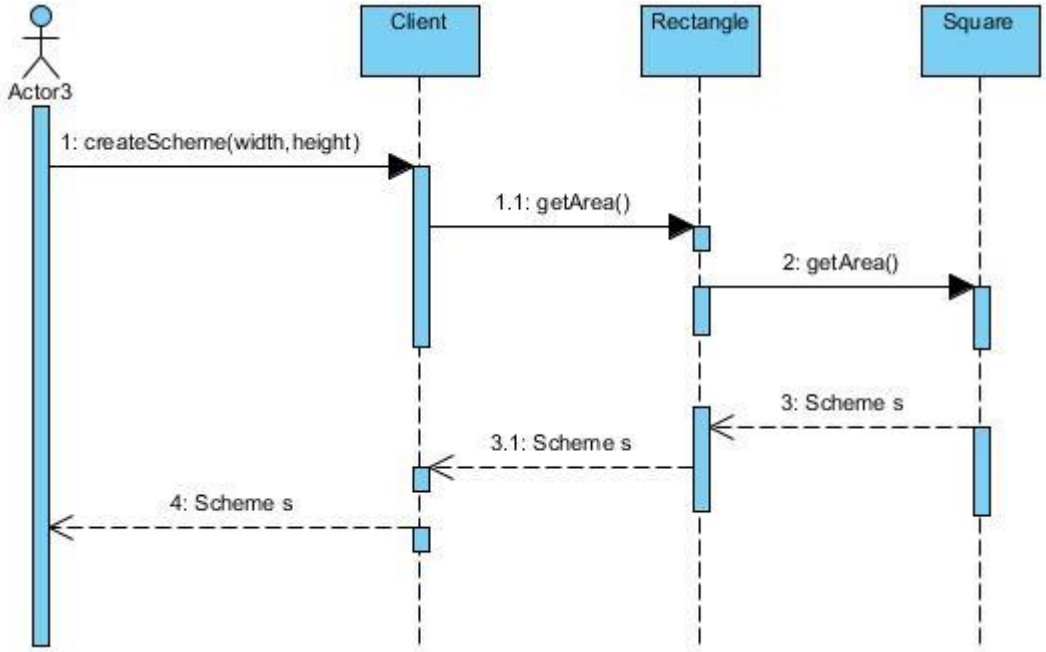
Πρόβλημα:

Με αυτό το πρότυπο υπάρχει η δυνατότητα να κλωνοποιήσουμε κάποιο αντικείμενο. Ένα παράδειγμα βρίσκεται σε έναν υπολογιστή όπου υπάρχει ένας χρήστης που θέλει να αποκτήσει πρόσβαση σε μια βάση δεδομένων μέσα σε έναν η/υ.Για να το πετύχει αυτό χρειάζεται να ξέρει ποια είναι η βάση που θα μπει, να έχει ένα username και password. Τα αντικείμενα αυτά θα είναι πανομοιότυπα σε όλες τις κλάσεις. Κάθε φορά που θα αλλάζει ο χρήστης θα αλλάζουν και οι τιμές των αντικειμένων.

Class diagram



Sequence diagram



JAVA CODE

```
abstract class RectanglePrototype {

   protected int m_width;
   protected int m_height;

        public void setWidth(int width){
                m_width = width;
        }

        public void setHeight(int height){
```

```
            m_height = height;
        }


        public int getWidth(){
            return m_width;
        }

        public int getHeight(){
            return m_height;
        }

        public int getArea(){
            return m_width * m_height;
        }
}




public class Square extends RectanglePrototype{


    public void setWidth(int width){
            m_width = width;
            m_height = width;
        }

        public void setHeight(int height){
            m_width = height;
            m_height = height;
        }


}

public class Client {

    private static RectanglePrototype getRectanglePrototype()
        {
            // it can be an object returned by some factory ...
            return new RectanglePrototype() {};
        }

        public static void main (String args[])
        {
            RectanglePrototype r = Client.getRectanglePrototype();

            r.setWidth(5);
```

```
            r.setHeight(10);
            // user knows that r it's a rectangle.
            // It assumes that he's able to set the width and height as for the
base class

            System.out.println(r.getArea());
            // now he's surprised to see that the area is 100 instead of 50.
        }


}
```

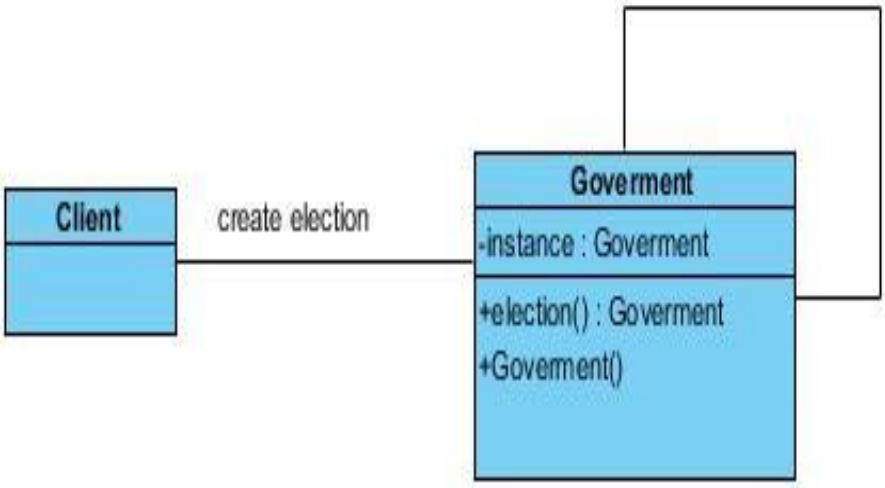
Singleton

Στην τεχνολογία λογισμικού , το πρότυπο είναι μονήρεις πρότυπο σχεδίασης που χρησιμοποιείται για την εφαρμογή της μαθηματικής έννοιας του μεμονωμένου , περιορίζοντας τη συγκεκριμενοποίηση της κατηγορίας σε ένα αντικείμενο . Αυτό είναι χρήσιμο όταν ένα αντικείμενο χρειάζεται ακριβώς για να συντονίσουν τις δράσεις σε όλο το σύστημα. Η ιδέα είναι μερικές φορές γενικευμένη στα συστήματα που λειτουργούν πιο αποτελεσματικά, όταν μόνο ένα αντικείμενο υπάρχει, ή που περιορίζουν την συγκεκριμενοποίηση σε έναν ορισμένο αριθμό αντικειμένων. Υπάρχει κριτική από τη χρήση του προτύπου μονήρεις, όπως μερικοί το θεωρούν ένα αντι-σχέδιο , κρίνοντας ότι πρόκειται για υπερκατανάλωση, εισάγει περιττούς περιορισμούς σε περιπτώσεις όπου ένα και μόνο παράδειγμα μιας κατηγορίας δεν είναι στην πραγματικότητα απαιτείται, και εισάγει παγκόσμια κατάσταση σε μια εφαρμογή . Στην C + + χρησιμοποιείται επίσης για την απομόνωση από το απρόβλεπτο της τάξης του δυναμική εκκίνηση, επιστρέφει τον έλεγχο στον προγραμματιστή.

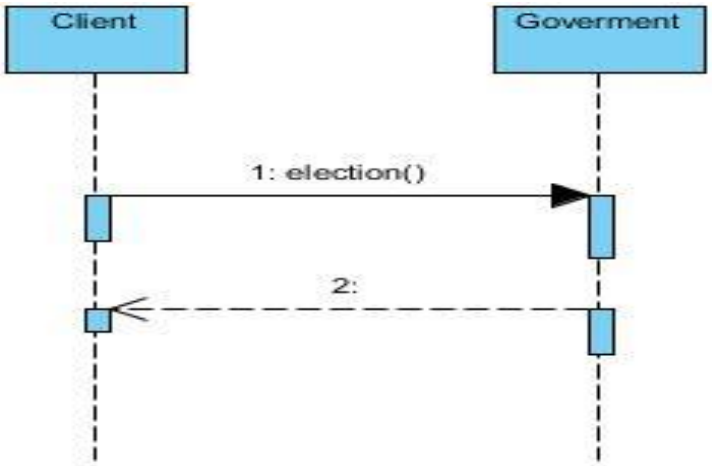
Πρόβλημα

Το πρότυπο αυτό μπορεί να χρησιμοποιηθεί για εφαρμογές που οι τιμές των μεθόδων μπορούν να υπάρξουν μόνο μια φορά. Για παράδειγμα το αξίωμα του Προέδρου των Ηνωμένων Πολιτειών είναι ένα Singleton. Το σύνταγμα των Ηνωμένων Πολιτειών καθορίζει τα μέσα με τα οποία ένας πρόεδρος εκλέγεται, περιορίζει τη θητεία του, και καθορίζει τη σειρά της διαδοχής. Άρα δεν μπορεί να υπάρξει πάνω από ένας πρόεδρος σε κάθε δεδομένη στιγμή.
Οι κλάσεις θα είναι μόνο μια με όνομα Government στην οποία θα γίνεται η εκλογή και η ανακοίνωση του προέδρου κάθε φορά.

Class diagram



Sequence diagram



JAVA CODE

```
public class Goverment {

  private static Goverment instance=new Goverment();

        /** A private Constructor prevents any other class from instantiating. */
        private Goverment(){}

        public static synchronized Goverment election()
        {
           if (instance == null){
                instance = new Goverment();

           }
           return instance;
        }


}




public class Client {

public static void main(String args[]){
//              Singleton obj = new Singleton();

                //create the Singleton Object..
                Goverment obj = Goverment.election();

                // Your Business Logic
                System.out.println("Barak Obama is the new President");

        }
}
```

ΚΕΦΑΛΑΙΟ 2 - Διαρθρωτικά πρότυπα σχεδίασης (Structural design patterns)

Τα δομικά πρότυπα ασχολούνται με το πώς κλάσεις και αντικείμενα θα σχηματίσουν μεγαλύτερες δομές. Τα πρότυπα αυτής της κατηγορίας χρησιμοποιούν την κληρονομικότητα για να συνθέσουν διεπαφές ή εφαρμογές. Ως ένα απλό παράδειγμα, αναφέρεται ο τρόπος με τον οποίο η πολλαπλή κληρονομικότητα αναμιγνύει δύο ή περισσότερες κλάσεις σε μία, με αποτέλεσμα μια κλάση που να συνδυάζει τις ιδιότητες των «γονικών» της κλάσεων. Αυτά τα πρότυπα είναι ιδιαίτερα χρήσιμα για την δημιουργία βιβλιοθηκών ανεξάρτητων κλάσεων οι οποίες να μπορούν να συνεργαστούν. Τα πρότυπα αυτής της κατηγορίας είναι τα εξής: Adapter (ή Wrapper),Bridge, Composite, Decorator, Facade, Flyweight και Proxy.

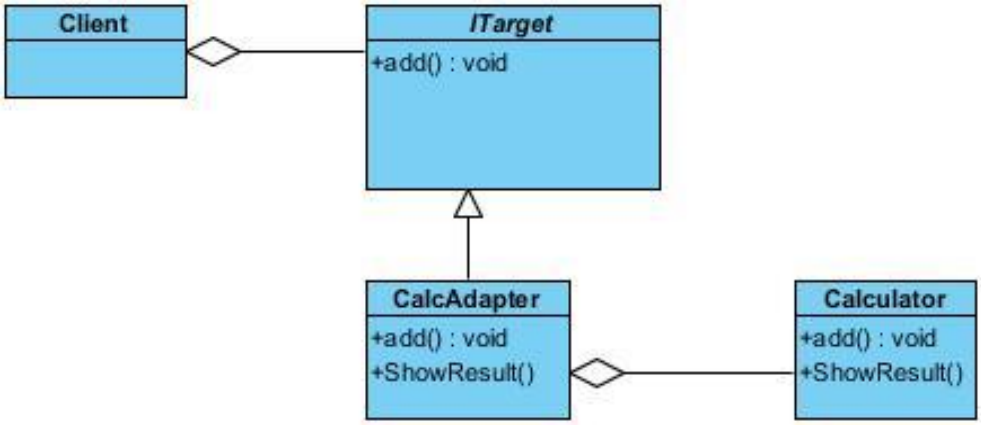
Adapter

Σε προγραμματισμό ηλεκτρονικών υπολογιστών , το πρότυπο προσαρμογέα (συχνά αναφέρεται ως το πρότυπο περιτύλιγμα ή απλά ένα περιτύλιγμα) είναι ένα πρότυπο σχέδιο που μετατρέπει ένα περιβάλλον για μια κατηγορία σε μια συμβατή διεπαφή. Ένας *προσαρμογέας* επιτρέπει τάξεις να συνεργάζονται από κοινού όταν κανονικά δεν θα μπορούσε, λόγω των ασυμβίβαστων διασυνδέσεων, με την παροχή διασύνδεσης προς τους πελάτες της, ενώ με την αρχική διεπαφή. Ο προσαρμογέας μετατρέπει τις κλήσεις της διεπαφής χρήστη σε κλήσεις στο αρχικό περιβάλλον, και το ποσό του κώδικα που είναι απαραίτητος για να γίνει αυτό είναι συνήθως μικρό. Ο προσαρμογέας είναι επίσης υπεύθυνος για τη μετατροπή των δεδομένων σε κατάλληλη μορφή. Για παράδειγμα, εάν οι πολλαπλές boolean τιμές αποθηκεύονται ως ένα ενιαίο ακέραιο (π.χ. σημαίες), αλλά ο πελάτης σας απαιτεί μια «αληθινή» / «ψευδείς», ο προσαρμογέας θα είναι υπεύθυνος για την εξαγωγή των κατάλληλων τιμών από την ακέραια τιμή. Ένα άλλο παράδειγμα είναι η μετατροπή της μορφής των ημερομηνιών (π.χ. ΕΕΕΕΜΜΗΗ να ΜΜ / ΗΗ / ΕΕΕΕ ή ΗΗ / ΜΜ / ΕΕΕΕ).

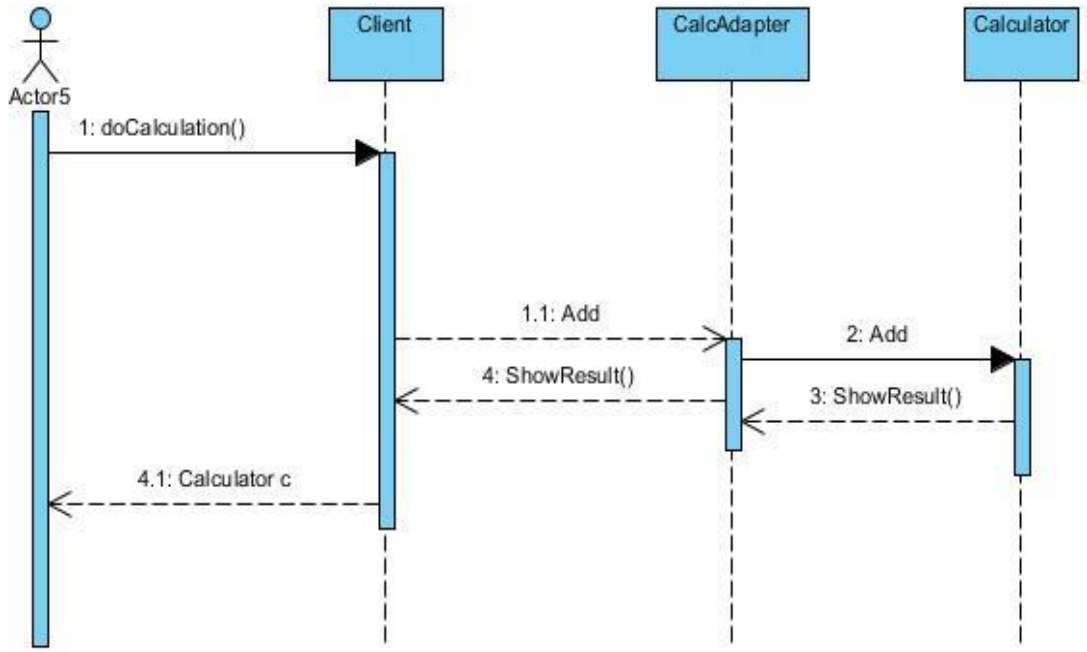
Πρόβλημα:

Το πρότυπο Adapter ορίζει τάξεις που μπορούν να εργαστούν από κοινού με τη μετατροπή της διεπαφής μιας τάξης σε μια άλλη τάξη. Ένα παράδειγμα για το πρότυπο είναι η λειτουργία της αριθμομηχανής.Έχουμε χρησιμοποιήσει κάποια βιβλιοθήκη, όπου έχουμε λειτουργία η οποία λαμβάνει δύο ακέραιους και παρέχει το άθροισμα τους. Τώρα για την αναβάθμιση της βιβλιοθήκης η βιβλιοθήκη θα αλλάξει τη λειτουργία Add.Μια επιλογή θα μπορούσε να είναι, να αλλάξουμε όλον τον κώδικα πελάτη όπου έχουμε χρησιμοποιήσει τη μέθοδο Add ή άλλη επιλογή είναι να έχουμε έναν προσαρμογέα(adapter).

Class diagram



Sequence diagram



JAVA CODE

```
abstract public class ITarget {

 abstract public void add(int integer);

}

public class Calculator {

     private ITarget target;

  public Calculator() {  }

    public Calculator(ITarget target)
    {
        this.target = target;
    }


    public void add(int integer) {
        System.out.println("add adapter "+ integer);
    }

    public void ShowResult()
    {
        System.out.print("adapter is "+target);
    }


}

public class CalcAdapter extends ITarget {

    private ITarget target;

    public CalcAdapter(Calculator c) {}

    public CalcAdapter(ITarget target)
    {
        this.target = target;
    }

    public void add(int integer) {
        System.out.println("add adapter "+ integer);

    }

    public void ShowResult(ITarget target)
```

```
    {
        System.out.print("adapter is "+ target);
    }

}

public class Client {


    public static void main(String args[]) {

        Calculator c=new Calculator();
        ITarget t=new ITarget() {
            public void add(int integer) {}
        };
          c.add(20);
        CalcAdapter adapter = new CalcAdapter(c);
         adapter.add(10);
         adapter.ShowResult(t);


    }
}
```

Bridge

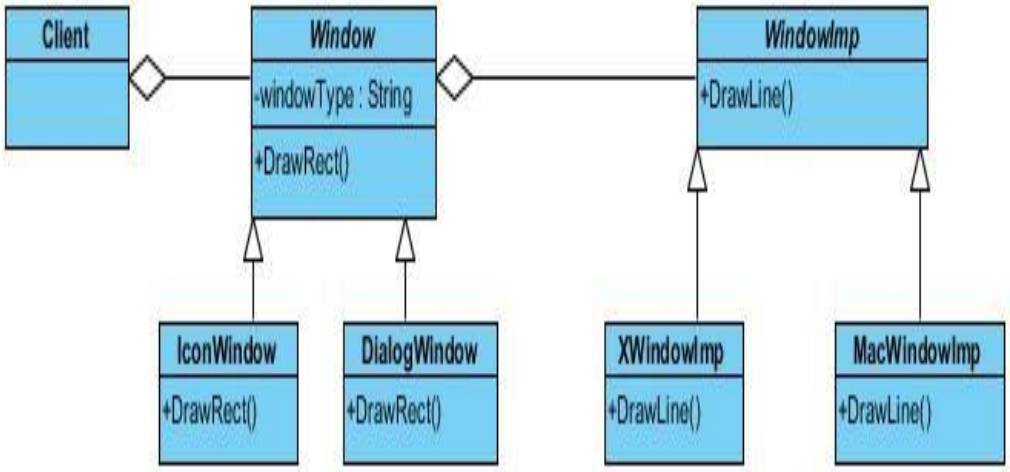
Το σχέδιο γέφυρα είναι ένα πρότυπο σχεδίασης που χρησιμοποιείται στην τεχνολογία λογισμικού που προορίζεται να *«αποσυνδέσει την αφαίρεση της από την εφαρμογή έτσι ώστε οι δύο μπορεί να ποικίλουν ανεξάρτητα ».* Η *γέφυρα* χρησιμοποιεί ενθυλάκωση , συγκέντρωση , και μπορεί να χρησιμοποιήσει κληρονομικότητα για το διαχωρισμό των αρμοδιοτήτων σε διαφορετικές τάξεις . Όταν μια κατηγορία ποικίλλει συχνά, τα χαρακτηριστικά του αντικειμενοστραφούς προγραμματισμού γίνονται πολύ χρήσιμα, διότι οι αλλαγές σε ένα πρόγραμμα του κώδικα μπορεί να γίνουν εύκολα με την ελάχιστη προηγούμενη γνώση σχετικά με το πρόγραμμα. Το σχέδιο γέφυρα είναι τόσο χρήσιμο όταν η τάξη δεν διαφέρει συχνά. Η κατηγορία από μόνη της μπορεί να θεωρηθεί ως *εφαρμογή* και ποια είναι η τάξη που μπορεί να κάνει την *αφαίρεση.* Το πρότυπο γέφυρα μπορεί επίσης να θεωρηθεί ως δύο στρώματα της αφαίρεσης. Το πρότυπο γέφυρα συχνά συγχέεται με το πρότυπο προσαρμογέα . Στην πραγματικότητα, το σχέδιο γέφυρα εφαρμόζεται συχνά με το πρότυπο προσαρμογέα τάξη Πρόβλημα:

