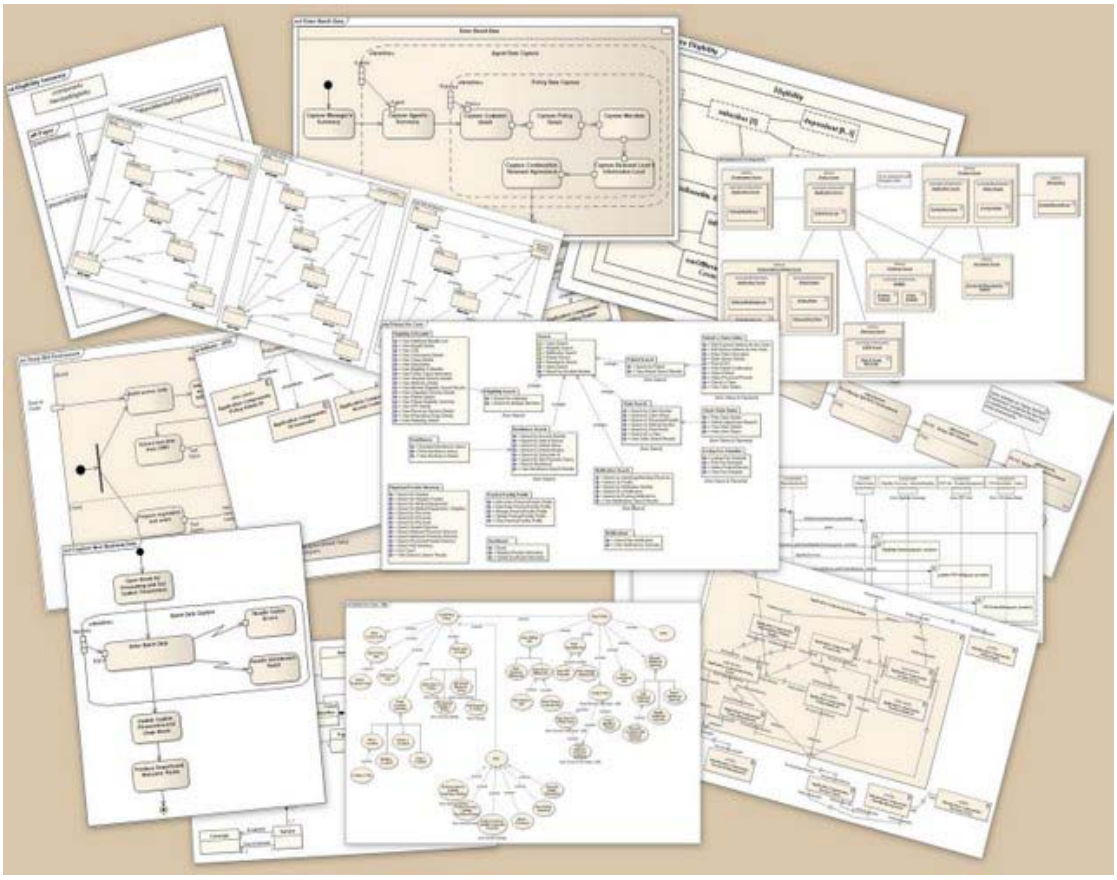
ΑΛΕΞΑΝΔΡΕΙΟ Τ.Ε.Ι. ΘΕΣΣΑΛΟΝΙΚΗΣ

ΣΧΟΛΗ ΤΕΧΝΟΛΟΓΙΚΩΝ ΕΦΑΡΜΟΓΩΝ

ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ

ΠΤΥΧΙΑΚΗ ΕΡΓΑΣΙΑ



Αποτύπωση κώδικα αντικειμενοστρεφούς γλώσσας

σε διαγράμματα UML

Χαραλάμπους Νεκτάριος

Επιβλέπουσα καθηγήτρια: Κουμπή Βασιλάντα

ΕΥΡΕΤΗΡΙΟ ΠΕΡΙΕΧΟΜΕΝΩΝ

ΕΥΡΕΤΗΡΙΟ ΣΧΗΜΑΤΩΝ

ΕΥΡΕΤΗΡΙΟ ΕΙΚΟΝΩΝ

ΕΥΡΕΤΗΡΙΟ ΠΙΝΑΚΩΝ

ΠΡΟΛΟΓΟΣ

Τις τελευταίες δυο δεκαετίες σημειώθηκε μεγάλη εξέλιξη στην τεχνολογία των υπολογιστών γεγονός που έδωσε την δυνατότητα στους προγραμματιστές να αναπτύσσουν συνεχώς μεγαλύτερα και πολυπλοκότερα προγράμματα με σκοπό να καλύψουν τις ανάγκες του επιστημονικού, επιχειρηματικού και εμπορικού κόσμου σε λογισμικό. Αυτή η αύξηση του μεγέθους και της πολυπλοκότητας των εφαρμογών οδήγησε από την σύνταξη απλών προγραμμάτων στην σχεδίαση και ανάπτυξη συστημάτων λογισμικού τα οποία αποτελούνται από μικρότερα υποσυστήματα που επικοινωνούν μεταξύ τους αλλά ταυτόχρονα είναι ανεξάρτητα, όσο γίνεται, το ένα από το άλλο.

Μέσα από την υλοποίηση και κυρίως την συντήρηση των συστημάτων λογισμικού, έγινε εμφανής η ανάγκη να δημιουργηθεί ένας τρόπος σχεδίασης και μοντελοποίησης των συστημάτων λογισμικού μέσω του οποίου θα εμφανίζεται η δομή και οι λειτουργίες του συστήματος. Η γλώσσα που χρησιμοποιείται για την μοντελοποίηση των λογισμικών συστημάτων είναι η UML *(Unified Modeling Language)*. Ένα μοντέλο αποτελεί μία απλούστευση της πραγματικότητας. Φτιάχνουμε μοντέλα για να καταλάβουμε καλύτερα το σύστημα που αναπτύσσουμε καθώς προσφέρει γραφική απεικόνιση, προσδιορίζει δομή και συμπεριφορά και βοηθά στην οργάνωση της κατασκευής. Δεν υπάρχει ένα μοναδικό μοντέλο που να καλύπτει όλες τις πλευρές του συστήματος. Διαφορετικοί τύποι μοντέλων προσεγγίζουν διαφορετικές πλευρές.

Στα πλαίσια της πτυχιακής παρουσιάζονται και αναλύονται τα βασικά UML διαγράμματα, η διαδικασία που ακολουθείται για την μετατροπή από κώδικα σε UML διαγράμματα και δίνεται ένα παράδειγμα μιας τέτοιας μετατροπής.

Ο κώδικας από τον οποίο θα προκύψουν τα διαγράμματα UML θα είναι γραμμένος σε γλώσσα C++ και Java.

Η πτυχιακή αποτελείται από πέντε κεφάλαια τα οποία παρουσιάζονται συνοπτικά παρακάτω:

Το πρώτο κεφάλαιο είναι εισαγωγικό γίνεται ανάλυση της **γλώσσας μοντελοποίησης**, αναφέρονται κάποια ιστορικά στοιχεία και γίνεται αναφορά και ανάλυση των διαγραμμάτων μοντελοποίησης. Επίσης, υπάρχει και μια εκτεταμένη αναφορά για τον αντικειμενοστρεφή προγραμματισμό.

Στο δεύτερο κεφάλαιο αναφορά το εργαλείο μοντελοποίησης **Visual Paradigm** και δίνονται οδηγίες για την χρήση του εργαλείου ως προς την δημιουργία διαγραμμάτων.

Στο τρίτο κεφάλαιο παρουσιάζονται τα εργαλεία **Umbrello UML Modeller** και **Visual Paradigm for UML** και γίνεται σύγκριση μεταξύ τους. Στην αρχή της εργασίας η μοντελοποίηση κώδικα ξεκίνησε με το Umbrello UML Modeller και στην συνέχεια συνεχίστηκε με το Visual Paradigm for UML.

Το τέταρτο κεφάλαιο είναι ένα από τα βασικά κεφάλαια της πτυχιακής εργασίας όπου δίνονται τα βήματα πως από κώδικα C++ δημιουργούνται τα διαγράμματα UML. Ακόμη, σε αυτό το κεφάλαιο παρουσιάζεται και ένα παράδειγμα «Σύστημα διαχείρισης κινηματογράφου».

Στο πέμπτο κεφάλαιο δίνεται ένα δεύτερο παράδειγμα «Διαχείρισης Τραπεζικών Λογαριασμών», για την δημιουργία διαγραμμάτων από κώδικα.

Τέλος, παρουσιάζονται τα συμπεράσματα που προέκυψαν κατά την διάρκεια εκπόνησης της πτυχιακής εργασίας και εξηγείται η σχέση της γλώσσας μοντελοποίησης με τον αντικειμενοστρεφή προγραμματισμό και η ανάγκη μετατροπής του κώδικα σε μια *ενοποιημένη διεργασία η οποία διασφαλίζει μια πειθαρχημένη ανάθεση καθηκόντων και αρμοδιοτήτων μέσα σε ένα οργανισμό ανάπτυξης.*

ΚΕΦΑΛΑΙΟ 1: ΑΝΤΙΚΕΙΜΕΝΟΣΤΡΕΦΗΣ ΠΡΟΓΡΑΜΜΑΤΙΣΜΟΣ ΚΑΙ UML

1.1. Εισαγωγή στον αντικειμενοστρεφή προγραμματισμό

Αντικειμενοστρεφής προγραμματισμός *(Object-oriented Programming)* ονομάστηκε το προγραμματιστικό υπόδειγμα, που εμφανίστηκε στα τέλη της δεκαετίας του 1960 και καθιερώθηκε κατά τη δεκαετία του 1990, αντικαθιστώντας σε μεγάλο βαθμό το παραδοσιακό υπόδειγμα του δομημένου προγραμματισμού. Ο αντικειμενοστρεφής προγραμματισμός επεκτάθηκε την τελευταία δεκαετία σε δημοφιλή τεχνολογία λογισμικού. Δεδομένου ότι το υλικό και το λογισμικό έγιναν όλο και περισσότερο σύνθετα, οι ερευνητές μελέτησαν τους τρόπους στους οποίους η ποιότητα λογισμικού θα μπορούσε να διατηρηθεί. Ο αντικειμενοστρεφής προγραμματισμός επεκτάθηκε εν μέρει ως προσπάθεια να εξεταστεί αυτό το πρόβλημα με έντονα να υπογραμμίσει τις ιδιαίτερες μονάδες του προγραμματισμού της λογικής και της ικανότητας επαναχρησιμοποίησης του λογισμικού.

Η γλώσσα προγραμματισμού *Simula 67* ήταν η πρώτη η οποία είχε εισαγάγει έννοιες, που κρύβονται πίσω από τον αντικειμενοστρεφή προγραμματισμό (αντικείμενα, κατηγορίες, υποκατηγορίες, εικονικούς μεθόδους, συλλογές απορριμμάτων, ιδιαίτερη προσομοίωση γεγονότος κ.α.) στον πραγματικό κόσμο.

Οι αντικειμενοστρεφείς γλώσσες προγραμματισμού (Java, Eiffel, Smalltalk, C++) δίνουν έμφαση στα δεδομένα παρά στον κώδικα. Το πρόγραμμα αναπτύσσεται γύρω από τα δεδομένα τα οποία ορίζουν από μόνα τους τον τρόπο με τον οποίο μπορούμε να τα διαχειριστούμε.

Ένα απλό παράδειγμα που μπορούμε να χρησιμοποιήσουμε για την κατανόηση της φιλοσοφίας του αντικειμενοστρεφούς προγραμματισμού είναι το αυτοκίνητο. Κάθε αυτοκίνητο είναι ένα αντικείμενο που ανήκει σε μια κλάση που ορίζει τα βασικά χαρακτηριστικά του αυτοκινήτου. Αυτά μπορεί να διαφέρει ανάμεσα στους κατασκευαστές, αλλά όλα θα παρέχουν τα βασικά χαρακτηριστικά που ορίζει η κλάση "αυτοκίνητο" για τη χρήση του. Δηλαδή τιμόνι, γκάζι, φρένο, συμπλέκτης και ταχύτητες.

Αυτά είναι τα δεδομένα, κάθε ένα από αυτά ορίζει τον τρόπο χρήσης του. Το τιμόνι στρίβει αριστερά/δεξιά, τα πεντάλ πιέζονται ή αφήνονται και οι ταχύτητες αλλάζουν διακριτά έχοντας μηχανισμό ασφαλείας δε μπορούμε να αλλάξουμε ταχύτητα σε όπισθεν ενώ το αυτοκίνητο κινείται με ταχύτητα. Η υλοποίηση καθενός από αυτούς τους μηχανισμούς διαφέρει σε κάθε κατασκευαστή, αλλά η χρήση τους είναι η ίδια για όλους. Δηλαδή η χρήση του τιμονιού και των ταχυτήτων γίνεται με τον ίδιο τρόπο ανεξαρτήτως κατηγορίας, κατασκευαστή και μοντέλου του αυτοκινήτου. Επίσης, ο μηχανισμός με τον οποίο γίνεται η χρήση των δεδομένων αυτών είναι κρυμμένος από τον χρήστη (δηλαδή τον οδηγό). Ο χρήστης δεν ενδιαφέρεται για τον τρόπο μετάδοσης της κίνησης από το τιμόνι στους τροχούς ώστε το αυτοκίνητο να στρίψει. Επίσης, η χρήση ενός αυτοκινήτου δεν αλλάζει με την επέκτασή του ή αλλαγή ορισμένων χαρακτηριστικών του (όπως π.χ. νέα μηχανή, λάστιχα, κλπ). Αλλάζει η συμπεριφορά του αλλά όχι η χρήση του.

Με το παραπάνω παράδειγμα περιγράφονται απλά τα πιο σημαντικά χαρακτηριστικά του αντικειμενοστρεφούς προγραμματισμού, οι βασικές λοιπόν έννοιες του είναι:

Κλάση *(class),* είναι μία αυτοτελής και αφαιρετική αναπαράσταση κάποιας κατηγορίας αντικειμένων. Στην ουσία είναι ένας τύπος δεδομένων, ή αλλιώς το προσχέδιο μιας δομής δεδομένων με δικές της μεταβλητές, τύπους δεδομένων, παραστάσεις, τελεστές και διαδικασίες. Τα περιεχόμενα αυτά δηλώνονται είτε ως δημόσια (public) είτε ως ιδιωτικά (private), με τα ιδιωτικά να μην είναι προσπελάσιμα από κώδικα εκτός της κλάσης. Οι διαδικασίες των κλάσεων συνήθως καλούνται μέθοδοι *(methods)* και οι μεταβλητές τους γνωρίσματα *(attributes)* ή πεδία *(fields).*

Αντικείμενο *(object),* είναι το στιγμιότυπο μιας κλάσης, δηλαδή είναι μια δομή δεδομένων (με αποκλειστικά δεσμευμένο χώρο στη μνήμη) βασισμένη στο «καλούπι» που προσφέρει η κλάση. Τα αντικείμενα μιας κλάσης μπορούν να προσπελάσουν τις δημόσιες μεθόδους άλλων αντικειμένων της ίδιας κλάσης.

Ενθυλάκωση δεδομένων *(data encapsulation),* καλείται η ιδιότητα που προσφέρουν οι κλάσεις να «κρύβουν» τα ιδιωτικά δεδομένα τους από το υπόλοιπο πρόγραμμα και να εξασφαλίζουν πως μόνο μέσω των δημόσιων μεθόδων τους θα μπορούν αυτά να προσπελαστούν. Αυτή η τακτική παρουσιάζει μόνο οφέλη καθώς εξαναγκάζει κάθε εξωτερικό πρόγραμμα να φιλτράρει το χειρισμό που επιθυμεί να κάνει στα πεδία μίας

κλάσης μέσω των ελέγχων που μπορούν να περιέχονται στις δημόσιες μεθόδους της κλάσης.

Κληρονομικότητα ονομάζεται η ιδιότητα των κλάσεων να επεκτείνονται σε νέες κλάσεις, ρητά δηλωμένες ως κληρονόμους (υποκλάσεις ή 'θυγατρικές κλάσεις'), οι οποίες μπορούν να επαναχρησιμοποιήσουν τις μεταβιβάσιμες μεθόδους και ιδιότητες της γονικής τους κλάσης αλλά και να προσθέσουν δικές τους. Στιγμιότυπα των θυγατρικών κλάσεων μπορούν να χρησιμοποιηθούν όπου απαιτούνται στιγμιότυπα των γονικών (εφόσον η θυγατρική είναι κατά κάποιον τρόπο μία πιο εξειδικευμένη εκδοχή της γονικής), αλλά το αντίστροφο δεν ισχύει. Για παράδειγμα υπάρχει η υπερκλάση *Σχήμα* (class Shape{}) και οι δύο υποκλάσεις της *Τρίγωνο* (class Triangle: public Shape{}) και *Κύκλος* (class Circle: public Shape{}), οι οποίες κληρονομούν την υπερκλάση. Πολλαπλή κληρονομικότητα είναι η δυνατότητα που προσφέρουν ορισμένες γλώσσες προγραμματισμού μία κλάση να κληρονομεί ταυτόχρονα από περισσότερες από μία υπερκλάσεις. Ακόμη από μια υποκλάση μπορούν να προκύψουν νέες υποκλάσεις που κληρονομούν την υποκλάση, με αποτέλεσμα να δημιουργηθεί μια ιεραρχία κλάσεων που συνδέονται μεταξύ τους με σχέσεις κληρονομικότητας.

Υπερφόρτωση μεθόδου *(methods overloading),* είναι η κατάσταση κατά την οποία υπάρχουν, στην ίδια ή σε διαφορετικές κλάσεις, μέθοδοι με το ίδιο όνομα και πιθανώς διαφορετικά ορίσματα. Αν πρόκειται για μεθόδους της ίδιας κλάσης διαφοροποιούνται μόνο από τις διαφορές τους στα ορίσματα και στον τύπο επιστροφής.

Αφηρημένη κλάση *(abstract class),* είναι μία κλάση που ορίζεται μόνο για να κληρονομηθεί σε θυγατρικές υποκλάσεις και δεν υπάρχουν δικά της στιγμιότυπα (αντικείμενα). Η αφηρημένη κλάση ορίζει απλώς ένα πρότυπο το οποίο θα πρέπει να ακολουθούν οι υποκλάσεις της όσον αφορά τις υπογραφές των μεθόδων τους. Μία αφηρημένη κλάση μπορεί να έχει και μη αφηρημένες μεθόδους οι οποίες υλοποιούνται στην ίδια την κλάση.

Με τον **πολυμορφισμό** *(polymorphism)* αντικείμενα που ανήκουν σε παρόμοιες κλάσεις μπορούν να έχουν κοινό τρόπο προσπέλασης, με αποτέλεσμα ο χρήστης να μπορεί να τα χειριστεί με τον ίδιο τρόπο χωρίς να χρειάζεται να μάθει νέες διαδικασίες.

Οι βασικοί τύποι δεδομένων στην C++ είναι σχεδόν οι ίδιοι με αυτούς της C. Ακολουθεί πίνακας με τους διαθέσιμους τύπους δεδομένων της C++ και το μέγεθός τους σε bytes, για ένα συνηθισμένο PC (32-bit αρχιτεκτονική).

Πίνακας 1-1 Βασικοί τύποι δεδομένων στην C++

Όνομα	**Μέγεθος**	**Διάστημα**
char	1 byte	signed: -128 έως 127 unsigned: 0 έως 255
short	2 bytes	signed: -32768 έως 32767 unsigned: 0 έως 65535
int	4 bytes	signed: -2147483648 to 2147483647 unsigned: 0 to 4294967295
long int	4 bytes	signed: -2147483648 to 2147483647 unsigned: 0 to 4294967295
float	4 bytes	+/- 3.4e +/- 38 (~7 digits)
double	8 bytes	+/- 1.7e +/- 308 (~15 digits)
bool	1 byte	true / false
wchar	2 ή 4 bytes	1 wide character

Οι τύποι δεδομένων char, short, int, long int προορίζονται για χρήση με ακέραιους αριθμούς, ενώ οι τύποι float, double για χρήση αριθμών κινητής υποδιαστολής (δηλ. δεκαδικών αριθμών). Ο τύπος bool μπορεί να λαμβάνει μόνο δύο τιμές true και false. Ο τύπος wchar έχει μέγεθος 2 bytes γιατί έχει σχεδιαστεί να περιέχει χαρακτήρες Unicode (UTF-16 για την ακρίβεια).

Παραδείγματα δηλώσεων και αρχικοποιήσεων μεταβλητών:

```
int an_integer;

an_integer = 10;

long a_long = an_integer *1000;
```

```
double verysmallnumber = 0.000000000003;

bool am_i_hungry = false;

char alpha = 'a';
```

Τα πιο πάνω αποτελούν τις θεμελιώδεις έννοιες του αντικειμενοστρεφή προγραμματισμού. Τα τελευταία χρόνια η ανάπτυξη λογισμικού γίνεται με αντικειμενοστρεφείς προσεγγίσεις, με αποτέλεσμα τα συστήματα να γίνονται απλούστερα ως προς την ανάπτυξή τους και περισσότερο ποιοτικά και αξιόπιστα ως προς το αποτέλεσμά τους.

Ένα από τα σημαντικότερα προβλήματα στην ανάπτυξη λογισμικού είναι η επικοινωνία μεταξύ των μελών της ομάδας ανάπτυξης αλλά και μεταξύ αναλυτών και πελατών. Το έργο χωρίζεται σε πολλά τμήματα άρα θα δοθεί σε πολλούς σχεδιαστές-προγραμματιστές όπου αυτό απαιτεί την χρήση κοινής ορολογίας και μοντέλου σχεδίασης όπου θα επιτρέπει αφαιρέσεις, προσθήκες και βελτιώσεις στο έργο πριν αυτό παραδοθεί στην αγορά. Στα πλαίσια αυτά η **Ενοποιημένη Γλώσσα Μοντελοποίησης** *(Unified Modeling Language)* αποτελεί την ποιό δημοφιλή και πρότυπη γλώσσα για την αποτύπωση μιας λύσης, που θα λειτουργούσε και ως υποδομή για την επικοινωνία μεταξύ των ενδιαφερομένων.

1.2. Εισαγωγή στην UML

Η Ενοποιημένη Γλώσσα Μοντελοποίησης *(UML)* είναι μια γλώσσα διαμόρφωσης, που χρησιμοποιείται για να κάνει το σχεδιασμό των συστημάτων γρηγορότερο και εφόσον υπάρχει καλή επικοινωνία μεταξύ των αναλυτών του έργου και των πελατών το λογισμικό παραδίδεται πιο γρήγορα. Η UML είναι ένα οπτικό εργαλείο που βοηθά τους μηχανικούς λογισμικού και τους σχεδιαστές λογισμικού να καταλάβουν τις προδιαγραφές του συστήματος καλύτερα, να διευκρινίσει τις αμφιβολίες τους και να δημιουργήσει τις συνδεδεμένες λύσεις όπως απαιτείται από τους πελάτες έτσι ώστε να επιφέρει το επιθυμητό αποτέλεσμα. Πριν την δεκαετία του 90’ δεν υπήρχε καμία κοινή γλώσσα για το λογισμικό. Προτού η UML γίνει διεθνές πρότυπο, δύο μηχανικοί λογισμικού ακόμα κι αν μιλούσαν την ίδια γλώσσα, δεν είχαν κανέναν τρόπο να μιλήσουν για την ανάπτυξη του λογισμικού τους. Δεν υπάρχει αμφιβολία πως η πρόοδος

του έργου ήταν αργή. Με την εμφάνιση της UML οι μηχανικοί λογισμικού έχουν ένα κοινό γραφικό λεξιλόγιο για το καθετί που αφορά το λογισμικό. Ένα απλό παράδειγμα για το πόσο χρήσιμη είναι η UML είναι το εξής «αν σε μία παγκόσμια παρουσίαση με ακροατές διαφορετικών εθνικοτήτων ο ομιλητής γράψει σε ένα ασπροπίνακα 1+1= σίγουρα όλοι θα γνωρίζουν την απάντηση γιατί τα συγκεκριμένα σύμβολα είναι ενιαία σε όλο τον κόσμο».

1.2.1. Ιστορία της UML

Μετά την ευρεία εξάπλωση του αντικειμενοστρεφούς προγραμματισμού κατά τη δεκαετία του '90, το αντικειμενοστρεφές μοντέλο σχεδίασης (με κλάσεις, κληρονομικότητα, αντικείμενα και τυποποιημένες αλληλεπιδράσεις μεταξύ τους) επικράτησε ακόμη και για μοντελοποίηση που δεν περιελάμβανε προγραμματισμό (π.χ. σχήματα βάσεων δεδομένων). Έτσι αναπτύχθηκαν διάφορες πρότυπες γλώσσες μοντελοποίησης λογισμικού οι οποίες τυποποιούσαν οπτικά σύμβολα και συμπεριφορές με στόχο την αφαιρετική περιγραφή της λειτουργίας και της δομής ενός υπολογιστικού συστήματος. Οι γλώσσες αυτές είχαν εξαρχής έναν εμφανή αντικειμενοστρεφή προσανατολισμό. Τελικά οι πιο δημοφιλείς από αυτές ενοποιήθηκαν στο κοινό πρότυπο UML που η πρώτη του έκδοση οριστικοποιήθηκε το 1997. Η UML πλέον είναι η πρότυπη γλώσσα μοντελοποίησης στη μηχανική λογισμικού. Χρησιμοποιείται για την γραφική απεικόνιση, τον προσδιορισμό, την κατασκευή και την τεκμηρίωση των στοιχείων ενός συστήματος λογισμικού. Δεδομένου ότι η UML χρησιμοποιεί την οπτική απεικόνιση του λογισμικού στο σχέδιο συστημάτων, καθιστά την επικοινωνία απλούστερη και τους στόχους ευκολότερους να υλοποιηθούν. Η ενοποιημένη γλώσσα διαμόρφωσης χρησιμοποιεί τα διαγράμματα για να δημιουργήσει αυτές τις οπτικές απεικονίσεις αποκαλούμενες διαγράμματα περίπτωσης χρήσης. Μπορεί να χρησιμοποιηθεί σε διάφορες φάσεις ανάπτυξης, από την ανάλυση απαιτήσεων ως τον έλεγχο ενός ολοκληρωμένου συστήματος. Αποτελείται από ένα σύνολο προσυμφωνημένων όρων, συμβόλων και διαγραμμάτων που επιτρέπουν:

- την εμφάνιση των ορίων ενός συστήματος και των βασικών λειτουργιών του, χρησιμοποιώντας «περιπτώσεις χρήσης» *(use-cases)* και «actors».

- την επεξήγηση της πραγματοποίησης των περιπτώσεων χρήσης με «διαγράμματα αλληλεπίδρασης».
- την αναπαράσταση μιας στατικής δομής ενός συστήματος χρησιμοποιώντας «διαγράμματα κλάσεων».
- την μοντελοποίηση της συμπεριφοράς των αντικειμένων με «διαγράμματα καταστάσεων».
- την αποκάλυψη της υλοποίησης της αρχιτεκτονικής με «διαγράμματα συστατικών» και «ανάπτυξης».
- την επέκταση της λειτουργικότητας με «στερεότυπα».

Συμπερασματικά είναι σχεδόν αδύνατο στις μέρες μας κάποιος να ασχοληθεί με την ανάπτυξη λογισμικού χωρίς να γνωρίζει την δομή των αντικειμενοστρεφών συστημάτων. Αναγκαία είναι και η γνώση της UML για την μοντελοποίηση αυτών των συστημάτων.

1.2.2. Διαγράμματα της UML

Τα βασικά διαγράμματα που υπάρχουν στην UML είναι τα ακόλουθα:

- **Διαγράμματα περιπτώσεων χρήσης** *(Use Case Diagram)*: Αποτυπώνουν τις προδιαγραφές του υπό κατασκευή συστήματος.
- **Διαγράμματα κλάσεων** *(Class Diagram)*: Περιγράφουν τις οντότητες που απαρτίζουν ένα σύστημα και τις στατικές συσχετίσεις μεταξύ τους.
- **Διαγράμματα αλληλεπίδρασης** *(Interaction Diagram)*
  - ✓ Διάγραμμα ακολουθίας *(Sequence Diagram)*: Αναπαριστάνει τα συμβάντα των εξωτερικών χρηστών με το σύστημα.
  - ✓ Διάγραμμα συνεργασίας *(Collaboration Diagram)*: Είναι ισοδύναμο με το διάγραμμα ακολουθίας με την έννοια ότι αποτυπώνονται σε αυτό οι ίδιες οι πληροφορίες.
- **Διαγράμματα καταστάσεων** *(State chart Diagram)*: Τα διαγράμματα καταστάσεων εμφανίζουν μια μηχανή καταστάσεων με τις δυνατές καταστάσεις μιας οντότητας και τις δυνατές μεταπτώσεις μεταξύ των καταστάσεων.

- **Διαγράμματα δραστηριοτήτων** *(Activity Diagram)*: Τα διαγράμματα δραστηριοτήτων χρησιμοποιούνται για την απεικόνιση διαδικασιών, επιχειρηματικών διεργασιών και ροής εργασίας.
- **Διαγράμματα συστατικών** *(Component Diagram)*: Δίνει τη δυνατότητα να εμφανίσουμε συστατικά με τις δημόσιες διασυνδέσεις που αυτά παρέχουν.
- **Διαγράμματα ανάπτυξης** *(Deployment Diagram)*:Τα διαγράμματα ανάπτυξης περιγράφουν την φυσική υποδομή του συστήματος.