Πρόβλημα:

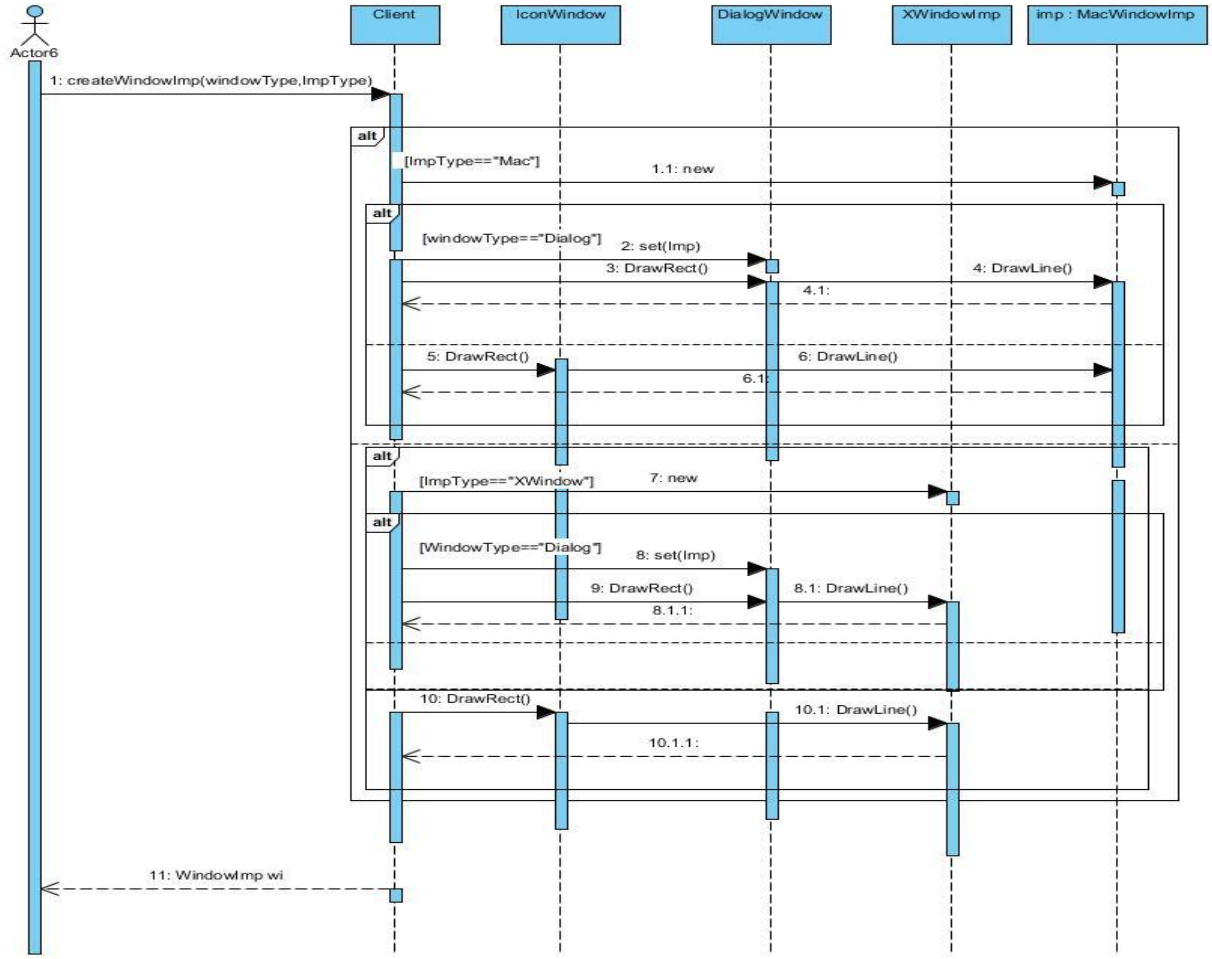
Για το πρότυπο bridge έχουμε παράδειγμα τη δημιουργία μιας βιβλιοθήκης από το μηδέν δεν είναι ποτέ μια καλή ιδέα και γι 'αυτό μπορεί να χρειαστεί να χρησιμοποιήσετε κάποιο από τα υπάρχουσες υποδομές ή τις διαθέσιμες βιβλιοθήκες. Μπορούμε να χρησιμοποιήσουμε την XWindow εργαλείων σαν

κλάση ή MacWindow εργαλείων ως βάση σαν κλάση επίσης ανάλογα με την πλατφόρμα χρήστη και την εργαλειοθήκη που είναι διαθέσιμη. Η drawRect χρησιμοποιεί πράγματι τη DrawLine λειτουργία, η οποία στην πραγματικότητα εξαρτάται από το είδος της εφαρμογής και θα διαχωρίζει την εφαρμογή.

Class diagram



Sequence diagram



JAVA CODE

```
abstract public class WindowImp {

  abstract  public String DrawLine(String str);


}


abstract public class Window {

  abstract  public Object DrawRect(String string);


}

public class IconWindow extends Window {

    private WindowImp implementor = null;

          public IconWindow(WindowImp imp) {

                    this.implementor = imp;

          }

          public String DrawRect(String str) {
            System.out.println("drawrect is "+str);
                    return implementor.DrawLine(str);

          }


}

public class DialogWindow extends Window {

    private WindowImp implementor = null;

          public DialogWindow(WindowImp imp) {

                    this.implementor = imp;

          }

          public String DrawRect(String str) {
             System.out.println("drawrect is "+str);
```

```
                return implementor.DrawLine(str);
        }
}
public class XWindowImp extends WindowImp {
    private Window implementor = null;
  public XWindowImp() {}
        public XWindowImp(Window imp) {
                this.implementor = imp;
        }

    public String DrawLine(String str) {
        return (String) implementor.DrawRect(str);
    }
}
public class MacWindowImp extends WindowImp {
    private Window implementor = null;
  public MacWindowImp() {}
        public MacWindowImp(Window imp) {
                this.implementor = imp;
        }

    public String DrawLine(String str) {
       return (String) implementor.DrawRect(str);
    }
}
public class Client {
```

```
    public static void main(String[] args) {
            WindowImp implementor = null;

            if(window()){
                implementor = new MacWindowImp();
            }else{
                implementor = new XWindowImp();
            }
            Window w = new IconWindow(implementor);
            Object o = w.DrawRect("12343755");

            w = new DialogWindow(new XWindowImp());
            w.DrawRect("2323");
        }
        private static boolean window() {
            return false;
        }
}
```

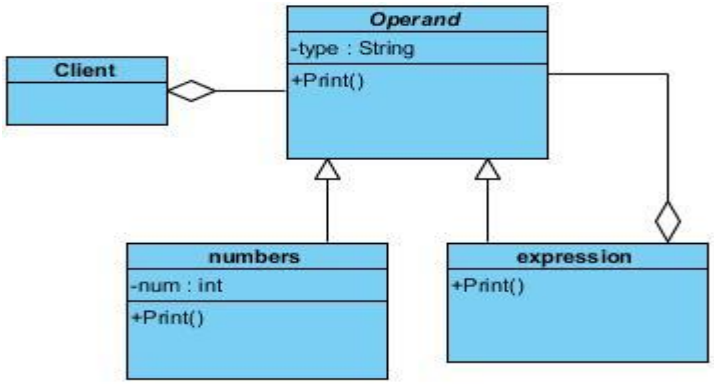
Composite

Στην τεχνολογία λογισμικού , το σύνθετο πρότυπο είναι μια διαμέριση πρότυπο σχεδιασμού . Το σύνθετο σχέδιο περιγράφει ότι μια ομάδα αντικειμένων, πρέπει να αντιμετωπίζεται με τον ίδιο τρόπο όπως ένα μόνο παράδειγμα ενός αντικειμένου. Η πρόθεση ενός σύνθετου είναι να "συνθέσει" αντικείμενα στις δομές δέντρο να εκπροσωπεί ολόκληρο το μέρος-ιεραρχίες. Εφαρμογή του σύνθετου πρότυπου επιτρέπει στους πελάτες να αντιμετωπίζουν μεμονωμένα αντικείμενα και συνθέσεις ομοιόμορφα.
Συνθέσεις μπορεί να χρησιμοποιηθεί όταν οι πελάτες θα πρέπει να αγνοήσουν τη διαφορά μεταξύ συνθέσεις των αντικειμένων και των μεμονωμένων αντικειμένων.Αν διαπιστώσετε ότι οι προγραμματιστές που χρησιμοποιούν περισσότερα από ένα αντικείμενα με τον ίδιο τρόπο, και συχνά έχουν σχεδόν ταυτόσημο κώδικα για να χειριστεί το καθένα από αυτά, τότε είναι σύνθετα μια καλή επιλογή, είναι λιγότερο πολύπλοκο σε αυτή την κατάσταση για την αντιμετώπιση της αρχέτυπα και συνθέτων, ως ομοιογενής.

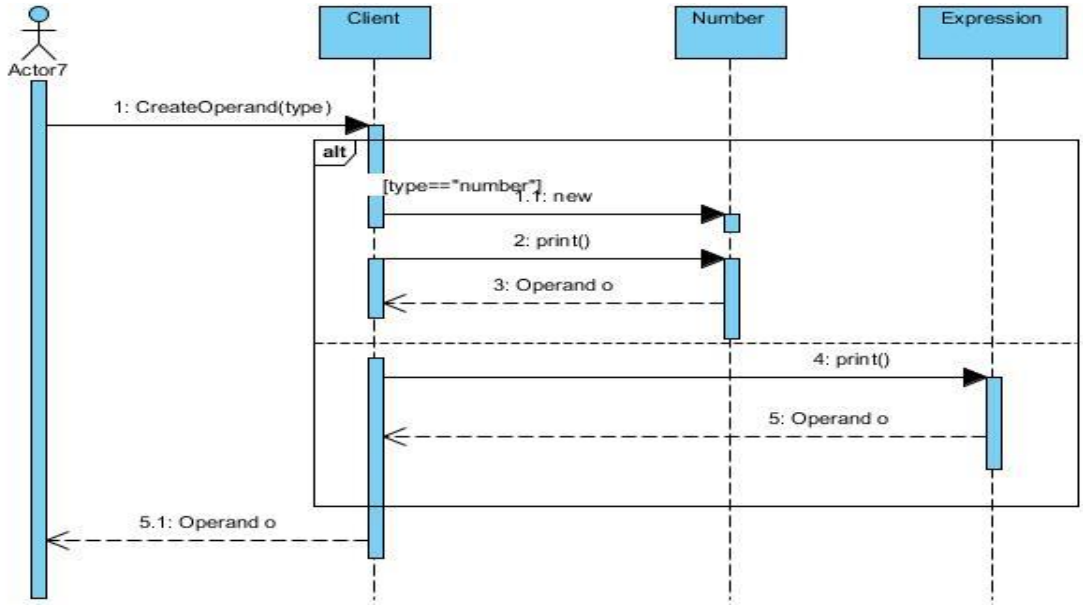
Πρόβλημα:

Το Composite συνθέτει αντικείμενα σε δομές δέντρων και δίνει τη δυνατότητα στους πελάτες να αντιμετωπίζουν μεμονωμένα αντικείμενα και συνθέσεις ομοιόμορφα. Αν και το παράδειγμα είναι αφηρημένο, αριθμητικές εκφράσεις είναι σύνθετα. Μια έκφραση αριθμητική αποτελείται από έναν τελεστή, (+ - * /), και ένα άλλο τελεστή. Ο τελεστής μπορεί να είναι ένας αριθμός, ή ένα άλλο expresssion. Έτσι, 2 + 3 και (2 + 3) + (4 * 6) είναι και οι δύο έγκυρες εκφράσεις.Άρα αν έχουμε μια κλάση numbers που θα έχει τους αριθμούς και μια κλάση expression που θα έχει ολόκληρες εκφράσεις τότε θα μπορούμε να κάνουμε ανάλογες πράξεις.

Class diagram



Sequence diagram



JAVA CODE

```
abstract public class Operand {

    protected String type;

    public Operand(String s) {type=s;}
    public abstract void print();

}

public class numbers extends Operand {

    private int a;
    private int b;

    public numbers(String s, int a0, int b0) {
        super(s);
        a=a0;
        b=b0;
    }

    public void print() {
        System.out.print(a + " " + type+ " " + b);
    }

}


import java.util.ArrayList;

public class Expression extends Operand {

    private ArrayList<Operand> opList = new ArrayList<Operand>();

    public Expression(String arg){
        super(arg);
    }

    public void addExpression(Operand a) {
        opList.add(a);
    }

    public void print(){
        for (int i=0;i<opList.size();i++) {
            opList.get(i).print();
            if (i!=opList.size()-1) System.out.print(" " + type + " ");
        }
    }
}
```

```
public class Client {

   public static void main(String args[])
   {

      Operand o = new numbers("+", 2, 8);
      o.print();
      System.out.println(" ");
      Expression o1 = new Expression("+");
      Operand o2 = new numbers("+", 4, 11);
      Operand o3 = new numbers("+", 1, 6);

      o1.addExpression(o2);
      o1.addExpression(o3);
      o1.print();
      System.out.println(" ");

   }

}
```

Decorator

Το πρότυπο διακοσμητής μπορεί να χρησιμοποιηθεί για την επέκταση (διακοσμήσετε) της λειτουργικότητας του συγκεκριμένου αντικειμένου στο χρόνο εκτέλεσης , ανεξάρτητα από άλλες περιπτώσεις της ίδιας κατηγορίας , εφόσον κάποια προεργασία γίνεται κατά το χρόνο σχεδίασης. Αυτό επιτυγχάνεται με το σχεδιασμό μιας νέας κατηγορίας *διακοσμητή* που τυλίγει την αρχική κατηγορία. Η συσκευασία θα μπορούσε να επιτευχθεί με την ακόλουθη σειρά βημάτων:

1. Υποκατηγορία η αρχική "διακοσμητής" τάξη σε μια "συνιστώσα" τάξη
2. Στην κατηγορία Διακοσμητής, προσθέστε ένα δείκτη στοιχείου ως ένα πεδίο
3. Περάστε μια συνιστώσα για τον κατασκευαστή διακοσμητή να προετοιμαστεί το δείκτη του στοιχείου
4. Στην κατηγορία Διακοσμητής, ανακατευθύνει όλα τα "συνιστωσών" μεθόδους για να το "Component" δείκτη Και
5. Στην κατηγορία ConcreteDecorator, υπερισχύει κάθε μέθοδο Στοιχείο η συμπεριφορά του οποίου πρέπει να τροποποιηθεί.

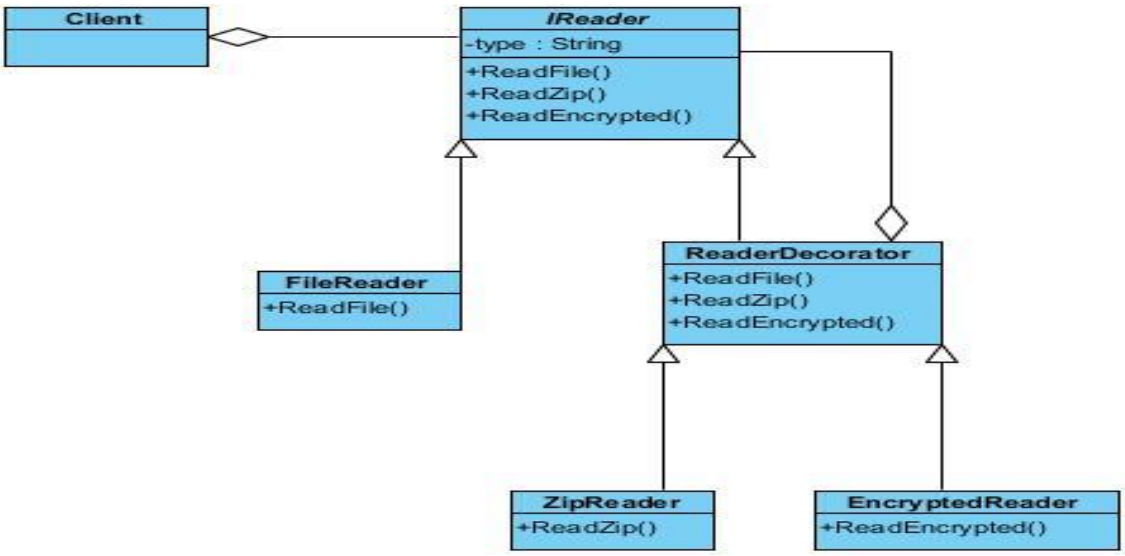
Αυτό το πρότυπο έχει σχεδιαστεί έτσι ώστε πολλαπλοί διακοσμητές μπορούν να στοιβάζονται ο ένας πάνω στον άλλο, προσθέτοντας κάθε φορά μια νέα λειτουργικότητα στη μέθοδο (-ους). Το σχέδιο διακοσμητής είναι μια εναλλακτική λύση Το subclassing . Το subclassing προσθέτει συμπεριφορά κατά τη μεταγλώττιση , και η αλλαγή επηρεάζει όλες τις εμφανίσεις της αρχικής κατηγορίας, η διακόσμηση μπορεί να δώσει νέα συμπεριφορά κατά το χρόνο

εκτέλεσης για μεμονωμένα αντικείμενα. Αυτή η διαφορά γίνεται πιο σημαντική όταν υπάρχουν πολλοί *ανεξάρτητοι* τρόποι επέκτασης της λειτουργικότητας. Σε κάποιες αντικειμενοστρεφείς γλώσσες προγραμματισμού , οι τάξεις δεν μπορούν να δημιουργηθούν κατά το χρόνο εκτέλεσης, και δεν είναι συνήθως δυνατό να προβλεφθούν, κατά το χρόνο σχεδίασης, τι συνδυασμούς των επεκτάσεων θα χρειαστούν. Αυτό θα σήμαινε ότι μια νέα τάξη θα πρέπει να γίνει για κάθε πιθανό συνδυασμό. Αντίθετα, διακοσμητές είναι αντικείμενα, που δημιουργήθηκαν κατά το χρόνο εκτέλεσης, και μπορούν να συνδυαστούν σε ένα ανά χρήση βάση.

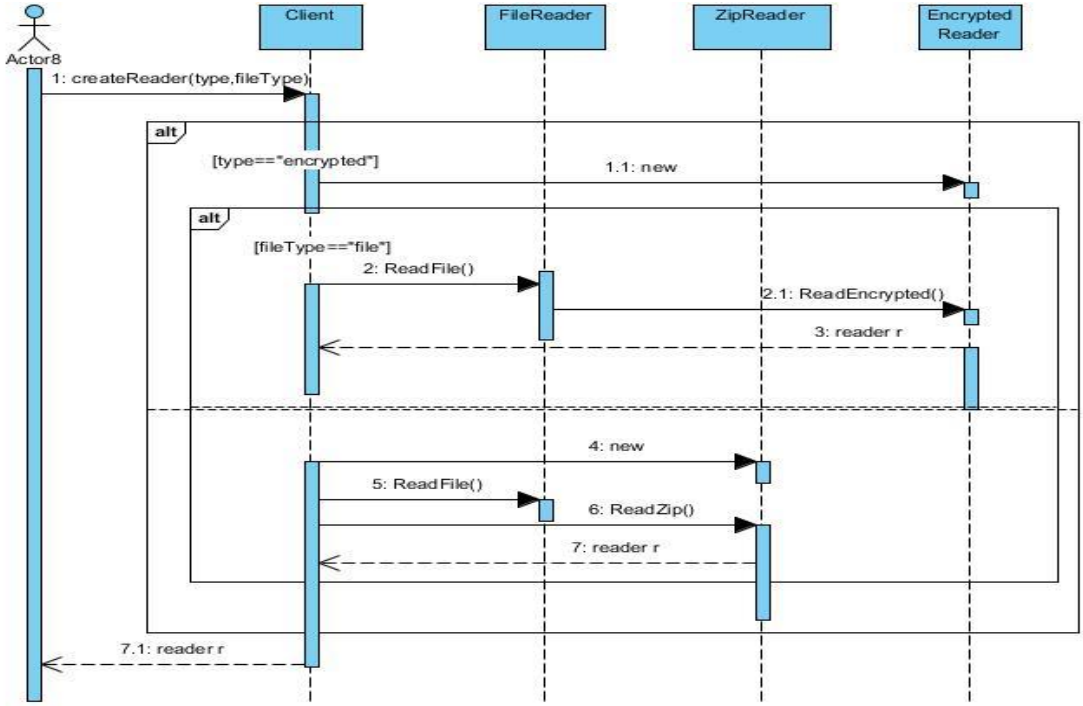
Πρόβλημα:

Ο Διακοσμητής(Decorator) αποδίδει πρόσθετες ευθύνες σε ένα αντικείμενο δυναμικά. Έστω ότι έχουμε ένα FileReader κατηγορία όπου βρίσκεται το αρχείο που μπορεί να διαβαστεί με βάση το συνδυασμό της εφαρμογής οποιουδήποτε από τους τύπους, όπως θα μπορούσε να είναι είτε κρυπτογραφημένη είτε συμπιεσμένα και κρυπτογραφημένα.Κρυπτογραφημένα και συμπιεσμένα τότε ecrypted πάλι.