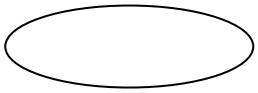
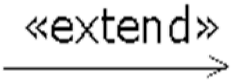
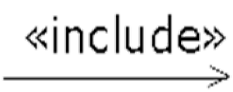
1.2.2.1. Διαγράμματα περιπτώσεων χρήσης

Ένα σημαντικό μέρος της UML είναι η υλοποίηση όλων των περιπτώσεων χρήσης σε διάγραμμα περίπτωσης χρήσης. Οι περιπτώσεις χρήσης χρησιμοποιούνται κατά τη διάρκεια της φάσης ανάλυσης ενός προγράμματος για να προσδιοριστεί και να χωριστεί η λειτουργία του συστήματος. Χωρίζουν το σύστημα στους χειριστές ή χρήστες συστήματος και τις περιπτώσεις χρήσης.

Οι χρήστες μπορούν να είναι άνθρωποι, άλλοι υπολογιστές, κομμάτια του υλικού, ή ακόμα και άλλα συστήματα λογισμικού. Το μόνο κριτήριο είναι ότι πρέπει να είναι εξωτερικοί στο χωρισμό του συστήματος στις περιπτώσεις χρήσης δηλαδή να παρέχουν εξωτερικά ερεθίσματα και να περιμένουν την απόκριση του συστήματος.

Οι περιπτώσεις χρήσης περιγράφουν την συμπεριφορά του συστήματος όταν στέλνει ένας από τους εξωτερικούς χρήστες ένα ιδιαίτερο ερέθισμα. Αυτή η συμπεριφορά καταγράφεται και δημιουργείται ένα κείμενο με όλες τις απαιτήσεις του συστήματος. Το κείμενο της περίπτωσης χρήσης επίσης συνήθως περιγράφει τις εναλλακτικές ροές κατά τη διάρκεια της διευκρινισμένης συμπεριφοράς, καθώς και ποια διορθωτικά μέτρα θα πρέπει να πάρει το σύστημα. Παραδείγματος χάριν, στην περίπτωση χρήσης ΠΧ01-Πληρωμή όπου ο πελάτης δεν έχει το απαιτούμενο ποσό σε μετρητά: μια εναλλακτική λύση είναι να ακυρωθεί η παραγγελία. Ο ακόλουθος πίνακας συνοψίζει τις βασικές έννοιες που συναντάμε σε ένα διάγραμμα περιπτώσεων χρήσης.

Πίνακας 1- 2 Βασικές έννοιες σε ένα διάγραμμα περιπτώσεων χρήσης

Κατασκευή	**Περιγραφή**	**Συντακτικό**
Περίπτωση χρήσης	Μία ακολουθία ενεργειών, συμπεριλαμβανομένων και των παραλλαγών τους, που μπορεί να επιτελέσει ένα σύστημα σε αλληλεπίδραση με τους ρόλους που υπάρχουν στο σύστημα αυτό	 *Περίπτωση Χρήσης*
Σύστημα	Αποτελεί το σύνολο των περιπτώσεων χρήσης	
Ρόλος	Ένας ρόλος είναι ένας ρόλος που μπορεί να παίξει ένας χρήστης του συστήματος όταν αλληλεπιδρά με τις περιπτώσεις χρήσης του συστήματος	
Συσχέτιση	Δηλώνει την συμμετοχή ενός ρόλου σε μια περίπτωση χρήσης	
Γενίκευση	Μια συσχέτιση μιας πιο γενικής περίπτωσης χρήσης με μια πιο ειδική περίπτωση χρήσης.	
Επέκταση	Η σχέση μιας επεκταμένης και μιας περίπτωσης χρήσης βάσης, που προσδιορίζει πως θα γίνει αυτή η επέκταση (τα σημεία επέκτασης).	
Περιεκτικότητα	Μία τέτοια συσχέτιση δείχνει ότι μια περίπτωση χρήσης περιλαμβάνει τις λειτουργίες μιας άλλης περίπτωσης χρήσης. Δηλαδή	

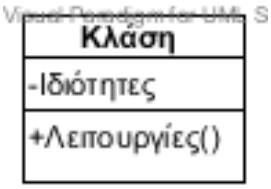
	η λειτουργικότητα της περιλαμβανόμενης περίπτωσης χρήσης εισάγεται στην περίπτωση χρήσης βάσης.	

1.2.2.2. Διαγράμματα κλάσεων

Σκοπός ενός διαγράμματος κλάσεων είναι να απεικονίσει τις κλάσεις μέσα σε ένα πρότυπο. Σε μια αντικειμενοστρεφή εφαρμογή, οι κλάσεις έχουν ιδιότητες (μεταβλητές μελών), διαδικασίες (λειτουργίες μελών) και σχέσεις με άλλες κλάσεις. Το διάγραμμα κλάσεων της UML μπορεί να απεικονίσει όλα αυτά αρκετά εύκολα.

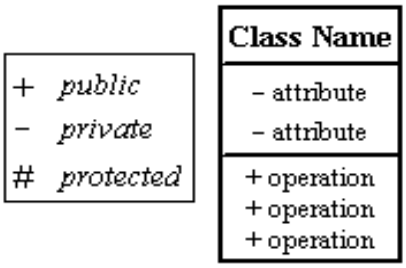
Το θεμελιώδες σύμβολο του διαγράμματος κλάσεων είναι ένα παραλληλόγραμμο με τρία διαμερίσματα το πρώτο αναφέρεται το όνομα της κλάσης, στο μεσαίο απεικονίζονται οι ιδιότητες και στο τελευταίο οι λειτουργίες της κλάσης.

Στο παρακάτω σχήμα βλέπουμε πως συμβολίζεται μια κλάση.



Σχήμα 1-1 Συμβολισμός μιας κλάσης

Όπως φαίνεται στο *Σχήμα 1-2***,** οι δείκτες διαφάνειας χρήσης δηλώνουν ποιος μπορεί να έχει πρόσβαση στις πληροφορίες που περιλαμβάνονται μέσα σε μια κλάση. Η **ιδιωτική –** *(private)* διαφάνεια κρύβει τις πληροφορίες δηλαδή ότι είναι ιδιωτικό δεν είναι προσπελάσιμο από άλλες κλάσεις ενώ η **δημόσια** + *(public)* διαφάνεια είναι προσπελάσιμη από τις άλλες κλάσεις. Η **προστατευμένη** # *(protected)* διαφάνεια επιτρέπει στις κατηγορίες παιδιών να έχουν πρόσβαση στις πληροφορίες που κληρονόμησαν από μια κατηγορία γονέων.



Σχήμα 1-2 Δείκτες Διαφάνειας (προσβασιμότητας)

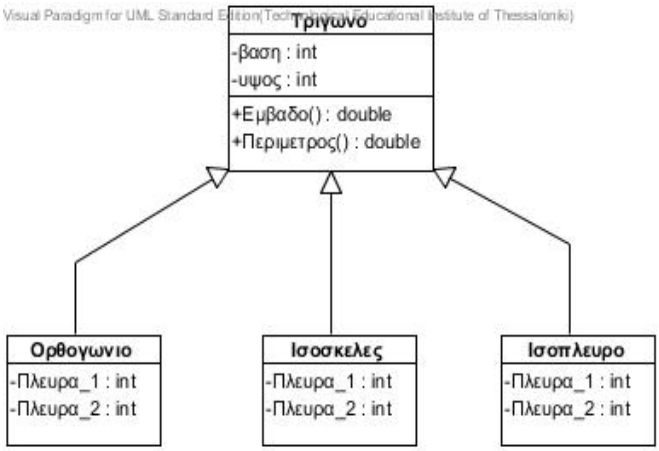
Μία **συσχέτιση** *(association)* μεταξύ δύο κλάσεων απεικονίζει μία *στατική σχέση* μεταξύ των δύο κλάσεων. Ο ρόλος της συσχέτισης είναι για να μας υποδείξει τον τρόπο που οι δύο κλάσεις βλέπουν η μια την άλλη. Το όνομα της συσχέτισης μπορεί να είναι μία λέξη που υποδηλώνει το νόημα της συσχέτισης. Το όνομα όταν αναγράφεται, πρέπει να δηλώνει το μέσο της ένωσης, όπως φαίνεται στο *Σχήμα 1-3*:

Σχήμα 1-3 Συσχέτιση μεταξύ δύο κλάσεων

Η **πολυπλοκότητα** αφορά ένα άκρο μιας συσχέτισης και είναι *το πλήθος των αντικειμένων* που μπορεί πιθανώς να συσχετίζονται με ένα αντικείμενο μιας άλλης κλάσης κατά τη διάρκεια της εκτέλεσης του προγράμματος. Στο *Σχήμα 1-3* για παράδειγμα έχει την δυνατότητα ο χρήστης να κάνει μία (**1**) κράτηση που να αποτελείται από πολλές (**1...***) θέσεις. Άλλες πιθανές τιμές για την πολλαπλότητα είναι από 0 ή περισσότερες (*****), 0 ή 1 (**0..1**),κάποιος συγκεκριμένος αριθμός (π.χ. **20**) ή ακόμα μία συγκεκριμένη περιοχή τιμών (π.χ. **5...31**).

Η **γενίκευση** είναι μία ειδική μορφή συσχέτισης κατά την οποία μία γενική κλάση αποτελεί τη βάση για τη δήλωση μιας ή περισσοτέρων ειδικότερων κλάσεων. Η γενική κλάση ονομάζεται υπερκλάση και οι ειδικότερες κλάσεις υποκλάσεις. Στην ουσία η γενίκευση στις περισσότερες γλώσσες προγραμματισμού θεωρείται ως κληρονομικότητα ή ως επέκταση μιας κλάσης σε υποκατηγορίες. Με την υλοποίηση της υπερκλάσης οι ιδιότητες και οι λειτουργίες της είναι χρήσιμες και προσπελάσιμες από τις

υποκλάσεις. Σημαντικό χαρακτηριστικό είναι ότι οι υποκλάσεις επεκτείνουν την λειτουργικότητα της υπερκλάσης με τις επιπλέον λειτουργίες τους και με αυτόν τον τρόπο εξειδικεύουν την συμπεριφορά τους.



Σχήμα 1-4 Γενίκευση

Όπως βλέπουμε στο *Σχήμα 1-4* υπάρχουν τέσσερις κλάσεις, η κλάση *Τρίγωνο* αποτελεί την υπερκλάση και οι υπόλοιπες τρεις τις υποκλάσεις. Οι υποκλάσεις έχουν κληρονομήσει τις ιδιότητες και τις λειτουργίες της υπερκλάσης, όμως η κάθε μια ξεχωριστά έχει ορίσει και δικές της ιδιότητες.

1.2.2.3. Διαγράμματα αλληλεπίδρασης

Από το όνομα αλληλεπίδραση είναι σαφές ότι το διάγραμμα χρησιμοποιείται για να περιγράψει κάποιο τύπο αλληλεπιδράσεων μεταξύ των διαφορετικών στοιχείων στο πρότυπο. Έτσι αυτή η αλληλεπίδραση είναι ένα μέρος της δυναμικής συμπεριφοράς του συστήματος. Αυτή η διαλογική συμπεριφορά αντιπροσωπεύεται σε UML από δύο διαγράμματα γνωστά ως *διάγραμμα ακολουθίας* και *διάγραμμα συνεργασίας*. Οι βασικοί σκοποί και των δύο διαγραμμάτων είναι παρόμοιοι. Το διάγραμμα ακολουθίας υπογραμμίζει εγκαίρως την ακολουθία μηνυμάτων και το διάγραμμα συνεργασίας υπογραμμίζει την δομική οργάνωση των αντικειμένων που στέλνουν και λαμβάνουν τα μηνύματα. Σκοπός των διαγραμμάτων αλληλεπίδρασης είναι να απεικονιστεί η διαλογική συμπεριφορά του συστήματος. Τα διαγράμματα ακολουθίας και συνεργασίας χρησιμοποιούνται για να συλλάβουν την δυναμική φύση αλλά από μια διαφορετική γωνία.

1.2.2.3.1. Διαγράμματα ακολουθίας

Στα διαγράμματα ακολουθίας έχουμε την απεικόνιση μιας συνεργασίας αντικειμένων. Τα αντικείμενα συμβολίζονται με παραλληλόγραμμα στα οποία αναγράφεται το όνομα του αντικειμένου ακολουθούμενο από μία άνω-κάτω τελεία και στην συνέχεια το όνομα της κλάσης. Κάτω από κάθε αντικείμενο εκτείνεται μία διακεκομμένη γραμμή που ονομάζεται ζωή γραμμής του αντικειμένου. Αν θελήσουμε να καταστρέψουμε το αντικείμενο τότε μπορούμε να βάλουμε ένα (X) στο τέλος της γραμμής ζωής του. *Τα* αντικείμενα ανταλλάσουν μηνύματα μεταξύ *τους* που στην γλώσσα της UML ονομάζονται ερεθίσματα.

Ερέθισμα μπορεί να είναι οτιδήποτε από τα εξής:

- **Κλήση μιας λειτουργίας:** όταν γίνει κλήση μιας λειτουργίας ο αποστολέας του μηνύματος θα πρέπει να περιμένει την ολοκλήρωση της λειτουργίας για να συνεχίσει. Συμβολίζεται με ένα βέλος από τον αποστολέα προς τον παραλήπτη.

  Η κεφαλή του βέλους είναι γεμισμένη με μαύρο χρώμα (⟶). Πάνω από το βέλος αναγράφεται το όνομα της λειτουργίας που καλείται.

- **Σήμα:** όταν ένα αντικείμενο αποστέλλει ένα ασύγχρονο μήνυμα σε ένα άλλο αντικείμενο. Η μόνη διαφορά από την κλήση είναι ότι ο αποστολέας δεν περιμένει την ολοκλήρωση της επεξεργασίας του μηνύματος που έστειλε. Συμβολίζεται με ένα ανοιχτό βέλος (⟶).

- **Δημιουργία αντικειμένου**: είναι ένα ερέθισμα που καταλήγει στη δημιουργία ενός αντικειμένου και όχι στην γραμμή ζωής κάπου υπάρχοντος αντικειμένου.

- **Επιστροφή κλήσης:** είναι ένα διακεκομμένο βέλος (<------) το οποίο συμβολίζει την επιστροφή από μια κλήση. Θα πρέπει φυσικά να έχει προηγηθεί η κλήση και να έχουν ολοκληρωθεί και άλλες πιθανές υποκλήσεις που περιέχει η εκτέλεση της συγκεκριμένης λειτουργίας. Πάνω στο βέλος αναγράφεται η τιμή επιστροφής.

- **Καταστροφή αντικειμένου**: είναι ένα ερέθισμα που καταλήγει στο συμβολισμό τερματισμού της ζωής ενός άλλου αντικειμένου. Σηματοδοτεί την καταστροφή του αντικειμένου ως αποτέλεσμα ενός μηνύματος από ένα άλλο αντικείμενο.

Για να γίνουν πιο κατανοητά τα παραπάνω ας εξετάσουμε το διάγραμμα ακολουθίας που αφορά την δημιουργία ενός ταμία insert_ cashier (). Στο διάγραμμα ακολουθίας στο *Σχήμα 1-5* δίνεται η σειρά με την οποία γίνονται οι κλήσεις των λειτουργιών για την δημιουργία του ταμία.

Όπως φαίνεται και στο πιο κάτω σχήμα τα αντικείμενα συνεργάζονται μεταξύ τους περιμένοντας κάποιο ερέθισμα από ένα άλλο αντικείμενο. Η ανταλλαγή των ερεθισμάτων αριθμείται με αύξουσα σειρά δηλαδή το πρώτο μήνυμα που στέλνεται καταχωρείται με τον αριθμό ένα. Στο πιο κάτω παράδειγμα θα δείτε πως γίνεται η καταχώρηση ενός καινούργιου ταμία στο σύστημα. Ας πάρουμε με την σειρά τα μηνύματα και να τα αναλύσουμε:

1: Το σύστημα επιστρέφει στον ταμία μια κάρτα εγγραφής για νέους υπαλλήλους.

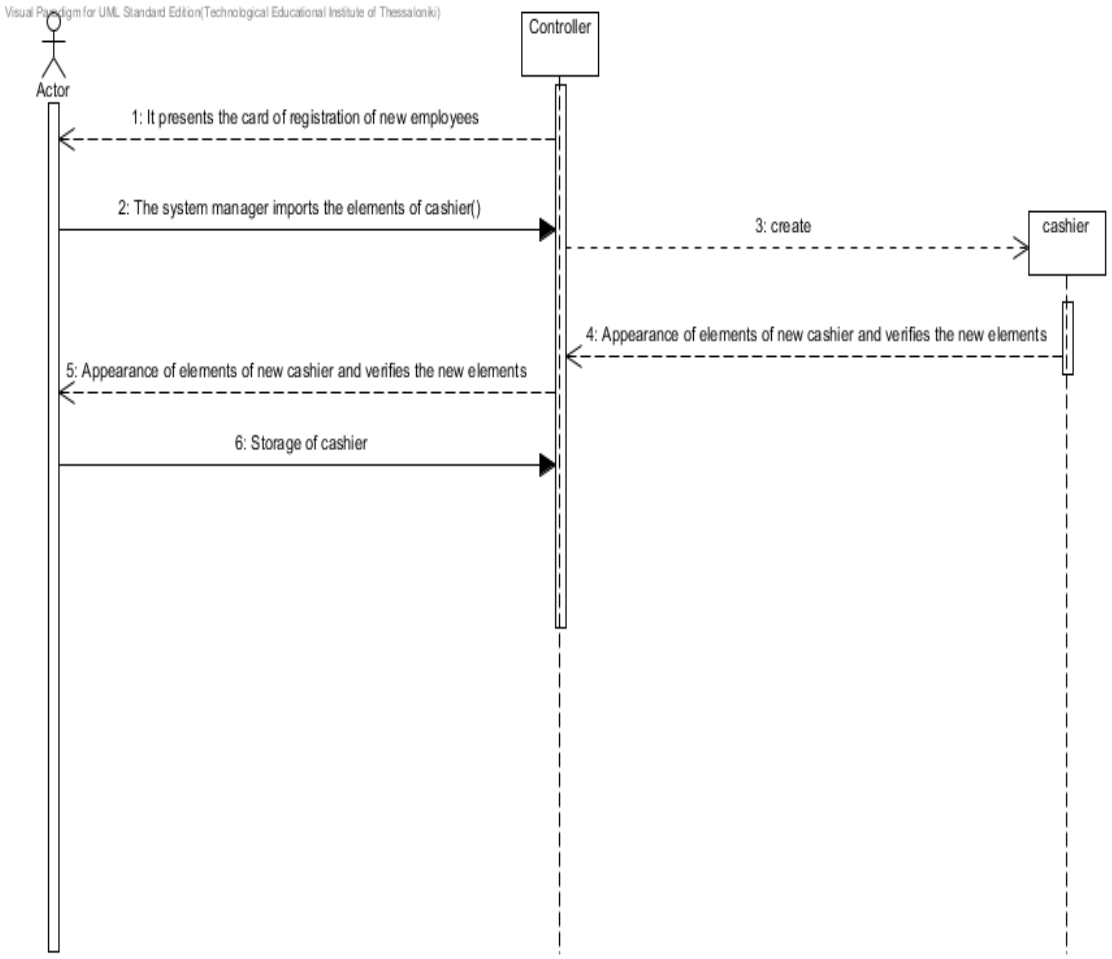
2: Ο χειριστής του λογισμικού που στην περίπτωση μας είναι ο ταμίας θα πρέπει να καταχωρίσει τα στοιχεία του καινούργιου ταμία, γι' αυτό καλείται η μέθοδος *import_elements_of_new_cashier()*.

3: Με το που καλείται η μέθοδος *import_elements_of_new_cashier()* δημιουργείται ένα αντικείμενο τύπου *Cashier* στην ίδια ευθεία που καλείται η μέθοδος γιατί πριν το συγκεκριμένο αντικείμενο δεν υπήρχε.

4: Το αντικείμενο *cashier* επιστρέφει τα στοιχεία του νέου ταμία στο σύστημα.

5: Το σύστημα εμφανίζει τα στοιχεία του νέου ταμία στον χειριστή του συστήματος και τα επαληθεύει.

6: Τέλος αποθηκεύεται ο νέος ταμίας στο σύστημα.



Σχήμα 1-5 Παράδειγμα διαγράμματος ακολουθίας

1.2.3.3.2. Διαγράμματα συνεργασίας

Ένα διάγραμμα συνεργασίας είναι ισοδύναμο με ένα διάγραμμα ακολουθίας. Η διαφορά του έγκειται στο γεγονός ότι τα αντικείμενα μπορούν να είναι διάσπαρτα στο χώρο. Επειδή ακριβώς έχουμε τυχαία τοποθέτηση των αντικειμένων στον χώρο οι ανταλλαγές μηνυμάτων θα πρέπει να αριθμηθούν έτσι ώστε να φαίνεται η σειρά τους.

Στο *Σχήμα 1-6* δίνεται ένα διάγραμμα συνεργασίας όπου φαίνονται οι ανταλλαγές μηνυμάτων μεταξύ του χρήστη και των κλάσεων έτσι ώστε να βρεθεί το όνομα του καλύτερου φοιτητή στο μάθημα το οποίο υπάρχει στην κλάση Course. Ακολουθώντας με την σειρά τους αριθμούς φαίνεται με ποια σειρά γίνεται η ανταλλαγή των μηνυμάτων.

1: Ο χρήστης του συστήματος καλεί την μέθοδο *getBestName()*.

2: Το αντικείμενο *course* παίρνει τα ονόματα από την κλάση *student.*

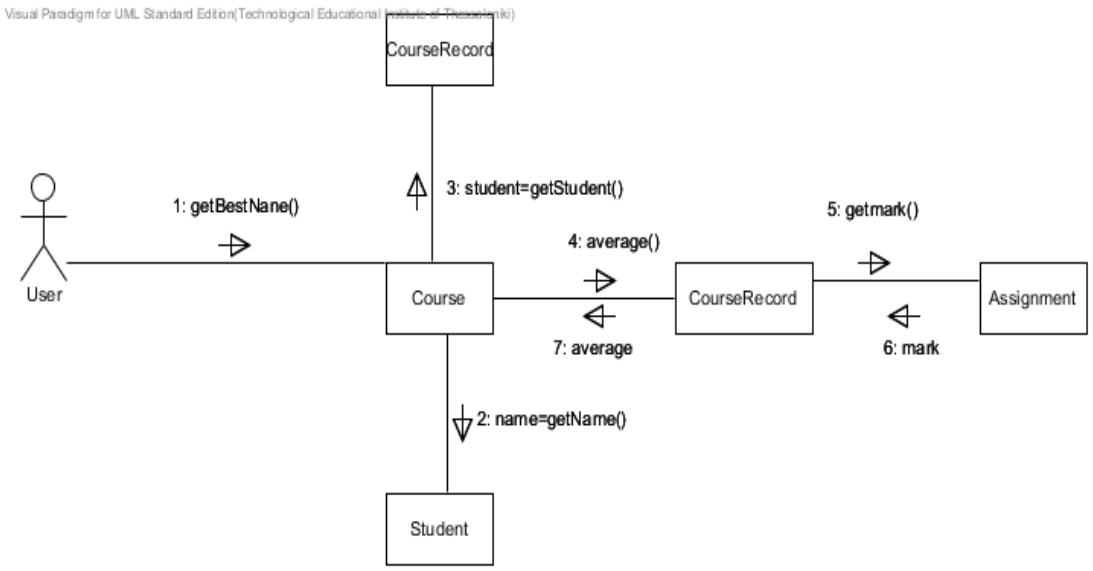
3: Από την κλάση *Course Record* παίρνουμε τον μαθητή με τον καλύτερο βαθμό.

4: Το αντικείμενο *course* καλεί την μέθοδο *average().*

5: Η μέθοδος *average()* βρίσκεται στην κλάση *Course Record* όπου για να βγάλει τον μέσο όρο της βαθμολογίας παίρνει τους βαθμούς από την κλάση Assignment μέσω της μεθόδου getmark().

6: Η κλάση επιστρέφει τους βαθμούς στην κλάση *Course Record.*

7: Επιστρέφεται ο μέσος όρος των μαθημάτων στην κλάση *Course*



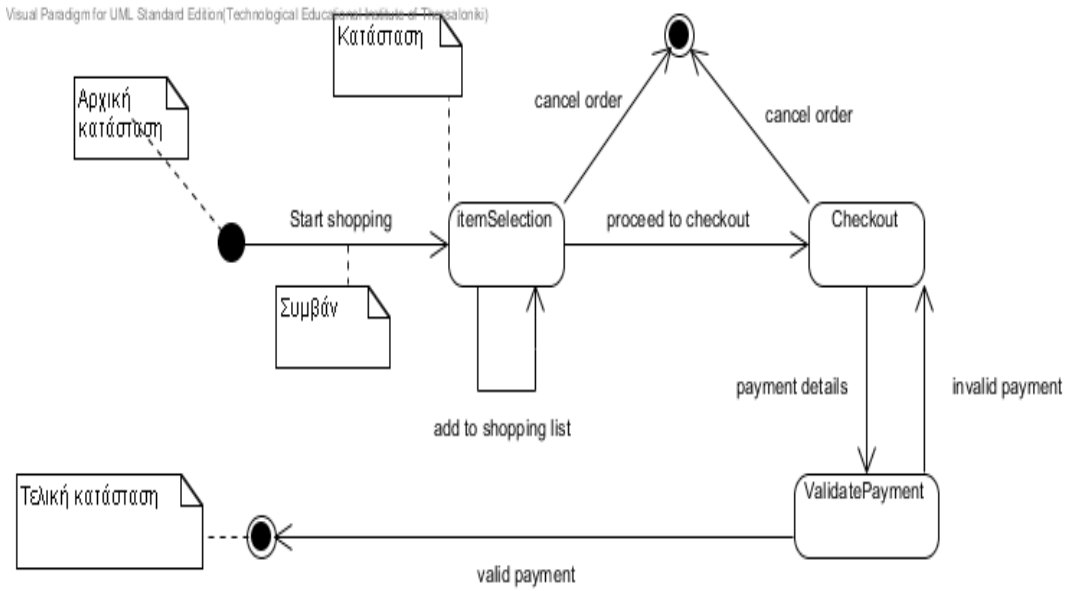
Σχήμα 1-6 Παράδειγμα διαγράμματος συνεργασίας

1.2.2.4. Διαγράμματα καταστάσεων

Το ίδιο το όνομα του διαγράμματος διευκρινίζει το σκοπό του διαγράμματος και άλλων λεπτομερειών.

Περιγράφει τις διαφορετικές καταστάσεις ενός συστατικού σε ένα σύστημα, τον κύκλο ζωής των στιγμιότυπων των κλάσεων και την εκτέλεση μίας πράξης ενός στιγμιότυπου μίας κλάσης.

Τα διαγράμματα καταστάσεων χρησιμοποιούνται για να μοντελοποιήσουν τις πιθανές καταστάσεις των στιγμιότυπων της κλάσης, τις πιθανές μεταβάσεις από την μια κατάσταση στην άλλη, τα συμβάντα που προκαλούν τις μεταβάσεις και τις πράξεις που εκτελούνται στην κατάσταση ή κατά την μετάβαση.



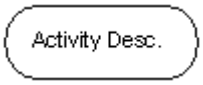
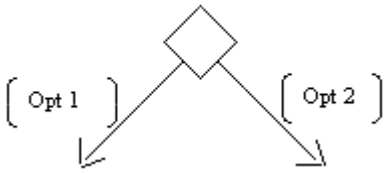
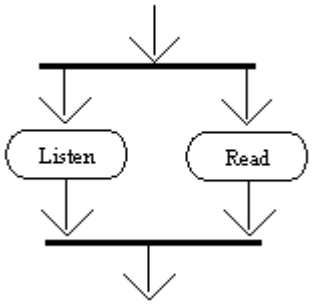
Σχήμα 1-7 Διάγραμμα κατάστασης για ψώνια

Όπως βλέπουμε στο παραπάνω σχήμα κατάστασης για ψώνια, το διάγραμμα αρχίζει με έναν μαύρο κύκλο όπου είναι η αρχική μας κατάσταση. Από την αρχική κατάσταση έχει προκύψει το συμβάν start shopping. Η *κατάσταση* συμβολίζεται με *παραλληλόγραμμο με στρογγυλεμένες γωνίες*. Στο πάνω μέρος εμφανίζεται το όνομα της κατάστασης και αποκάτω προαιρετικά μπορούν να εμφανισθούν οι δραστηριότητες που παίρνουν μέρος στην είσοδο και στην έξοδο της *κατάστασης*. Πέρα από τα συμβάντα που γίνονται από την μια κατάσταση στην άλλη μπορούμε να έχουμε και αυτομεταβάσεις. **Αυτομετάβαση** είναι ένα συμβάν που αρχίζει και τελειώνει στην ίδια κατάσταση όπως φαίνεται και στο *Σχήμα 1-7* στην κατάσταση item Selection όπου αρχίζει το συμβάν προσθήκης προϊόντων (add to shopping list) και τελειώνει στην ίδια κατάσταση. Για να υποδηλώσουμε το τέλος του διαγράμματος κατάστασης χρησιμοποιούμε ένα κύκλο και στο κέντρο του ένα άλλο μαυρισμένο κύκλο.