Class diagram



Sequence diagram



JAVA CODE

```
package decorator;
abstract public class IReader {
IReader reader;
   public IReader(){};

   String type;
   abstract public void ReadFile();
   abstract public void ReadZip();
   abstract public void ReadEncrypted();
}
package decorator;

public class FileReader1 extends IReader {

   IReader reader;

   public FileReader1(IReader reader){
     super();
      this.reader=reader;
   }
   public FileReader1() {
      // throw new UnsupportedOperationException("Not yet implemented");
   }
   public void ReadFile() {
     System.out.print("it's a simple file ");
   }
```

```
    @Override
   public void ReadZip() {
        throw new UnsupportedOperationException("Not supported yet.");
    }
    @Override
    public void ReadEncrypted() {
        throw new UnsupportedOperationException("Not supported yet.");
    }
}

package decorator;

public abstract class ReaderDecorator extends IReader {

   abstract public void ReadZip();
   abstract public void ReadEncrypted();

}
package decorator;


public class ZipReader extends ReaderDecorator {
    IReader reader;

    public ZipReader(IReader reader){
      super();
       this.reader=reader;

    }

    public void ReadZip(){
     System.out.println(" the file is zip ");

    }

    public void ReadFile()
    {
        reader.ReadFile();
         ReadZip();
    }

    @Override
    public void ReadEncrypted() {
        throw new UnsupportedOperationException("Not supported yet.");
    }

} package decorator;

public class EncryptedReader extends ReaderDecorator {
```

```
   IReader reader;

    public EncryptedReader(IReader reader){
      super();
       this.reader=reader;

    }

    EncryptedReader(){}

    public void ReadEncrypted(){
     System.out.println(" the file is encrypt ");

    }

    public void ReadFile()
    {
        reader.ReadFile();
         ReadEncrypted();
    }

    @Override
    public void ReadZip() {
        throw new UnsupportedOperationException("Not supported yet.");
    }
} import decorator.*;

public class Client {
   // private static File C;


    public static void main(String arg[])
    {

       IReader read = new FileReader1();
       read.ReadFile();
       ReaderDecorator decorator = new ZipReader(read);
       decorator.ReadZip();
       ReaderDecorator decorator1 = new EncryptedReader(read);
       decorator1.ReadEncrypted();
    }
}
```

Façade

Το σχέδιο είναι μια πρόσοψη μηχανικής λογισμικού πρότυπο σχεδίασης που χρησιμοποιούνται συνήθως με Αντικειμενοστρεφή προγραμματισμό . Το όνομα είναι κατ 'αναλογία σε μια αρχιτεκτονική πρόσοψη .

Η πρόσοψη είναι ένα αντικείμενο που παρέχει μια απλοποιημένη διεπαφή για ένα μεγαλύτερο σώμα του κώδικα, όπως μια βιβλιοθήκη κατηγορίας . Η πρόσοψη μπορεί να:

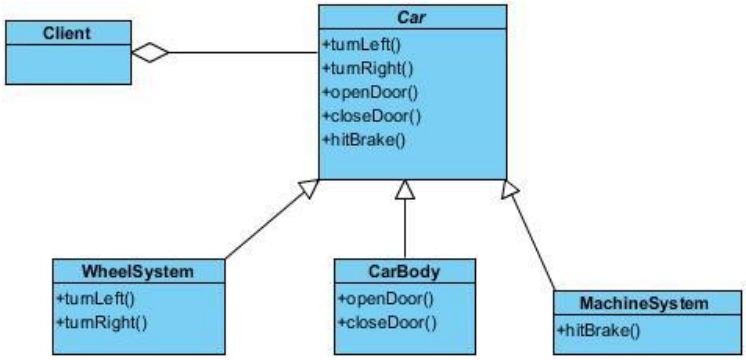
- κάνει μια βιβλιοθήκη λογισμικού ευκολότερη στη χρήση, να κατανοήσουν και να δοκιμάσουν, εφόσον η πρόσοψη έχει βολικές μεθόδους για κοινές εργασίες
- καθιστούν τη βιβλιοθήκη πιο ευανάγνωστη, για τον ίδιο λόγο
- να μειώσει τις εξαρτήσεις του κώδικα έξω από τις εσωτερικές λειτουργίες μιας βιβλιοθήκης, δεδομένου ότι ο περισσότερος κώδικας χρησιμοποιεί την πρόσοψη, επιτρέποντας έτσι μεγαλύτερη ευελιξία στην ανάπτυξη του συστήματος

Ένας προσαρμογέας χρησιμοποιείται όταν το περιτύλιγμα πρέπει να τηρεί ένα συγκεκριμένο περιβάλλον και πρέπει να υποστηρίξει μια πολυμορφική συμπεριφορά. Από την άλλη πλευρά, μια πρόσοψη χρησιμοποιείται όταν κάποιος θέλει μια πιο εύκολη ή πιο απλή διεπαφή για να εργαστεί.

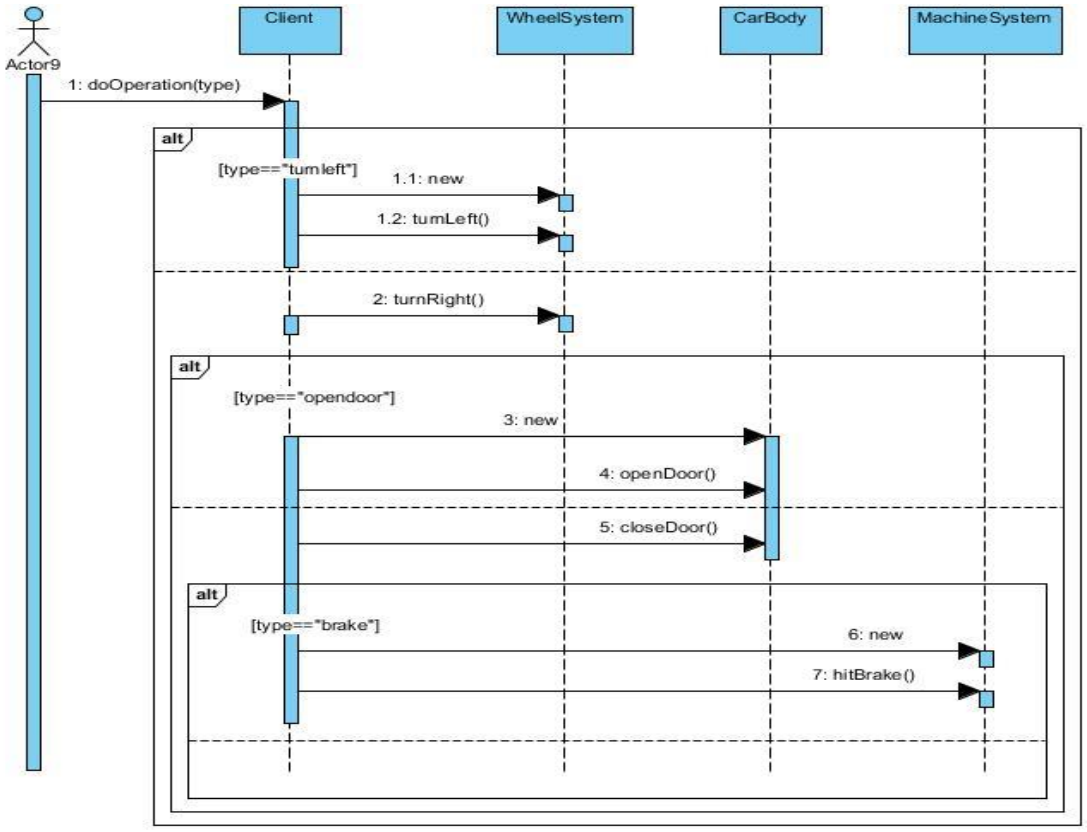
Πρόβλημα:

Η πρόσοψη(facade) καθορίζει ένα ενιαίο, υψηλότερο επίπεδο διασύνδεσης σε ένα υποσύστημα που καθιστά πιο εύκολο στη χρήση. Έχουμε μια δημιουργία ενός συστήματος αυτοκινήτου σαν αφηρημένη κλάση και με τη σειρά της δημιουργούνται υποκλάσεις με διάφορα εξαρτήματα του αυτοκινήτου, όταν το αυτοκίνητο δημιουργήθηκε με βάση τα πολύπλοκα υποσυστήματα, όπως τιμόνι, το σύστημα διεύθυνσης, πλαίσιο, αμάξωμα κ.τ.λ.

Class diagram



Sequence diagram



JAVA CODE

```
public class Car {

    public WheelSystem wheel;
    public CarBody body;
    public MachineSystem machine;

    public Car() {
        this.wheel = new WheelSystem();
        this.body = new CarBody();
        this.machine = new MachineSystem();
    }

    public void startCar() {
        wheel.turnLeft();
        body.closeDoor();
        machine.hitBrake();
    }
```

```
}

public class CarBody {

    public CarBody(){};

    public void openDoor()
    {
     System.out.println("open");
    }
    public void closeDoor()
    {
      System.out.println("close");
    }

}

public class MachineSystem {

    public MachineSystem(){};

    public void hitBrake()
    {
         System.out.println("hit");
    }

}

public class WheelSystem{

    public WheelSystem(){};

    public void turnLeft()
    {
        System.out.println("left");
    }
    public void turnRight()
    {
     System.out.println("right");
    }

}

public class Client {

    public static void main(String[] args) {
        Car facade = new Car();
        facade.startCar();
```

```
        System.out.print("car "+facade);
    }

}
```

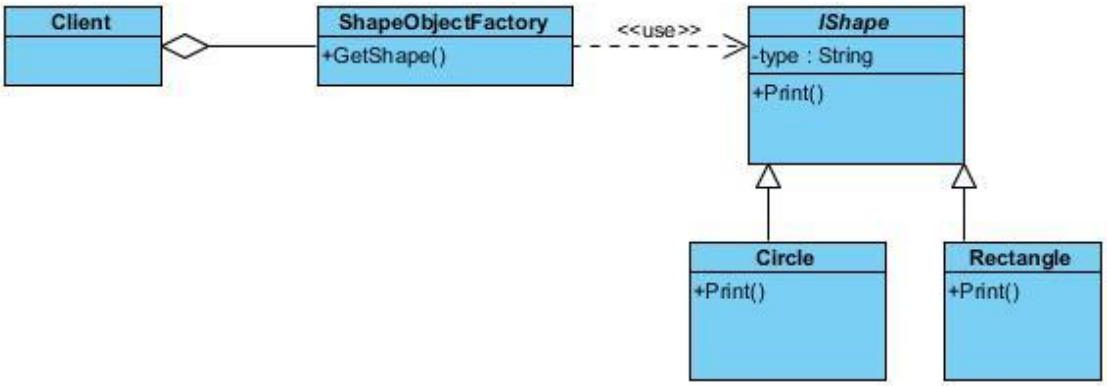
Flyweight

Flyweight είναι ένα λογισμικό πρότυπο σχεδιασμού . Μια flyweight είναι ένα αντικείμενο που ελαχιστοποιεί τη χρήση της μνήμης με την ανταλλαγή δεδομένων όσο το δυνατόν περισσότερο με άλλα παρόμοια αντικείμενα. Είναι ένας τρόπος να χρησιμοποιήσετε τα αντικείμενα σε μεγάλους αριθμούς όταν μια απλή επαναλαμβανόμενη παράσταση θα χρησιμοποιήσει ένα ποσό της μνήμης. Συχνά ορισμένα τμήματα του κράτους αντικείμενο μπορεί να μοιραστούν και είναι κοινά για τα εντάξουμε σε εξωτερικές δομές δεδομένων και να περάσουν στις flyweight αντικείμενα προσωρινά όταν χρησιμοποιούνται. Ένα κλασικό παράδειγμα χρήσης του flyweight πρότυπο είναι οι δομές δεδομένων για τη γραφική αναπαράσταση των χαρακτήρων σε ένα επεξεργαστή κειμένου . Μπορεί να είναι επιθυμητό να διαθέτουν, για κάθε χαρακτήρα σε ένα έγγραφο, ένα ιερογλυφικό αντικείμενο που περιέχει περιγραφή της γραμματοσειράς, μετρικές γραμματοσειράς και άλλα στοιχεία μορφοποίησης, αλλά αυτό θα ανέλθει σε εκατοντάδες ή χιλιάδες bytes για κάθε χαρακτήρα. Αντ 'αυτού, για κάθε χαρακτήρα, ίσως να υπάρξει μια αναφορά σε ένα αντικείμενο flyweight να μοιράζονται κάθε εμφάνιση του ίδιου του χαρακτήρα του εγγράφου μόνο η θέση του κάθε χαρακτήρα (στο έγγραφο και / ή τη σελίδα) θα πρέπει να αποθηκευτεί στο εσωτερικό τους. Σε άλλες περιπτώσεις η ιδέα της ανταλλαγής ίδιες δομές δεδομένων ονομάζεται hash consing .

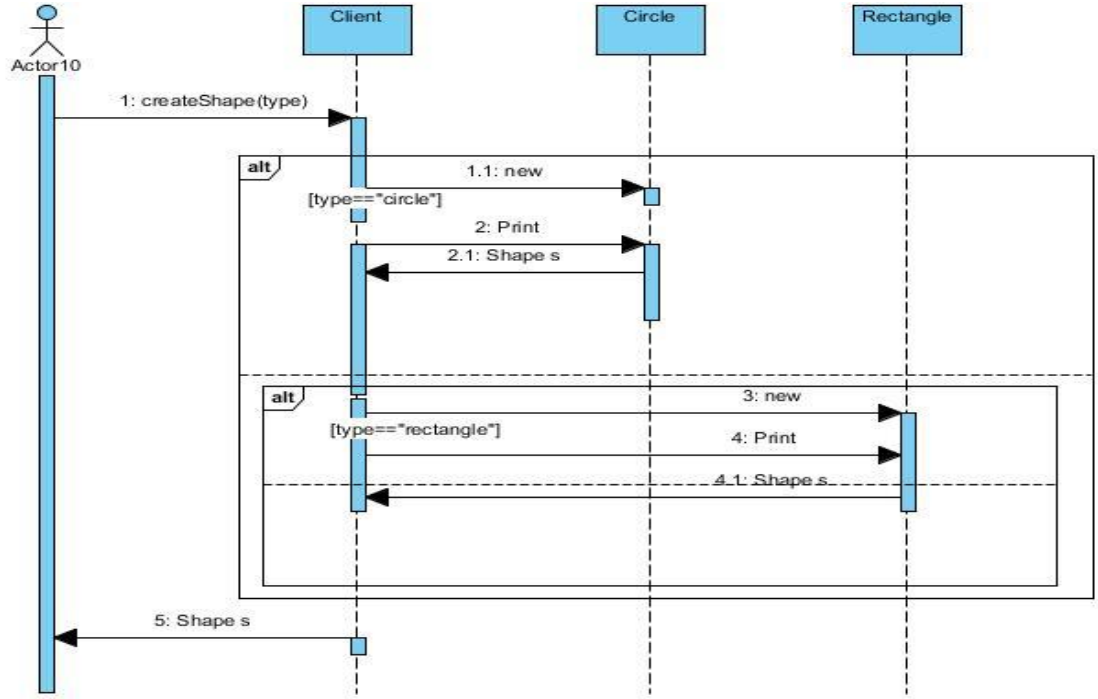
Πρόβλημα:

Το Flyweight χρησιμοποιεί κοινή χρήση για την υποστήριξη μεγάλου αριθμού αντικειμένων αποτελεσματικά. Έχουμε μια κλάση σχήμα και χωρίζεται σε δύο υποκλάσεις με διάφορα σχήματα.Τα δυο σχήματα που δεν είναι ίδια έχουν τις ίδιες μεθόδους. Όταν θα χρειαστεί να γίνει εκτύπωση των σχημάτων θα πρέπει τότε να φανεί πιο σχήμα θα εκτυπωθεί αναλόγως της κλάσης.

Class diagram



Sequence diagram



JAVA CODE

```
public class ShapeObjectFactory {

    IShape Shape;
    public void setShape(IShape s)
    {
      Shape=s;
    }
    public IShape getShape()
    {
        return Shape;
    }
}
```

```
abstract public class IShape extends ShapeObjectFactory{

    public IShape(){};

    public String type;
    abstract public IShape print();
   abstract  public IShape print(IShape shape);

}

public class Circle extends IShape{

    public Circle(){};

    public IShape print(IShape circle)
    {
        circle.setShape(circle);
        circle.getShape();
        return circle;
    }

    @Override
    public IShape print() {
        throw new UnsupportedOperationException("Not supported yet.");
    }



}

public class Rectangle extends IShape {

    public Rectangle(){};


    public IShape print(IShape rectangle)
           {
          rectangle.setShape(rectangle);
          rectangle.getShape();
          return rectangle;
    }

    @Override
    public IShape print() {
        throw new UnsupportedOperationException("Not supported yet.");
    }
}

public class Client {
```

```
    public static void main(String args[])
    {
     ShapeObjectFactory s=new ShapeObjectFactory();

     Circle c=new Circle();
     Rectangle r=new Rectangle();

    //System.out.print(c.setShape());

     System.out.println(" shape is "+c.print(c));
     System.out.println(" shape is "+r.print(r));

    }

}
```

Proxy

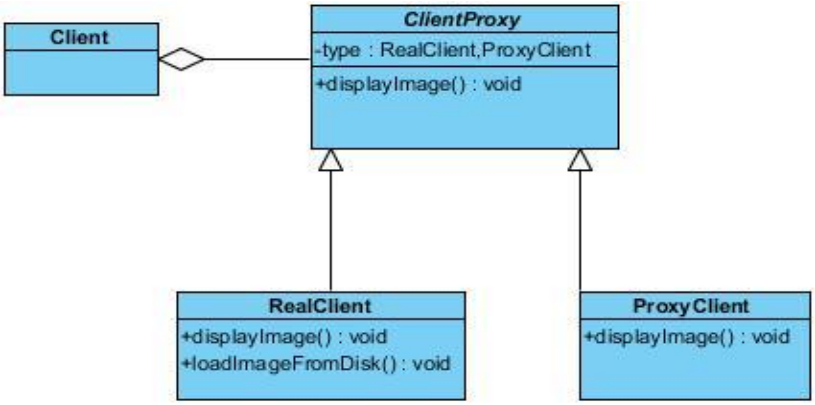
Σε προγραμματισμό ηλεκτρονικών υπολογιστών , το πληρεξούσιο πρότυπο είναι ένα πρότυπο σχεδίασης λογισμικού . Ο διαμεσολαβητής, στην πιο γενική μορφή της, είναι μια λειτουργία τάξη ως διεπαφή για κάτι άλλο. Ο πληρεξούσιος θα μπορούσε να διασυνδεθεί με : μια σύνδεση με το δίκτυο, ένα μεγάλο αντικείμενο στη μνήμη, ένα αρχείο, ή κάποια άλλη πηγή που είναι ακριβό ή αδύνατο να επιτευχθούν. Ένα γνωστό παράδειγμα της μεσολάβησης είναι ένα πρότυπο αναφοράς καταμέτρησης δείκτη αντικειμένου. Σε περιπτώσεις όπου πολλαπλά αντίγραφα ενός σύνθετου αντικειμένου πρέπει να υπάρχει, το πληρεξούσιο πρότυπο μπορεί να προσαρμόζεται για να ενσωματώσει την flyweight πρότυπο προκειμένου να μειώσει το αποτύπωμα μνήμης της εφαρμογής. Συνήθως, μία εμφάνιση του συγκροτήματος και αντικείμενο πολλών αντικειμένων μεσολάβησης δημιουργούνται, τα οποία περιέχουν αναφορά στην ενιαία αρχική σύνθετο αντικείμενο. Τυχόν εργασίες που αφορούν τα πληρεξούσια διαβιβάζεται στο αρχικό αντικείμενο. Όταν όλες οι περιπτώσεις της μεσολάβησης είναι εκτός του πεδίου εφαρμογής, η μνήμη του σύνθετου αντικειμένου μπορεί να είναι αριθμητική τιμή, σύμβολο.

Πρόβλημα:

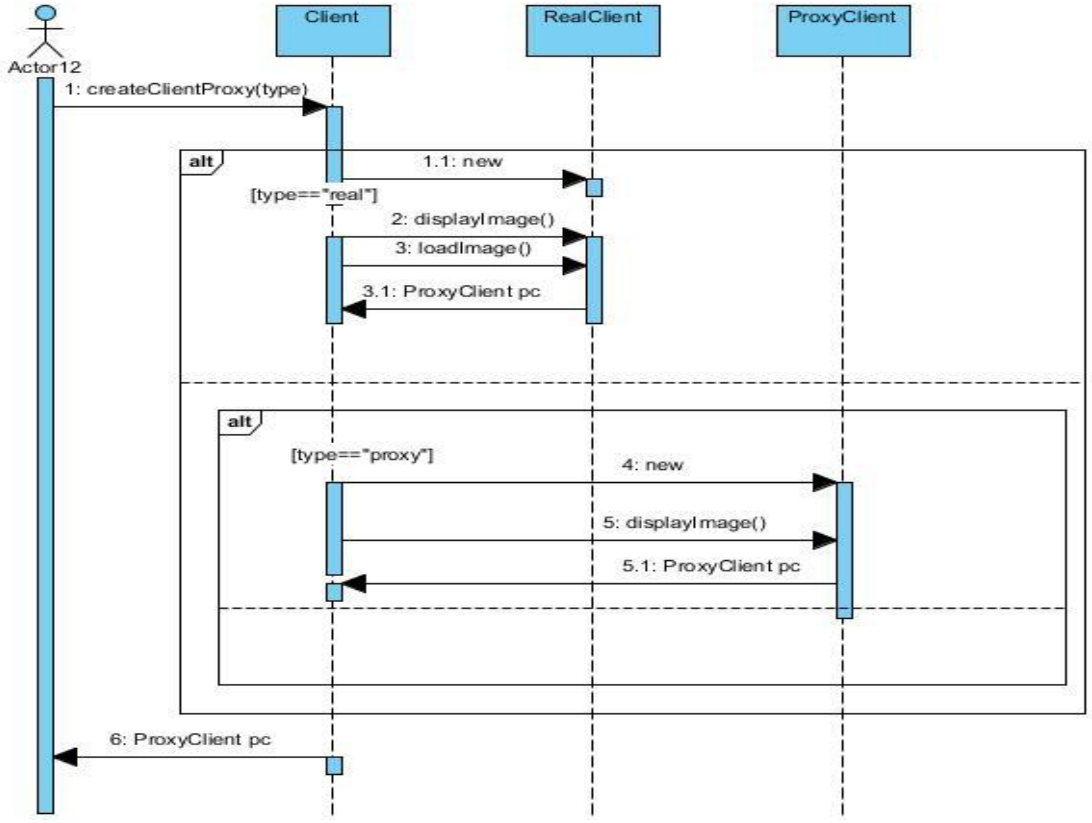
Το πρότυπο πληρεξούσιο(Proxy) παρέχει έναν αντικαταστάτη ενός άλλου αντικειμένου στο οποίο δεν είναι δυνατή η πρόσβαση υπό κανονικές συνθήκες.Ας πούμε οτι έχουμε ένα σύστημα ρύθμισης όπου θα στέλνουν και να λαμβάνουν δεδομένα μέσω του δικτύου. Τώρα, λόγω ορισμένων λόγων ασφαλείας των δεδομένων είναι κρυπτογραφημένα και θα πρέπει να αποκρυπτογραφηθούν πριν από την επεξεργασία και έτσι τώρα είμαστε σε δύσκολη θέση, διότι όλος ο κώδικας

πελάτης πρέπει να αλλάξει για να αποκρυπτογραφήσει τα δεδομένα ή μπορεί να χρειαστεί να αλλάξετε τον κωδικό τον οποίο χρησιμοποιήσετε για να στείλετε / λάβετε τα δεδομένα. Τώρα, η λύση θα ήταν να έχουν την εξουσιοδότηση που θα κάνει την πρόσθετη ευθύνη που δόθηκε στο σύστημα και στη συνέχεια να στείλει τα δεδομένα χρησιμοποιώντας το δοκιμασμένο σύστημα.

Class diagram



Sequence diagram



JAVA CODE

```
abstract public class ClientProxy {


    void displayImage() {
        throw new UnsupportedOperationException("Not yet implemented");
    }

}

public class RealClient extends ClientProxy{

    private String filename;

    public RealClient(String filename) {
        this.filename = filename;
        loadImageFromDisk();
    }

    private void loadImageFromDisk() {
        System.out.println("Loading   " + filename);
    }

    public void displayImage() {
        System.out.println("Displaying " + filename);
    }


}

public class ProxyClient extends ClientProxy {
private String filename;
    private RealClient image;

    public ProxyClient(String filename) {
        this.filename = filename;
    }

    public void displayImage() {
        if (image == null) {
           image = new RealClient(filename);
        }
        image.displayImage();
    }

   }
```

```
public class Client {

    public static void main(String[] args) {
        ClientProxy image1 = new ProxyClient("HiRes_10MB_Photo1");
        ClientProxy image2 = new ProxyClient("HiRes_10MB_Photo2");

        image1.displayImage();
        image2.displayImage();
        image1.displayImage();
    }



}
```

ΚΕΦΑΛΑΙΟ 3 - Σχεδιαστικά πρότυπα συμπεριφοράς (Behavioral design patterns)

Τα πρότυπα συμπεριφοράς ασχολούνται με αλγορίθμους και την ανάθεση των αρμοδιοτήτων μεταξύ των αντικειμένων. Τα πρότυπα σχεδίασης αυτής της κατηγορίας δεν περιγράφουν απλώς το είδος των αντικειμένων ή κλάσεων, αλλά και της επικοινωνίας μεταξύ τους. Αυτά τα πρότυπα συναντώνται, κυρίως, σε περιπτώσεις όπου υπάρχουν περίπλοκες ροές ελέγχου που είναι δύσκολο να τις παρακολουθήσει κανείς κατά το χρόνο εκτέλεσης. Τα συγκεκριμένα πρότυπα βοηθάνε τον αναγνώστη του κώδικα να επικεντρωθεί αποκλειστικά στον τρόπο με τον οποίο συνδέονται μεταξύ τους τα αντικείμενα, παραβλέποντας την κάθε ροή ελέγχου. Τα πρότυπα συμπεριφοράς που ορίστηκαν από τους E. Gamma, R. Helms, R. Johnson και J. Vlissides είναι τα εξής: Chain of responsibility, Command, Interpreter, Iterator, Mediator, Memento, Observer, State, Strategy, Template method, Visitor.

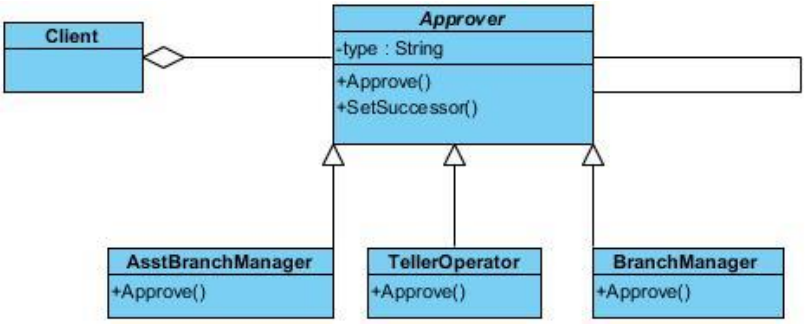
3.1 Chain of responsibility

Σε Αντικειμενοστρεφή Σχεδιασμό , η αλυσίδα της ευθύνης-πρότυπο είναι ένα πρότυπο σχέδιο που αποτελείται από μια πηγή των αντικειμένων εντολή και μια σειρά από αντικείμενα επεξεργασίας. Κάθε αντικείμενο επεξεργασίας περιέχει λογική που καθορίζει τα είδη των αντικειμένων εντολών που μπορεί να χειριστεί. Τα υπόλοιπα περνάνε στο επόμενο αντικείμενο επεξεργασίας στην αλυσίδα. Ο μηχανισμός υπάρχει και για την προσθήκη νέων αντικειμένων επεξεργασία μέχρι το τέλος αυτής της αλυσίδας. Σε μια παραλλαγή του το καθιερωμένο μοντέλο της αλυσίδας του κύκλου ευθύνης, κάποιοι διαχειριστές μπορούν να ενεργούν ως αποστολείς , ικανό να στέλνει εντολές σε ποικίλες κατευθύνσεις, σχηματίζοντας ένα *δέντρο της ευθύνης.* Σε ορισμένες περιπτώσεις, αυτό μπορεί να συμβεί κατ 'επανάληψη, με την επεξεργασία των αντικειμένων καλώντας υψηλότερη μέχρι αντικείμενα επεξεργασίας με εντολές που προσπαθούν να λύσουν ένα μικρότερο μέρος του προβλήματος. Στην περίπτωση αυτή αναδρομή συνεχίζεται έως ότου η εντολή επεξεργασία, ή ολόκληρο το δέντρο έχει εξερευνηθεί. Ο διερμηνέας XML θα μπορούσε να λειτουργήσει με αυτόν τον τρόπο.

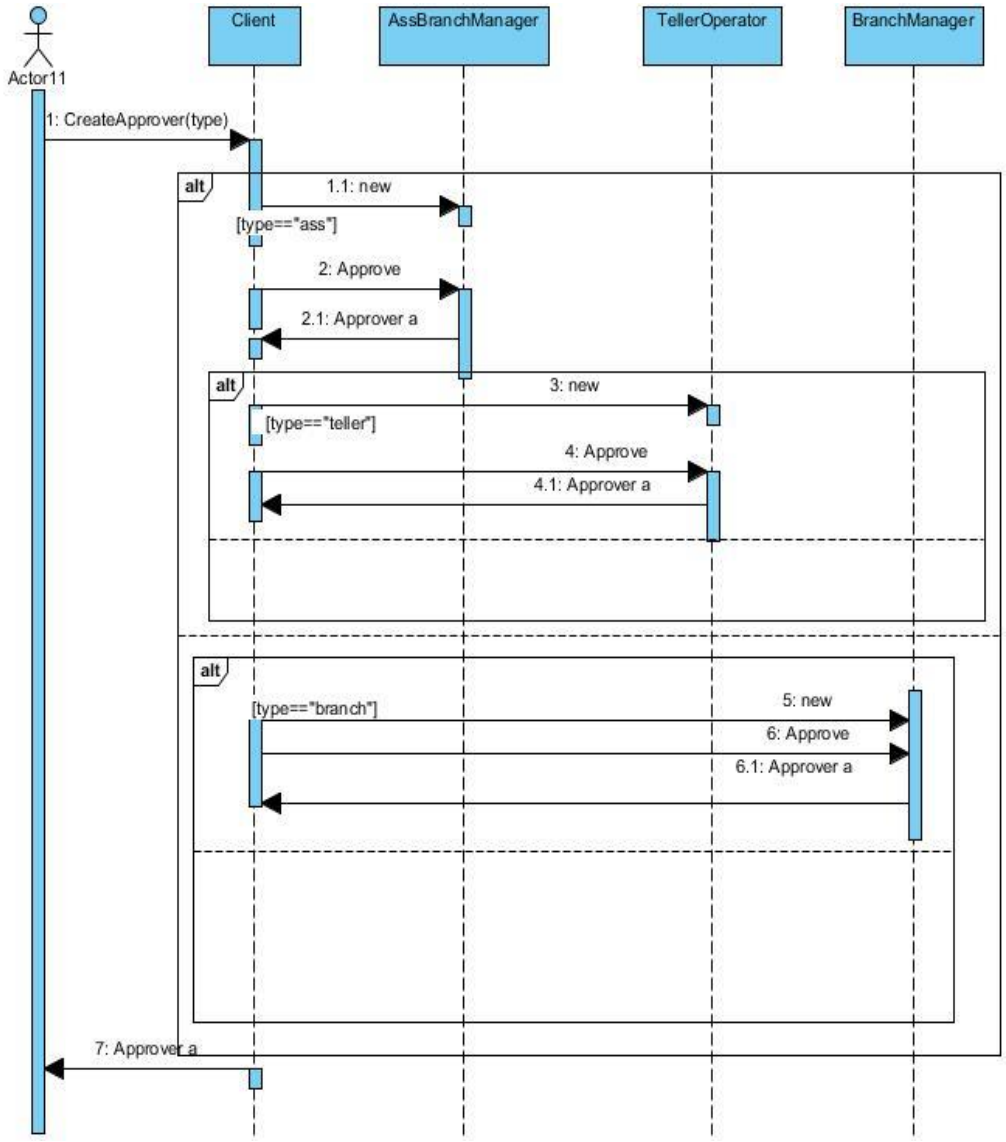
3.1.1 Πρόβλημα

Η αλυσίδα Ευθύνης(Chain of responsibility) αποφεύγει σύζευξη του αποστολέα ενός αιτήματος στο δέκτη, δίνοντας σε περισσότερα από ένα αντικείμενα την ευκαιρία να χειριστεί το αίτημα. Στο τραπεζικό σύστημα, όπου υπάρχει έλεγχος για την εκκαθάριση έχει εγκριθεί από το πρόσωπο, αλλά αν ο έλεγχος δείξει ότι το ποσό έχει υπερβεί ορισμένα όρια, η ευθύνη για την έγκριση κινείται στο υψηλότερο πρόσωπο της τράπεζας.

3.1.2 Class diagram



3.1.2 Sequence diagram



3.1.3 Ενδεικτικός Κώδικας

```
abstract public class Approver {

    public Approver(){}

    public String type;
    public void approve(Request request){};

    protected Approver m_successor;

   public void setSuccessor(Approver successor)
       {
              m_successor = successor;
       }

}

public class AssBranchManager extends Approver {

public AssBranchManager(){};
    public void approve(Request request)
       {
              if (request.getValue() < 0)
              {
                     System.out.println("Negative values are handled by
AssBranchManager:");
                  System.out.println("\tAssBranchManager.HandleRequest : " +
request.getDescription()
                                              + request.getValue());
              }
              else
              {
                     super.approve(request);
              }
       }


}

public class BranchManager extends Approver {

public BranchManager(){};
    public void approve(Request request)
       {
              if (request.getValue() > 0)
              {
                     System.out.println("Positive values are handled by
BranchManager:");
                  System.out.println("\tBranchManager.HandleRequest : " +
request.getDescription()
                                              + request.getValue());
              }
         else
              {
                     super.approve(request);
              }
       }
}



public class TellerOperator extends Approver {

    public TellerOperator(){};
```

```
    public void approve(Request request)
       {
              if (request.getValue() > 0)
              {
                   System.out.println("Positive values are handled by
                                       TellerOperator:");
                  System.out.println("\tTellerOperator.HandleRequest : " +
                           request.getDescription() + request.getValue());
              }
         else
              {
                    super.approve(request);
              }
       }
}


public class Request {


    private int m_value;
       private String m_description;

       public Request(String description, int value)
       {
              m_description = description;
              m_value = value;
       }

       public int getValue()
       {
              return m_value;
       }

       public String getDescription()
       {
              return m_description;
       }
}

public class Client {


    public static void main(String[] args)
       {

              AssBranchManager h1 = new AssBranchManager();
              TellerOperator h2 = new TellerOperator();
              BranchManager h3 = new BranchManager();
              h1.setSuccessor(h2);
              h2.setSuccessor(h3);


              h1.approve(new Request("Negative Value ", -1));
              h1.approve(new Request("Negative Value ",  0));
              h1.approve(new Request("Negative Value ",  1));
              h1.approve(new Request("Negative Value ",  2));
              h1.approve(new Request("Negative Value ", -5));
               h2.approve(new Request("Negative Value ",  2));
              h3.approve(new Request("Negative Value ",  5));

       }
}
```

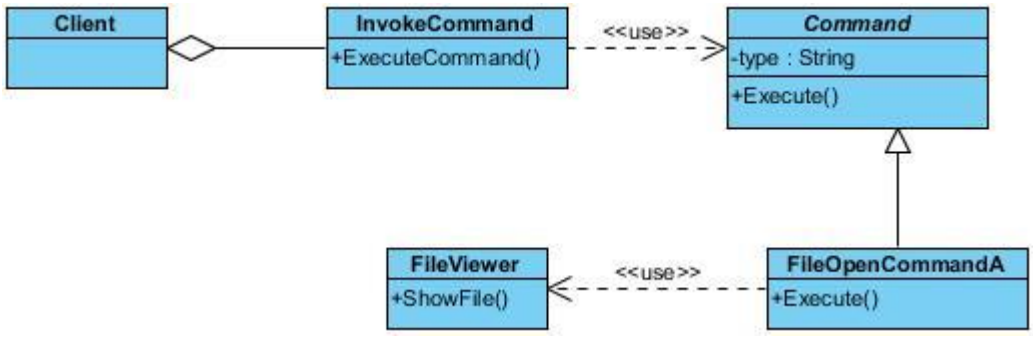
Command

Στον αντικειμενοστρεφή προγραμματισμό , το σχέδιο εντολής είναι ένα πρότυπο σχέδιο στο οποίο ένα αντικείμενο χρησιμοποιείται για να αντιπροσωπεύσει και να ενσωματώσει όλες τις πληροφορίες που απαιτούνται για την κλήση μιας μεθόδου σε μεταγενέστερο χρόνο. Αυτές οι πληροφορίες περιλαμβάνουν το όνομα της μεθόδου, το αντικείμενο που κατέχει τη μέθοδο και τιμές για τις παραμέτρους μέθοδο. Τρεις όρους πάντα που σχετίζονται με το σχέδιο εντολής είναι *πελάτης, invoker* και του *δέκτη.* Ο *πελάτης* το αντικείμενο εντολή και παρέχει τις πληροφορίες που απαιτούνται για να καλέσει τη μέθοδο σε μεταγενέστερο χρόνο. Η *invoker* αποφασίζει πότε η μέθοδος θα πρέπει να ονομάζεται. Ο *δέκτης* είναι ένα στιγμιότυπο της κλάσης που περιέχει κώδικα της μεθόδου. Χρησιμοποιώντας αντικείμενα εντολή καθιστά ευκολότερο να κατασκευάσει τα γενικά συστατικά που πρέπει να αναθέσει, ή να εκτελέσει ακολουθία κλήσεων μεθόδου σε μια στιγμή της επιλογής τους χωρίς την ανάγκη να γνωρίζουν τον ιδιοκτήτη της μεθόδου ή των παραμέτρων της μεθόδου.

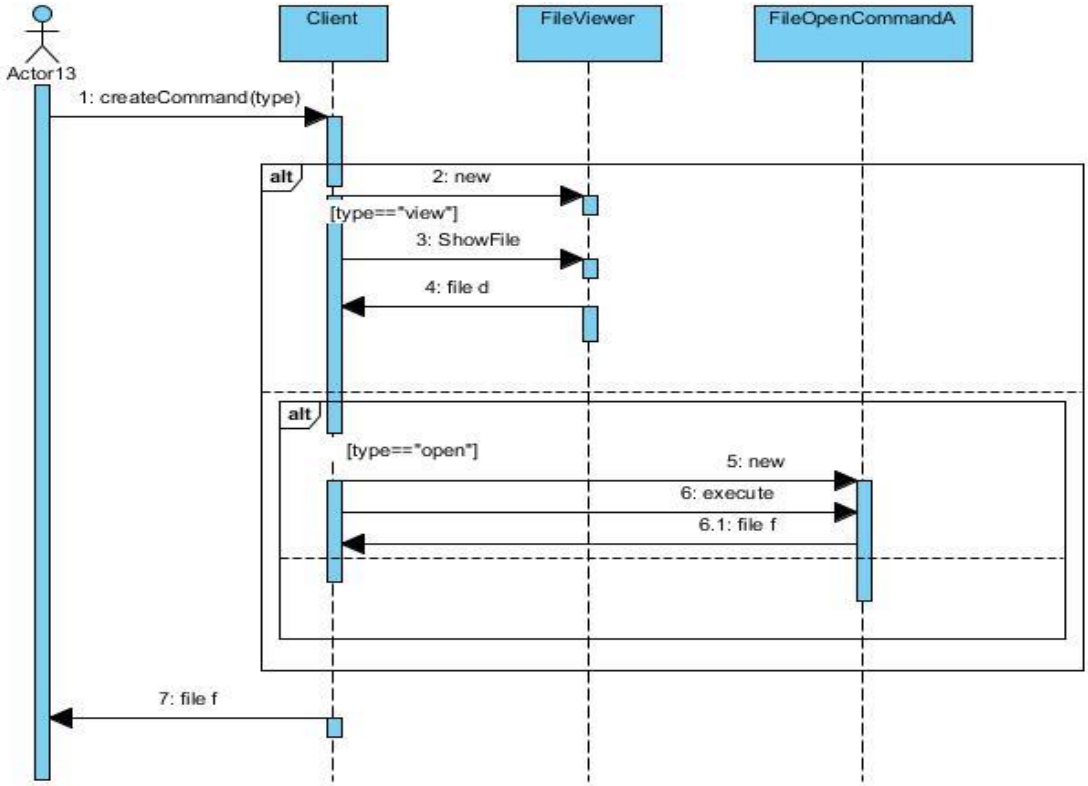
Πρόβλημα:

Για το πρότυπο αυτό έστω ότι έχουμε σχεδιάσει ένα πρόγραμμα επεξεργασίας εικόνας και ο χρήστης μπορεί να έχει τη δυνατότητα ανοίγματος αρχείου από διάφορους τρόπους, όπως μενού, γραμμή εργαλείων, διπλό κλικ σε ένα αρχείο στον εξερευνητή. Η λύση είναι το πρότυπο όπου η εντολή FileOpen εντολή συνδέεται στο χρήστη και το ίδιο όταν η εντολή εκτελεί το αρχείο εμφανίζεται στο χρήστη.