1.2.2.5. Διαγράμματα δραστηριοτήτων

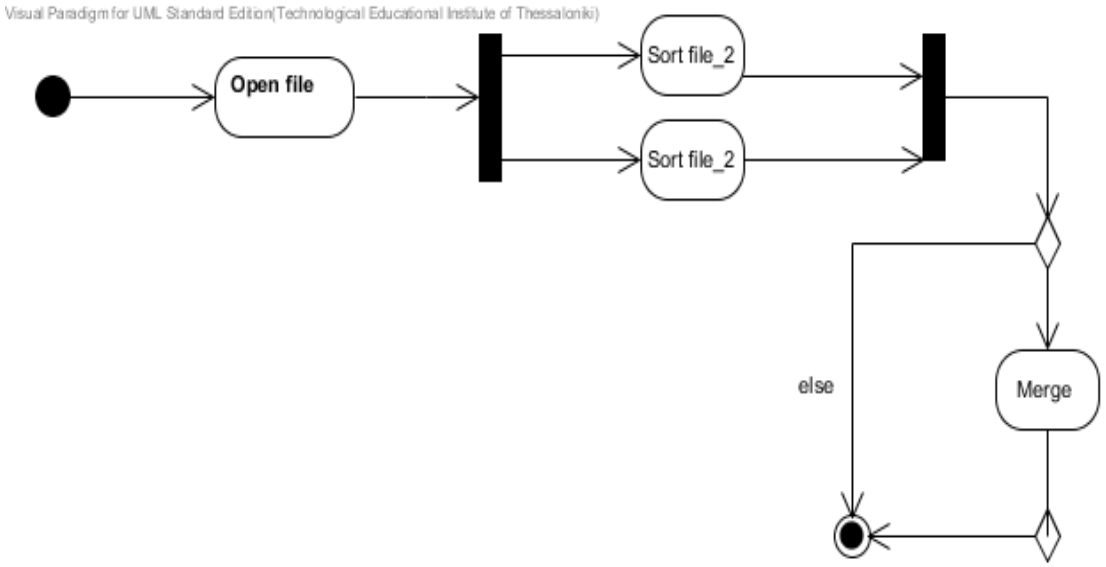
Το διάγραμμα δραστηριοτήτων είναι μια παραλλαγή του διαγράμματος καταστάσεων στο οποίο οι κόμβοι αναπαριστούν καταστάσεις ενεργειών και οι μεταβάσεις λαμβάνουν χώρα με την ολοκλήρωση αυτών των ενεργειών και όχι ως συνέπεια ενός συμβάντος όπως συμβαίνει με το διάγραμμα καταστάσεων. Στην ουσία είναι μια απλή και διαισθητική απεικόνιση αυτό που συμβαίνει σε μια ροή εργασιών, ποιες δραστηριότητες μπορούν να γίνουν παράλληλα, και εάν υπάρχουν εναλλακτικές πορείες μέσω της ροής εργασιών. Τα σύμβολα που χρησιμοποιούνται στο συγκεκριμένο διάγραμμα αναφέροντα στον πιο κάτω πίνακα:

Πίνακας 1- 3 Σύμβολα διαγράμματος δραστηριοτήτων

Στοιχείο και περιγραφή του	Σύμβολο
Αρχική δραστηριότητα: Η αφετηρία ή η πρώτη δραστηριότητα της ροής. Συμβολίζεται με έναν μαύρο κύκλο.	
Δραστηριότητα: Απεικονίζεται από ένα παραλληλόγραμμο με στρογγυλευμένες άκρες.	
Αποφάσεις: Παρόμοια με τα διαγράμματα ροής. Η λογική απόφαση που πρόκειται να ληφθεί απεικονίζεται με ένα ρόμβο, και εκτελείται η συνθήκη που είναι αληθής	
Ταυτόχρονες δραστηριότητες: Μερικές δραστηριότητες εμφανίζονται ταυτόχρονα ή παράλληλα. Τέτοιες δραστηριότητες καλούνται ταυτόχρονες δραστηριότητες. Παραδείγματος χάριν, ακούοντας τον ομιλητή και εξετάζοντας τον πίνακα είναι μια παράλληλη δραστηριότητα. Αυτό αντιπροσωπεύεται από μια διακλάδωση (fork), δύο ταυτόχρονες δραστηριότητες η μια δίπλα στην άλλη, και μια οριζόντια γραμμή για να παρουσιάσει το τέλος της παράλληλης	

δραστηριότητας.	
Τελική δραστηριότητα: Το τέλος του διαγράμματος δραστηριότητας παρουσιάζεται από ένα κύκλο και στο κέντρο του έχει έναν άλλο κύκλο μαυρισμένο. Αυτό το σύμβολο καλείται επίσης ως τελική κατάσταση.	

Όπως φαίνεται στο *Σχήμα 1-8* πιο κάτω, το διάγραμμα αρχίζει με την αρχική κατάσταση, καλείται η δραστηριότητα open file, επιλέγονται δύο αρχεία γι αυτό χρησιμοποιείται η διακλάδωση *(fork),* μετά από την επιλογή των αρχείων βάζουμε το σύμβολο απόφασης και ανάλογα αν ικανοποιείται η συνθήκη τότε καλείται η δραστηριότητα merge και τερματίζει, αν όχι τότε τερματίζει κατευθείαν.

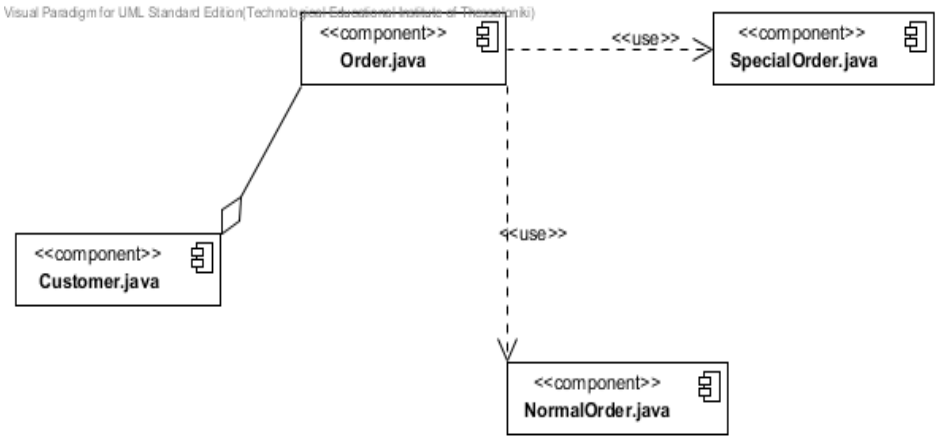


Σχήμα 1-8 Διάγραμμα δραστηριοτήτων

1.2.2.6. Διαγράμματα συστατικών

Τα συστατικά διαγράμματα είναι διαφορετικά από την άποψη της φύσης και της συμπεριφοράς χρησιμοποιούνται για να διαμορφώσουν τις φυσικές πτυχές ενός συστήματος. Οι φυσικές πτυχές είναι τα στοιχεία όπως τα executable αρχεία, οι βιβλιοθήκες, η γενικότερα *αρχεία και έγγραφα.* Έτσι τα συστατικά διαγράμματα χρησιμοποιούνται για να απεικονίσουν την οργάνωση και τις σχέσεις μεταξύ των συστατικών σε ένα σύστημα. Αυτά τα διαγράμματα χρησιμοποιούνται επίσης για να

κάνουν τα εκτελέσιμα συστήματα. Ο σκοπός είναι επίσης διαφορετικός από όλα τα άλλα διαγράμματα που συζητούνται μέχρι τώρα. Δεν περιγράφει τη λειτουργία του συστήματος αλλά περιγράφει τα συστατικά που χρησιμοποιούνται για να κάνουν εκείνες τις λειτουργίες. Τα συστατικά διαγράμματα μπορούν επίσης να περιγραφούν ως στατική άποψη εφαρμογής ενός συστήματος. Η στατική εφαρμογή αντιπροσωπεύει την οργάνωση των συστατικών σε μια ιδιαίτερη στιγμή. Ένα ενιαίο συστατικό διάγραμμα δεν μπορεί να αντιπροσωπεύσει ολόκληρο σύστημα, αλλά μια συλλογή διαγραμμάτων χρησιμοποιείται για να αντιπροσωπεύσει το σύνολο.



Σχήμα 1-9 Διάγραμμα συστατικών

Στο *Σχήμα 1-9*, υπάρχουν τέσσερα αρχεία όπου προσδιορίζονται και οι σχέσεις που παράγονται μεταξύ τους . Το συστατικό διάγραμμα δεν μπορεί να αντιστοιχηθεί άμεσα με τα άλλα διαγράμματα UML που αναφέρθηκαν μέχρι τώρα.

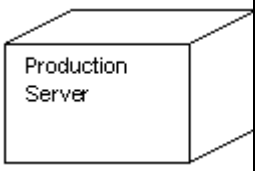
1.2.2.7. Διαγράμματα ανάπτυξης

Το διάγραμμα ανάπτυξης (διάταξης) παρέχει μια διαφορετική άποψη της εφαρμογής, συλλαμβάνει την διαμόρφωση των στοιχείων χρόνου εκτέλεσης της εφαρμογής. Αυτό το διάγραμμα είναι κατά πολύ πιο χρήσιμο όταν χτίζεται ένα σύστημα και είναι σε θέση το σύστημα να επεκταθεί. Αυτό το διάγραμμα ανάπτυξης εξελίσσεται έπειτα και αναθεωρείται έως ότου χτίζεται το σύστημα και χρησιμοποιείται κυρίως σε κατανεμημένα συστήματα για να δείξει την φυσική διάταξη των διαφόρων τμημάτων του λογισμικού. Το βασικό στοιχείο στα διαγραμμάτων ανάπτυξης είναι ο κόμβος. Ο κόμβος

αντιπροσωπεύει το περιβάλλον στο οποίο ένα συστατικό ή ένα σύνολο συστατικών εκτελείται. Οι κόμβοι του συστήματος συμβολίζονται με κύβους στους οποίους αναγράφεται το όνομα του κόμβου.

Ένα διάγραμμα ανάπτυξης αποτελείται από τα ακόλουθα στοιχεία:

Πίνακας 1- 4 Στοιχεία διαγράμματος ανάπτυξης

Στοιχείο και η περιγραφή του	**Σύμβολο**
Κόμβος: Το στοιχείο που παρέχει το περιβάλλον εκτέλεσης για τα συστατικά ενός συστήματος. Απεικονίζεται με έναν κύβο με το όνομα του αντικειμένου μέσα σε αυτόν.	
Σύνδεση: Παρόμοια με την σχέση/ ένωση που χρησιμοποιείται στα διαγράμματα κατηγορίας για να καθορίσει την διασύνδεση μεταξύ των κόμβων.	

ΚΕΦΑΛΑΙΟ 2: ΤΟ UML ΕΡΓΑΛΕΙΟ VISUAL PARADIGM

Το **Visual Paradigm for UML** *(VP-UML)* είναι ένα UML μοντέλο το οποίο υποστηρίζει όλα τα UML 2.x διαγράμματα που συντελούν στην ανάλυση και τον σχεδιασμό ενός συστήματος. Η μετατροπή των UML διαγραμμάτων σε πηγαίο κώδικα δεν είναι εύκολη υπόθεση. Το Visual Paradigm for UML, μπορεί από UML διαγράμματα, να παράγει πηγαίο κώδικα σε πάνω από 10 προγραμματιστικές γλώσσες καθώς και το αντίστροφο.

2.1. Εισαγωγή

Το VP-UML είναι ένα πολύ εύκολο και ισχυρό cross-platform εργαλείο, που χρησιμοποιείται στην μοντελοποίηση διαγραμμάτων UML. Το VP-UML παρέχει στους προγραμματιστές λογισμικού την δυνατότητα να δημιουργούν εφαρμογές γρηγορότερα, καλύτερα και φθηνότερα.

Με το εργαλείο αυτό μπορούν να σχεδιαστούν όλα τα είδη των διαγραμμάτων UML 2.x που αναφέρθηκαν στο προηγούμενο κεφάλαιο στην παράγραφο 1.2.2 (διαγράμματα της UML) και τα οποία είναι:

- Use Case Diagram — Διάγραμμα περιπτώσεων χρήσης
- Class Diagram — Διάγραμμα κλάσεων
- Sequence Diagram — Διάγραμμα ακολουθίας
- Communication Diagram — Διάγραμμα επικοινωνίας
- Activity Diagram — Διάγραμμα δραστηριοτήτων
- Component Diagram — Διάγραμμα συστατικών
- Deployment Diagram — Διάγραμμα ανάπτυξης
- Package Diagram — Διάγραμμα πακέτων
- Object Diagram — Διάγραμμα Αντικειμένων

2.2. Διαλειτουργικότητα

Η υποστήριξη της διαλειτουργικότητας από το εργαλείο VP-UML δίνει τη δυνατότητα να ανταλλάσσονται δεδομένα από το μοντέλο με άλλα εργαλεία δημιουργίας διαγραμμάτων UML.Υποστηρίζονται τα εξής εργαλεία:

Πίνακας 2-1 Στήριξη διαλειτουργικότητας από το Visual Paradigm

	Import	**Export**
Telelogic Modeler	✓	
Rational Rose	✓	
Erwin Data Modeler project	✓	
Rational Software Architect	✓	
Rational DNX	✓	
NetBeans 6.x UML diagrams	✓	
Visio drawing	✓	
BPEL for Oracle workflow engine		✓
BPEL for JBoss workflow engine		✓
Diagram (JPG, PNG, SVG, EMF, PDF)		✓
Microsoft Excel	✓	✓
EMF based UML2 model	✓	✓
XMI (1,0, 1,2, 2,1)	✓	✓
XML (native)	✓	✓
VP project	✓	✓
Microsoft Word document (for use case model)	✓	✓

2.3. Μετατροπή διαγραμμάτων από και σε πηγαίο κώδικα

Το *Instant Reverse* και *Instant Generator* παρέχονται για δημιουργία κώδικα από τα διαγράμματα ή και το αντίστροφο δημιουργία διαγραμμάτων από κώδικα. Στο πιο κάτω

πίνακα βλέπουμε τα είδη του πηγαίου κώδικα που μπορεί να ανατραπεί ή να δημιουργηθεί μέσω Instant Reverse και Instant Generator.

Πίνακας 2-2 Instant Reverse και Instant Generator

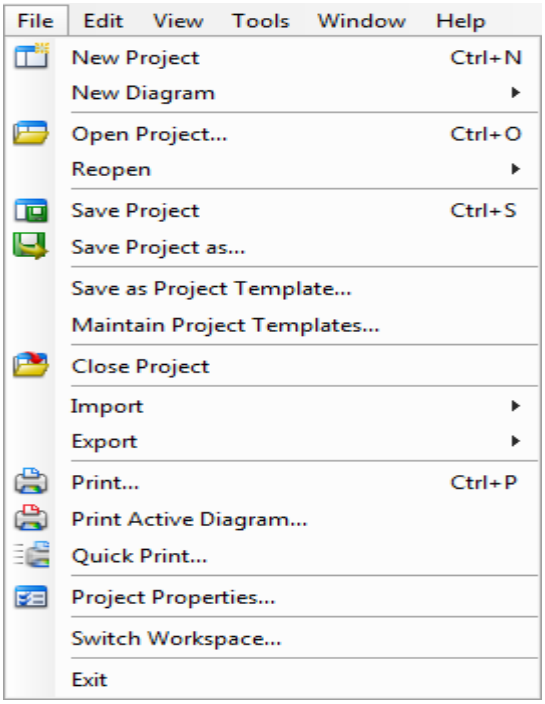
	Instant Reverse	**Instant Generator**
Java	✓	✓
C++	✓	✓
XML Schema	✓	✓
PHP	✓	✓
Python Source	✓	✓
Objective C	✓	✓
Corba IDL	✓	✓
.Net dll or .exe	✓	
JDBC	✓	
Hibernate	✓	
C#		✓
VB.Net		✓
ODL		✓
Action Script		✓
Delph		✓
Perl		✓
Ada95		✓
Ruby		✓

2.4. Διεπαφή Χρήστη – Μενού VP-UML

Το **κύριο μενού** του VP-UML , που βρίσκεται στο επάνω μέρος του παραθύρου, σας επιτρέπει να επιλέξετε και να εκτελέσετε διάφορες εργασίες στα διαγράμματα.

File Menu – Το μενού *Αρχείο* επιτρέπει:

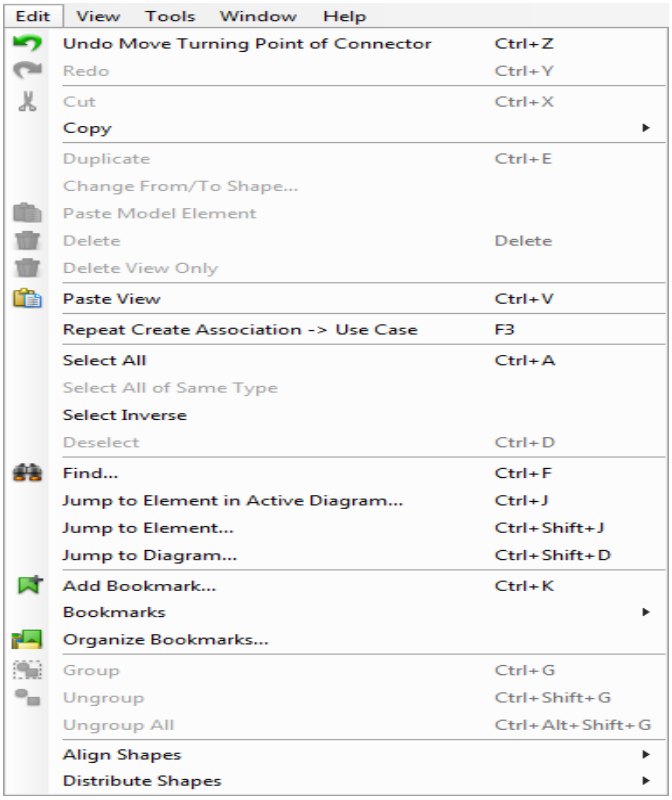
- Την δημιουργία ενός έργου.
- Δημιουργία ενός διαγράμματος UML.
- Άνοιγμα έργου.
- Αποθήκευση του έργου.
- Αποθήκευση και διαχείριση πρότυπου έργου.
- Εισαγωγή από τα ακόλουθα μέσα: **VP-UML Project, Rose Project, XMI, XML, Erwin Project, Telelogic Rhapsody Project, Telelogic System Architect, Rational Model, Rational DNX, MS Word (Use Case Model documentation exported from VP), Excel (Exported from VP), Visio, NetBeans.**
- Εξαγωγή του έργου στις εξής μορφές:VP -UML Project, XMI, XML, MS Word (for Use Case Model), Excel.
- Εξαγωγή εικόνων (JPG, PNG, SVG, EMF, PDF).
- Εκτύπωση.
- Ορισμός ιδιοτήτων του έργου.
- Έξοδος.



Εικόνα 2-1 File Menu (Μενού Αρχείο)

Edit menu - Το μενού *Επεξεργασίας* επιτρέπει :

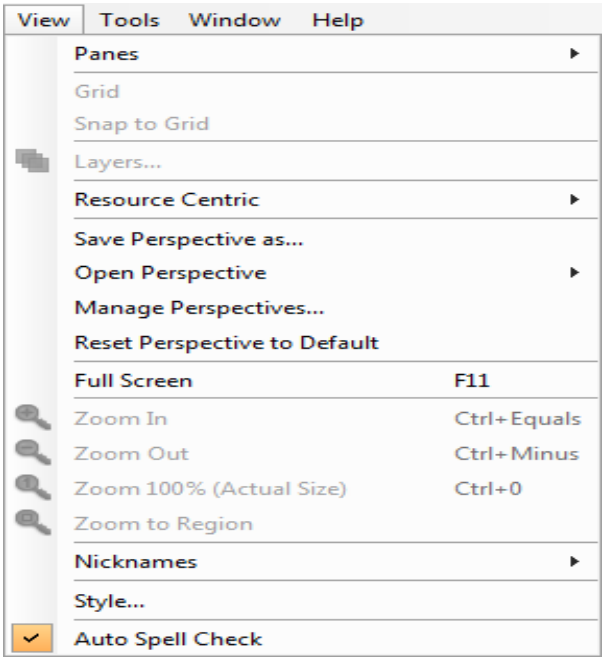
- Αναίρεση και Επανάληψη
- Αποκοπή
- Αντίτυπο
- Αντίγραφο
- Διαγραφή
- Επανάληψη μιας ενέργειας
- Επιλογή όλων στο διάγραμμα
- Εύρεση μοντέλου στοιχείου / διαγράμματος
- Μετάβαση σε ένα διάγραμμα ή ένα στοιχείο
- Προσθήκη ή διαχείριση σελιδοδεικτών
- Ομαδοποίηση
- Ευθυγράμμιση και διανομή σχημάτων



Εικόνα 2-2 Edit Menu (Μενού Επεξεργασίας)

View Menu - Το μενού *Εμφάνισης* επιτρέπει:

- Εμφάνιση / Απόκρυψη παραθύρου
- Προβολή και διαχείριση δικτύου
- Διαχείριση Layers
- Αποθήκευση, άνοιγμα και διαχείριση προοπτικής
- Αλλαγή του VP-UML ώστε να μπορεί να εμφανιστεί σε πλήρη οθόνη
- Δυνατότητα ζουμ στα διάγραμμα
- Διαχείριση ψευδώνυμου
- Διαχείριση στυλ
- Εμφάνιση ορθογραφικού ελέγχου

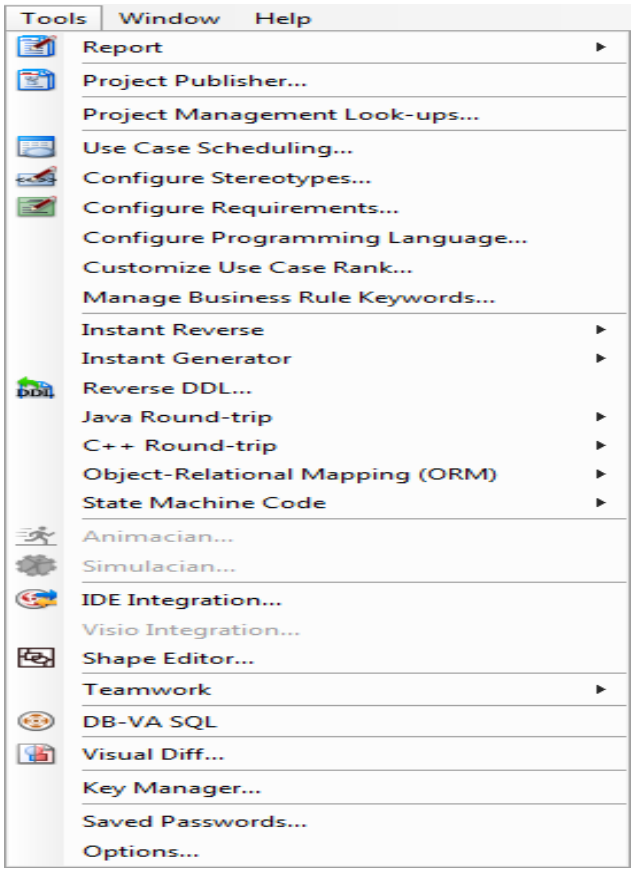


Εικόνα 2-3 View Menu (Μενού Εμφάνισης)

Tools menu -Το μενού *Εργαλείων* επιτρέπει:

- Δημιουργία αναφοράς
- Δημοσίευση του έργου
- Εκτέλεση του σχεδιασμού των περιπτώσεων χρήσης
- Διαμόρφωση στερεοτύπων
- Διαμόρφωση απαιτήσεων
- Διαμόρφωση γλώσσας προγραμματισμού

- Reverse and Forward engineering with Instant Reverse and Instant Generator
- Reverse DDL
- Εκτέλεση Object Relational Mapping (ORM)
- Εκτέλεση της μηχανής δημιουργίας κώδικα
- Εκκίνηση Shape Editor
- Σύγκριση διαγραμμάτων με το Visual Diff
- Εκκίνηση License Key Manager

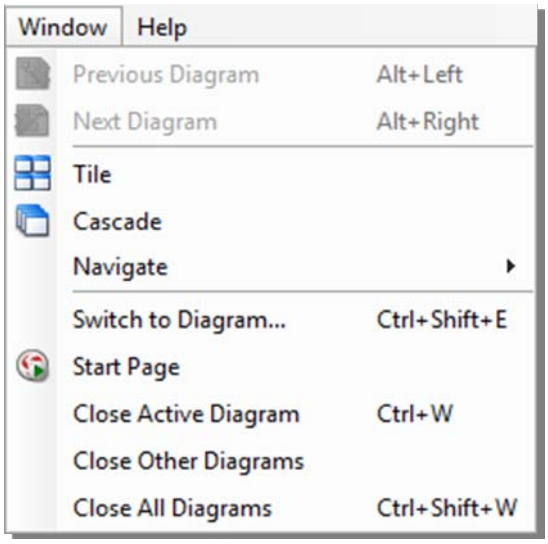


Εικόνα 2-4 Tools Menu (Μενού Εργαλείων)

Window menu - Το μενού *Παραθύρου* επιτρέπει :

- Πλοήγηση μεταξύ των διαγραμμάτων
- Αναδιάταξη διαγραμμάτων στα παράθυρα
- Αλλαγή σε άλλο διάγραμμα
- Εμφάνιση της αρχικής σελίδας

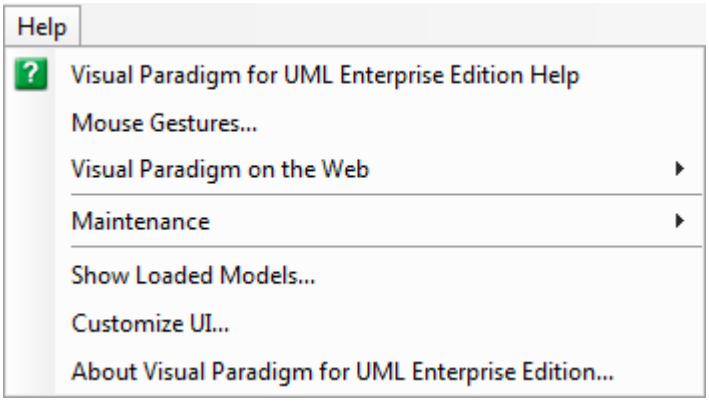
- Κλείσιμο Διαγραμμάτων



Εικόνα 2-5 Window Menu (Μενού Παραθύρου)

Help Menu - Το μενού *Βοήθεια* επιτρέπει:

- Αναζήτηση βοήθειας στα περιεχόμενα
- Έλεγχος οδηγιών του Mouse Gesture
- Ηλεκτρονική υποστήριξη του Visual Paradigm
- Επισκευή του έργου
- Προσαρμογή του περιβάλλοντος εργασίας χρήστη
- Έλεγχος περιβάλλοντος, χρησιμοποιώντας το παράθυρο διαλόγου



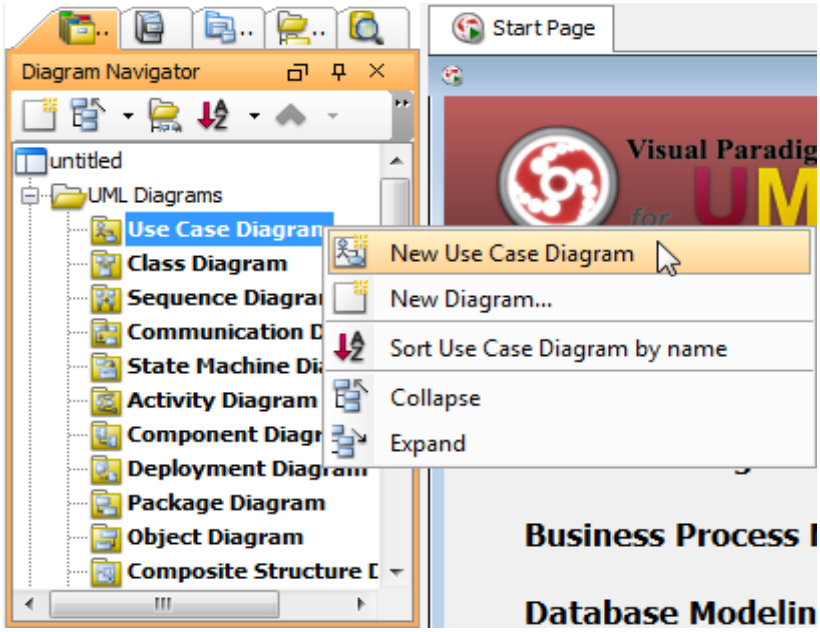
Εικόνα 2-6 Help Menu (Μενού Βοήθειας)

2.5. Δημιουργία διαγραμμάτων UML

Χρησιμοποιώντας το διάγραμμα «περίπτωσης χρήσης», επιτρέπεται να μοντελοποιηθούν οι λειτουργίες του συστήματος καθώς και οι χρήστες που αλληλεπιδρούν με αυτές τις λειτουργίες. Το VP-UML επιτρέπει την δημιουργία διαγραμμάτων περιπτώσεων χρήσης.

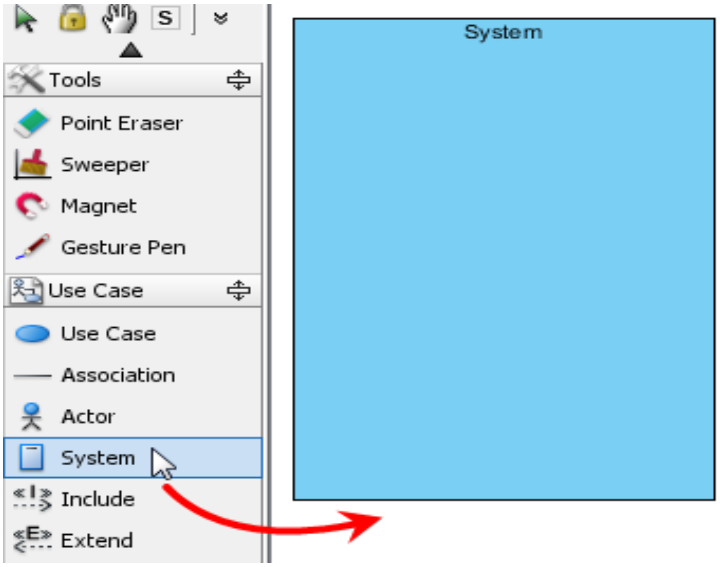
2.5.1. Δημιουργία διαγράμματος περίπτωσης χρήσης

Στην *Εικόνα 2-7*, φαίνεται πώς μπορεί να σχεδιαστεί ένα διάγραμμα περίπτωσης χρήσης. Δεξί click στο διάγραμμα *περίπτωσης χρήσης* (use case diagram) και επιλογή *νέου διαγράμματος περίπτωσης χρήσης* (new use case diagram) από το αναδυόμενο μενού. Πληκτρολογούμε το όνομα για το νεοσύστατο σχήμα περίπτωση χρήσης στον τομέα που βρίσκεται πάνω αριστερά κάνοντας διπλό click στο όνομα διαγράμματος.



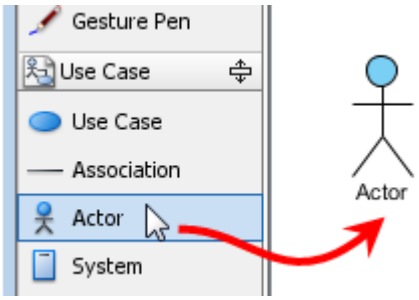
Εικόνα 2-7 Δημιουργία διαγράμματος περίπτωσης χρήσης

Στο νέο διάγραμμα περίπτωσης χρήσης που δημιουργήθηκε μπορούν να προστεθούν διάφορα στοιχεία. Έτσι για να δημιουργηθεί ένα σύστημα, επιλέγουμε *Σύστημα* στη γραμμή εργαλείων και στη συνέχεια κάνουμε click στο παράθυρο διαγράμματος. Τέλος, δίνουμε το όνομα του νεοσύστατου συστήματος όταν αυτό δημιουργηθεί.



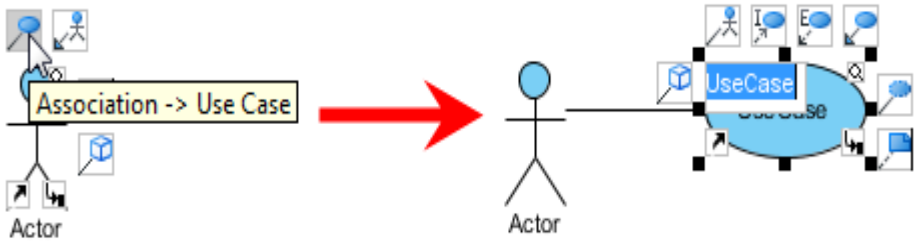
Εικόνα 2-8 Δημιουργία συστήματος

Για να σχεδιαστεί ένας χειριστής του συστήματος, επιλέγουμε **χειριστής** *(actor)* από την γραμμή εργαλείων και στη συνέχεια κάνουμε click στο παράθυρο διαγράμματος .Τέλος, δώστε το όνομά του, όταν αυτός δημιουργηθεί.



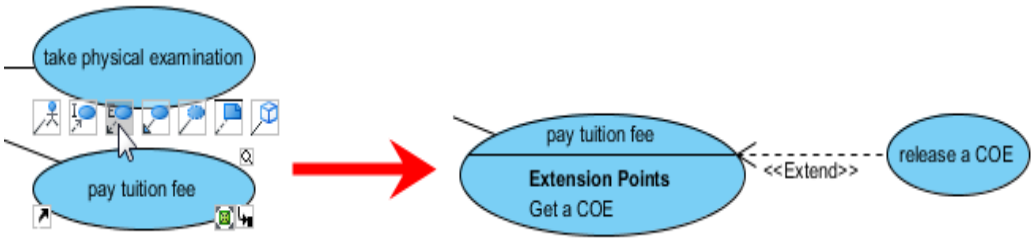
Εικόνα 2-9 Δημιουργία ενός ηθοποιού

Εκτός από τη δημιουργία μιας υπόθεσης χρήστη από το μενού εργαλείων, μπορούμε να το δημιουργήσουμε και μέσω του εικονιδίου πόρων. Μετακινώντας το ποντίκι πάνω σε ένα σχήμα και πατώντας το εικονίδιο το οποίο μπορεί να δημιουργήσει την περίπτωση χρήσης σέρνουμε και αφήνουμε το κουμπί του ποντικιού μέχρι να φτάσει στην επιθυμητή θέση. Το υπάρχον σχήμα και η νεοσυσταθείσα περίπτωση χρήσης είναι συνδεδεμένα. Τέλος, δίνουμε το όνομα του νεοσύστατης περίπτωσης χρήσης.



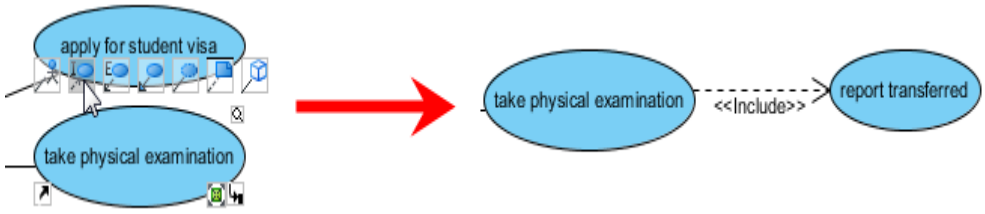
Εικόνα 2-10 Δημιουργία σεναρίου χρήσης

Για να δημιουργηθεί μια σχέση επέκτασης, μετακινούμε το ποντίκι πάνω από την περίπτωση χρήσης των πόρων και πατάμε το εικονίδιο της επέκτασης (E). Σέρνουμε το ποντίκι στην περίπτωση χρήσης που θέλουμε να γίνει και στη συνέχεια αφήνουμε το κουμπί του ποντικιού. Η περίπτωση χρήσης με τα σημεία επέκτασης και την νεοδημιουργηθείσα περίπτωση χρήσης είναι συνδεδεμένα.



Εικόνα 2-11 Δημιουργία επέκτασης

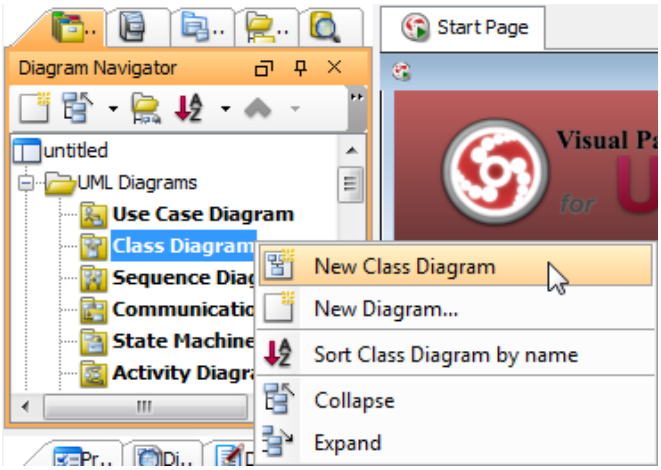
Για να δημιουργηθεί μια σχέση include (I), μετακινούμε το ποντίκι πάνω από την περίπτωση χρήσης των πόρων και πατάμε το εικονίδιο include. Σέρνουμε το ποντίκι στην περίπτωση χρήσης που θέλουμε και στη συνέχεια αφήνουμε το κουμπί του ποντικιού. Η περίπτωση χρήσης με τα σημεία include και την νεοδημιουργηθείσα περίπτωση χρήσης είναι συνδεδεμένα.



Εικόνα 2-12 Δημιουργία σχέσης include

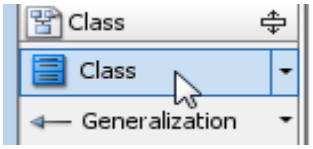
2.5.2. Δημιουργία διαγράμματος κλάσης

Κάνοντας δεξί click διάγραμμα κλάσης στον πλοηγό διαγραμμάτων και επιλέγοντας «Νέο Διάγραμμα κλάσης» από το αναδυόμενο μενού μπορούμε να δημιουργήσουμε ένα διάγραμμα κλάσης.



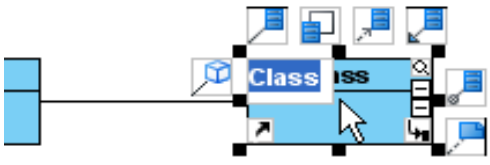
Εικόνα 2-13 Δημιουργία διαγράμματος κλάσης

Για να δημιουργήσουμε μια κλάση κάνουμε click στο κουμπί κλάση στη γραμμή εργαλείων και στη συνέχεια click στο διάγραμμα. Δώστε όνομα στην κλάση κάνοντας διπλό click επάνω στην νεοδημιουργηθείσα κλάση.



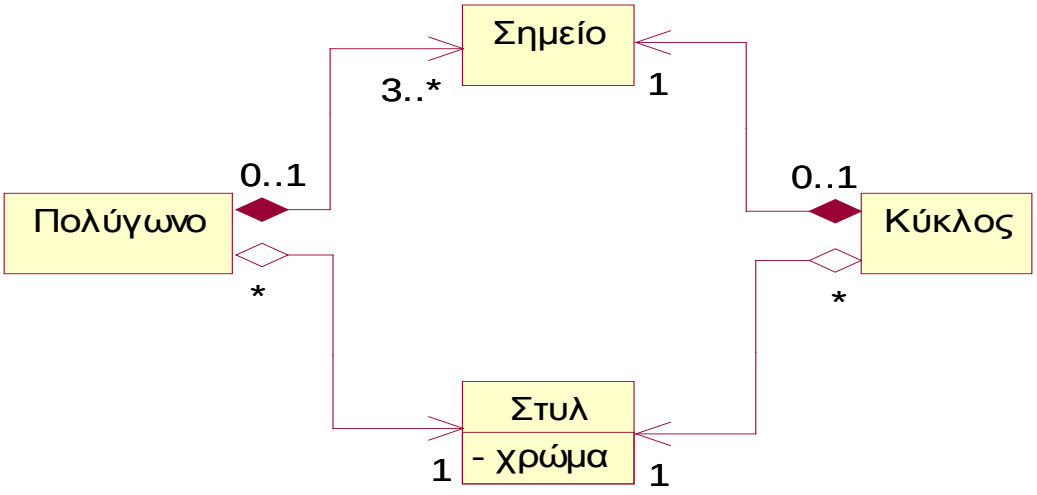
Εικόνα 2-14 Δημιουργία κλάσης

Για να δημιουργηθεί σύνδεση από την κλάση, μετακινούμε το ποντίκι στην κλάση ώστε να εμφανιστούν οι πόροι κάντε click στην ένωση και σέρνουμε το ποντίκι στο κενό χώρο. Έτσι δημιουργείται μια νέα κλάση, ή αν σύρουμε σε μια υπάρχουσα κλάση γίνεται σύνδεση με αυτήν.



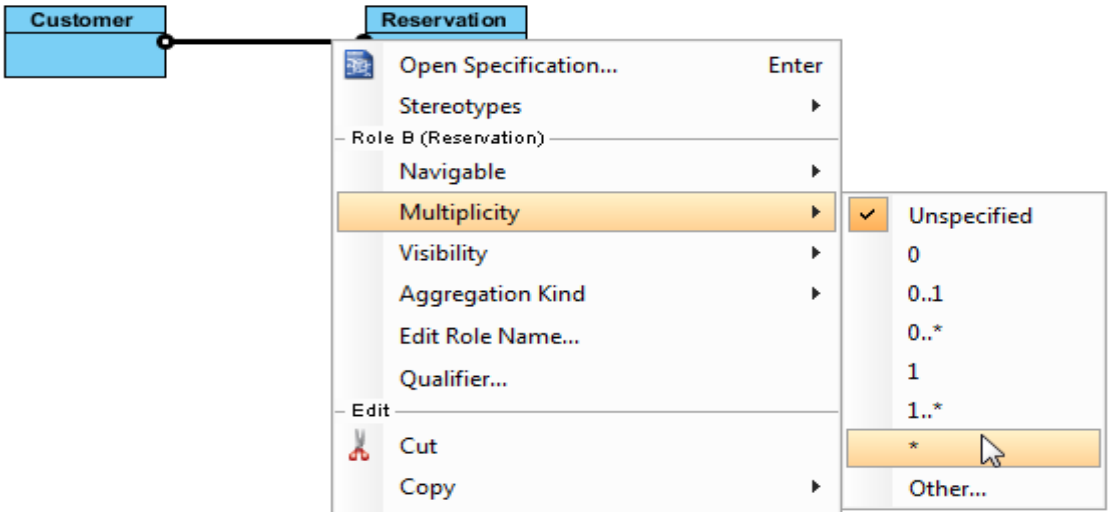
Εικόνα 2-15 Δημιουργία σύνδεσης κλάσης

Εκτός από την απλή σύνδεση υπάρχει και η *συσσωμάτωση, η σύνθεση και η γενίκευση*. Οι συνδέσεις στην κλάση "Σημείο" η οποία φαίνεται στο πιο κάτω σχήμα υποδηλώνουν ότι το σημείο αποτελεί τμήμα είτε ενός πολυγώνου είτε ενός κύκλου, αλλά καθώς η διαχείριση του πραγματοποιείται από αυτές τις κλάσεις, ένα σημείο δεν μπορεί να ανήκει ταυτοχρόνως και σε ένα Πολύγωνο και σε ένα Κύκλο. Η κλάση "Στυλ" ωστόσο, συσχετίζεται με συσσωμάτωση και κατά συνέπεια ένα στυλ μπορεί να μοιράζεται μεταξύ πολλών πολυγώνων και πολλών κύκλων. Επιπλέον, η διαγραφή ενός Πολυγώνου συνεπάγεται την διαγραφή των συσχετισμένων με αυτό Σημείων αλλά όχι την διαγραφή του συσχετισμένου με αυτό Στυλ.



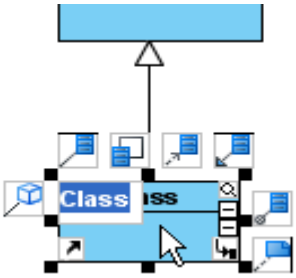
Σχήμα 2-1 Συνδέσεις στο διάγραμμα κλάσης

Στο πιο πάνω παράδειγμα βλέπουμε στις συσχετίσεις να υπάρχουν πολλαπλότητες, για παράδειγμα ο κύκλος έχει μόνο ένα σημείο και μόνο ένα στυλ. Για να μπορέσετε να επεξεργαστείτε τις πολλαπλότητες , κάντε δεξί click κοντά στο τέλος σύνδεσης, επιλέξτε **πολλαπλότητα** *(multiplicity)* από το αναδυόμενο μενού και στη συνέχεια επιλέξτε μια πολλαπλότητα.



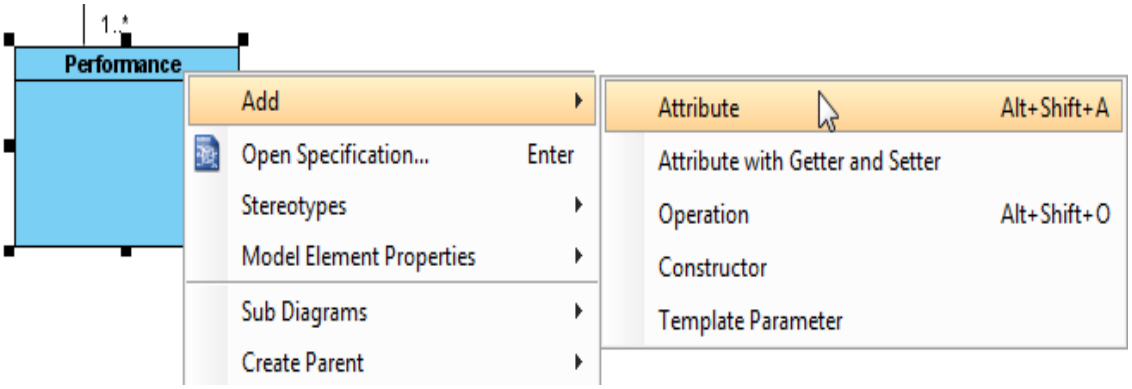
Εικόνα 2-16 Εισαγωγή πολλαπλότητας

Η τρίτη σχέση που προαναφέρθηκε είναι η γενίκευση, όπου σκοπός της γενίκευσης είναι η σταδιακή περιγραφή ενός στοιχείου, αξιοποιώντας την περιγραφή των προγόνων του, αρχή που συναντάται ως κληρονομικότητα. Για να δημιουργηθεί γενίκευση από την κλάση, μετακινούμε το ποντίκι προς την κλάση και όταν εμφανιστούν οι πόροι της κλάσης κάνουμε click στη Γενίκευση και σέρνουμε το ποντίκι δίπλα στον κενό χώρο.



Εικόνα 2-17 Δημιουργία Γενίκευσης

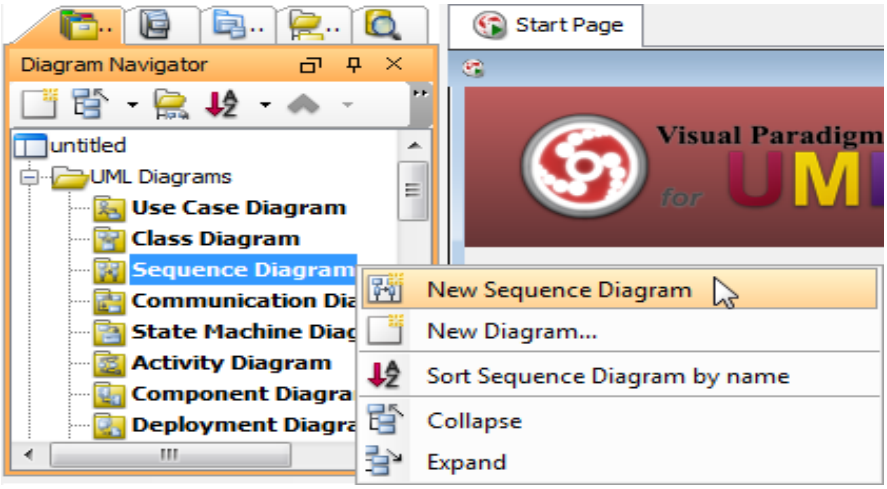
Η κάθε κλάση απαρτίζεται από ιδιότητες και λειτουργίες. Για την προσθήκη αυτών κάνουμε δεξί click **Προσθήκη-Ιδιότητες** *(add/attribute)* και πληκτρολογούμε το όνομα της ιδιότητας και με το enter γίνεται κατοχύρωση. Το ίδιο γίνεται και με τις λειτουργίες δεξί click **Προσθήκη-Λειτουργίες** *(add/operation)* πάλι πληκτρολογούμε το όνομα της λειτουργίας και με το enter το όνομα κατοχυρώνεται.



Εικόνα 2-18 Προσθήκη χαρακτηριστικών στην κλάση

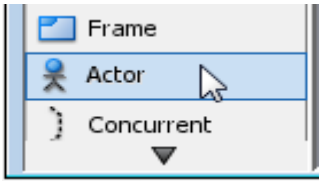
2.5.3. Δημιουργία διαγράμματος ακολουθίας

Χρησιμοποιώντας το διάγραμμα ακολουθίας μπορούμε να μοντελοποιήσετε τις λειτουργίες του συστήματος καθώς και τους χρήστες που αλληλεπιδρούν με αυτές τις λειτουργίες. Το VP-UML σας επιτρέπει την δημιουργία διαγραμμάτων ακολουθίας. Σε αυτή τη σελίδα, θα δείτε πώς μπορείτε να σχεδιάσετε ένα διάγραμμα ακολουθίας. Κάντε δεξί click στο διάγραμμα ακολουθίας που βρίσκεται στον περιηγητή διαγραμμάτων και επιλέξτε νέο διάγραμμα ακολουθίας από το αναδυόμενο μενού.



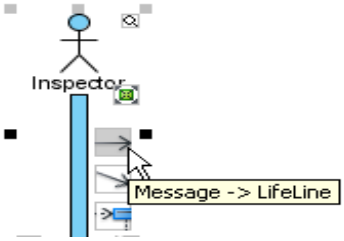
Εικόνα 2-19 Διάγραμμα ακολουθίας

Για να δημιουργήσουμε ένα χρήστη του συστήματος, κάνουμε click στο κουμπί χρήστη της γραμμής εργαλείων και στη συνέχεια click στο διάγραμμα.



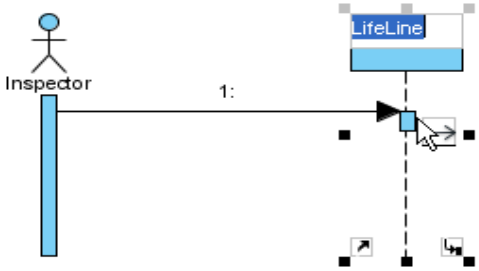
Εικόνα 2-20 Δημιουργία χρήστη

Για να δημιουργήσουμε μια γραμμή ζωής ενός αντικειμένου μπορούμε να κάνουμε click στην γραμμή ζωής (Lifeline) που βρίσκεται στη γραμμή εργαλείων και στη συνέχεια click στο διάγραμμα.



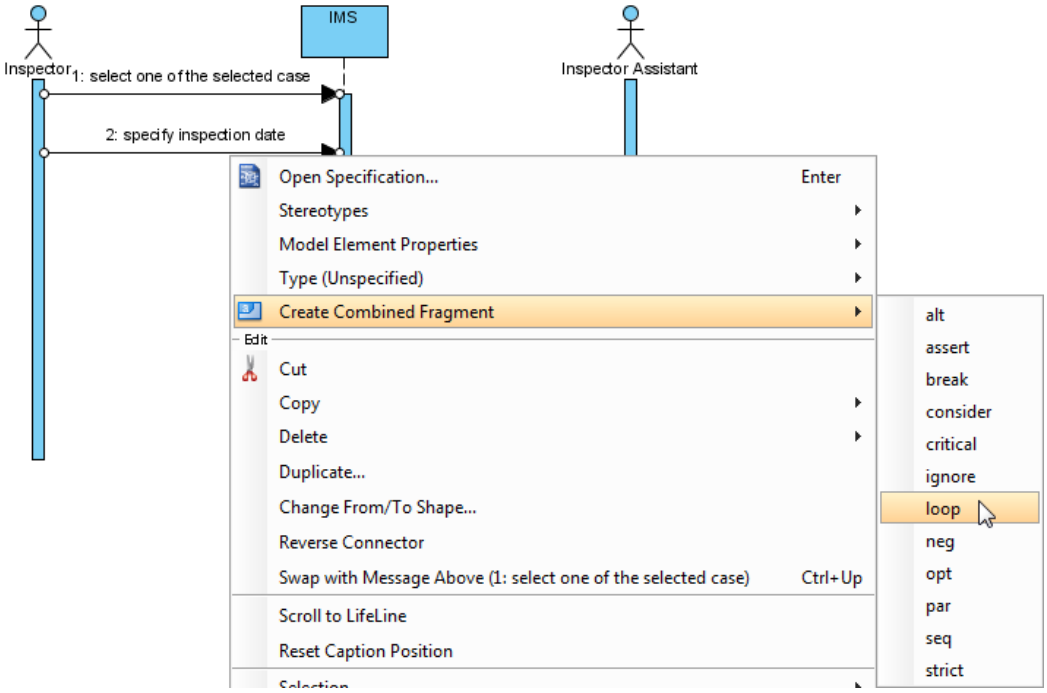
Εικόνα 2-21 Δημιουργία γραμμής ζωής

Εναλλακτικά, ένας πολύ πιο γρήγορος και πιο αποτελεσματικός τρόπος είναι να χρησιμοποιήσουμε του πόρους που εμφανίζονται στον χρήστη που δημιουργήσαμε. Κάνουμε click στον πόρο μήνυμα και σέρνουμε προς το αντικείμενο που θέλουμε να αλληλεπιδράσει με τον χρήστη μας.



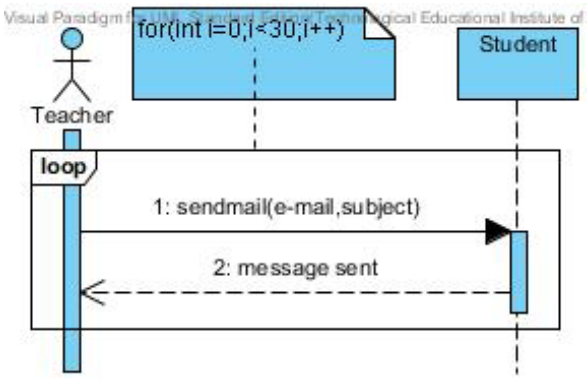
Εικόνα 2-22 Αλληλεπίδραση με αντικείμενο

Για να δημιουργήσουμε συνδυασμό από συνθήκες πρέπει να χρησιμοποιήσουμε κάποιες συγκεκριμένες καταστάσεις έτσι ώστε να ικανοποιούνται οι συνθήκες μας. Αυτό γίνεται, κάνοντας δεξί click στο διάγραμμα και επιλογή «Δημιουργία συνδυασμένων Τμημάτων». Στη συνέχεια επιλέγουμε τον τύπο που ικανοποιεί την συνθήκη (π.χ. βρόχο) από το αναδυόμενο μενού.



Εικόνα 2-23 Δημιουργία συνδυαζόμενου τμήματος

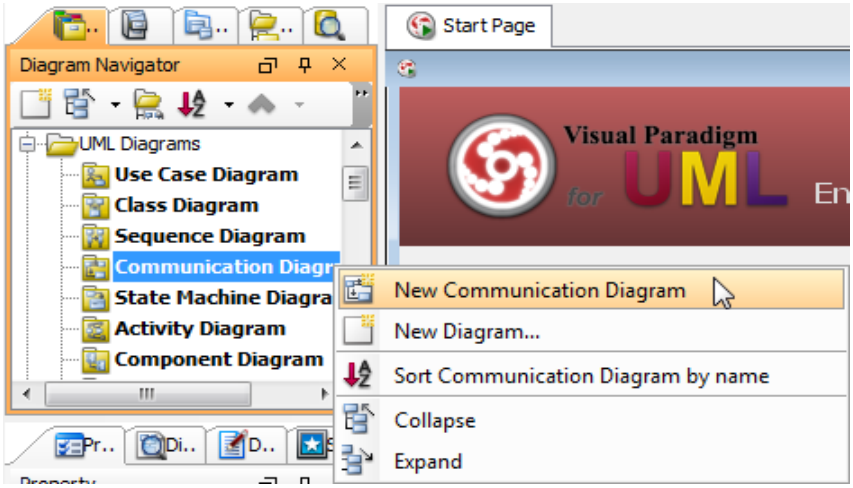
Ένα συνδυασμένο τεμάχιο του επιλεγμένου τύπου θα δημιουργηθεί για να καλύψει τα μηνύματα που αφορούν ένα ή περισσότερα αντικείμενα τα οποία αλληλεπιδρούν μεταξύ τους, ικανοποιώντας τις απαιτήσεις του συστήματος. Για παράδειγμα όταν ένας καθηγητής θέλει να στείλει ηλεκτρονικό μήνυμα σε όλα τα παιδιά της τάξης του ξεχωριστά στο καθένα θα πρέπει να χρησιμοποιηθεί βρόγχος επανάληψης με αριθμό επανάληψης ίσον με τον σύνολο των μαθητών.



Σχήμα 2-2 Παράδειγμα διαγράμματος ακολουθίας

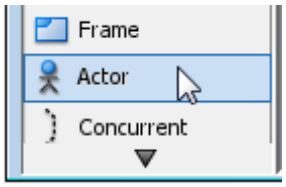
2.5.4. Δημιουργία διαγραμμάτων επικοινωνίας

Το διάγραμμα επικοινωνίας είναι σχεδιασμένο για να αποτυπώνει την δυναμική πλευρά του συστήματος. Δίνει έμφαση στην οργανωτική δομή των αντικειμένων όταν στέλνουν και λαμβάνουν μηνύματα. Για να δημιουργήσουμε ένα διάγραμμα επικοινωνίας κάνουμε δεξί click στο *διάγραμμα επικοινωνίας* το οποίο βρίσκετε στον πλοηγό διαγραμμάτων και επιλέγουμε «*νέο διάγραμμα επικοινωνίας*» από το αναδυόμενο μενού.



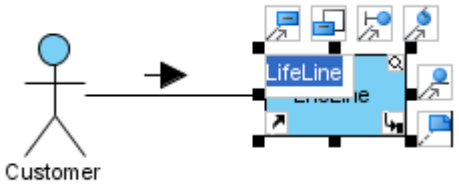
Εικόνα 2-24 Δημιουργία διαγράμματος επικοινωνίας

Για να δημιουργήσουμε ένα χρήστη του συστήματος, κάνουμε click στο κουμπί χρήστη της γραμμής εργαλείων και στη συνέχεια click στο διάγραμμα.