Class diagram



Sequence diagram



JAVA CODE

```
import java.util.List;
import java.util.ArrayList;

public class InvokeCommand {
  private List<Command> history = new ArrayList<Command>();

  public InvokeCommand() {
  }

  public void ExecuteCommand(Command cmd) {
    this.history.add(cmd); // optional
    cmd.Execute();
  }
}

abstract public class Command {

   public Command(){};

   abstract public void Execute();

}
```

```
public class FileOpenCommandA extends Command {

     private FileViewer file;

     public FileOpenCommandA(){}

    public FileOpenCommandA ( FileViewer st) {
        file = st;
    }
    public void Execute( ) {
        file.ShowFile( );
    }


}
public class FileViewer {

    public FileViewer(){};

    public void ShowFile() {
        System.out.println("Show the file");
    }

}

public class Client {

    public static void main(String args[]){

      FileViewer file = new FileViewer();
      Command com = new FileOpenCommandA(file);
      InvokeCommand invoke = new InvokeCommand();
           invoke.ExecuteCommand(com);



   }
}
```

Interpreter

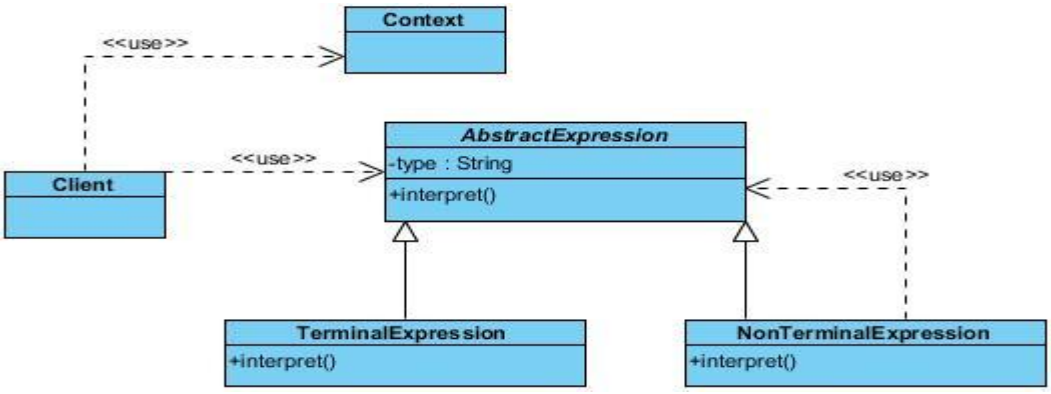
Σε προγραμματισμό ηλεκτρονικών υπολογιστών , το πρότυπο διερμηνέας είναι ένα πρότυπο σχέδιο . Το σχέδιο καθορίζει τον τρόπο διερμηνέα να αξιολογεί προτάσεις σε μια γλώσσα. Η βασική ιδέα είναι να έχουμε μια τάξη για κάθε σύμβολο (τερματικό ή μη τερματικό σύμβολο) σε μια εξειδικευμένη γλώσσα του

υπολογιστή . Το δέντρο σύνταξη μιας πρότασης στη γλώσσα είναι ένα παράδειγμα της σύνθετης πρότυπο και χρησιμοποιούνται για να αξιολόγησει (ερμηνεύσει) την πρόταση.

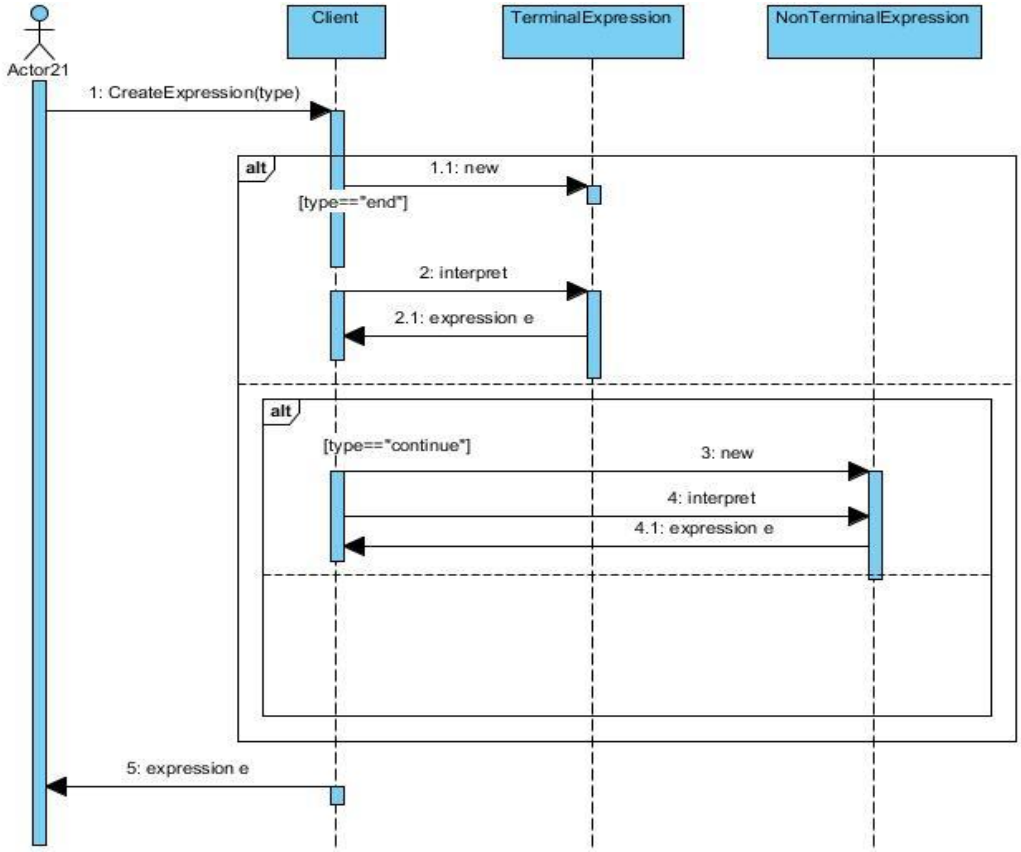
Πρόβλημα:

Το πρότυπο διερμηνέας(interpreter) χρησιμοποιείται για να ερμηνεύει τις διάφορες εκφράσεις.Το κλασικό παράδειγμα η ερμηνεία των λατινικών αριθμών. Η expresion είναι ένα string που τίθεται στο πλαίσιο. Το πλαίσιο αποτελείται από την υπόλοιπη μη αναλυμένη Ρωμαϊκή σειρά αρίθμησης και του αποτελέσματος της numerral που έχουν ήδη αναλυθεί. η κλάση context διατηρεί την τρέχουσα σειρά που πρέπει να αναλυθεί και το δεκαδικό που περιέχει τη μετατροπή έχει ήδη γίνει. Αρχικά το πλαίσιο διατηρεί την πλήρη συμβολοσειρά που πρέπει να μετατραπεί και 0 για τον δεκαδικό εξόδου.Οι κλάσεις expression αποτελείται από τη μέθοδο ερμηνείας που δέχεται το πλαίσιο. Με βάση το τρέχων αντικείμενο που χρησιμοποιεί συγκεκριμένες τιμές για χιλιάδες, εκατό, δέκα, ένα και ένα συγκεκριμένο πολλαπλασιαστή. Στην περίπτωσή μας η μέθοδος έχει ήδη οριστεί στην κατηγορία Έκφραση βάση της κάθε τάξης TerminalExpression που καθορίζει τη συμπεριφορά του από τις αφηρημένες μεθόδους: ένα, τέσσερα (), πέντε (), εννέα (), πολλαπλασιαστή (). Πρόκειται για ένα πρότυπο μέθοδος πρότυπο. Η client τάξη είναι υπεύθυνη για την κατασκευή του συντακτικού δέντρου που αντιπροσωπεύει μια συγκεκριμένη πρόταση στη γλώσσα που ορίζεται από τη γραμματική. Μετά το συντακτικό δέντρο είναι χτισμένη η κύρια μέθοδος που επικαλείται τη μέθοδο ερμηνείας.

Class diagram



Sequence diagram



JAVA CODE

```
public class Context {

    private String input;
    private int output;
    public Context(String input)
    {
        this.input = input;
    }
    public String getInput()
    {
        return input;
    }
    public void setInput(String input)
    {
        this.input = input;
    }
    public int getOutput()
    {
        return output;
```

```
    }
    public void setOutput(int output)
    {
        this.output = output;
    }

}

public abstract class AbstractExpression {

    abstract public boolean interpret(String str);
}

import java.util.StringTokenizer;

public class TerminalExpression extends AbstractExpression {

            private String literal = null;
          public TerminalExpression(String str) {
              literal = str;
          }

          public boolean interpret(String str)
          {
              StringTokenizer st = new StringTokenizer(str);
              while (st.hasMoreTokens()) {
                  String test = st.nextToken();
                  if (test.equals(literal)) {
                      return true;
                  }
              }
              return false;
          }
}



public class NonTerminalExpression extends AbstractExpression {

    private AbstractExpression expression1 = null;
    private AbstractExpression expression2 = null;

    public          NonTerminalExpression(AbstractExpression          expression1,
AbstractExpression expression2)
    {
        this.expression1 = expression1;
        this.expression2 = expression2;
    }
    public boolean interpret(String str)
    {
```

```
        return expression1.interpret(str) && expression2.interpret(str);
    }


}

public class Client {

   static AbstractExpression buildInterpreterTree()
   {

      AbstractExpression terminal = new TerminalExpression("John");
      AbstractExpression terminal2 = new TerminalExpression("Henry");
      AbstractExpression terminal3 = new TerminalExpression("Mary");

      AbstractExpression alternation1 = new NonTerminalExpression(terminal2, terminal3);

       return new NonTerminalExpression(terminal,terminal2);
   }


   public static void main(String[] args) {
      String context = "Mary Owen";
      AbstractExpression define = buildInterpreterTree();
      System.out.println(context + " is " + define.interpret(context));
   }
}
```

Iterator

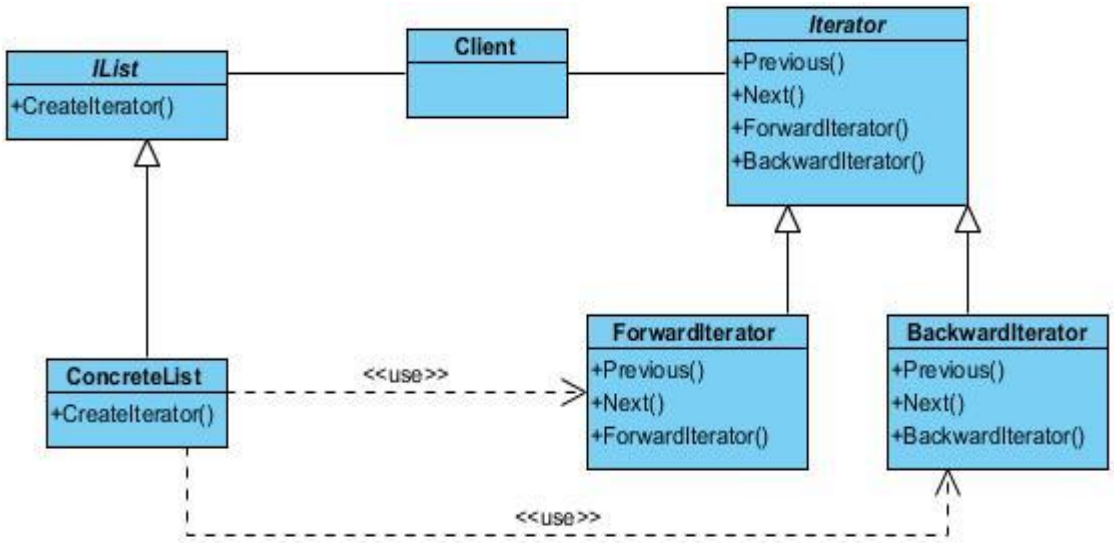
Στον αντικειμενοστρεφή προγραμματισμό , το σχέδιο επαναλήπτης είναι ένα πρότυπο σχέδιο στο οποίο ένας επαναλήπτης χρησιμοποιείται για να διασχίσει ένα δοχείο και πρόσβαση στα στοιχεία του εμπορευματοκιβωτίου. Ο επαναλήπτης πρότυπο αποσυνδέει αλγόριθμους από τα δοχεία σε ορισμένες περιπτώσεις, οι αλγόριθμοι είναι κατ 'ανάγκην ειδικά εμπορευματοκιβώτια και συνεπώς δεν μπορεί να αποσυνδεθεί.

Πρόβλημα:

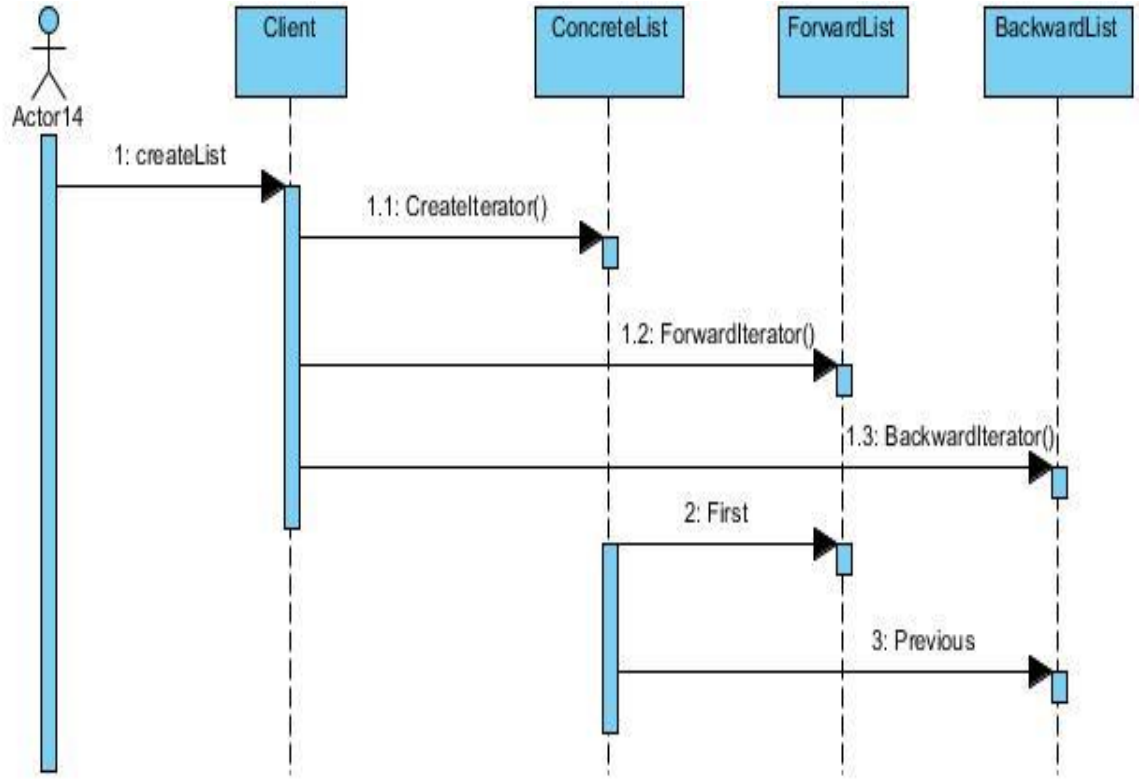
Η Iterator παρέχει τρόπους για την πρόσβαση στα στοιχεία ενός αντικειμένου. Παρέχετε έναν τρόπο να έχουν πρόσβαση στα στοιχεία των συνολικών διαδοχικών αντικειμένων. Για παράδειγμα έχουμε μια λίστα και ένα δρομέα και

υπάρχει ανεξάρτητη κίνηση του εμπρός ή πίσω.Για να γίνει εφικτή αυτή η λειτουργία θα πρέπει να υπάρχει ο iterator.

Class diagram



Sequence diagram



JAVA CODE

```
abstract public class IList {

 abstract public Iterator createIterator();

}

public class ConcreteList extends IList {

    private String list[] = {"Design Patterns","1","2","3","4"};
    public Iterator createIterator()
    {
        Forward result = new Forward();
        return result;
    }

}

abstract public class Iterator {

abstract public boolean previous();
abstract public boolean next();
abstract public Object ForwardIterator();
abstract public Object BackwardIterator();

}

public class Forward extends Iterator{

    private int l_position=10;
    private Object list[]=new Object[11];
    String next;
    public boolean previous()
    {
        if (l_position > list.length){
            System.out.println("there is previous");
            return true;
        }
        else
            return false;
    }
    public boolean next()
```

```
    {
        if (l_position < list.length){
            System.out.println("there is next");
           return true;
        }
        else
           return false;

    }

    public Object ForwardIterator()
    {
        if (this.next())
           return list[l_position++];
        else
           return null;
    }

    public Object BackwardIterator() {
       if (this.previous())
           return list[l_position++];
        else
           return null;
    }
}


public class Backward extends Iterator {

   private int l_position=10;
    private Object list[]=new Object[11];
    public boolean previous()
    {
        if (l_position > list.length){
           System.out.println("there is previous");
           return true;
        }
        else
            return false;
    }
    public boolean next()
    {
        if (l_position < list.length){
            System.out.println("there is next");
           return true;
        }
        else
           return false;

    }
```

```
    public Object ForwardIterator()
    {
        if (this.next())
            return list[l_position++];
        else
            return null;
    }

    public Object BackwardIterator() {
       if (this.previous())
            return list[l_position++];
        else
            return null;
    }
}


public class Client{
    public static void main(String args[])
    {
      ConcreteList list=new ConcreteList();
      Forward f=new Forward();
      Backward b=new Backward();

      list.createIterator();
      f.next();
      f.previous();
      f.BackwardIterator();
      f.ForwardIterator();

      b.next();
      b.previous();
      b.BackwardIterator();
      b.ForwardIterator();



    }

}
```

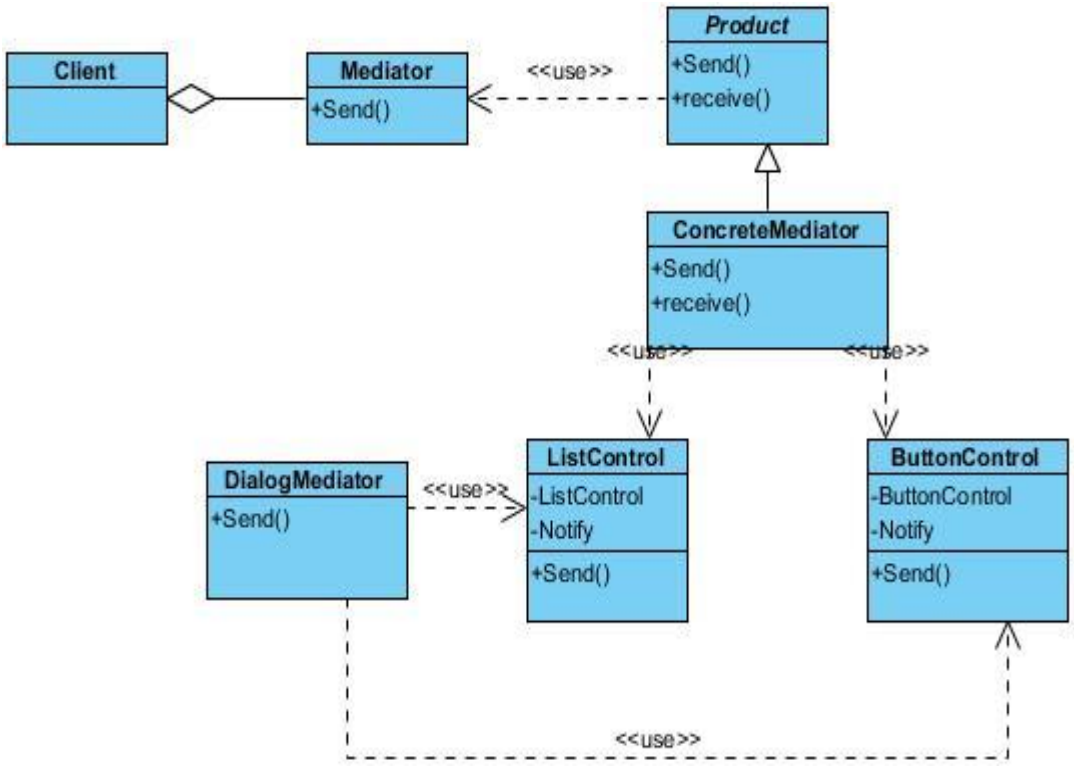
Mediator

Το πρότυπο μεσολαβητής καθορίζει ένα αντικείμενο που συμπυκνώνει με τον τρόπο ένα σύνολο αντικειμένων που αλληλεπιδρούν. Αυτό το σχέδιο θεωρείται ότι είναι ένα πρότυπο συμπεριφοράς που οφείλεται στον τρόπο που μπορεί να αλλάξει τη συμπεριφορά λειτουργίας του προγράμματος. Συνήθως ένα πρόγραμμα αποτελείται από ένα (σημαντικό) αριθμό των τάξεων . Έτσι, η λογική και υπολογισμός κατανέμεται μεταξύ αυτών των κατηγοριών. Ωστόσο, δεδομένου ότι οι περισσότερες τάξεις αναπτύσσονται σε ένα πρόγραμμα, ειδικά κατά τη διάρκεια της συντήρησης και / ή refactoring , το πρόβλημα της επικοινωνίας μεταξύ αυτών των κατηγοριών μπορεί να γίνει πιο περίπλοκη. Αυτό κάνει το πρόγραμμα πιο δύσκολο να διαβάσει και να διατηρηθεί . Επιπλέον, μπορεί να γίνει δύσκολο να αλλάξετε το πρόγραμμα, δεδομένου ότι οποιαδήποτε αλλαγή μπορεί να επηρεάσει τον κώδικα σε πολλές άλλες κατηγορίες. Με το πρότυπο μεσολαβητή, η επικοινωνία μεταξύ αντικειμένων είναι ενσωματωμένη με ένα αντικείμενο μεσολαβητή. Αντικείμενα πλέον μπορούν να επικοινωνούν απευθείας μεταξύ τους, αλλά επικοινωνούν μέσω του διαμεσολαβητή. Αυτό μειώνει τις εξαρτήσεις μεταξύ των αντικειμένων επικοινωνία, μειώνοντας έτσι τη σύζευξη .