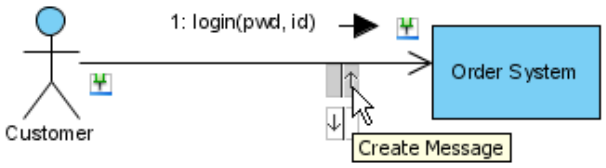
Εικόνα 2-25 Δημιουργία χρήστη

Για να δημιουργήσουμε μια γραμμή ζωής ενός, αντικειμένου μπορούμε να κάνουμε click στην γραμμή ζωής *(Lifeline)* που βρίσκεται στη γραμμή εργαλείων και στη συνέχεια κάνουμε click στο διάγραμμα. Εναλλακτικά, ένας πολύ πιο γρήγορος και πιο αποτελεσματικός τρόπος είναι να χρησιμοποιήσουμε τους πόρους που εμφανίζονται στον χρήστη που δημιουργήσαμε, κάνοντας click στον πόρο μήνυμα και σέρνοντας προς το αντικείμενο που θέλουμε να αλληλεπιδράσει με τον χρήστη μας.



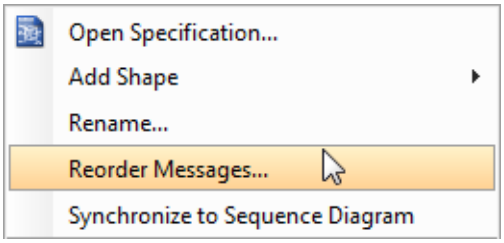
Εικόνα 2-26 Σύνδεση και δημιουργία μηνύματος

Για να δημιουργήσουμε μήνυμα στο σύνδεσμο, κάνουμε click στην επιλογή *Δημιουργία μηνύματος* του πόρου, όπως φαίνεται στην πιο κάτω εικόνα.



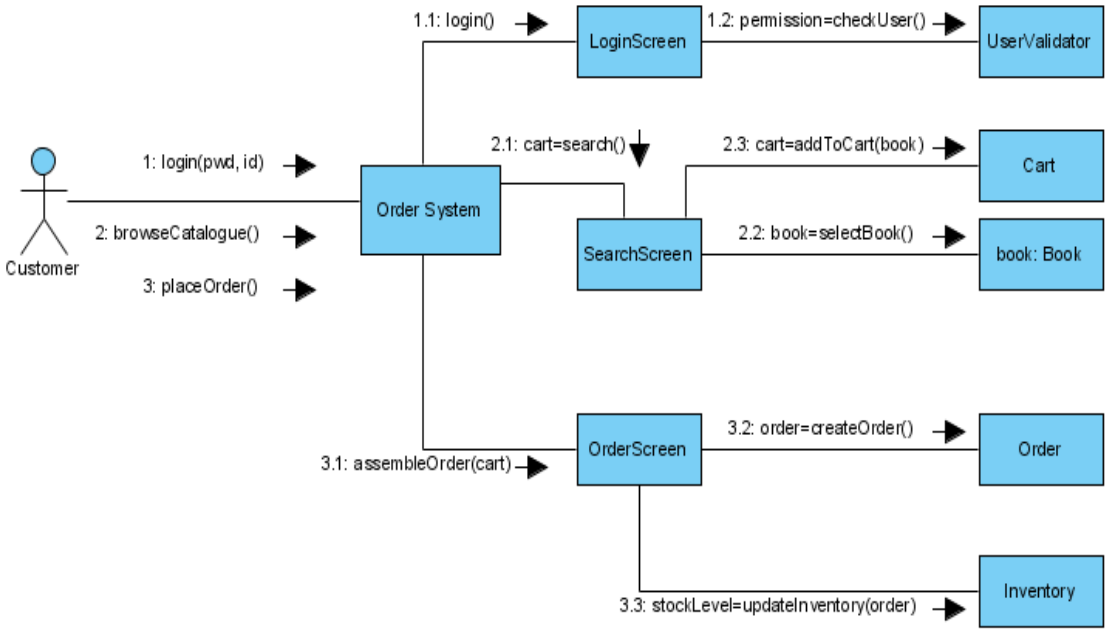
Εικόνα 2-27 Νέο μήνυμα στο σύνδεσμο

Για να επεξεργαστούμε τον αύξοντα αριθμό των μηνυμάτων, π.χ. αν θέλουμε να δείξουμε κάποια μηνύματα τα οποία έχουν ένθετο επίπεδο αλληλεπίδρασης, κάνουμε δεξί click στο διάγραμμα και επιλέγουμε *Αναδιάταξη Μηνυμάτων* από το αναδυόμενο μενού.



Εικόνα 2-28 Αναδιάταξη Μηνυμάτων

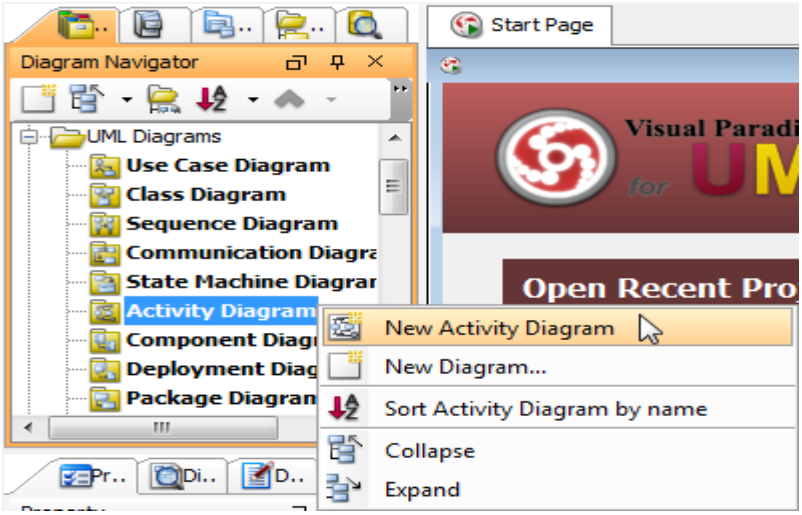
Ένα παράδειγμα διαγράμματος επικοινωνίας δίνετε στο ακόλουθο σχήμα. Μιας και η τοποθέτηση των αντικειμένων στον χώρο είναι τυχαία, οι ανταλλαγές μηνυμάτων θα πρέπει να αριθμηθούν έτσι ώστε να φαίνεται η σειρά τους.



Σχήμα 2-3 Παράδειγμα διαγράμματος επικοινωνίας

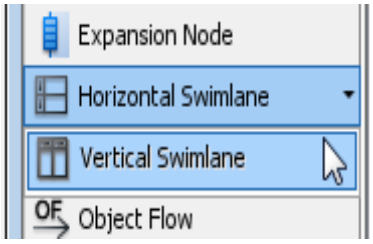
2.5.5. Δημιουργία διαγράμματος δραστηριοτήτων

Κάνουμε δεξί click στο *διάγραμμα δραστηριοτήτων* το οποίο βρίσκετε στον πλοηγό διαγραμμάτων και επιλέγουμε «*νέο διάγραμμα δραστηριοτήτων*» από το αναδυόμενο μενού για να δημιουργήσουμε ένα διάγραμμα δραστηριοτήτων.



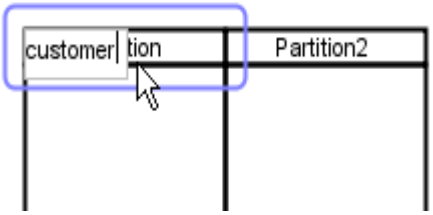
Εικόνα 2-29 Δημιουργία διαγράμματος δραστηριοτήτων

Μπορείτε να δημιουργήσουμε είτε οριζόντια είτε κάθετα τμήματα/λωρίδες *(Swimlane)* από την γραμμή εργαλείων όπως φαίνεται στην πιο κάτω εικόνα.



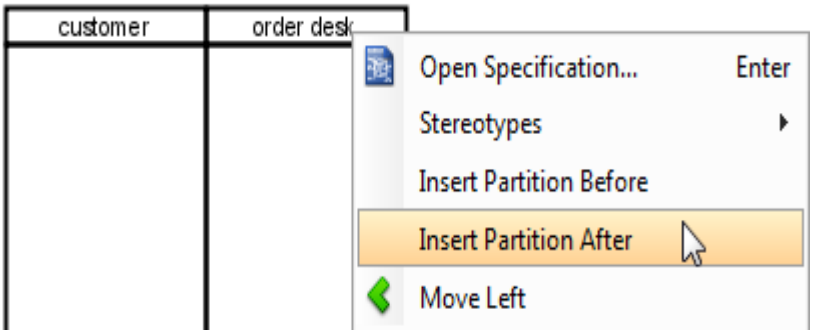
Εικόνα 2-30 Δημιουργία swimlane

Κάνοντας διπλό click στο όνομα του διαμερίσματος μπορείτε να το μετονομάσουμε ανάλογα με την κατάσταση που εκτελείται την δεδομένη στιγμή.



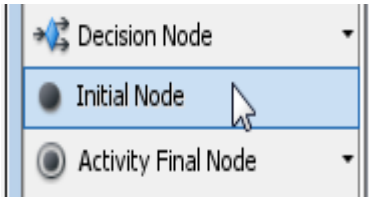
Εικόνα 2-31 Μετονομασία διαμερίσματος

Για να τοποθετήσουμε μια καινούργια λωρίδα, κάνουμε δεξί click σε ένα διαμέρισμα και από το αναδυόμενο μενού επιλέγουμε είτε εισαγωγή τμήματος πριν, είτε εισαγωγή τμήματος μετά, όπως φαίνεται πάρα κάτω.



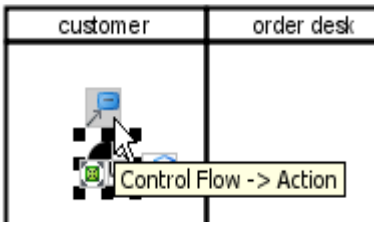
Εικόνα 2-32 Εισαγωγή τμήματος

Κάνοντας click στο κουμπί αρχικός κόμβος (initial node), που βρίσκεται στο μενού εργαλείων μπορούμε να τοποθετήσουμε το κόμβο σε όποια λωρίδα θέλουμε –η λωρίδα έχει ήδη δημιουργηθεί από πριν.



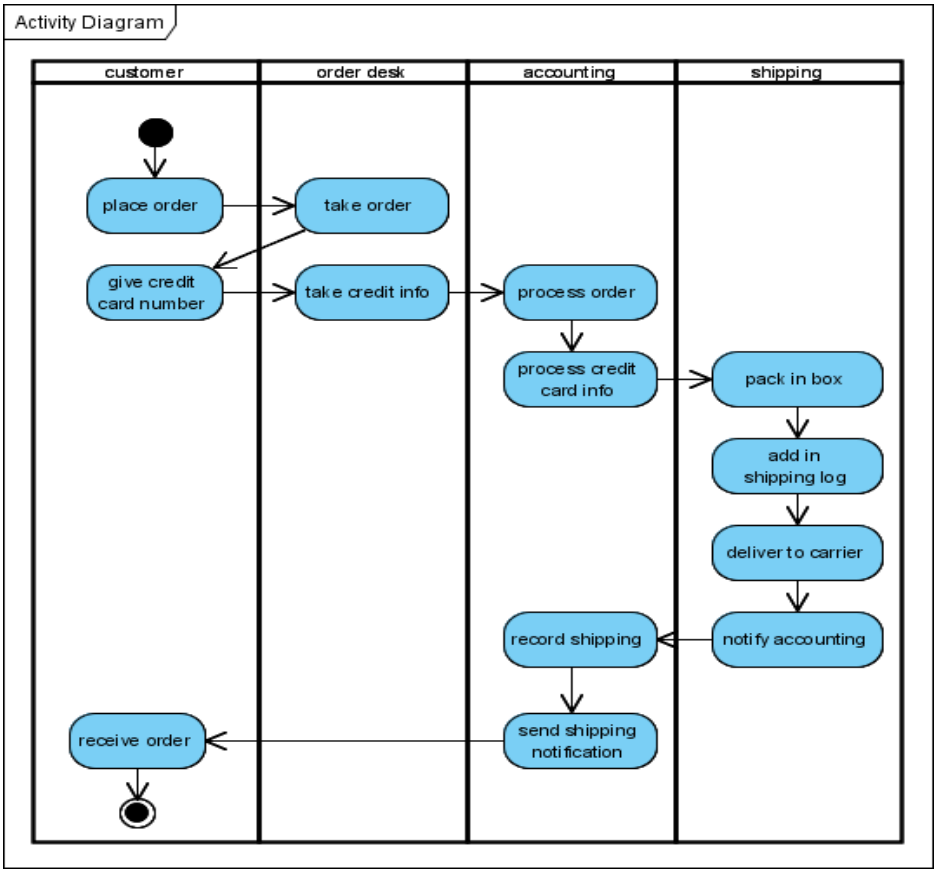
Εικόνα 2-33 Δημιουργία αρχικού κόμβου

Μετακινώντας το ποντίκι πάνω στον αρχικό κόμβο εμφανίζονται οι δυο πόροι με τους οποίους ο κόμβος μπορεί να συνδεθεί (*Control Flow→ Action* και *Generic Resource).* Κάνουμε click στο *Έλεγχος ροής →Δράση* και σέρνουμε το ποντίκι όπου θέλουμε να τοποθετήσουμε την ενέργεια.



Εικόνα 2-34 Δημιουργία δράσης

Με τον τρόπο αυτό δημιουργείται μια ενέργεια η οποία είναι συνδεδεμένη με το αρχικό κόμβο. Μπορούμε να ενώσουμε δραστηριότητες μεταξύ τους και να δημιουργηθεί μια σειρά ενεργειών που συνδέουν αρκετές δραστηριότητες μαζί δημιουργώντας το λεγόμενο διάγραμμα δραστηριοτήτων.

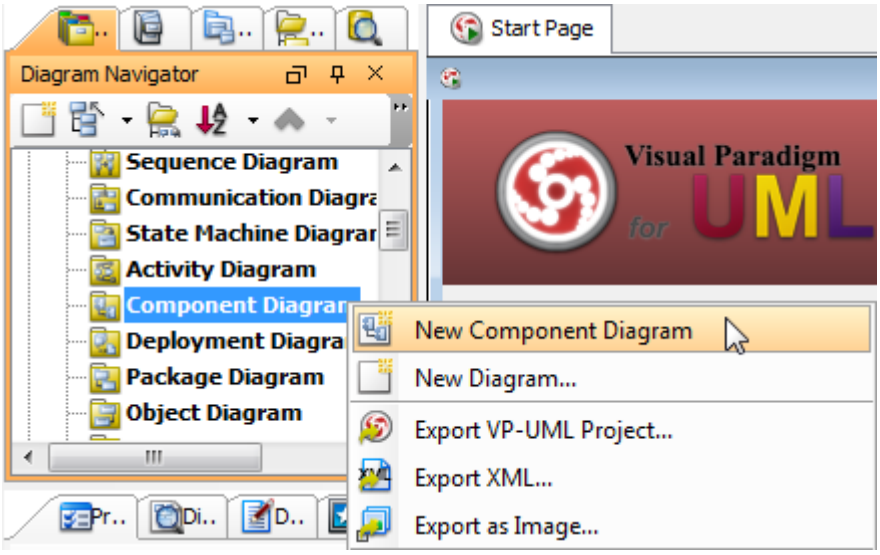


Σχήμα 2-4 Παράδειγμα διαγράμματος δραστηριοτήτων

Στο *Σχήμα 2-4* παρατηρούμε τις δραστηριότητες που εμπλέκονται για την δημιουργία παραγγελίας μέσω διαδικτύου. Το διάγραμμα χωρίζεται σε τέσσερα διαμερίσματα που λαμβάνουν μέρος στην παραγγελία, ο πελάτης *(customer),* το γραφείο παραγγελιών *(order desk),* το λογιστήριο *(accounting)* και το τμήμα φόρτωσης *(shipping).* Ακολουθώντας τον αρχικό κόμβο και τα βέλη που ορίζουν την ροή μεταξύ των δραστηριοτήτων, ο πελάτης κάνει μια παραγγελία το γραφείο παραγγελιών παίρνει την παραγγελία και ζητά από τον πελάτη να δώσει τον αριθμό της πιστωτικής του κάρτας. Ο πελάτης δίνει τον αριθμό, το γραφείο παραγγελιών παίρνει τα στοιχεία της κάρτας και προωθεί την παραγγελία στο λογιστήριο. Το λογιστήριο εξακριβώνει τα στοιχεία της κάρτας, και δίνει εντολή να πακετάρουν την παραγγελία στο τμήμα φόρτωσης. Το τμήμα φόρτωσης προσθέτει την παραγγελία στην λίστα φόρτωσης, και παραδίδει το πακέτο στον μεταφορέα. Ειδοποιείται το τμήμα λογιστικής για την μεταφορά, καταγράφει την φόρτωση στα αρχεία της και στέλνει ειδοποίηση στον πελάτη για την μεταφορά. Ο πελάτης λαμβάνει την παραγγελία και έτσι τελειώνει η διαδικασία παραγγελίας.

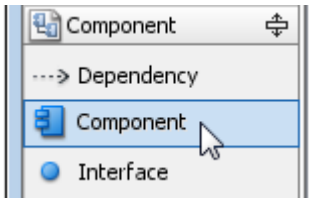
2.5.6. Δημιουργία διαγράμματος συστατικών

Το διάγραμμα συστατικών δείχνει τη φυσική πτυχή ενός αντικειμενοστρεφούς λογισμικού συστήματος. Καταδεικνύει την αρχιτεκτονική των στοιχείων λογισμικού και τις εξαρτήσεις μεταξύ τους. Κάνοντας δεξί click στο διάγραμμα συστατικών που βρίσκεται στον πλοηγό Διαγραμμάτων και επιλέγοντας Νέο Διάγραμμα Συστατικών από το αναδυόμενο μενού μπορούμε να δημιουργήσουμε ένα διάγραμμα συστατικών.



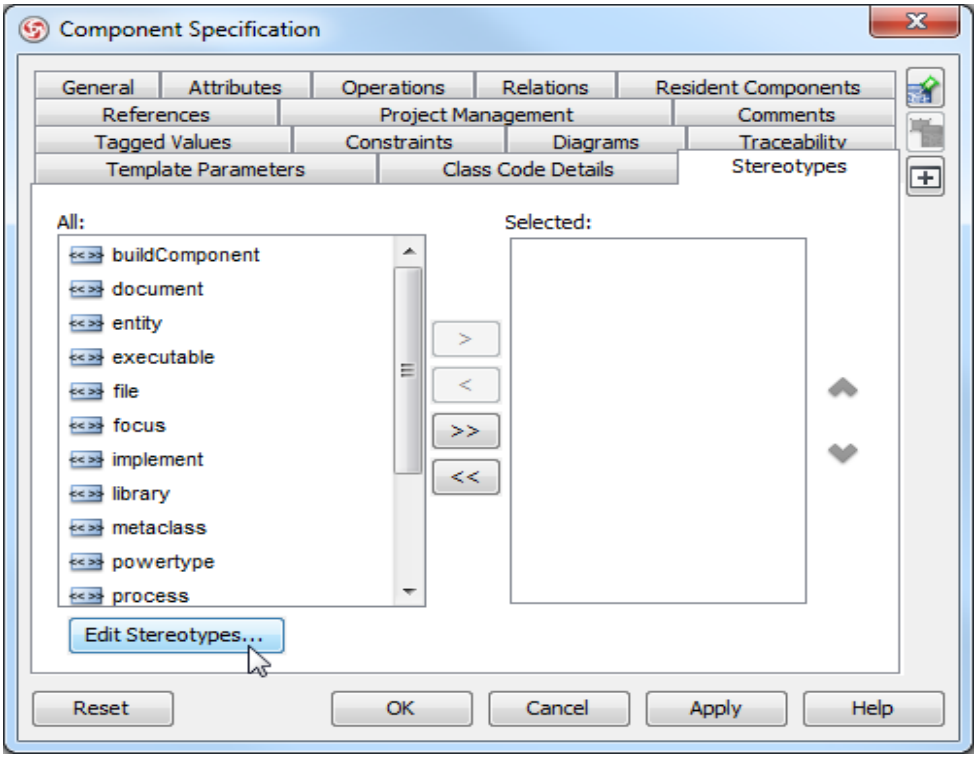
Εικόνα 2-35 Δημιουργία διαγράμματος συστατικών

Για να δημιουργήσουμε ένα στοιχείο/συστατικό *(component)*, κάνουμε click στην εντολή στοιχείο της γραμμής εργαλείων του διαγράμματος και στη συνέχεια click στο διάγραμμα.



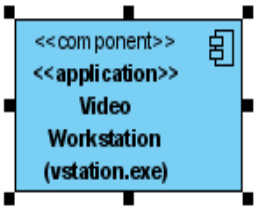
Εικόνα 2-36 Δημιουργία στοιχείου

Όταν το παράθυρο διαλόγου Προδιαγραφές Συστατικών *(Open Specification)* ανοίξει, η καρτέλα στερεότυπα ανοίγει από προεπιλογή. Η λίστα στα αριστερά εμφανίζει τα επιλεγόμενα στερεότυπα. Αν το στερεότυπο που θέλουμε να χρησιμοποιήσουμε δεν βρίσκεται στη λίστα, κάνουμε click στο κουμπί Επεξεργασία Στερεότυπων**.**



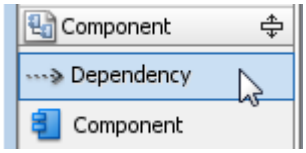
Εικόνα 2-37 Επεξεργασία στερεότυπων

Δημιουργούμε το στερεότυπο που θέλουμε και στην συνέχεια προσθέτουμε το στερεότυπο στο πακέτο. Αν για παράδειγμα, ονομάσουμε “application το στερεότυπο που δημιουργήσαμε, τότε θα εφαρμοστεί στο πακέτο μας όπως φαίνεται στο παρακάτω σχήμα.



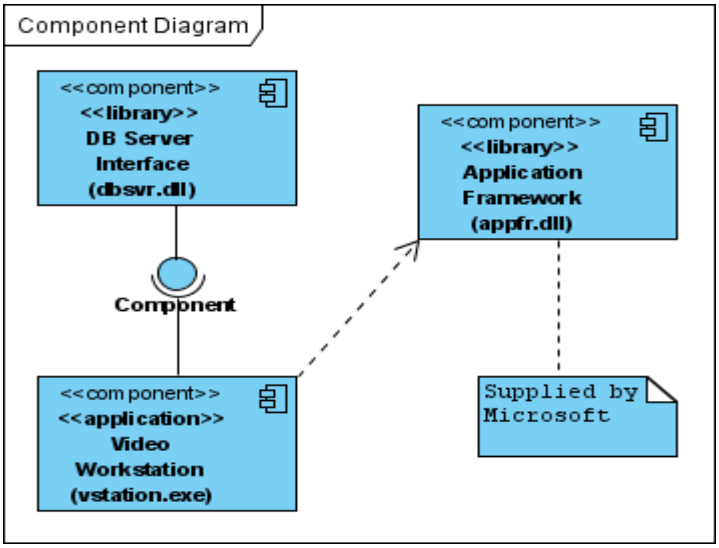
Εικόνα 2-38 Τα στερεότυπα έχουν ανατεθεί

Για να δημιουργήσουμε provided interface για ένα στοιχείο, μετακινούμε το ποντίκι πάνω από το συστατικό-στόχο και πατάμε το εικονίδιο πόρων του Realization→ Interface. Για να δημιουργήσουμε την εξάρτηση, κάνουμε click στην εξάρτηση από τη γραμμή εργαλείων των Διαγραμμάτων



Εικόνα 2-39 Δημιουργία εξάρτησης

Ένα παράδειγμα ολοκληρωμένου διαγράμματος συστατικών είναι το παρακάτω όπου φαίνονται οι εξαρτήσεις ,τα στερεότυπα και οι διεπαφές μεταξύ των συστατικών.



Σχήμα 2-5 Παράδειγμα ολοκληρωμένου Διαγράμματος Συστατικών

2.6. Μετατροπή κώδικα C++ με VP UML

Στην ενότητα αυτή θα δοθεί ένα απλό παράδειγμα κώδικα έτσι ώστε να γίνει κατανοητή η ανάλυση και μετατροπή αρχείων κώδικα από μία αντικειμενοστρεφή γλώσσα προγραμματισμού, που στην περίπτωση αυτή θα είναι η C++ σε διαγράμματα UML.. Το παράδειγμα που ακολουθεί είναι ένα μοντέλο βάσης δεδομένων των υπαλλήλων μιας εταιρείας κατασκευής συσκευών. Στην εταιρεία αυτή υπάρχουν τρεις κατηγορίες υπαλλήλων: οι διευθυντές που διευθύνουν , οι επιστήμονες που κάνουν έρευνα για να αναπτύξουν καλύτερες συσκευές και οι εργάτες που χειρίζονται τις μηχανές κατασκευής συσκευών.

Η βάση δεδομένων περιέχει το όνομα και τον αριθμό ταυτότητας κάθε υπαλλήλου ανεξάρτητα από την κατηγορία στην οποία ανήκει. Για τους διευθυντές αποθηκεύει επιπλέον τους τίτλους τους και τις συνδρομές στην λέσχη του γκολφ. Για τους

επιστήμονες αποθηκεύει τον αριθμό των επιστημονικών άρθρων που έχουν δημοσιεύσει. Για τους εργάτες δεν χρειάζεται τίποτα άλλο εκτός από το όνομα και τον αριθμό ταυτότητας.

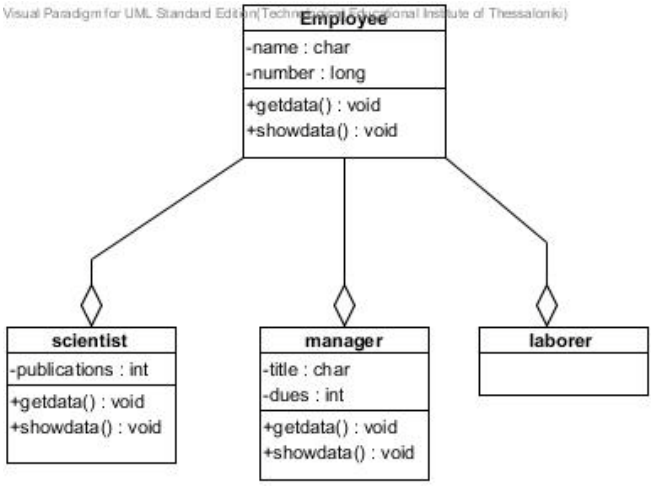
Η λίστα του προγράμματος C++, που υλοποιεί το μοντέλο βάσης δεδομένων των υπαλλήλων με την χρήση κληρονομικότητας είναι η παρακάτω:.

```
#include <iostream>
using namespace std;
const int size=80;
class employee // κλάση υπάλληλος
{
	private:
		char name[size];
		unsigned long number;
	public:
		void getdata()
		{
		cout << "Enter name  \n";
		cin >> name;
		cout << "Enter number of id \n";
		cin >> number;
		}
		void showdata() const
		{
		cout << "\n Name is : " << name << "\n";
		cout << "\n Number of id is : " << number << "\n";
		}
};
//////////////////////////////////////////////////////////////////////////
class laborer: public employee //κλάση εργάτης
{

};
//////////////////////////////////////////////////////////////////////////
class manager : public employee //κλάση διευθυντή
{
	private:
		char title[size];
		double dues;
	public:
		void getdata()
		{
```

```
                    employee::getdata();
                    cout << "Enter the title : \n";
                    cin >> title;
                    cout << "Enter the dues : \n";
                    cin >> dues;
            }
            void showdata() const
            {
                    cout << "MANAGER\n";
                    employee::showdata();
                    cout << "\n Title  is : " << title  << "\n";
                    cout << "\n Dues  are : " << dues  << "\n";
            }

};
//////////////////////////////////////////////////////////////////////
class scientist: public employee //κλάση επιστήμονα
{
      private:
            int publications;
      public:
            void getdata()
            {
                    employee::getdata();
                    cout << "\nEnter the no of publications : ";
                    cin >> publications;
            }
            void showdata() const
            {
                    cout << "SIENTIST\n";
                    employee::showdata();
                    cout << "number of publications : ";
                    cout << publications<<"\n";
            }
};
//////////////////////////////////////////////////////////////////////
#include "employee.cpp"
#include "laborer.cpp"
#include "manager.cpp"
#include "scientist.cpp"
#include <cstdlib>
#include <iostream>
#include <conio.h>
using namespace std;
```

```
int main()
{

	manager m1;
	manager m2;
	scientist  s;
	labour l;

	cout << "MANAGER GET DATA\n";
	m1.getdata();
	m2.getdata();


	cout << "SCIENTIST GET DATA\n";
	s.getdata();


	cout << "LABOUR GET DATA\n";
	l.getdata();
	cout << "\n SHOW DATA\n";
	m1.showdata();
	m2.showdata();
	s.showdata();
	l.showdata();

	system("Pause");
	return 0;

}
```

Από το πιο πάνω απόσπασμα κώδικα δημιουργούνται τέσσερεις κλάσεις. Η βασική κλάση employee χειρίζεται το επώνυμο και τον αριθμό ταυτότητας όλων το υπαλλήλων. Από αυτήν την κλάση παράγονται τρεις άλλες κλάσεις: ο manager *(διευθυντής),* ο scientist *(επιστήμονας)* και ο laborer *(εργάτης).* Οι κλάσεις περιέχουν πρόσθετες πληροφορίες για τις αντίστοιχες κατηγορίες υπαλλήλων καθώς και συναρτήσεις-μέλη για το χειρισμό αυτών των πληροφοριών, όπως φαίνεται στο *Σχήμα 2-6*.



Σχήμα 2-6 Διάγραμμα κλάσεων Employee

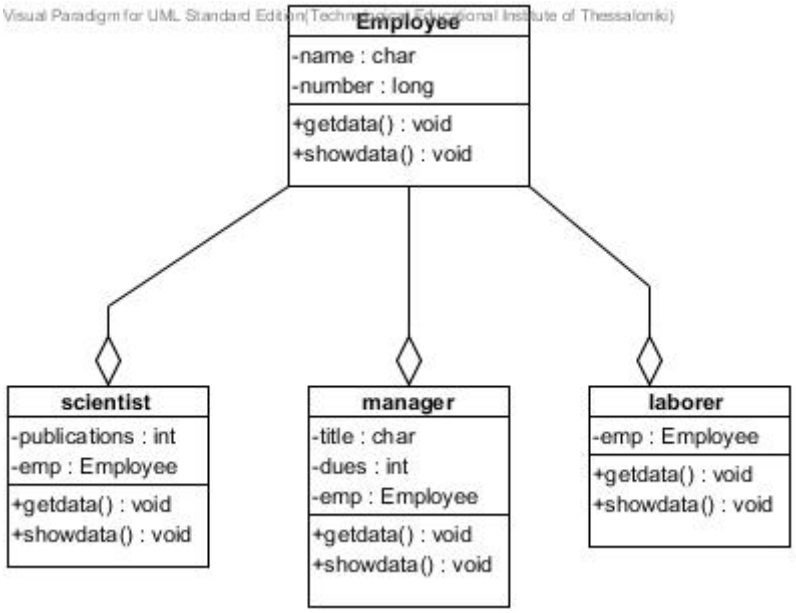
Τον πιο πάνω κώδικα employee μπορούμε να τον τροποποιήσουμε και αντί κληρονομικότητας να χρησιμοποιήσουμε συσσώρευση, δηλαδή οι κλάσεις *manager*, *scientist* και *laborer* να περιέχουν παρουσίες της κλάσης *employee* ως χαρακτηριστικά τους. Ο τροποποιημένος κώδικας φαίνεται πιο κάτω :

```
#include <iostream>
using namespace std;
const int size=80;
class employee // κλάση υπάλληλος
{
      private:
            char name[size];
            unsigned long number;
      public:
            void getdata()
            {
            cout << "Enter name  \n";
            cin >> name;
            cout << "Enter number of id \n";
            cin >> number;
            }
            void showdata() const
            {
            cout << "\n Name is : " << name << "\n";
            cout << "\n Number of id is : " << number << "\n";
            }
};
////////////////////////////////
```

```
class scientist
        private:
                int publications;
                employee emp ;// το emp είναι αντικείμενο της κλάσης employee
        public:
                void getdata()
                {
                        emp.getdata();
                        cout << "\nEnter the no of publications : ";
                        cin >> publications;
                }
                void showdata() const
                {
                        cout << "SIENTIST\n";
                        emp.showdata();
                        cout << "number of publications : ";
                        cout << publications<<"\n";
                }
};
//////////////////////////////////////////////////////////
class manager
{
        private:
                char title[size];
                double dues;
                employee emp ;// το emp είναι αντικείμενο της κλάσης employee
        public:
                void getdata()
                {
                        emp.getdata();
                        cout << "Enter the title : \n";
                        cin >> title;
                        cout << "Enter the dues : \n";
                        cin >> dues;
                }
                void showdata() const
                {
                        cout << "MANAGER\n";
                        emp.showdata();
                        cout << "\n Title  is : " << title  << "\n";
                        cout << "\n Dues  are : " << dues  << "\n";
                }

};
```

```
//////////////////////////////////////////////////////////////////////
class laborer
{
private :
    employee emp; // το emp είναι αντικείμενο της κλάσης employee
public :
     void getdata()
    {
         emp.getdata();
    }
    void showdata()
    {
         emp.showdata();
    }
};
```

Σε κάθε κλάση έχει δηλωθεί ένα αντικείμενο emp τύπου *employee* όπου μέσω του αντικειμένου αυτού καλείται η ανάλογη μέθοδος της κλάσης *employee*. Αυτό σχηματικά παρουσιάζεται στο *Σχήμα 2-7*



Σχήμα 2-7 Διάγραμμα κλάσεων-συσσώρευση

Στο τμήμα της main(), δηλώνονται τέσσερα αντικείμενα διαφορετικών κλάσεων, καλούνται οι συναρτήσεις-μέλη getdata() για να δώσουν πληροφορίες για κάθε υπάλληλο

και τέλος η συνάρτηση showdata() για να εμφανίσει αυτές τις πληροφορίες. Πιο κάτω δίνεται η main() και πως από αυτή προκύπτει το διάγραμμα ακολουθίας, δηλαδή η ανταλλαγή μηνυμάτων μεταξύ των αντικειμένων. Στην main() κάθε "cout" δηλώνει μια επιστροφή κλήσης ,δηλαδή εμφάνιση ενός μηνύματος από το σύστημα και κλήση μίας λειτουργίας .

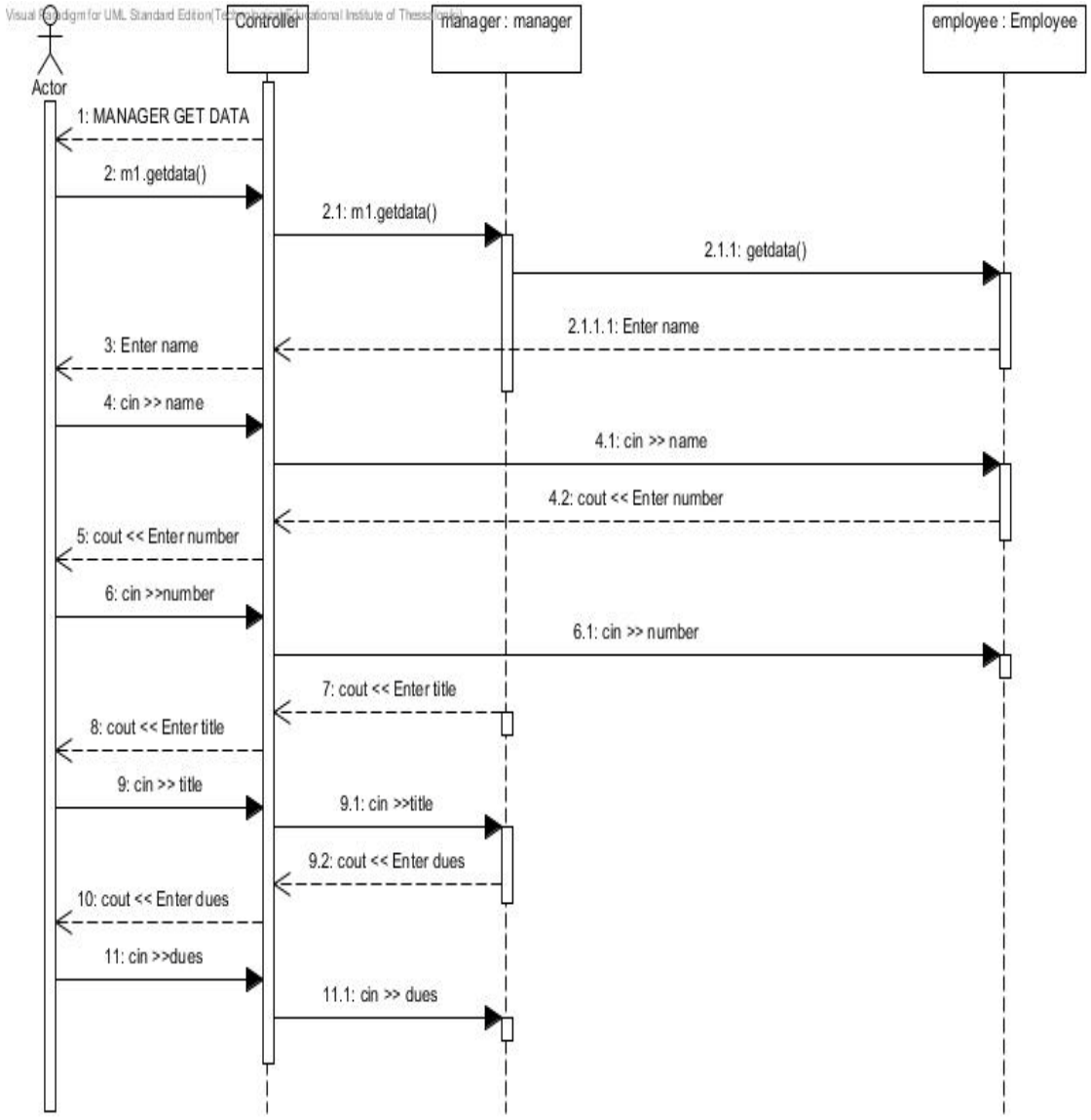
```
1.  int main()
2.  {
3.
4.  manager m1;
5.  manager m2;
6.  scientist  s;
7.  labour l;

8.  cout << "MANAGER GET DATA\n";
9.  m1.getdata();
10. m2.getdata();

11. cout << "SCIENTIST GET DATA\n";
12. s.getdata();

13. cout << "LABORER GET DATA\n";
14. l.getdata();
15. cout << "\n SHOW DATA\n";
16. m1.showdata();
17. m2.showdata();
18. s.showdata();
19. l.showdata();

20. system("Pause");
21. return 0;
22. }
```

Στις γραμμές 4-7 δηλώνονται τα τέσσερα αντικείμενα, στην γραμμή 8 γίνεται η πρώτη επιστροφή κλήσης από τον controller.

Στην γραμμή 9-10 καλείται η μέθοδος getdata() της κλάσης manager , η οποία υλοποιείται στην κλάση employee. Στην γραμμή 11 πάλι γίνεται επιστροφή κλήσης για να δώσει ο χρήστης τα στοιχεία του επιστήμονα και στην συνέχεια καλείται η μέθοδος

getdata() η οποία υλοποιείται στην κλάση employee. Το ίδιο γίνεται και με τους εργαζόμενους.

Στην συνέχεια ο controller στην γραμμή 15 επιστρέφει μήνυμα στον χρήστη ότι θα εμφανιστούν τα δεδομένα. Στις γραμμές 16-19 καλείται η μέθοδος showdata() της κάθε κλάσης ξεχωριστά ανάλογα με το αντικείμενο που καλεί την μέθοδο. Στο πιο κάτω Σχήμα 2-8 φαίνεται το διάγραμμα ακολουθίας για την περίπτωση χρήσης ΠΧ-Λήψη δεδομένων για τους υπαλλήλους τύπου manager.



Σχήμα 2-8 Διάγραμμα ακολουθίας ΠΧ- Λήψη δεδομένων

Το διάγραμμα συνεργασίας ή επικοινωνίας είναι ισοδύναμο με ένα διάγραμμα ακολουθίας. Η διαφορά τους έγκειται στο γεγονός ότι τα αντικείμενα μπορούν να είναι διάσπαρτα στο χώρο. Επειδή ακριβώς έχουμε τυχαία τοποθέτηση των αντικειμένων στον χώρο οι ανταλλαγές μηνυμάτων θα πρέπει να αριθμηθούν έτσι ώστε να φαίνεται η σειρά τους.

Στο *Σχήμα 2-9* όπου φαίνεται το διάγραμμα επικοινωνίας της ΠΧ- Λήψη δεδομένων *manager*, οι αριθμοί δηλώνουν την σειρά με την οποία ανταλλάσσονται τα μηνύματα μεταξύ των αντικειμένων. Παρακάτω εμφανίζονται με την σειρά τα μηνύματα :

1. Ο controller εμφανίζει στην οθόνη του χρήστη μήνυμα για την λήψη δεδομένων.
2. Καλείται η μέθοδος m1.getdata().
    2.1. Η μέθοδος m1.getdata() καλείται μέσω του αντικειμένου m1 το οποίο είναι τύπου manager.
        2.1.1. Η μέθοδος getdata() της κλάσης manager καλεί την μέθοδο getdata() μέσω του αντικειμένου emp το οποίο είναι τύπου employee.
            2.1.1.1. Η κλάση *employee* επιστρέφει μήνυμα για εισαγωγή του ονόματος του manager
3. Ο controller εμφανίζει μήνυμα επιστροφής στον χρήστη για να εισάγει το όνομα του manager.
4. Ο χρήστης πληκτρολογεί το όνομα και αποθηκεύεται στην μεταβλητή name.
5. Ο controller εμφανίζει μήνυμα επιστροφής στον χρήστη για να εισάγει τον αριθμό ταυτότητας του manager.
6. Ο χρήστης πληκτρολογεί τον αριθμό και αποθηκεύεται στην μεταβλητή number.
7. Εφόσον εκτελεστεί η μέθοδος getdata() της κλάσης employee συνεχίζει το πρόγραμμα στην μέθοδο getdata() της κλάσης manager.
8. Ο controller εμφανίζει μήνυμα επιστροφής στον χρήστη για να εισάγει τον τίτλο του manager.
9. Ο χρήστης πληκτρολογεί τον τίτλο.

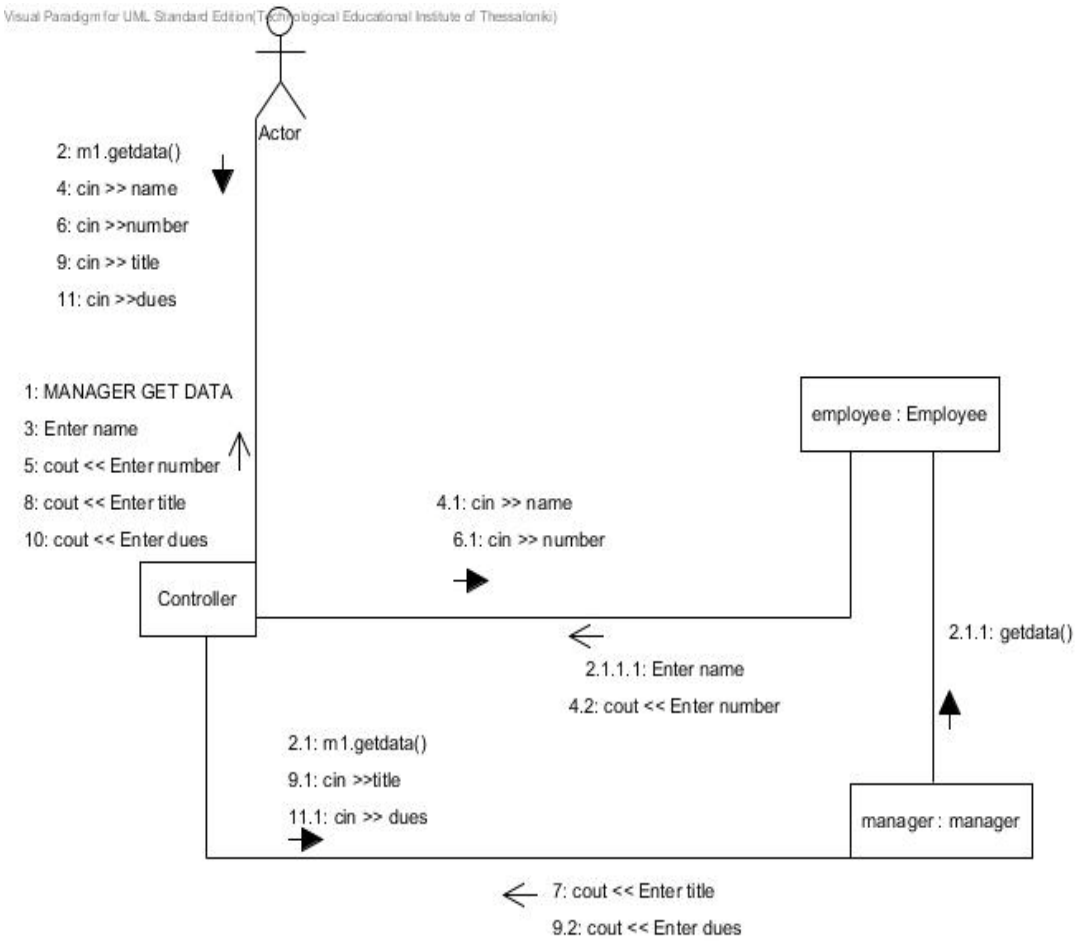
9.1. Η μεταβλητή title πλέον παίρνει την τιμή που πληκτρολόγησε ο χρήστης.

9.2. Εμφανίζεται μήνυμα επιστροφής για να εισάγει το ποσό πληρωμής του manager.

10. Ο controller εμφανίζει μήνυμα επιστροφής στον χρήστη για να εισάγει το ποσό πληρωμής του manager.

11. Ο χρήστης πληκτρολογεί το ποσό.

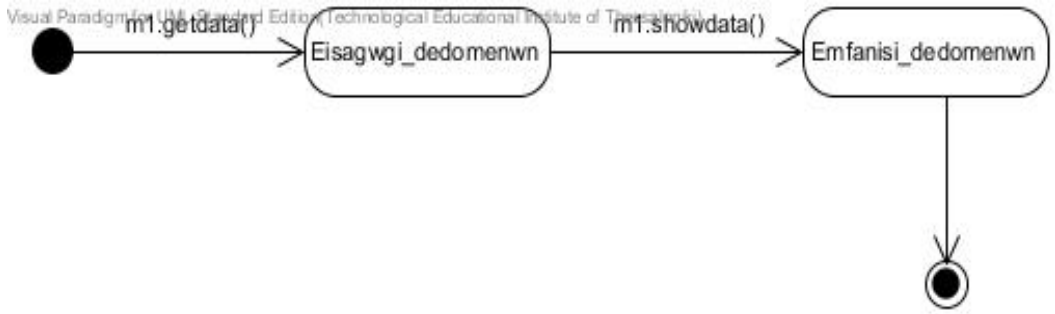
11.1. Το ποσό αποθηκεύεται στην μεταβλητή dues.



Σχήμα 2-9 Διάγραμμα επικοινωνίας ΠΧ-Λήψη δεδομένων manager

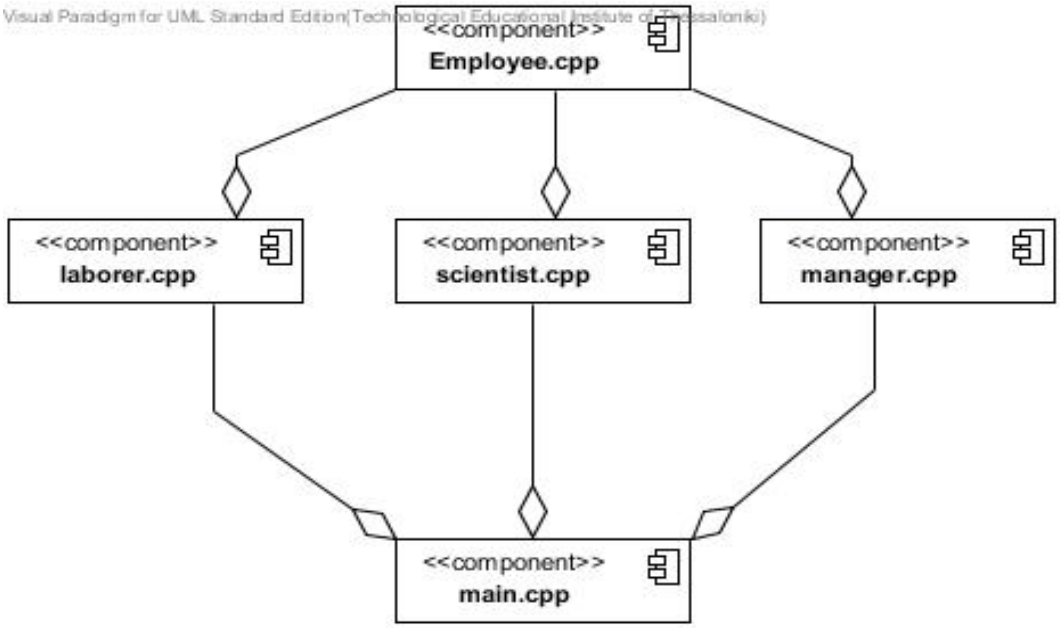
Στο προηγούμενο απόσπασμα κώδικα, στο τμήμα main() του προγράμματος γίνεται κλήση της μεθόδου getdata() από το αντικείμενο m1 και γίνεται εισαγωγή των δεδομένων του manager. Εφόσον περαστούν τα δεδομένα γίνεται κλήση της μεθόδου

showdata() από το αντικείμενο m1 και εμφανίζονται όλα τα στοιχεία του υπαλλήλου manager. Αυτό φαίνεται από το πιο κάτω Σχήμα 2-10 το διάγραμμα δραστηριοτήτων.



Σχήμα 2-10 Διάγραμμα δραστηριοτήτων ΠΧ-Λήψη δεδομένων manager

Έχοντας υπόψη ότι κάθε αρχείο (.cpp) της C++ αποτελεί ένα ξεχωριστό συστατικό, προκύπτει από το παράδειγμα μας που είναι ένα μοντέλο βάσης δεδομένων για υπαλλήλους μιας εταιρείας το ακόλουθο σχήμα. Το πρόγραμμα του παραδείγματος μας αποτελείται από τέσσερις κλάσεις συνεπώς τέσσερα διαφορετικά αρχεία (.cpp), τα οποία έχουν γίνει όπως φαίνεται στο Σχήμα 2-11 συστατικά της εφαρμογής.



Σχήμα 2-11 Διάγραμμα συστατικών

ΚΕΦΑΛΑΙΟ 3: ΣΥΓΚΡΙΣΗ VISUAL PARADIGM FOR UMLME ΤΟ UMBRELLO UML MODELER

3.1. Εισαγωγή

Το Umbrello UML Modeller είναι ένα εργαλείο δημιουργίας διαγραμμάτων UML και μπορεί να υποστηρίξει τη διαδικασία ανάπτυξης λογισμικού. Ειδικά κατά τη διάρκεια του σχεδιασμού και την φάση της ανάλυσης, το εργαλείο Umbrello UML Modeller παρέχει ένα προϊόν υψηλής ποιότητας.

Ένα καλό μοντέλο του λογισμικού είναι ο καλύτερος τρόπος επικοινωνίας με άλλους προγραμματιστές που εργάζονται στο ίδιο έργο μ' εμάς και με τους πελάτες μας

Η UML είναι γλώσσα δημιουργίας διαγραμμάτων που χρησιμοποιείται για την περιγραφή των μοντέλων αυτών. Μέσω αυτής της γλώσσας μπορούμε να παρουσιάσουμε τις ιδέες μας στην UML χρησιμοποιώντας διαφορετικούς τύπους διαγραμμάτων. Το εργαλείο Umbrello UML Modeller 1.2 υποστηρίζει τα εξής:

- Class Diagram Διάγραμμα κλάσεων
- Sequence Diagram Διάγραμμα ακολουθίας
- Collaboration Diagram Διάγραμμα συνεργασίας
- Use Case Diagram Διάγραμμα περίπτωσης χρήσης
- State Diagram Διάγραμμα καταστάσεων
- Activity Diagram Διάγραμμα δραστηριοτήτων
- Component Diagram Διάγραμμα συστατικών
- Deployment Diagram Διάγραμμα ανάπτυξης

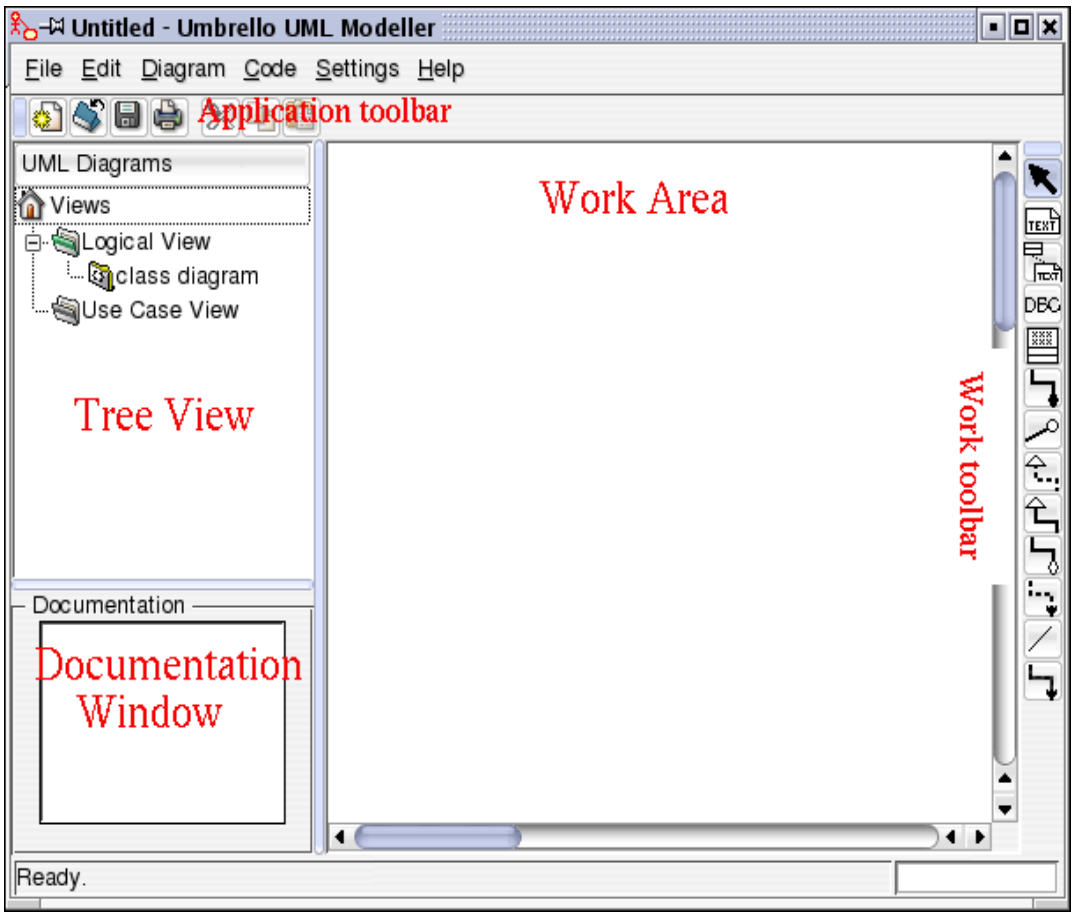
3.2. Διεπαφή χρήστη

Σε αυτήν την ενότητα παρουσιάζεται η διεπαφή χρήστη του εργαλείου Umbrello UML Modeller καθώς και όλα όσα πρέπει να γνωρίζει κάποιος για να ξεκινήσει τη μοντελοποίηση. Όλες οι ενέργειες στο Umbrello UML Modeller είναι προσβάσιμες μέσω του μενού και τις γραμμές εργαλείων. Μπορούμε να κάνουμε δεξί click στο ποντίκι επάνω στην επιφάνεια εργασίας του Umbrello UML Modeller για να έχουμε πρόσβαση σε όλα τα στοιχεία του εργαλείου ή από την γραμμή εργαλείων μπορούμε να πάρουμε ένα μενού με τις πιο χρήσιμες λειτουργίες που μπορούν να εφαρμοστούν για το συγκεκριμένο στοιχείο που εργαζόμαστε.

Το κύριο παράθυρο του Umbrello UML Modeller χωρίζεται σε τρεις περιοχές που βοηθούν να γίνει επισκόπηση ολόκληρου του συστήματος καθώς και άμεση πρόσβαση στα διάφορα διαγράμματα ενώ εργαζόμαστε με το μοντέλο μας.

Οι τρεις περιοχές του Umbrello UML Modeller είναι:

- Tree View
- Work Area
- Documentation Window



Εικόνα 3-1 Umbrello UML Modeller Διεπαφή χρήστη

Η περιοχή *Tree View* βρίσκεται συνήθως στην πάνω αριστερή πλευρά του παραθύρου και δείχνει το σύνολο των διαγραμμάτων, τις κλάσεις, τους φορείς και περιπτώσεις χρήσης που ενισχύουν το μοντέλο μας. Το Tree View επιτρέπει να έχουμε μια γρήγορη επισκόπηση των στοιχείων που συνθέτουν το μοντέλο μας. Ακόμη το Tree View μας δίνει ένα γρήγορο τρόπο εναλλαγής μεταξύ των διαφορετικών διαγραμμάτων στο μοντέλο μας καθώς και την δυνατότητα εισαγωγής στοιχείων από το μοντέλο μας στο τρέχον διάγραμμα.

Η περιοχή *Documentation Window* είναι το μικρό παράθυρο που βρίσκεται στο αριστερό κάτω μέρος της Umbrello UML Modeller, και μας δίνει μια γρήγορη προεπισκόπηση των εγγράφων για το επιλεγμένο στοιχείο. Η *Documentation Window* επιτρέπει την προβολή εγγράφων με περισσότερες λεπτομέρειες, μπορούμε δε να ανοίξουμε τις ιδιότητες του στοιχείου όποτε αυτό χρειαστεί.

Η περιοχή *Work Area* είναι το κύριο παράθυρο στο Umbrello UML Modeller και σε αυτήν γίνονται όλες οι ενέργειες που αφορούν το μοντέλο. Μπορούμε να χρησιμοποιήσουμε την *Work Area* για να επεξεργαστούμε και να προβάλουμε τα διαγράμματα στο μοντέλο μας. Η περιοχή *Work Area* δείχνει το τρέχον ενεργό διάγραμμα. Επί του παρόντος, μόνο ένα διάγραμμα μπορεί να αναδειχθεί στην περιοχή *Work Area* ανά πάσα στιγμή.