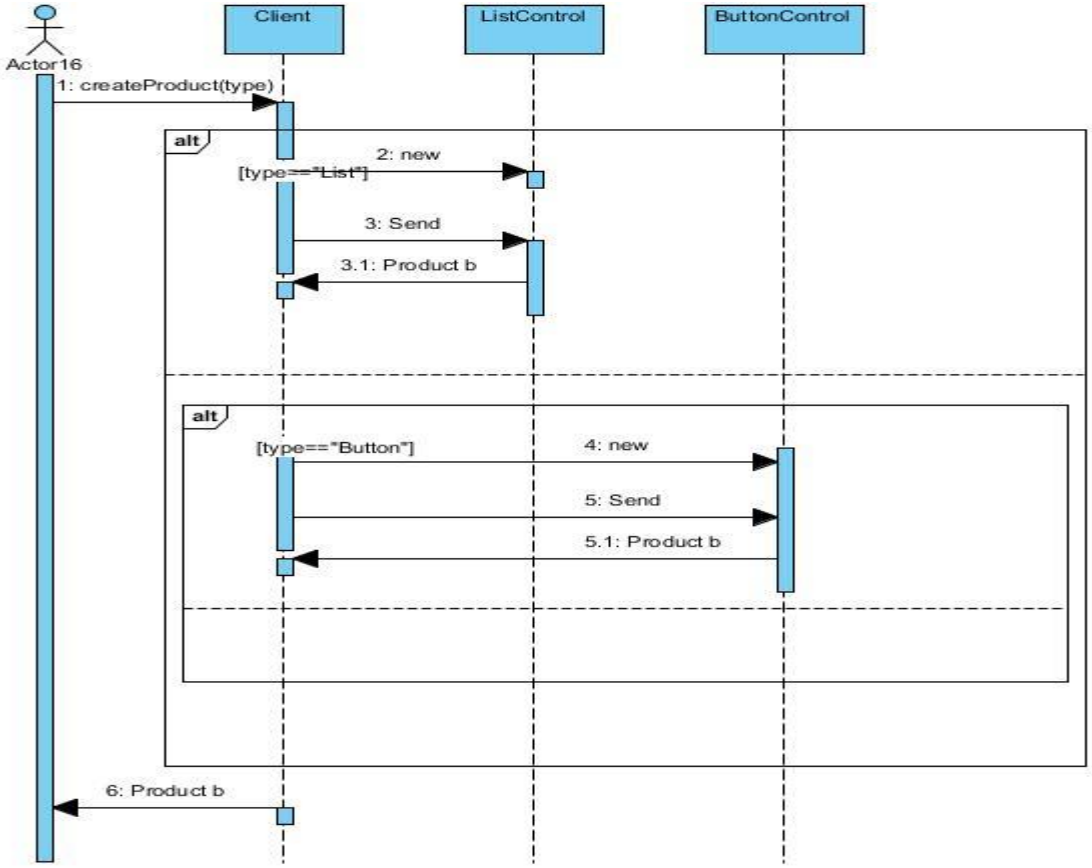
Πρόβλημα:

Ο διαμεσολαβητής(Mediator) ορίζει ένα αντικείμενο που ελέγχει μια σειρά από αντικείμενα που αλληλεπιδρούν. Έχουμε μια βιβλιοθήκη ελέγχου, όπου έχουμε ένα παράθυρο και κάποια χειριστήρια πάνω από το παράθυρο.Οι έλεγχοι μπορούν να αλληλεπιδράσουν με την DialogMediator . Τώρα, αν τα αδέλφια του διαλόγου θα ήθελα να αλληλεπιδράσουν τότε θα πράξει μέσω του DialogMediator του επειδή η DialogMediator έχει τις πληροφορίες για το υπόλοιπο της υπο-σύστημα.

Class diagram



Sequence diagram



JAVA CODE

```
abstract public class Product {

    private Mediator mediator;

    public Product(){}

public Product(Mediator m)
  {
   mediator = m;
  }


public void send(String message)
{
```

```
 mediator.send(message, this);
}



public Mediator getMediator()
{
return mediator;
}

abstract public void receive(String message);



}

abstract public class Mediator {

  abstract  public void send(String message, Product product);


}

import java.util.ArrayList;

public class ConcreteMediator extends Mediator {

    private ArrayList<Product> products;

public ConcreteMediator()

{
    products = new ArrayList<Product>();
}

public void addColleague(Product product)

{
    products.add(product);
}

public void send(String message, Product product)

if(product != product)

{

product.receive(message);

}
```

```
    }

}

public class ListControl extends Product{

  public ListControl(ConcreteMediator mediator) {}

    public void receive(String message)
    {
      System.out.println("List Received: " + message);

    }

}

public class ButtonControl extends Product {

  public ButtonControl(ConcreteMediator mediator) { }


public void receive(String message)
    {
         System.out.println("Button Received: " + message);

    }
}


public class Client {

public static void main(String[] args)
{

     ConcreteMediator mediator = new ConcreteMediator();
     ListControl list = new ListControl(mediator);
     ButtonControl button = new ButtonControl(mediator);
      mediator.addColleague(list);

      mediator.addColleague(button);

      list.receive("list")
      button.receive("button");


}
```

Memento

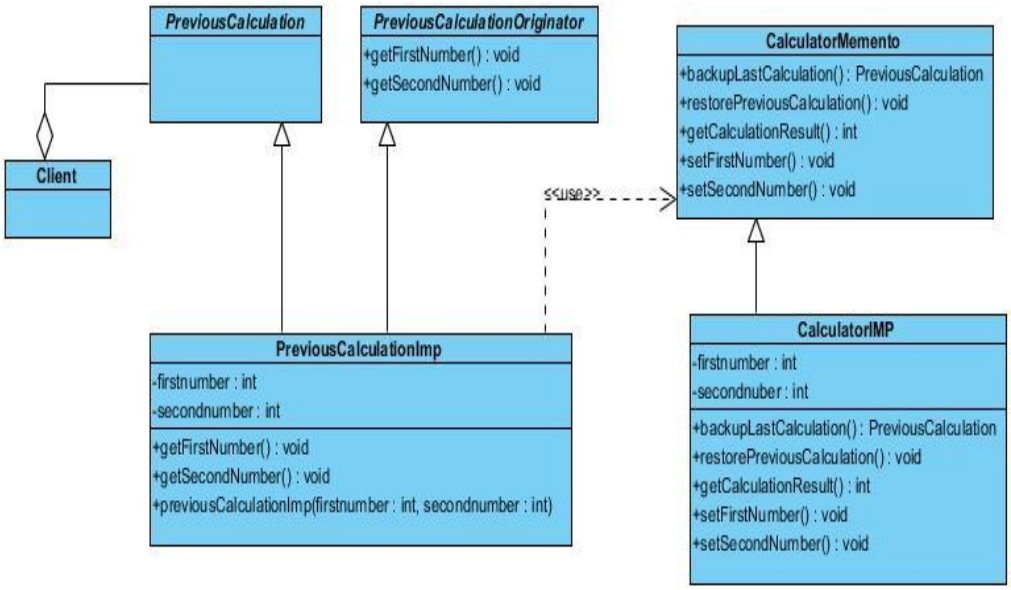
Το αναμνηστικό σχέδιο είναι ένα σχέδιο του σχεδιασμού του λογισμικού που παρέχει τη δυνατότητα να επαναφέρετε ένα αντικείμενο στην προηγούμενη κατάσταση (αναίρεση μέσω επαναφοράς).

Το αναμνηστικό σχέδιο υλοποιείται με δύο αντικείμενα: ο *εντολέας και* φροντιστής. Ο δημιουργός τους είναι κάποιο αντικείμενο που έχει μια εσωτερική κατάσταση. Ο επιστάτης πρόκειται να κάνει κάτι με το αρχικό, αλλά θέλει να είναι σε θέση να αναιρέσετε την αλλαγή. Ο επιστάτης πρώτα ζητεί από το συντάκτη για ένα αναμνηστικό αντικείμενο. Στη συνέχεια, κάνει ό, τι λειτουργία (ή η ακολουθία των εργασιών), επρόκειτο να κάνει. Για να επανέλθει στην κατάσταση πριν από τις εργασίες, επιστρέφει το αναμνηστικό αντικείμενο στον εντολέα. Το ίδιο το ενθύμιο αντικείμενο είναι ένα αδιαφανές αντικείμενο (μια που ο επιστάτης δεν μπορεί, ή δεν πρέπει, να αλλάξει). Όταν χρησιμοποιείτε αυτό το πρότυπο, πρέπει να ληφθεί μέριμνα, εάν ο δημιουργός μπορεί να αλλάξει ή άλλα αντικείμενα μέσα - η ενθύμιο πρότυπο λειτουργεί σε ένα μόνο αντικείμενο.

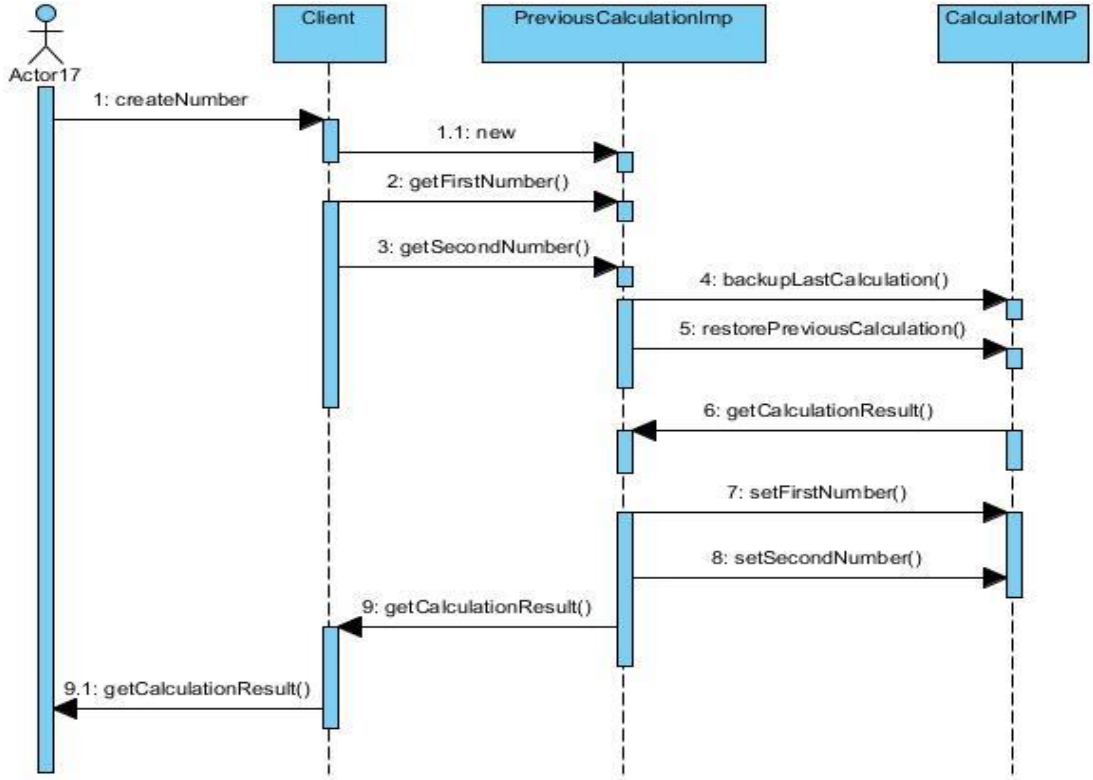
Πρόβλημα:

Το Memento δείχνει την εσωτερική κατάσταση ενός αντικειμένου ώστε το αντικείμενο να μπορεί αργότερα να αποκατασταθεί στην αρχική του κατάσταση. Ας υποθέσουμε ότι έχουμε μια αριθμομηχανή και θέλουμε να κάνουμε κάποιες πράξεις. Το memento μας βοηθάει στο να κρατήσει ότι ενέργιες κάνουμε έτσι ώστε αν θέλουμε να ξαναγυρίσουμε σε μια προηγούμενη κατάσταση να μπορεί να γίνει.

Class diagram



Sequence diagram



JAVA CODE

```
public class CalculatorImp extends CalculatorMemento {

   private int firstNumber;
   private int secondNumber;

        public PreviousCalculation backupLastCalculation() {

                // create a memento object used for restoring two numbers
                return new PreviousCalculationIMP(firstNumber,secondNumber);
        }


        public int getCalculationResult() {

                // result is adding two numbers
                return firstNumber + secondNumber;
        }


        public void restorePreviousCalculation(PreviousCalculation memento) {

                this.firstNumber                                                =
((PreviousCalculationOriginator)memento).getFirstNumber();
                this.secondNumber                                               =
((PreviousCalculationOriginator)memento).getSecondNumber();
        }


        public void setFirstNumber(int firstNumber) {

                this.firstNumber =  firstNumber;
        }


        public void setSecondNumber(int secondNumber) {

                this.secondNumber = secondNumber;
        }

}

abstract public class CalculatorMemento {
// Create Memento
        abstract public PreviousCalculation backupLastCalculation();
```

```
        // setMemento
        abstract public void restorePreviousCalculation(PreviousCalculation
memento);

        // Actual Services Provided by the originator
        abstract public int getCalculationResult();
        abstract public void setFirstNumber(int firstNumber);
        abstract public void setSecondNumber(int secondNumber);

}

public interface PreviousCalculation {

}

public interface PreviousCalculationOriginator {

  abstract public int getFirstNumber();
  abstract public int getSecondNumber();

}

public class PreviousCalculationIMP implements
PreviousCalculation,PreviousCalculationOriginator {

      private int firstNumber;
        private int secondNumber;

        public PreviousCalculationIMP(int firstNumber, int secondNumber) {

                this.firstNumber =  firstNumber;
                this.secondNumber = secondNumber;
        }


        public int getFirstNumber() {

                return firstNumber;
        }


        public int getSecondNumber() {

                return secondNumber;
        }
}


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

```
        CalculatorMemento calculator = new CalculatorImp();


        calculator.setFirstNumber(10);
        calculator.setSecondNumber(100);


        System.out.println(calculator.getCalculationResult());

        // Store result of this calculation in case of error
        PreviousCalculation memento = calculator.backupLastCalculation();

        // user enters a number
        calculator.setFirstNumber(17);

        // user enters a wrong second number and calculates result
        calculator.setSecondNumber(-290);

        // calculate result
        System.out.println(calculator.getCalculationResult());

        // user hits CTRL + Z to undo last operation and see last result
        calculator.restorePreviousCalculation(memento);

        // result restored
        System.out.println(calculator.getCalculationResult());
    }