Για την μετάβαση μεταξύ των διαγραμμάτων και εφαρμογής, το Umbrello UML Modeller επιτρέπει να δημιουργηθεί ο πηγαίος κώδικα σε διάφορες γλώσσες προγραμματισμού. Επίσης, εάν θέλουμε να χρησιμοποιήσουμε UML σε ένα αρχείο C ++, το εργαλείο Umbrello UML Modeller μπορεί να μας βοηθήσει να δημιουργήσουμε ένα πρότυπο του συστήματός από τον πηγαίο κώδικα, αναλύοντας τον πηγαίο κώδικα και εισάγοντας τις κλάσεις με όλα τα χαρακτηριστικά που βρέθηκαν σε αυτό το αρχείο.

Επίσης, το Umbrello UML Modeller μπορεί να παράγει πηγαίο κώδικα βασισμένο στο UML μοντέλο σε διάφορες γλώσσες προγραμματισμού έτσι ώστε να βοηθήσει στην υλοποίηση του έργου μας. Ο παραγόμενος κώδικας αποτελείται από τις δηλώσεις της κλάσης, τις μεθόδους και τα χαρακτηριστικά τους, έτσι ώστε να έχουμε την δυνατότητα να συμπληρώσουμε τα κενά ανάλογα με την λειτουργικότητα των κλάσεων μας.

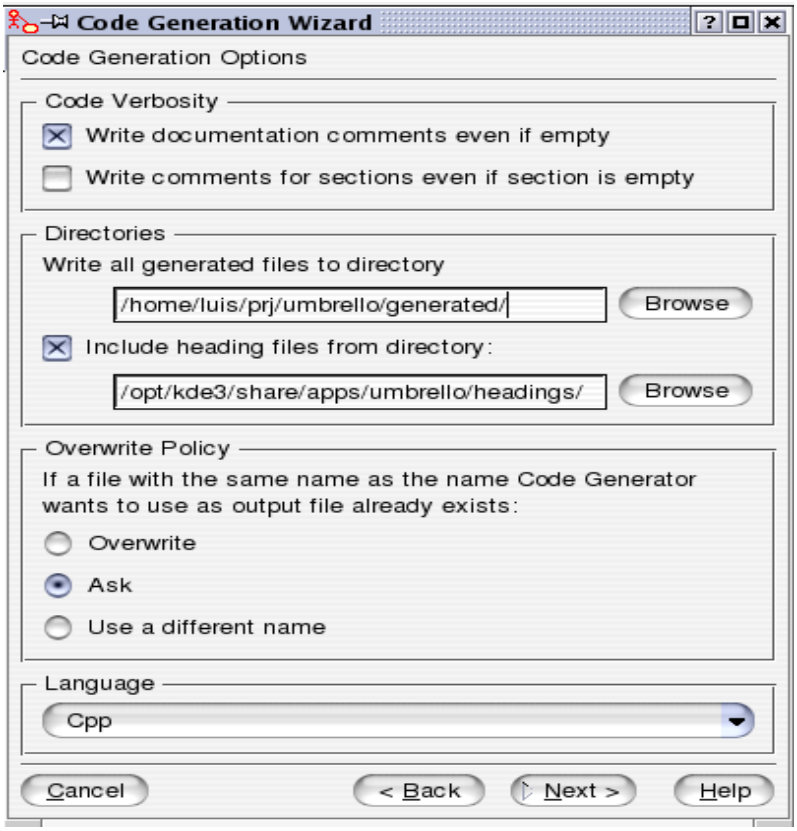
Το Umbrello UML Modeller 1.2 υποστηρίζει την δημιουργία κώδικα για Action Script, Ada, C+ +, CORBA IDL, Java ™, JavaScript, PHP, Perl, Python, SQL και XMLSchema.

Για να δημιουργηθεί κώδικας με το Umbrello UML Modeller, πρέπει πρώτα να δημιουργηθεί ή να φορτωθεί ένα υπόδειγμα που περιέχει τουλάχιστον μια κλάση. Όταν είμαστε έτοιμοι να αρχίσουμε την συγγραφή κώδικα, επιλέγουμε το *Code Generation Wizard* από το μενού κώδικα *(Code)* για να ξεκινήσει έναν οδηγό ο οποίος θα μας καθοδηγήσει στην διαδικασία παραγωγής κώδικα.

Το πρώτο βήμα είναι να επιλέξουμε τις κλάσεις για τις οποίες θέλουμε να δημιουργήσουμε τον πηγαίο κώδικα. Εξ ορισμού όλες οι κλάσεις του μοντέλου είναι

επιλεγμένες, και μπορούμε να καταργήσουμε αυτές για τις οποίες δεν θέλουμε να δημιουργήσουμε κώδικα με τη μετακίνησή τους στον κατάλογο στην αριστερή πλευρά.

Το επόμενο βήμα του οδηγού, μας επιτρέπει να τροποποιήσουμε τις παραμέτρους του χρησιμοποιεί ο οδηγός κώδικα (code generator) καθώς γράφουμε το πρόγραμμα. Οι ακόλουθες επιλογές φαίνονται στην πιο κάτω εικόνα:



Εικόνα 3-2 Επιλογές για την παραγωγή κώδικα στο Umbrello UML Modeller

Η επιλογή «*Write documentation comments even if empty*» αναθέτει στο *Code generator* να γράψει σχόλια του τύπου «/ * μπλα μπλα */» , ακόμη και αν το μπλοκ σχολίων είναι άδειο. Αν προσθέσουμε τεκμηρίωση στις κλάσεις, μεθόδους, ή στα χαρακτηριστικά του μοντέλου μας, το *Code Generator* θα γράψει αυτά τα σχόλια, ως Doxygen τεκμηριώσεις ανεξάρτητα με το τι έχουμε ορίσει. αλλά αν κάνουμε αυτήν την επιλογή το Umbrello UML Modeller θα γράψει μπλοκ σχολίων για όλες τις κλάσεις, τις μεθόδους και τα χαρακτηριστικά που ακόμη και αν δεν υπάρχει τεκμηρίωση στο μοντέλο

μας. Στην οποία περίπτωση θα πρέπει να τεκμηριωθούν αργότερα απευθείας στον πηγαίο κώδικα.

Η επιλογή «*Write comments for sections even if section is* empty» είναι η αιτία για να γράψει το Umbrello UML Modeller σχόλια στον πηγαίο κώδικα και να προσδιορίσει τα διάφορα τμήματα μιας κλάσης. Για παράδειγμα, δημόσιες μέθοδοι ή χαρακτηριστικά πριν από τα αντίστοιχα τμήματα. Αν επιλεχθεί αυτήν η επιλογή το Umbrello UML Modeller θα γράψει σχόλια για όλα τα τμήματα της κλάσης ακόμη και αν η ενότητα είναι άδεια. Για παράδειγμα, θα γράψει ένα σχόλιο για τις προστατευόμενες μεθόδους ακόμη και αν δεν υπάρχουν τέτοιου είδους μέθοδοι στην κλάση σας.

Το Umbrello UML Modeller από προεπιλογή θα δημιουργήσει κώδικα στη γλώσσα που έχει επιλέξει ως ενεργή γλώσσα, αλλά με το Code Generator Wizard υπάρχει η δυνατότητα να γίνει αλλαγή της προεπιλογής σε μια άλλη γλώσσα.

3.3. Σύγκριση Umbrello UML Modeler με το εργαλείο Visual Paradigm For UML

Σε αυτήν την ενότητα γίνεται μια σύγκριση μεταξύ των δύο εργαλείων μοντελοποίησης. Τα δύο εργαλεία έχουν τις ίδιες δυνατότητες όπως για παράδειγμα την εισαγωγή κώδικα και την δημιουργία διαγράμματος κλάσεων. Το Umbrello UML Modeller είναι πιο απλό εργαλείο με ευκολία στον χειρισμό όμως βασικό κριτήριο στην επιλογή εργαλείου μοντελοποίησης στα πλαίσια της πτυχιακής εργασίας ήταν η ευκρίνεια των διαγραμμάτων. Έτσι επιλέχθηκε το Visual Paradigm for UML καθώς τα διαγράμματα που δημιουργήθηκαν με αυτό ήταν πιο ευδιάκριτα και πιο ωραία σχηματικά από ότι αυτά που δημιουργήθηκαν με το Umbrello UML Modeller. Ένας άλλος λόγος που λειτούργησε υπέρ του Visual Paradigm for UML ήταν η προηγούμενη εμπειρία με το συγκεκριμένο εργαλείο σε μάθημα της σχολής.

Εκτός από αυτά τα δύο υπάρχουν αρκετά ακόμη εργαλεία για τα οποία αν διαβάσουμε τα εγχειρίδια χρήσης θα διαπιστώσουμε ότι υποστηρίζουν τις ίδιες λειτουργίες όπως το StarUML, Rational Rose, Dia, Poseidon for UML, Papyrus, ArgoUML και άλλα πολλά.

ΚΕΦΑΛΑΙΟ 4: ΣΥΣΤΗΜΑ ΔΙΑΧΕΙΡΙΣΗΣ ΚΙΝΗΜΑΤΟΓΡΑΦΟΥ

Ο κινηματογράφος ή αλλιώς σινεμά αποτελεί σήμερα την αποκαλούμενη και έβδομη τέχνη, δίπλα στη γλυπτική, την ζωγραφική, το χορό, την αρχιτεκτονική, την μουσική και την λογοτεχνία. Αρχικά εμφανίστηκε περισσότερο ως μια νέα τεχνική καταγραφής της κίνησης και οπτικοποίησής της, όπως δηλώνει και ο ίδιος ο όρος (*κινηματογράφος = κίνηση + γραφή*). Πλέον στις μέρες μας είναι ένα μέσο ψυχαγωγίας αλλά και μεταφοράς κάποιον μηνυμάτων μέσα από την κινηματογραφική ζωή των πρωταγωνιστών. Επέλεξα το σύστημα διαχείρισης κινηματογράφου γιατί πιστεύω ότι οι περισσότεροι από εμάς γνωρίζουν την διαδικασία που χρειάζεται να γίνει για την παρακολούθηση ενός κινηματογραφικού έργου.

Αρχικά εξηγείται η λογική και οι απαιτήσεις του συστήματος, κατόπιν εκφράζονται οι λειτουργίες του σε C++ και τελικά γίνεται μετατροπή του κώδικα C++ σε διαγράμματα UML χρησιμοποιώντας το *εργαλείο Visual Paradigm.*

4.1. Εισαγωγή

Ένα σύστημα διαχείρισης Κινηματογραφικών έργων έχει σε γενικές γραμμές τις παρακάτω απαιτήσεις:

- ✓ Καταχώρηση στοιχείων (αίθουσες, εισιτήρια, έργα, συντελεστές …)
- ✓ Κράτηση θέσεων για παράσταση ή προβολή
- ✓ Πληρωμή
- ✓ Ακύρωση / επιστροφή
- ✓ Επανεξέταση πελατών

Καταχώρηση στοιχείων

Με την εγκατάσταση του συστήματος, καταχωρούνται αρχικά οι τρεις αίθουσες που διαθέτει για παραστάσεις το Θέατρο. Η κάθε αίθουσα χαρακτηρίζεται από το όνομά της

καθώς και την χωρητικότητα σε θέσεις, οι οποίες κατατάσσονται σε τρεις κατηγορίες : Α, Β, και εξώστης. Το αντίτιμο του εισιτηρίου διαφέρει για κάθε κατηγορία.

Ωστόσο μία από τις αίθουσες η οποία δεν διαθέτει εξώστη, μπορεί να φιλοξενεί και κινηματογραφικά έργα, στην διάρκεια των οποίων δεν ισχύει ο διαχωρισμός των αιθουσών σε κατηγορίες. Οι παραστάσεις των έργων προκαθορίζονται για συγκεκριμένο χρονικό διάστημα. Για κάθε κινηματογραφική ταινία καταχωρούνται: ο κωδικός της, ο τίτλος, ο σκηνοθέτης, τρεις βασικοί ηθοποιοί , το είδος (αστυνομική, περιπέτεια, πολεμική, ιστορική, αισθηματική, κωμωδία, δράμα, τρόμου) και το έτος παραγωγής. Καταχωρούνται επίσης προαιρετικά και οι πελάτες. Για κάθε πελάτη, το σύστημα θα πρέπει θα καταχωρεί το ονοματεπώνυμό του, τον αριθμό τηλεφώνου, και προαιρετικά τον λογαριασμό της πιστωτικής του κάρτας. Η καταχώρηση γίνεται από τον ταμία. Ένας πελάτης μπορεί να γίνει συνδρομητής για ένα έτος.

Κράτηση θέσεων για παράσταση ή προβολή

Ο πελάτης έρχεται στο ταμείο με σκοπό να κλείσει μία ή περισσότερες θέσεις για παράσταση ή προβολή.

Ο ταμίας τον ρωτάει ποιο είναι το έργο που επιθυμεί.

Ο πελάτης απαντά ποιο έργο επιθυμεί να δει.

Ο ταμίας ζητάει από το σύστημα να του εμφανίσει τις παραστάσεις/προβολές του έργου (εμφανίζεται μαζί και η πληρότητα).

Ο ταμίας τον ρωτάει ποια προβολή του έργου και ποια μέρα (ημερ/νία) επιθυμεί.

Ο πελάτης απαντά.

Ο ταμίας ζητά από το σύστημα να του εμφανίσει την κατάσταση των θέσεων της αίθουσας που προβάλλεται το έργο.

Το σύστημα εμφανίζει την κατάσταση των θέσεων της αίθουσας που προβάλλεται το έργο (κατειλημμένες, κρατημένες, ελεύθερες).

Αρχικά ο ταμίας του παρουσιάζει στην οθόνη του υπολογιστή την εικόνα της κατάστασης των θέσεων της αίθουσας παράστασης/προβολής και τον ρωτάει να του

υποδείξει την/ις θέση/εις που επιθυμεί. Το σύνολο των θέσεων παρουσιάζεται υπό μορφή πίνακα, όπου οι κατειλημμένες φαίνονται με μικρό γεμάτο τετράγωνο, οι κρατημένες με ένα Χ και οι ελεύθερες με κενό τετράγωνο.

Ο πελάτης επιλέγει τις θέσεις που επιθυμεί και ο ταμίας προχωρά στην κράτησή τους. Αμέσως τα επιλεγμένα κενά τετράγωνα μετατρέπονται σε x.

Ο ταμίας ζητά από τον πελάτη το όνομά του και εφ' όσον δεν υπάρχει καταχωρημένος στο σύστημα τον καταχωρεί ζητώντας του και αριθμό τηλεφώνου επικοινωνίας.

Στη συνέχεια ρωτάει τον πελάτη αν θα πληρώσει εκείνη την στιγμή, οπότε προχωρεί στην διαδικασία <<πληρωμής>>, διαφορετικά η πληρωμή μένει σε εκκρεμότητα.

Μία κράτηση που αφορά θεατρικά έργα, έχει ισχύ μέχρι μία ώρα πριν την ημέρα και ώρα της παράστασης, ενώ για κινηματογραφικά έργα 20 λεπτά πριν την έναρξη της προβολής.

Πληρωμή

Το αντίτιμο του εισιτηρίου καθορίζεται από την κατηγορία της αίθουσας αλλά και αυτή του πελάτη.

Η πληρωμή / χρέωση μπορεί να γίνει επίσης με δύο τρόπους:

- ✓ Μετρητοίς
- ✓ Με χρεωστική κάρτα

Ακύρωση/επιστροφή

Η ακύρωση θέσης /ων μπορεί να γίνει με δύο τρόπους:

Αυτόματα, με την παρέλευση του χρόνου κράτησης.

Με τηλεφωνική επικοινωνία του πελάτη με τον ταμία.

Με το πέρας των παραστάσεων το σύστημα χαρακτηρίζει το έργο ως τετελεσμένο, ώστε να υπάρχει η δυνατότητα εξαγωγής στατιστικών στοιχείων.

Επανεξέταση πελατών

Στο τέλος κάθε έτους, με γνώμονα το σύνολο των κινήσεων τους, το κατάστημα προβαίνει σε επανεξέταση των πελατών του.

Όσοι έχουν πραγματοποιήσει κινήσεις που υπερβαίνουν σε αξία ένα συγκεκριμένο ποσό, τους παραχωρείται μία κάρτα ετήσιων παραστάσεων - ΚΕΠ, η οποία ισχύει κατ' ελάχιστον για ένα έτος. Η κάρτα, τους αποστέλλεται ταχυδρομικά και ενημερώνεται το σύστημα.

Όσοι είχαν ΚΕΠ και πραγματοποίησαν κινήσεις αξίας κάτω ενός καθορισμένου ορίου, δεν τους ανανεώνεται η ειδική κάρτα και ενημερώνεται το σύστημα.

4.2. Περιγραφή περιπτώσεων χρήσης

Σ' αυτή την ενότητα θα δοθεί η περιγραφή μίας περίπτωσης χρήσης όπου θα αναλυθεί η λειτουργικότητα ενός συστήματος, όπως αυτή είναι ορατή από τους εξωτερικούς χρήστες του συστήματος. Σ' αυτό το κομμάτι θα γίνει η προδιαγραφή των απαιτήσεων που γίνεται με γράψιμο κειμένου και όχι με την οργάνωση των περιπτώσεων χρήσης, το οποίο είναι προορατικό βήμα για την πιθανή βελτίωση της κατανόησης ή της μείωσης της επανάληψης.

4.3. Μοντέλο περιπτώσεων χρήσης

Από τον κώδικα που υλοποιεί την εφαρμογή διαχείρισης κινηματογράφου και από τις απαιτήσεις που προαναφέρθηκαν στην εισαγωγή του κεφαλαίου προκύπτει το διάγραμμα του *Σχήματος 4-1*.

Οι χειριστές της εφαρμογής έχουν την δυνατότητα να κοιτάξουν τις παραστάσεις από τις πληροφορίες προγράμματος. Αυτό γίνεται μέσα από τον παρακάτω κώδικα.

```
void Ergo::emfanParast()
{
    cout << " Εμφάνιση Παράστασης: \n";
    std::vector<Parastasi*>::const_iterator itr=parastaseis.begin();
    while (itr != parastaseis.end())
```

```
        {
                Parastasi *parast = static_cast<Parastasi*>(*itr);
                cout << parast;
                itr++;
        }
```

Στην πρώτη γραμμή του κώδικα καλείται η μέθοδος Ergo::emfanParast(), η οποία εμφανίζει τις παραστάσεις που υπάρχουν. Αυτό γίνεται με την βοήθεια του επαναλήπτη *(Iterator)*, ο οποίος σαρώνει τα στοιχεία μιας συλλογής αντικειμένων στην περίπτωση μας, το αντικείμενο Parastasi parast. Για την δημιουργία κράτησης ο ταμίας καλεί την μέθοδο neaKratisi (Ergo ergo) όπως φαίνεται στο πιο κάτω απόσπασμα κώδικα της τάξης *TameioCtrl*, που καλεί τον δομητή της τάξης Kratisi για μια νέα κράτηση.

```
public void neaKratisi(Ergo ergo) {

        trexKratisi= new Kratisi(arithmosKratisis++);

}
```

Ο δομητής της τάξης Kratisi αρχικοποιεί τον κωδικό κράτησης με τον αριθμό κράτησης που αυξάνεται κατά ένα κάθε φορά, που καλείται η μέθοδος neaKratisi(Ergo ergo).

```
public Kratisi (int arithmoskratisis) {

        this.kwdikos =arithmoskratisis;

    }
```

Εφόσον έχει αρχίσει η κράτηση καλείται η περίπτωση χρήσης Διαθεσιμότητα. Στην υλοποίηση του κώδικα η τάξη που ελέγχει την διαθεσιμότητα των παραστάσεων είναι η τάξη Parastasi. Με την εκτέλεση της τάξης Parastasi αυτόματα καλείται και η διαθεσιμότητα των θέσεων.

```
void Parastasi::emfanTheseis(){
	cout << "Διαθέσιμες θέσεις παράστασης \n";
	for (int i=0; i<theseisParast.size(); i++)
	{
		cout << theseisParast.at(i);
	}
}
```

Όπως φαίνεται στο διάγραμμα περίπτωσης χρήσης το «4.include ΠΧ-Επιλογή θέσης » παρουσιάζει κοινή συμπεριφορά είτε κάνεις κράτηση είτε κάνεις αλλαγή κράτησης, για αυτό δημιουργείται καινούργια περίπτωση χρήσης. Γίνεται έλεγχος για τις διαθέσιμες θέσεις και από αυτές επιλέγει ο χρήστης τις θέσεις που θέλει και το σύστημα κάνει την κράτηση θέσεων.

```
wchar_t Parastasi::getKatastasiThesis(int t)
{
	ThesiParastasis *thesiParast = theseisParast.at(t);
	return thesiParast.getKatastasi();
}
```

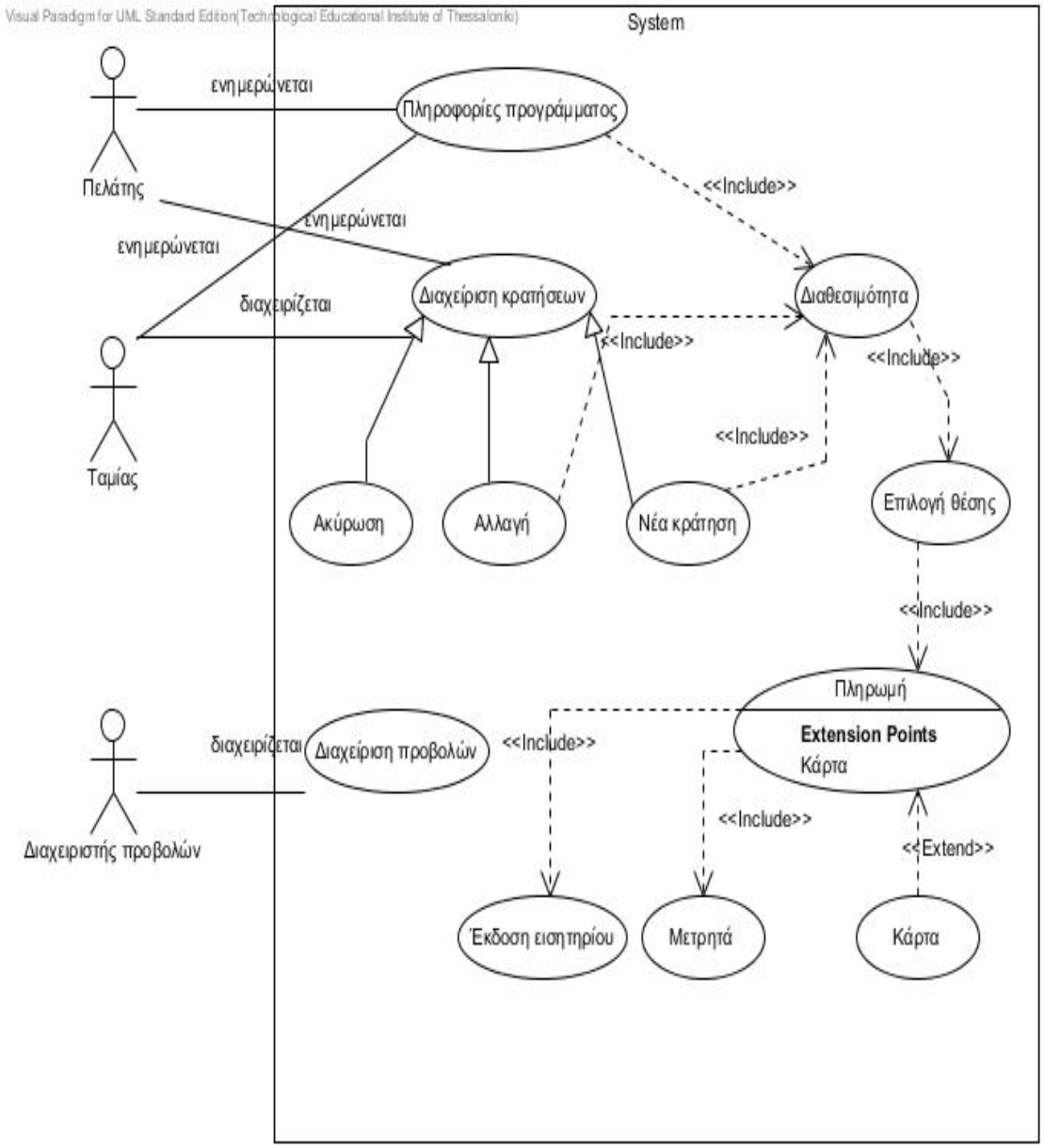
Εφόσον η εφαρμογή καλέσει την μέθοδο getKatastasiThesis(int t) μέσα στη μέθοδο δημιουργείται το αντικείμενο thesiParast που είναι τύπου ThesiParastasis. Άρα καλείται η τάξη ThesiParastasis να αρχικοποιήσει ένα αντικείμενο thesiParast και να κάνει τον έλεγχο διαθεσιμότητας όπως φαίνεται στο πιο κάτω απόσπασμα κώδικα.

```
1. #ifndef __ThesiParastasis_CPP
2. #define __ThesiParastasis_CPP
```

```
3. #include "ThesiParastasis.hpp"

4. ThesiParastasis::ThesiParastasis(Thesi *thesi)

5. {

6.    this.thesi = thesi;

7.    this.katastasi = ' ';

8. }



9. void ThesiParastasis::setDiathesimi(int i)

10.{

11.   katastasi = ' ';

12. }



13. void ThesiParastasis::setKatillimeni(int i)

14.{

15.   katastasi = 'K';

16.}


17. wchar_t ThesiParastasis::getKatastasi()

18.{

19.   return katastasi;

20.}
```

```
21. void ThesiParastasis::setKatastasi(wchar_t k)
22.{
23.   katastasi = k;
24.}
25. int ThesiParastasis::getArithmos()
26.{
27.  return thesi.getArithmos();
28.}
29. std::string ThesiParastasis::ToString()
30.{
31.  cout << "    Αριθμός θέσης,: " << getArithmos() << "  [" << katastasi
32. << "]";
33.}
34. #endif
```

Αφού έχει ολοκληρωθεί η επιλογή παράστασης και θέσεων τερματίζεται η κράτηση, γίνεται προσθήκη της τρέχουσας κράτησης στην λίστα κρατήσεων και καλείται η «5.include ΠΧ-Πληρωμή». Στην πληρωμή καλείται η υπολειτουργία «8.Include ΠΧ-Μετρητά», όπου ο πελάτης πληρώνει με μετρητά, αν όμως θελήσει να πληρώσει με κάρτα τότε καλείται η επέκταση «7. Extend ΠΧ-Κάρτα». Τέλος εκδίδεται το/α εισιτήριο/α. Πιο κάτω το ολοκληρωμένο διάγραμμα περιπτώσεων χρήσης.



Σχήμα 4-1 Διάγραμμα Περιπτώσεων Χρήσης

Πριν αρχίσουμε την υλοποίηση μιας εφαρμογής πρέπει πρώτα να βρούμε τις απαιτήσεις που θα έχει το σύστημα. Δηλαδή πρέπει να βρούμε και να αναλύσουμε όλες τις πιθανές περιπτώσεις χρήσης που απαρτίζουν την εφαρμογή που θα υλοποιήσουμε. Στην ενότητα αυτή θα αναπτυχθεί μία περίπτωση χρήσης «ΠΧ- Κράτηση θέσης» θα προσδιοριστούν οι χρήστες της εφαρμογής, θα δοθεί περιγραφή της ΠΧ και θα δοθεί η βασική ροή, δηλαδή οι ενέργειες που κάνει ο χρήστης και η ανταπόκριση που έχει από το σύστημα.

Παράδειγμα:

Περίπτωση Χρήσης ΠΧ1: **Κράτηση θέσης**

Κύριος χρήστης: **Ταμίας**

Περιγραφή: Αφορά την κράτηση μιας ή περισσότερων θέσεων.

Εμπλεκόμενοι και Ενδιαφέροντα**.**

Ταμίας: θέλει μια γρήγορη και ασφαλή κράτηση θέσης/θέσεων.

Πελάτης: θέλει μια γρήγορη, ευχάριστη και ασφαλή εξυπηρέτηση.

Διευθυντής: θέλει ικανοποιημένους πελάτες.

Βασική ροή ή Κύριο Σενάριο

Πίνακας 4- 1 Βασική ροή ή Κύριο Σενάριο

	Ενέργειες Χρήστη	**Απόκριση Συστήματος**
1	Ο πελάτης ζητάει Χ θέσεις για κάποιο έργο	
2	Ο ταμίας ζητά από το σύστημα το πρόγραμμα παραστάσεων του έργου	
3		Το σύστημα εμφανίζει το πρόγραμμα παραστάσεων του έργου
4	Ο ταμίας ρωτάει τον πελάτη για ποια ημερομηνία / ώρα από τις διαθέσιμες παραστάσεις επιθυμεί	
5	Ο πελάτης επιλέγει την παράσταση	
6	Ο ταμίας ζητά την συγκεκριμένη παράσταση από το σύστημα	
7		Το σύστημα εμφανίζει το σύνολο διαθέσιμων θέσεων για την συγκεκριμένη παράσταση
8	Ο ταμίας ρωτάει τον πελάτη για ποια κατηγορία ενδιαφέρεται	

	(κανονική θέση, θέση για άτομο με ειδικές ανάγκες κτλ.)	
9	Ο πελάτης επιλέγει την/τις θέση/εις κάποιας κατηγορίας που επιθυμεί	
10	Ο ταμίας ζητά το σύστημα να κάνει δέσμευση θέσεων	
11		Το σύστημα δεσμεύει της θέσεις και προβάλλει το κόστος της δέσμευση αυτής
12	Ο ταμίας ενημερώνει τον πελάτη για το κόστος (include ΠΧ Πληρωμή)	

Εναλλακτικές ροές ή Επεκτάσεις :

10^{α}. Έλεγχος διαθεσιμότητες :include ΠΧ Έλεγχος διαθεσιμότητας.