}
```

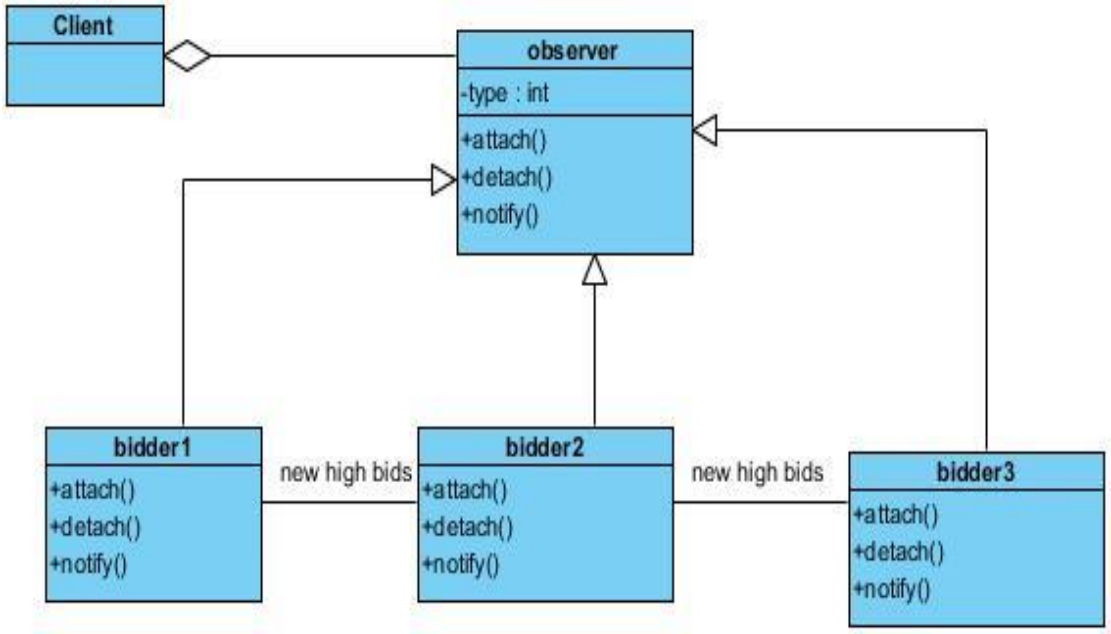
Observer

Το πρότυπο του παρατηρητή (aka. Εξαρτώμενα, publish / subscribe) είναι ένα πρότυπο σχεδίασης λογισμικού στην οποία ένα αντικείμενο , διατηρεί μια λίστα των εξαρτώμενων μελών της, που ονομάζεται παρατηρητές, και ειδοποιεί αυτόματα για τυχόν αλλαγές της κατάστασης, καλώντας συνήθως μία από τις τους μεθόδους . Χρησιμοποιείται κυρίως για την εφαρμογή που διανέμεται για εκδήλωση χειρισμού των συστημάτων. Παρατηρητής είναι επίσης ένας σημαντικός ρόλος στην οικεία MVC αρχιτεκτονικό πρότυπο. Στην πραγματικότητα, το πρότυπο του παρατηρητή εφαρμόστηκε για πρώτη φορά στο MVC πλαίσιο που βασίζεται σε διεπαφή χρήστη Smalltalk του.

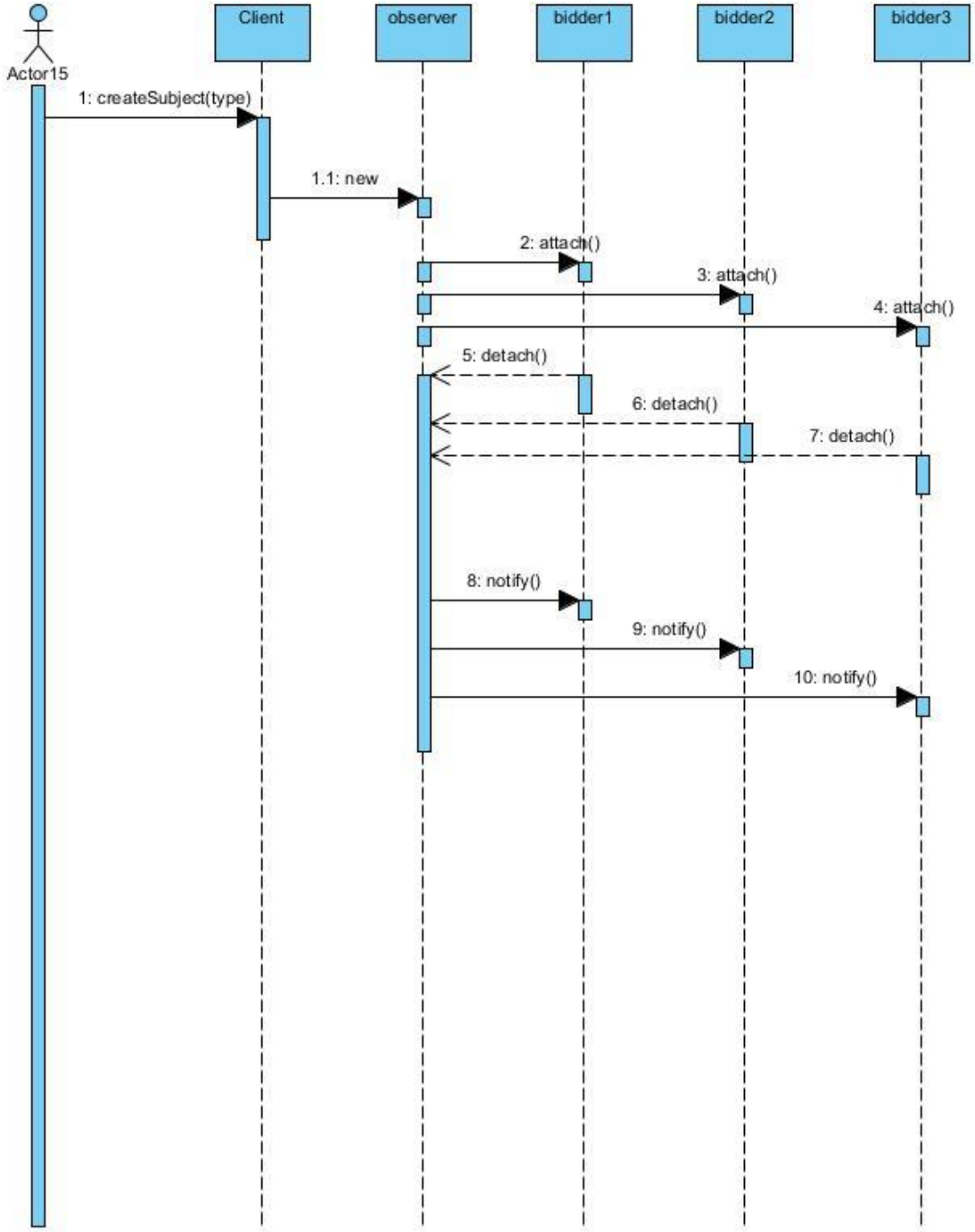
Πρόβλημα:

Ο παρατηρητής ορίζει ένα-προς-πολλά, έτσι ώστε, όταν ένα αντικείμενο αλλάζει κατάσταση, οι άλλοι κοινοποιούνται και να ενημερώνονται αυτόματα. Μερικές δημοπρασίες αποδεικνύουν αυτό το πρότυπο. Κάθε υποψήφιος έχει μια αριθμημένη πινακίδα που χρησιμοποιείται για να υποδείξει μια προσφορά. Ο διοργανωτής της δημοπρασίας ξεκινά την προσφορά, και «παρατηρεί» όταν μια πινακίδα αυξάνεται θα αποδεχθούν την προσφορά. Η αποδοχή της προσφοράς αλλάζει την τιμή προσφοράς η οποία μεταδίδεται σε όλους τους υποψηφίους με τη μορφή μιας νέας προσφοράς.

Class diagram



Sequence diagram



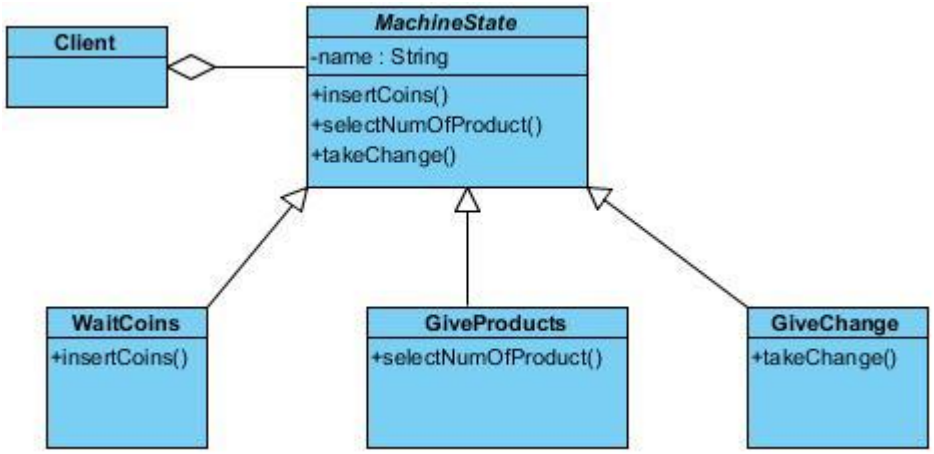
State

Το πρότυπο state, που μοιάζει πολύ με Πρότυπο στρατηγική , είναι μια συμπεριφορά πρότυπο σχεδιασμό του λογισμικού , γνωστό και ως τα αντικείμενα για τα κράτη πρότυπο. Αυτό το πρότυπο χρησιμοποιείται σε προγραμματισμό ηλεκτρονικών υπολογιστών για να αντιπροσωπεύουν την κατάσταση ενός αντικειμένου . Αυτός είναι ένας καθαρός τρόπος για ένα αντικείμενο να αλλάξει μερικώς τον τύπο του κατά την εκτέλεση.

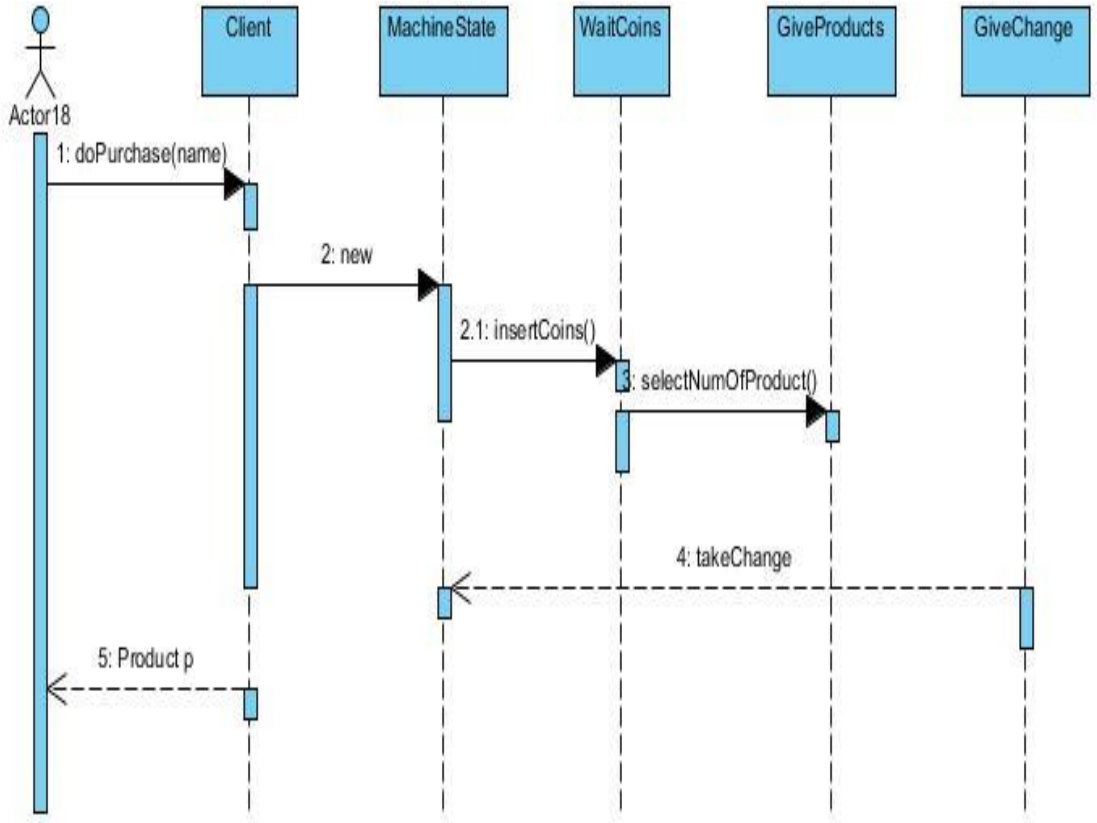
Πρόβλημα:

Το πρότυπο state επιτρέπει ένα αντικείμενο να αλλάξει τη συμπεριφορά του όταν υπάρχουν εσωτερικές αλλαγές της κατάστασης. Έχουμε ένα μηχάνημα αυτόματης πώλησης και τις υποκλάσεις της που καθορίζει τη λειτουργία του μηχανήματος.Ένα τέτοιο μηχάνημα λειτουργεί ως εξής:βάζουμε τα κέρματα, πατάμε του νούμερο που αντιστοιχεί στο προϊόν που θέλουμε και αφού μας παραδοθεί περιμένουμε τα ρέστα εάν αυτά υπάρχουν.

Class diagram



Sequence diagram



JAVA CODE

```
abstract public class MachineState implements StateTransittion {

   abstract public void insertCoins(MachineState state) ;

   abstract  public void selectNumOfProducts(MachineState state);

   abstract  public void takeChange(MachineState state);
}

public class GiveProducts extends MachineState {

      public void selectNumOfProducts(MachineState state) {
        System.out.println("take the product that you wanted");
```

```
        state.changeStateTo(MachineState.Product);
    }

    @Override
    public void insertCoins(MachineState state) {
        throw new UnsupportedOperationException("Not supported yet.");
    }

    @Override
    public void takeChange(MachineState state) {
        throw new UnsupportedOperationException("Not supported yet.");
    }

    public void changeStateTo(MachineState state) {
        throw new UnsupportedOperationException("Not supported yet.");
    }

}

public class WaitCoins extends MachineState {


    public void insertCoins(MachineState state) {
        System.out.println("the machine wait to insert the coins");
        state.changeStateTo(MachineState.Insert);
    }

    @Override
    public void selectNumOfProducts(MachineState state) {
        throw new UnsupportedOperationException("Not supported yet.");
    }

    @Override
    public void takeChange(MachineState state) {
        throw new UnsupportedOperationException("Not supported yet.");
    }

    public void changeStateTo(MachineState state) {
        throw new UnsupportedOperationException("Not supported yet.");
    }

}

public class GiveChange extends MachineState {


    public void takeChange(MachineState state){
        System.out.println("take your change");
        state.changeStateTo(MachineState.Change);
    }
```

```
    @Override
    public void insertCoins(MachineState state) {
        throw new UnsupportedOperationException("Not supported yet.");
    }

    @Override
    public void selectNumOfProducts(MachineState state) {
        throw new UnsupportedOperationException("Not supported yet.");
    }

    public void changeStateTo(MachineState state) {
        throw new UnsupportedOperationException("Not supported yet.");
    }

}

public interface StateTransittion {

    public static final MachineState Insert = new WaitCoins();
    public static final MachineState Product = new GiveProducts();
    public static final MachineState Change = new GiveChange();

    void changeStateTo(MachineState state);
}

public class Client {

   public static void main(String[] args) {
       WaitCoins w=new WaitCoins();
       GiveProducts g=new GiveProducts();
       GiveChange c=new GiveChange();
      MachineState ms=new MachineState() {

           @Override
           public void insertCoins(MachineState state) {     }

           @Override
           public void selectNumOfProducts(MachineState state) {}

           @Override
           public void takeChange(MachineState state) {}

           public void changeStateTo(MachineState state) { }
       };
           w.insertCoins(ms);
         System.out.println(w);
          g.selectNumOfProducts(ms);
          System.out.println(g);
          c.takeChange(ms);
```

```
        System.out.println(c);

    } }
```

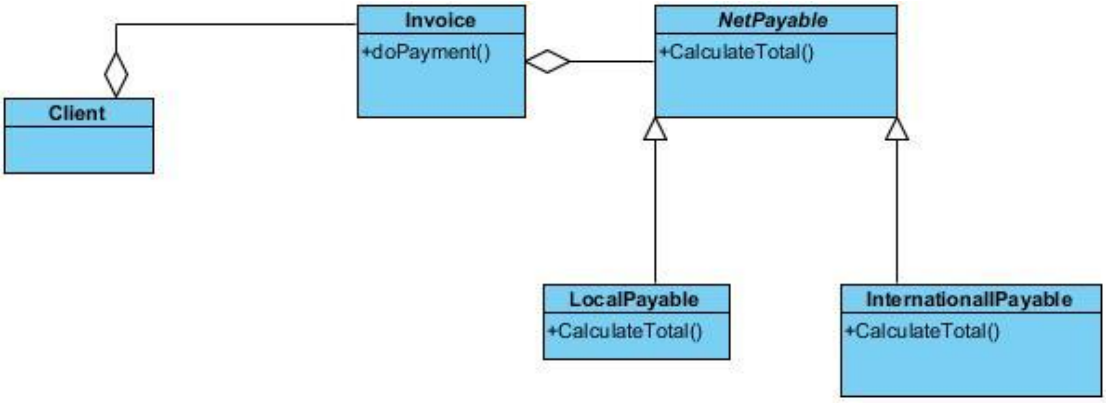
Strategy

Σε προγραμματισμό ηλεκτρονικών υπολογιστών , το σχέδιο στρατηγικής (επίσης γνωστό ως πρότυπο πολιτικής) είναι ένα συγκεκριμένο πρότυπο σχεδιασμού λογισμικού , σύμφωνα με την οποία αλγόριθμοι μπορούν να επιλεγούν κατά το χρόνο εκτέλεσης. Τυπικά, το πρότυπο στρατηγική καθορίζει μια οικογένεια αλγορίθμων , συμπυκνώνει το καθένα, και τους καθιστά εναλλάξιμα. Στρατηγική επιτρέπει ο αλγόριθμος να ποικίλλουν ανεξάρτητα από τους πελάτες που το χρησιμοποιούν. Για παράδειγμα, μια κατηγορία που εκτελεί την επικύρωση στα εισερχόμενα δεδομένα μπορούν να χρησιμοποιήσουν ένα σχέδιο στρατηγικής για να επιλέξετε έναν αλγόριθμο πιστοποίησης με βάση το είδος των δεδομένων, η πηγή των δεδομένων, επιλογή του χρήστη, και / ή άλλους συντελεστές διακριτικής μεταχείρισης. Αυτοί οι παράγοντες δεν είναι γνωστοί για κάθε περίπτωση μέχρι το χρόνο εκτέλεσης, και μπορεί να απαιτούν ριζικά διαφορετική επικύρωση που πρέπει να εκτελεστούν. Οι στρατηγικές επικύρωσης, έγκλειστα χωριστά από την επικύρωση αντικείμενο, μπορεί να χρησιμοποιηθεί από άλλα αντικείμενα επικύρωση σε διάφορες περιοχές του συστήματος (ή ακόμα και διαφορετικά συστήματα) χωρίς κωδικό επικάλυψη. Η απαραίτητη προϋπόθεση για την γλώσσα προγραμματισμού είναι η δυνατότητα να αποθηκεύσετε μια αναφορά σε κάποιο κωδικό σε μια δομή δεδομένων και να το επανακτήσει. Αυτό μπορεί να επιτευχθεί με μηχανισμούς όπως ο δείκτης λειτουργίας , την πρώτη θέση λειτουργίας , τάξεις ή την κατηγορία σε περιπτώσεις αντικειμενοστραφούς προγραμματισμού γλώσσες, ή την πρόσβαση σε εσωτερική μνήμη της γλώσσας εφαρμογή του κώδικα μέσω ανάκλασης .

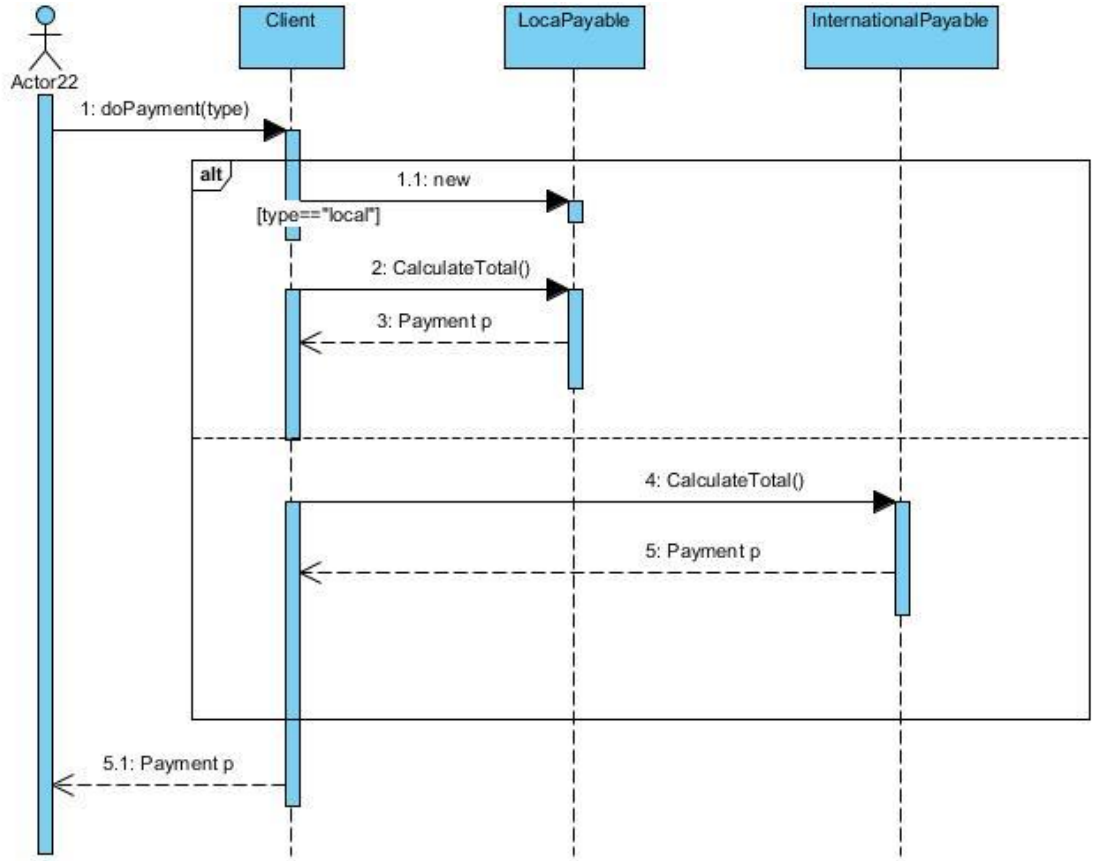
Πρόβλημα:

Η στρατηγική ορίζει μια σειρά από αλγόριθμους που μπορούν να χρησιμοποιηθούν εναλλακτικά. Έχουμε ένα σύστημα πληρωμών, όπου έχουμε μια τάξη Τιμολόγιο όπου υπολογίζει το σύνολο του ποσού που καταβάλλεται, αλλά η προϋπόθεση είναι ο υπολογισμός του ποσού εξαρτάται από το είδος των πελατών και την προσφορά έκπτωσης.Επίσης παίζει ρόλο αν η συναλλαγή πραγματοποιείται σε τοπικό επίπεδο ή διεθνές Και κατά τη στιγμή που οι προσφορές / εκπτώσεις μπορεί να διαφέρουν και μπορεί να χρειαστεί να αλλάξει την προσφορά κατά το χρόνο εκτέλεσης.

Class diagram



Sequence diagram



JAVA CODE

```
public class Invoice {

    NetPayable netpay;

    public Invoice(){    }


public void setNetPay(NetPayable netpay)
{
    this.netpay = netpay;
}
public NetPayable getNetPay()
{
    return netpay;
}
public String doPayment(){
String s=new String();
LocalPayable local=new LocalPayable();
if(local.equals(s))
{
    System.out.println("Payment is local");
}
else{
    System.out.println("Payment is international");
    }
return s;

}
}

abstract public class NetPayable {

    abstract public int calculateTotal();

}

public class InternationalPayable extends NetPayable {

    public int calculateTotal()
    {
        Invoice i=new Invoice();
        System.out.println(i.doPayment());
        return 0;
    }
```

```
}

public class LocalPayable extends NetPayable {

   public int calculateTotal()
   {
      Invoice i=new Invoice();
      System.out.println(i.doPayment());
      return 0;
   }

}




public class Client {

   public static void main(String args[])
         {

      LocalPayable l=new LocalPayable();
      System.out.print(l.calculateTotal());
      InternationalPayable ip=new InternationalPayable();
      System.out.print(ip.calculateTotal());
   }

}
```

Template Method

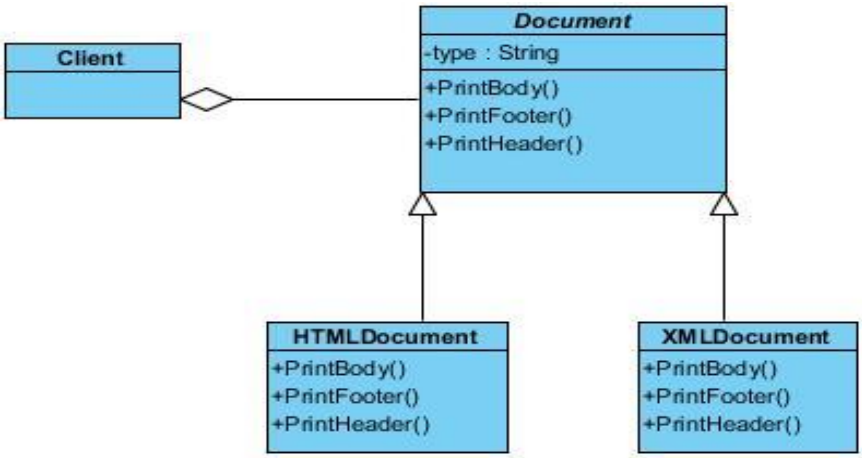
Μια *μέθοδος πρότυπο* καθορίζει το σκελετό προγράμματος ενός αλγορίθμου . Ένα ή περισσότερα από τα βήματα αλγόριθμος μπορεί να παρακαμφθεί από υποκλάσεις που να επιτρέπουν τη διαφορετική συμπεριφορά διασφαλίζοντας παράλληλα ότι ο πρωταρχικός αλγόριθμος ακολουθείται ακόμα. Στην αντικειμενοστρεφή προγραμματισμό, πρώτα μια τάξη που δημιουργείται παρέχει τα βασικά βήματα ενός σχεδιασμού αλγορίθμων . Αυτά τα μέτρα θα εφαρμόζονται με αφηρημένες μεθόδους . Αργότερα, υποκατηγορίες θα αλλάξουν τις αφηρημένες μεθόδους για να εφαρμόσουν πραγματικές δράσεις. Έτσι, ο γενικός αλγόριθμος είναι αποθηκευμένο σε ένα μέρος, αλλά τα συγκεκριμένα μέτρα μπορεί να αλλάξουν από τις υποκατηγορίες. Η μέθοδος πρότυπο διαχειρίζεται έτσι την ευρύτερη εικόνα του έργου σημασιολογία , και πιο εκλεπτυσμένες λεπτομέρειες υλοποίησης της επιλογής και την ακολουθία των μεθόδων. Αυτό απαιτεί μεγαλύτερη εικόνα αφηρημένη και μη αφηρημένες μεθόδους για την εργασία στο χέρι. Οι μη-αφηρημένες μεθόδους ελέγχεται πλήρως από τη μέθοδο πρότυπο,

αλλά οι αφηρημένες μεθόδους, που εφαρμόζονται σε υποκατηγορίες, παρέχουν την εκφραστική δύναμη του προτύπου και του βαθμού της ελευθερίας. Μερικά ή όλα από τις αφηρημένες μεθόδους μπορεί να ειδικεύεται σε μια υποκατηγορία, που επιτρέπει ο συγγραφέας της υποκλάσης να παρέχει συγκεκριμένη συμπεριφορά, με ελάχιστες τροποποιήσεις στις μεγαλύτερες σημασιολογία. Η μέθοδος πρότυπο (που είναι μη-αφηρημένη) παραμένει αμετάβλητο σε αυτό το πρότυπο, εξασφαλίζοντας ότι οι δευτερεύουσες μη αφηρημένες μεθόδους και τις μεθόδους που ονομάζεται αφηρημένη στο προορίζεται αρχικά-ακολουθία. Η μέθοδος πρότυπο εμφανίζεται συχνά, τουλάχιστον στην απλούστερη περίπτωση, όπου μια μέθοδος απαιτεί μόνο μία αφηρημένη μέθοδο, με αντικειμενοστρεφείς γλώσσες. Εάν ένας συγγραφέας λογισμικό χρησιμοποιεί μια πολυμορφική μέθοδο, αυτό το πρότυπο σχέδιο μπορεί να είναι μάλλον φυσική συνέπεια. Αυτό συμβαίνει επειδή μια μέθοδος καλώντας μια αφηρημένη ή πολυμορφική λειτουργία είναι απλώς ο λόγος για την ύπαρξη της αφηρημένης ή πολυμορφικό μέθοδο. Η μέθοδος πρότυπο μπορεί να χρησιμοποιηθεί για να προσθέσει αμέσως παρούσα αξία του λογισμικού ή με ένα όραμα για βελτιώσεις στο μέλλον. Το σχέδιο προτύπου μέθοδος έχει άμεση σχέση με τη μη-εικονική διεπαφή (NVI) πρότυπο. Το πρότυπο NVI αναγνωρίζει τα οφέλη της μη αφηρημένο τη μέθοδο που επικαλείται τις αφηρημένες μεθόδους υποδεέστερη. Αυτό το επίπεδο της έμμεσης επιτρέπει την προ και μετά δραστηριότητες σε σχέση με τις αφηρημένες ενέργειες τόσο άμεσα και με τις μελλοντικές απρόβλεπτες αλλαγές. Το πρότυπο NVI μπορεί να αναπτυχθεί με πολύ μικρή παραγωγή λογισμικού και το κόστος εκτέλεσης. Πολλά εμπορικά πλαίσια λογισμικό χρησιμοποιεί το πρότυπο NVI. Μέθοδος προτύπου υλοποιεί την Προστατευόμενη παραλλαγές GRASP αρχή, όπως το πρότυπο προσαρμογέα κάνει. Η διαφορά είναι ότι ο προσαρμογέας δίνει το ίδιο interface για πολλές επιχειρήσεις, ενώ Μέθοδος προτύπου το κάνει μόνο για ένα.

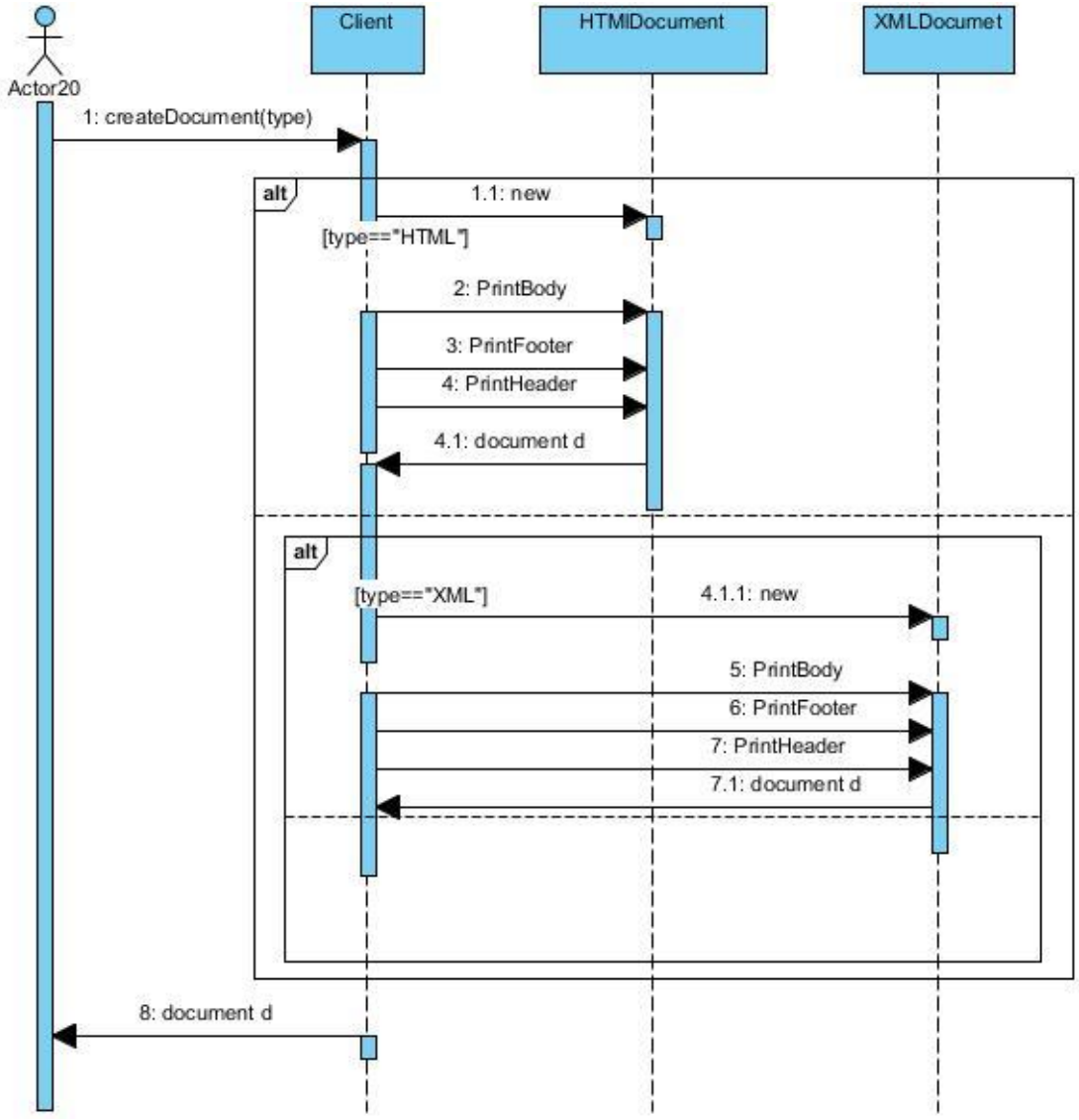
Πρόβλημα:

Η μέθοδος Πρότυπο ορίζει ένα σκελετό ενός αλγορίθμου σε μια επιχείρηση. Αλγόριθμος εκτύπωσης για διάφορους τύπους εγγράφων, όπου ο αλγόριθμος έχει καθοριστεί από την εκτύπωση της κεφαλίδας, τότε το σώμα και το υποσέλιδο που ορίζεται στην τάξη βάση το πώς ακριβώς τυπώνονται οι κεφαλίδες ορίζεται στις κατηγορίες που υποστηρίζονται έγγραφο.

Class diagram



Sequence diagram



JAVA CODE

```
abstract public class Document {

        public abstract void PrintBody();
        public abstract void PrintFooter();
        public abstract void PrintHeader();
        public void drawDocument() {
            PrintHeader();
            PrintBody();
            PrintFooter();
        }
}

public class HTMLDocument extends Document {

    public HTMLDocument(){}

    public void PrintBody() {
                System.out.println("<Body>");

        }
        public void PrintFooter() {
                System.out.println("<Foot>");
        }
        public void PrintHeader() {
                System.out.println("<Head>");
        }

}

public class XMLDocument extends Document {

    public XMLDocument(){}

    public void PrintBody() {
                System.out.println("<Body>");
        }
        public void PrintFooter() {
                System.out.println("<Foot>");
        }
        public void PrintHeader() {
                System.out.println("<?xml...>");
        }


}
```

```
public class Client {

   public static void main(String args[]) {
      HTMLDocument html=new HTMLDocument();
      html.drawDocument();
      XMLDocument xml=new XMLDocument();
      xml.drawDocument();

   }

}
```

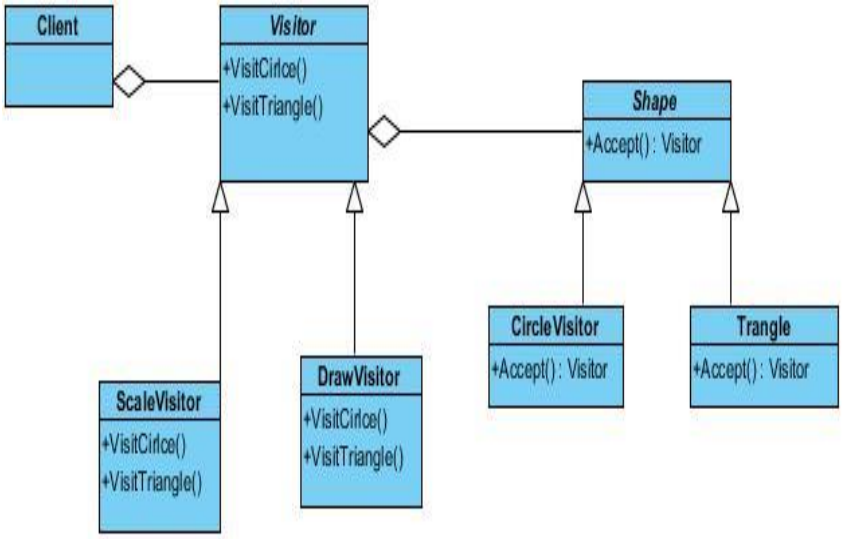
Visitor

Στην αντικειμενοστραφή προγραμματισμό και ανάπτυξη λογισμικού , ο επισκέπτης πρότυπο σχέδιο είναι ένας τρόπος διαχωρισμού ενός αλγορίθμου από ένα αντικείμενο δομή στην οποία δραστηριοποιείται. Ένα πρακτικό αποτέλεσμα αυτού του διαχωρισμού είναι η δυνατότητα να προσθέσετε νέες εργασίες στις υφιστάμενες δομές αντικείμενο χωρίς μετατροπή των εν λόγω δομών. Είναι ένας τρόπος για να ακολουθήσει εύκολα την ανοικτή / κλειστή αρχή . Στην ουσία, ο επισκέπτης επιτρέπει σε κάποιον να προσθέσει νέες λειτουργίες εικονικά σε μια οικογένεια των κατηγοριών χωρίς να τροποποιεί τις ίδιες τις τάξεις αντ 'αυτού, μία τάξη δημιουργεί ένα επισκέπτη που υλοποιεί όλες τις κατάλληλες ειδικότητες της εικονικής λειτουργίας. Ο επισκέπτης έχει την αναφορά παράδειγμα ως πρώτη ύλη, και υλοποιεί τον στόχο μέσω της διπλής αποστολής . Ενώ ισχυρό, το πρότυπο επισκέπτης είναι πιο περιορισμένο από ό, τι τα συμβατικά εικονική λειτουργίας . Δεν είναι δυνατόν να δημιουργήσει για τους επισκέπτες αντικείμενα, χωρίς την προσθήκη μιας μικρής μεθόδου επανάκλησης στο εσωτερικό κάθε κατηγορίας. Σε αφελείς εφαρμογές, η μέθοδος επανάκλησης σε κάθε μία από τις κατηγορίες δεν είναι κληρονομικό.

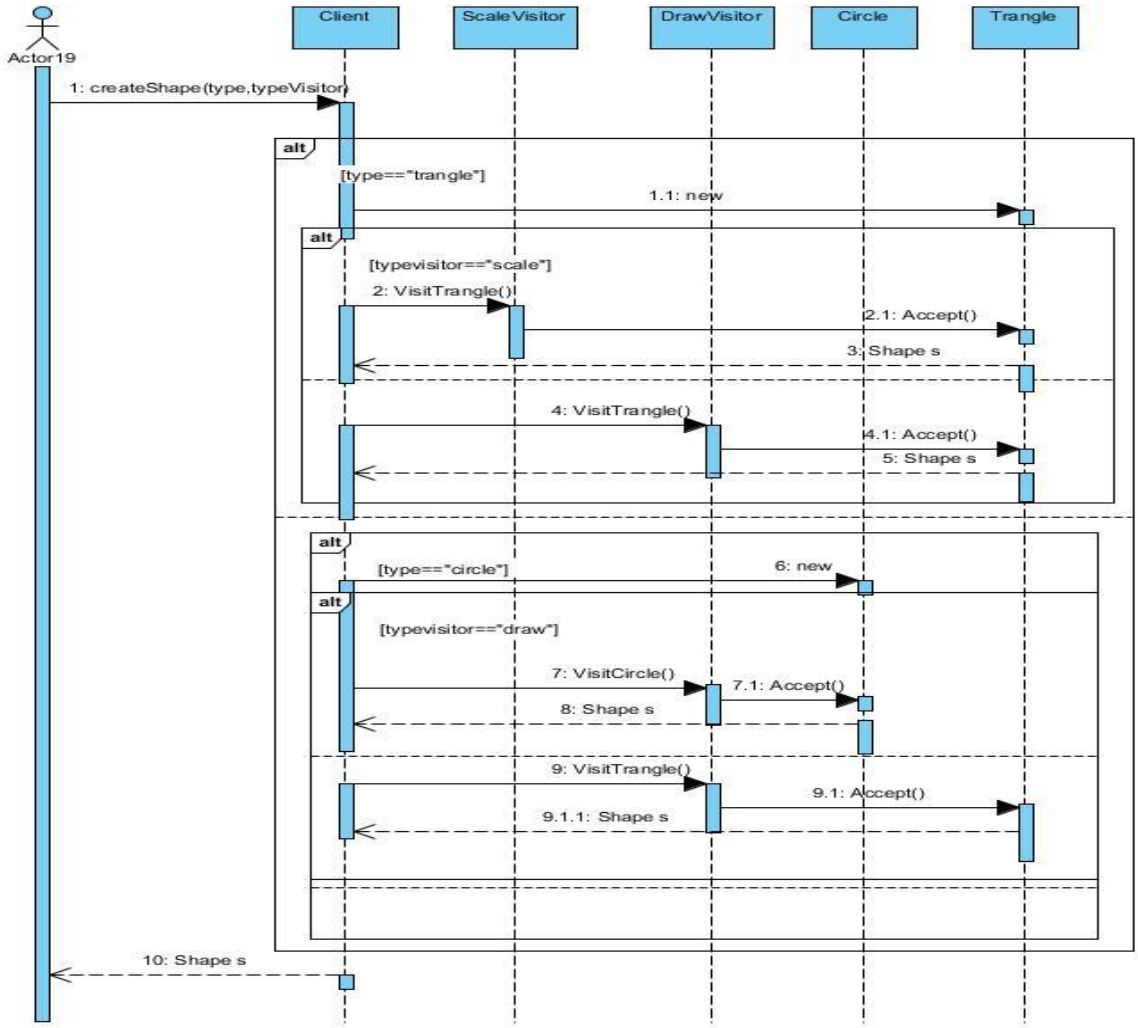
Πρόβλημα:

Το πρότυπο Visitor αποτελεί πράξη που πρόκειται να πραγματοποιηθεί στα στοιχεία μιας δομής αντικειμένου χωρίς να αλλάξετε τις κατηγορίες στις οποίες δραστηριοποιείται. Έστω ότι θέλουμε να σχεδιάσουμε ένα σχήμα είτε κύκλο,είτε τρίγωνο.Εάν θέλουμε να προσθέσουμε επιπλέον μεθόδους για το κάθε σχήμα χωρίς όμως να αλλάξουμε μόνιμα τα ίδια τα σχήματα τότε είναι χρήσιμο το visitor.

Class diagram



Sequence diagram



JAVA CODE

```
abstract public class Shape {

   abstract void accept(Visitor visitor);

}

abstract public class Visitor {

 abstract void visitCircle(Circle circle);
 abstract  void visitTrangle(Trangle trangle);

}

public class DrawVisitor extends Visitor {

    public void visitCircle(Circle circle) {
        System.out.println("DrawVisiting " + circle.getName());
    }

    public void visitTrangle(Trangle trangle) {
        System.out.println("DrawVisiting "+trangle.getName());
    }

}

class ScaleVisitor extends Visitor {

     public void visitCircle(Circle circle) {
        System.out.println("ScaleVisiting " + circle.getName());
    }

    public void visitTrangle(Trangle trangle) {
        System.out.println("ScaleVisiting "+trangle.getName());
    }


}


class Circle extends Shape {

    public String name="circle";

     Circle(){};

    public Circle(String name) {
        this.name = name;
```

```
    }

    public String getName() {
        return this.name;
    }

    public void accept(Visitor visitor) {

    visitor.visitCircle(this);


}
}

class Trangle extends Shape {

     public String name="trangle";

    public Trangle(String name) {
        this.name = name;
    }

    Trangle() {}

     public String getName() {
        return this.name;
    }


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


public class Client {
       static public void main(String[] args) {
         Circle c = new Circle();
         Trangle t=new Trangle();
         ScaleVisitor sc=new ScaleVisitor();
         c.accept(sc);
         t.accept(sc);
         DrawVisitor dv=new DrawVisitor();
         c.accept(dv);
         t.accept(dv);


}
```

ΣΥΜΠΕΡΑΣΜΑΤΑ

Ολοκληρώνοντας αυτή την εργασία, έγινε σαφές ότι υπήρξαν διαφορές ανάμεσα στα πρότυπα σχεδίασης.Είναι εμφανές ότι υπάρχουν πρότυπα που χρησιμοποιούνται, με μεγάλη διαφορά, συχνότερα από τα υπόλοιπα. Τα πρότυπα αυτά, αν και χρησιμοποιούνται σε διαφορετικό βαθμό από κάθε κατηγορία, είναι και για τις δυο περιπτώσεις κοινά. Πρόκειται, κυρίως, για τα πρότυπα State – Strategy, (Object) Adapter – Command και Singleton. Παράλληλα, υπάρχουν κι άλλα που η παρουσία τους στον κώδικα των εφαρμογών που μελετήθηκαν δεν ξεπερνούσε τα όρια του στατιστικού λάθους. Επίσης διαπιστώθηκε ότι μπορούμε να χρησιμοποιήσουμε διαφορετικά πρότυπα σε ένα ίδιο πρόβλημα, φυσικά η λύση θα είναι αντίστοιχη του κάθε προτύπου.

ΒΙΒΛΙΟΓΡΑΦΙΑ

Elisabeth Freeman, Eric Freeman,Bert Bates,Kathy Sierra,Elizabeth Robson, "Head First Design Patterns",2004

Erich Gamma,Richard Helm,Ralph Johnson,John Vlissides,
"Design Patterns CD","Elements of Reusable Object-Oriented Software,
1994

John Vlissides,"Pattern Hatching: Design Patterns Applied",1998

Αλέξανδρος Ν. Χατζηγεωργίου, "Αντικειμενοστραφής σχεδίαση"

http://www.oodesign.com/
http://en.wikipedia.org/wiki/Abstract_Factory_pattern
http://en.wikipedia.org/wiki/Singleton_pattern
http://en.wikipedia.org/wiki/Bridge_pattern
http://en.wikipedia.org/wiki/Factory_Method_pattern
http://en.wikipedia.org/wiki/Builder_pattern
http://en.wikipedia.org/wiki/Prototype_pattern
http://en.wikipedia.org/wiki/Proxy_pattern
http://en.wikipedia.org/wiki/Mediator_pattern
http://en.wikipedia.org/wiki/Memento_pattern
http://en.wikipedia.org/wiki/Chain_of_responsibility_pattern
http://en.wikipedia.org/wiki/Observer_pattern
http://en.wikipedia.org/wiki/Visitor_pattern
http://en.wikipedia.org/wiki/Command_pattern
http://en.wikipedia.org/wiki/Composite_pattern
http://en.wikipedia.org/wiki/Adapter_pattern
http://en.wikipedia.org/wiki/Decorator_pattern
http://en.wikipedia.org/wiki/Facade_pattern
http://en.wikipedia.org/wiki/Flyweight_pattern
http://en.wikipedia.org/wiki/Interpreter_pattern
http://en.wikipedia.org/wiki/Iterator_pattern
http://en.wikipedia.org/wiki/State_pattern
http://en.wikipedia.org/wiki/Strategy_pattern
http://en.wikipedia.org/wiki/Template_Method_pattern