11^{α}. Δέσμευση θέσεων : include Π.Χ Επιλογή θέσης.

12^{α}. Πληρωμή με μετρητά: include ΠΧ Μετρητά.

12^{b}. Πληρωμή με κάρτα : extends ΠΧ Κάρτα.

Στο πιο πάνω παράδειγμα δίνονται τα στοιχεία που χρειάζονται για την «ΠΧ Κράτηση θέσης», δηλαδή οι χειριστές του συστήματος μια μικρή περιγραφή για την ΠΧ, οι εμπλεκόμενοι και τα ενδιαφέροντα τους, η βασική ροή (δηλαδή τα βήματα που γίνονται για να υπάρξει μια επιτυχημένη κράτηση θέσης) και τέλος, δίνονται οι εναλλακτικές ροές. Στις εναλλακτικές ροές συνήθως υπάρχουν συμπεριφορές οι οποίες είναι κοινές σε πολλές περιπτώσεις χρήσης γι αυτό τον λόγο χρησιμοποιείται η σχέση «include» για την αποφυγή επανάληψης κειμένου. Παραδείγματος χάριν η περιγραφή της πληρωμής με μετρητά εμφανίζεται σε διάφορες ΠΧ όπως της ΠΧ-Πώλησης, ΠΧ-Ενοικίαση, ΠΧ-Αγορά. Η σχέση «include» δημιουργεί μια ξεχωριστή αυτόνομη επαναχρησιμοποιήσιμη υπολειτουργία περίπτωσης χρήσης.

Τι γίνεται αν στο πιο πάνω παράδειγμα έπρεπε μια περίπτωση χρήσης να τροποποιηθεί για κάποιο λόγο. Όπως φαίνεται στο παράδειγμα πιο πάνω αν ο πελάτης θελήσει να πληρώσει με κάρτα αντί για μετρητά έπρεπε η ΠΧ-Πληρωμή να αλλάξει

όμως με την χρήση της σχέσης «extend» απλά δημιουργείται μια καινούργια ΠΧ που συμπληρώνει την προηγούμενη. Ο τρόπος λειτουργίας της «extend» είναι ο εξής: δημιουργείται μια επέκταση ή μια πρόσθετη ΠΧ και μέσα σε αυτήν περιγράφεται που και υπό ποιους όρους επεκτείνεται η συμπεριφορά κάποιας βασικής ΠΧ.

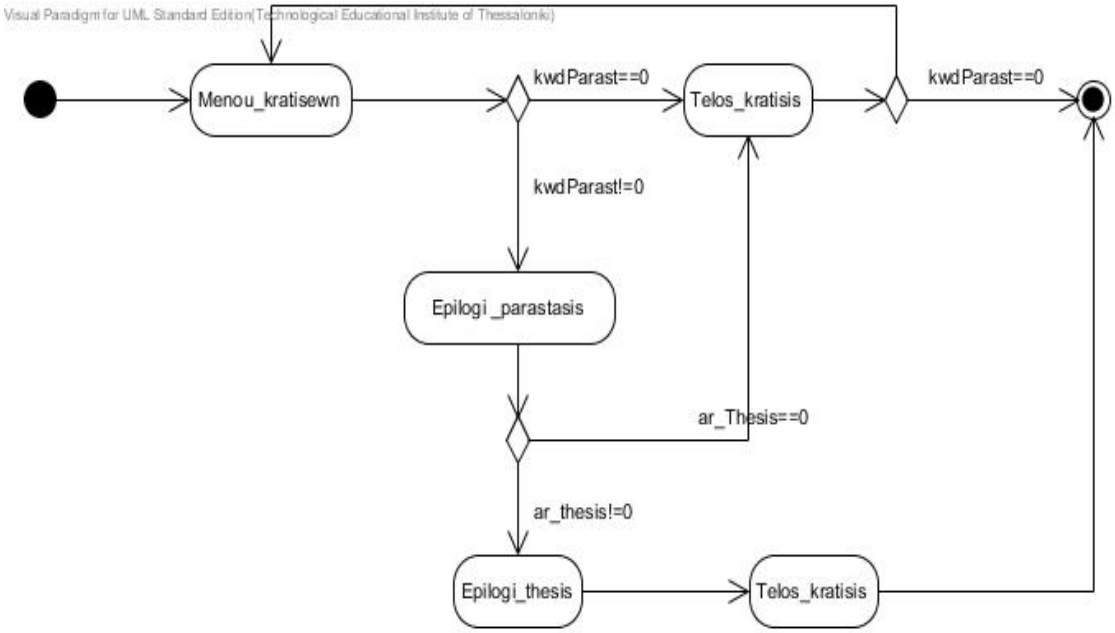
4.4. Διάγραμμα δραστηριοτήτων

Στην προηγούμενη ενότητα αναλύθηκε μια περίπτωση χρήσης ΠΧ-Κράτηση θέσης. Για να μπορεί ο χρήστης να επιλέξει μια ή περισσότερες θέσεις πρώτα πρέπει να κάνει μια κράτηση σε παράσταση. Αυτό μπορούμε να το δούμε μέσα από τον πιο κάτω κώδικα.

```
1.// ΠΧ04 Επαναληπτική διαδικασία κρατήσεων. Έξοδος με 0
2.      int kwdParast;
3.     cout <<"\n Δώσε αριθμό παράστασης,(1, 2, 0)=> ";
4.     cin >> kwdParast;
5.     while (kwdParast!=0) {
6.      if (kwdParast==1 || kwdParast==2) {
7.            tamCtrl.neaKratisi(ergo);
8.            // Επιλογή παράστασης και εμφάνιση αριθμού θέσεων
9.            tamCtrl.epilogiParast(kwdParast);
10.           tamCtrl.epilogiThesewn();
11.           tamCtrl.telosKratisis();
12.       }
13.      else
14.           System.out.println ("Οι κωδικοί παραστάσεων είναι 1 & 2.
```

```
Έξοδος με 0");
    15.          if (kwdParast!=0) {
    16.              ergo.emfanParast();
    17.              cout >>"Δώσε αριθμό παράστασης,(1, 2, 0)=> ";
    18.          }
    19.      }
```

Εφόσον υπάρχουν παραστάσεις στο κινηματογράφο ο χειριστής της εφαρμογής σ' αυτήν την περίπτωση είναι ο ταμίας, επιλέγει από το μενού κρατήσεων την παράσταση που θέλει ο πελάτης. Αυτό φαίνεται στον κώδικα από τις γραμμές 2-7, όπου γίνεται και έλεγχος συνθήκης τερματισμού που σ' αυτήν την περίπτωση είναι ο ακέραιος αριθμός μηδέν. Στην συνέχεια εμφανίζεται η λίστα ταινιών γίνεται επιλογή ταινίας στην γραμμή 7 του πιο πάνω κώδικα. Αφού έχει επιλεγεί το έργο τότε ο πελάτης πρέπει να επιλέξει την ώρα στη γραμμή 9, από την στιγμή που έχει δημιουργηθεί μία κράτηση πρέπει να γίνει και η επιλογή της/των θέσης/εων από το μενού θέσεων στην γραμμή 10. Εφόσον εκτελεστεί αυτή η σειρά των δραστηριοτήτων η διαδικασία κράτησης τερματίζεται και η τρέχουσα κράτηση προστίθεται στην λίστα κρατήσεων. Αλλιώς αν ο χειριστής του συστήματος δώσει λάθος κωδικό παράστασης τότε εμφανίζονται οι διαθέσιμες παραστάσεις με τους ανάλογους κωδικούς. Ζητείται ξανά από τον ταμία να δώσει αριθμό παράστασης που υπάρχει στην λίστα παραστάσεων. Εμφανίζεται η λίστα των θέσεων με τις διαθέσιμες και μη θέσεις και γίνεται η επιλογή από τις ελεύθερες θέσεις.



Σχήμα 4-2 Διάγραμμα δραστηριοτήτων

4.5. Διάγραμμα αλληλεπίδρασης

Σε αυτήν την ενότητα θα ασχοληθούμε με την συνεργασία μεταξύ των αντικειμένων της εφαρμογής μας. Η περιγραφή συμπεριφοράς γίνεται στο σύνολο των μηνυμάτων που ανταλλάσσονται από τα αντικείμενα. Ένα τέτοιο σύνολο μηνυμάτων μιας συνεργασίας ονομάζεται **αλληλεπίδραση** *(interaction). Ένα μήνυμα είναι μία μονόδρομη επικοινωνία μεταξύ δύο αντικειμένων, μία ροή ελέγχου με πληροφορία από έναν αποστολέα προς έναν αποδέκτη.* Ένα μήνυμα μπορεί να έχει παραμέτρους μεταφέροντας τιμές μεταξύ των αντικειμένων. Η ακολουθία των μηνυμάτων παρουσιάζεται είτε με διαγράμματα ακολουθίας που εστιάζουν στην χρονική ακολουθία των μηνυμάτων είτε με διαγράμματα επικοινωνίας που εστιάζουν στις σχέσεις μεταξύ των αντικειμένων που ανταλλάσσουν μηνύματα. Στο παράδειγμα διαχείρισης Κινηματογράφου θα δούμε πως ο ταμίας με τις ενέργειες του επηρεάζει τα αντικείμενα να ανταλλάζουν μηνύματα.

Ας δούμε πως ενεργούν τα αντικείμενα στην ΠΧ-Κράτησης θέσης στο διάγραμμα ακολουθίας.

Ο ταμίας μετά από επιλογή του πελάτη διαλέγει το έργο και την παράσταση, αυτόματα καλείτε η ΠΧ-Κράτηση θέσης. Για την επιλογή του έργου θα πρέπει να δημιουργηθεί μια νέα κράτηση. Αυτό γίνεται στην γραμμή 5, όπως φαίνεται και στο σχήμα ο ταμίας με την επιλογή της παράστασης ξεκινάει μια ανταλλαγή μηνυμάτων μεταξύ των αντικειμένων διαφορετικών κλάσεων.

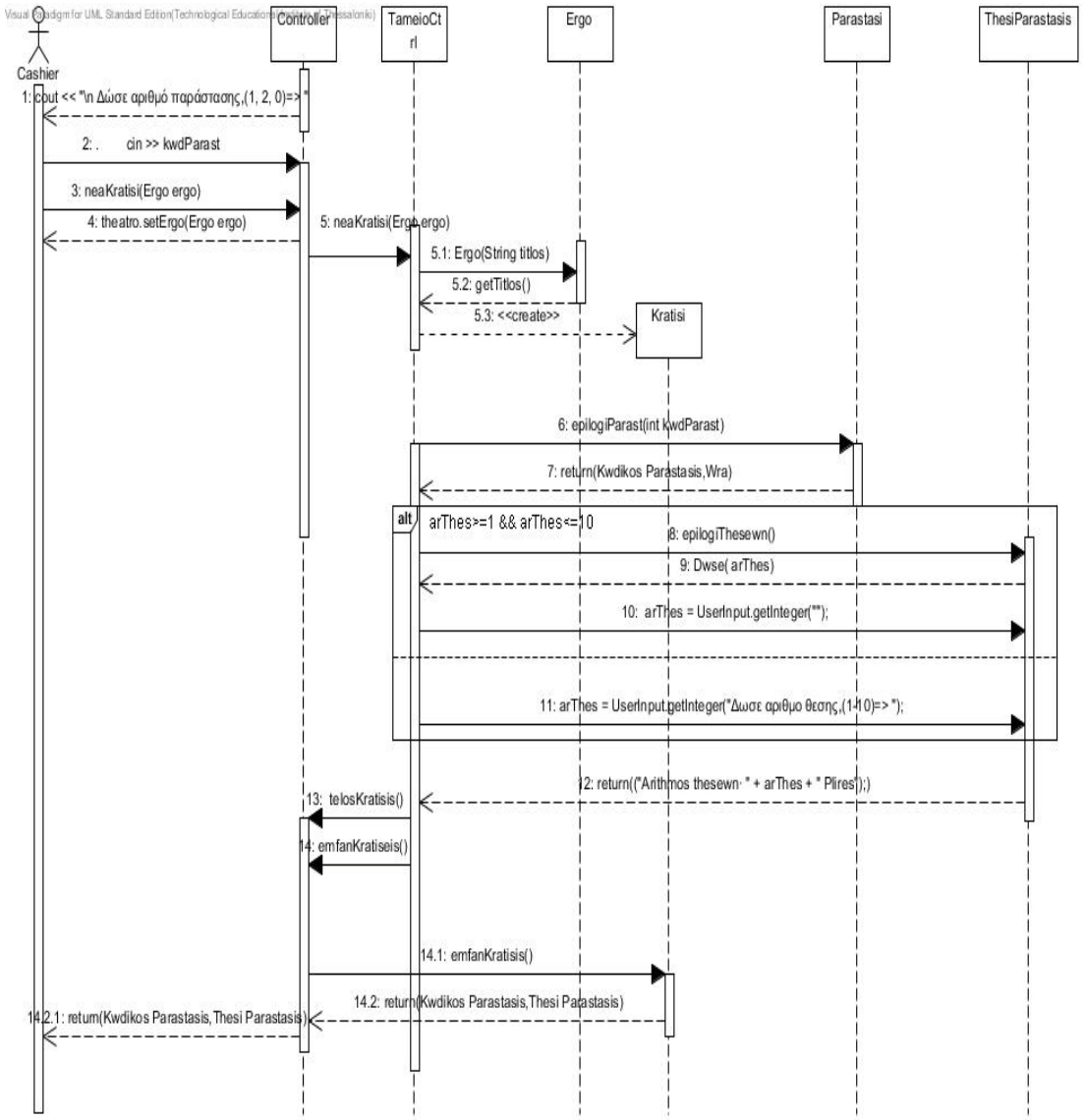
```
1.// ΠΧ04 Επαναληπτική διαδικασία κρατήσεων. Έξοδος με 0
2.      int kwdParast;
3.      cout << "\n Δώσε αριθμό παράστασης,(1, 2, 0)=> ";
4.       cin >> kwdParast;
5.       while (kwdParast!=0) {
6.       if (kwdParast==1 || kwdParast==2) {
7.              tamCtrl.neaKratisi(ergo);
8.              // Επιλογή παράστασης και εμφάνιση αριθμού θέσεων
9.              tamCtrl.epilogiParast(kwdParast);
10.             tamCtrl.epilogiThesewn();
11.             tamCtrl.telosKratisis();
12.     }
```

Το αντικείμενο tamCtrl που είναι τύπου TameioCtrl ενεργοποιεί τον δομητή της κλάσης Ergo όπου επιστρέφει τον τίτλο του έργου. Στην συνέχεια η κλάση TameioCtrl όπως φαίνεται στο πιο κάτω απόσπασμα κώδικα δημιουργεί μία νέα κράτηση.

```
1. TameioCtrl::TameioCtrl()
2. {
3. }

4. void TameioCtrl::setErgo(Ergo *ergo)
5. {
6.     this.ergo=ergo;
7. }

8. void TameioCtrl::neaKratisi(Ergo *ergo)
9. {
10.    trexKratisi= newKratisi(arithmosKratisis++);
11. }
```

Φυσικά αυτό δεν γίνεται μέσω της κλάσης TameioCtrl(), για την δημιουργία νέας κράτησης καλείται η κλάση Kratisi.

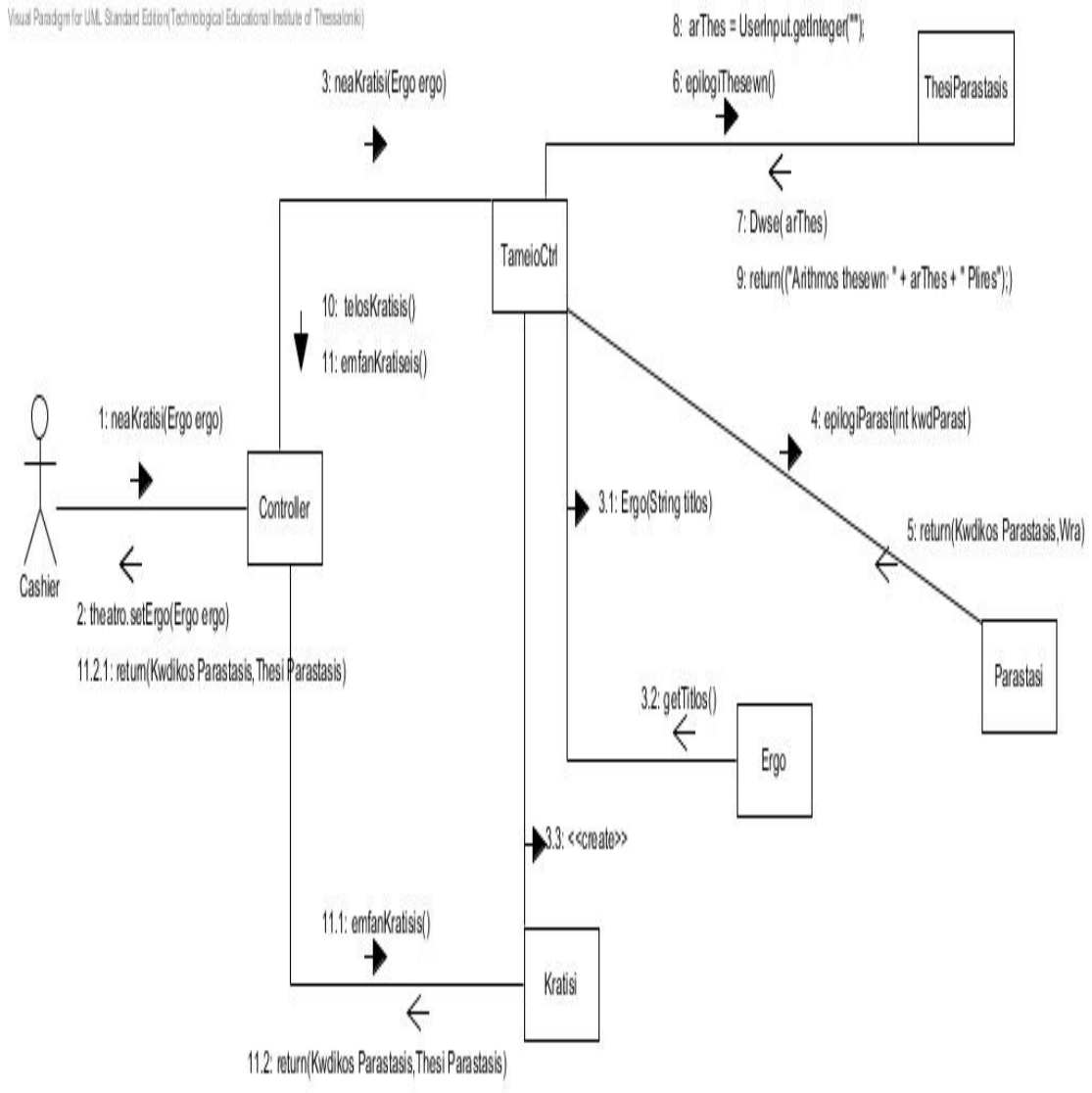
Εφόσον δημιουργηθεί η νέα κράτηση τότε η κλάση TameioCtrl() καλεί τις κλάσεις Parastasi και ThesiParastasis. Στην πρώτη κλάση δίνει τον κωδικό παράστασης που επιθυμεί ο πελάτης και επιστρέφεται ο κωδικός παράστασης και η ώρα έναρξης. Στην συνέχεια δημιουργείται μια συνθήκη εναλλακτικών περιπτώσεων *(alt)*, όπου γίνεται έλεγχος αν ο αριθμός θέσης που θέλει ο πελάτης βρίσκεται στο όριο ένα έως δέκα. Αν η πληκτρολόγηση του ταμία είναι σωστή τότε συνεχίζεται η κράτηση αλλιώς αρχίζει από την αρχή η διαδικασία επιλογής θέσεων. Αφού τελειώσει η επιλογή των θέσεων ολοκληρώνεται και η διαδικασία κράτησης. Η τρέχουσα κράτηση προστίθενται στην λίστα κρατήσεων και εμφανίζονται στον ταμία τα στοιχεία της κράτησης. Πιο κάτω θα δείτε το ολοκληρωμένο διάγραμμα ακολουθίας για την ΠΧ-Κράτηση Θέσης.



Σχήμα 4-3 Διάγραμμα Ακολουθίας (Sequence Diagram)

Για να δούμε τις σχέσεις μεταξύ των αντικειμένων που ανταλλάσσουν μηνύματα χρειάζεται να μελετήσουμε το διάγραμμα επικοινωνίας. Το σχήμα που ακολουθεί μας παρέχει τη δυναμική οπτική παρουσιάζοντας την αλληλεπίδραση των αντικειμένων. Τα διαγράμματα επικοινωνίας επεκτείνουν κατά κάποιο τρόπο τα διαγράμματα αντικειμένων, παρουσιάζοντας όχι μόνο τις σχέσεις μεταξύ των αντικειμένων αλλά και την επικοινωνία τους η οποία πραγματοποιείται με την ανταλλαγή μηνυμάτων. Τα διαγράμματα ακολουθίας και συνεργασίας ή επικοινωνίας θεωρούνται συμπληρωματικά καθώς περιέχουν τις ίδιες πληροφορίες αλλά το κάθε ένα δίνει μια διαφορετική οπτική

γωνία. Συνήθως χρησιμοποιούνται τα διαγράμματα ακολουθίας ως πιο κατάλληλα για την διερεύνηση διαφόρων σεναρίων μιας περίπτωσης χρήσης και τον εντοπισμό λειτουργιών αλλά και νέων κλάσεων ή σχέσεων μεταξύ τους. Το πιο κάτω διάγραμμα επικοινωνίας αντιστοιχεί στο διάγραμμα ακολουθίας του *Σχήματος 4-3*.



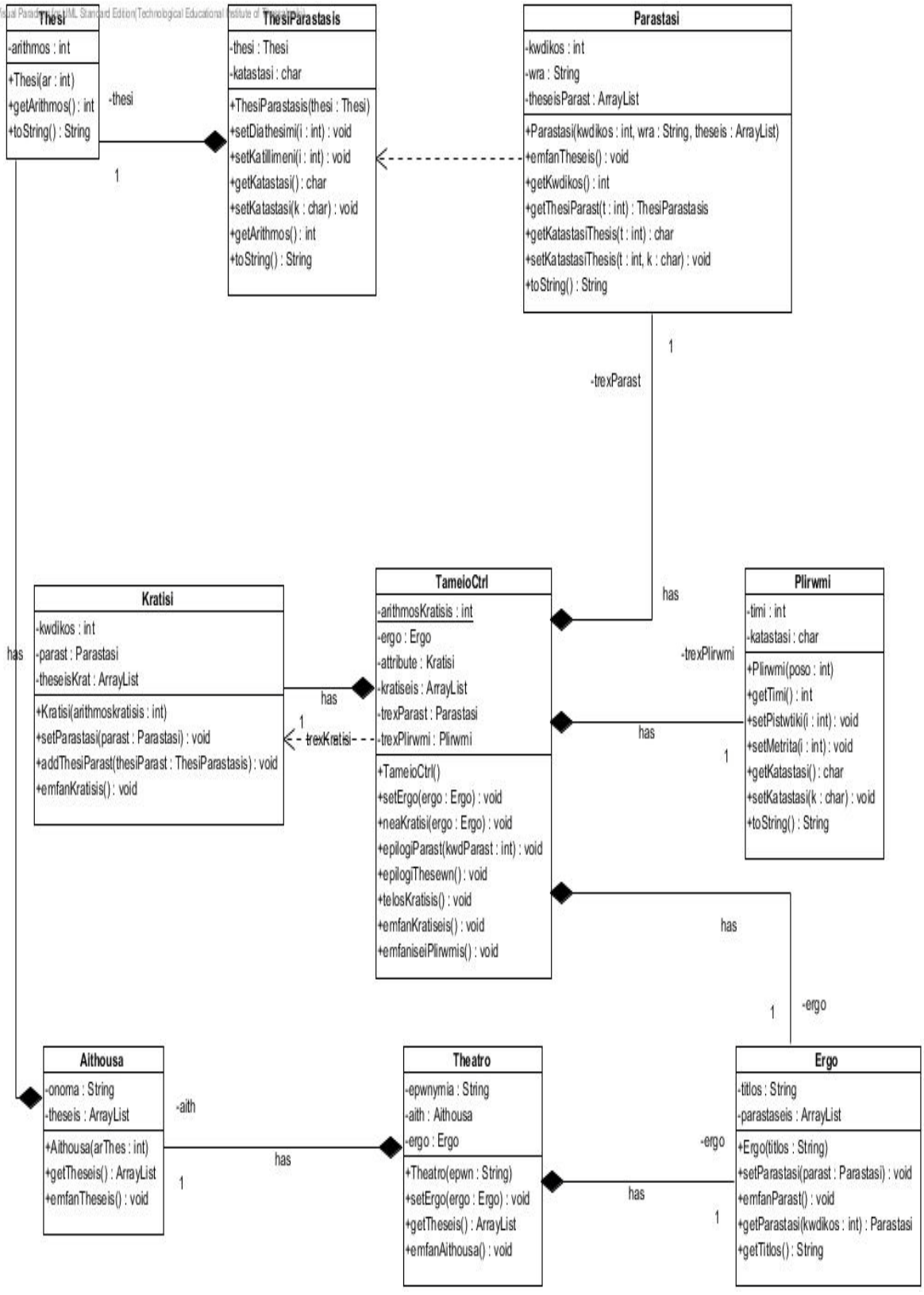
Σχήμα 4-4 Διάγραμμα Επικοινωνίας

4.6. Το στατικό μοντέλο του συστήματος

Ο κύριος σκοπός δημιουργίας των διαγραμμάτων ακολουθίας είναι η κατανομή της λειτουργικότητας (μεθόδων) στις κλάσεις του συστήματος και κατά δεύτερο λόγο ο

εντοπισμός τυχόν κλάσεων, ιδιοτήτων και μεθόδων που δεν αναδείχθηκαν στο στάδιο της ανάλυσης. Το στατικό μοντέλο του υπό ανάπτυξη συστήματος, δηλαδή το διάγραμμα κλάσεων που περιγράφει την αρχιτεκτονική του είναι αναμενόμενο να αναθεωρείται μετά την ολοκλήρωση των διαγραμμάτων ακολουθίας. Στο αναθεωρημένο διάγραμμα κλάσεων περιλαμβάνονται πλέον οι μέθοδοι κάθε κλάσης με την πλήρη υπογραφή τους, δηλαδή μαζί με τυχόν παραμέτρους που απαιτούνται και τον επιστρεφόμενο τύπο τους.

Η λήψη ενός μηνύματος από ένα αντικείμενο μιας κλάσης σε ένα διάγραμμα ακολουθίας υποδηλώνει την ύπαρξη μιας μεθόδου στην κλάση που είναι ο αποδέκτης του μηνύματος. Η μέθοδος συνιστά τη λειτουργία, που θα εκτελείται στην κλάση-αποδέκτη με τη λήψη του αντίστοιχου μηνύματος. Στο διάγραμμα κλάσεων παίρνουμε όλες τις κλάσεις της εφαρμογής με τις ιδιότητες και τις λειτουργίες της και τις τοποθετούμε στο διάγραμμα συμβολίζοντας τις κλάσεις με ορθογώνιο.



Σχήμα 4-5 *Διάγραμμα Κλάσεων*

4.7. Το δυναμικό μοντέλο του συστήματος

Τα δυναμικά μοντέλα απεικονίζουν τη δυναμική συμπεριφορά του συστήματος, π.χ. πως αποκρίνεται στις ενέργειες των χρηστών ή σε άλλα εξωτερικά ερεθίσματα και πως διαμορφώνεται η εσωτερική του κατάσταση κατά τη λειτουργία του. Το δυναμικό μοντέλο αποτελείται από τα διαγράμματα καταστάσεων, συστατικών και ανάπτυξης.

4.7.1. Διάγραμμα καταστάσεων

Στην ΠΧ-Κράτηση θέσης ο ταμίας επιλέγει από το μενού κρατήσεων μία παράσταση. Αυτό γίνεται εφόσον δώσει κωδικό παράστασης που να είναι διάφορος του μηδενός όπως γίνεται και στο πιο κάτω απόσπασμα κώδικα στην γραμμή 3.

```
// ΠΧ04 Επαναληπτική διαδικασία κρατήσεων. Έξοδος με 0
1.  int kwdParast;
2.      cout << "\n Δώσε αριθμό παράστασης,(1, 2, 0)=> 2."; cin >> kwdParast;
3.      while (kwdParast!=0) {
4.          if (kwdParast==1 || kwdParast==2) {
5.              tamCtrl.neaKratisi(ergo);
```

Στην συνέχεια δημιουργείται μια νέα κράτηση, το επόμενο βήμα είναι η επιλογή της παράστασης. Ο ταμίας επιλέγει την παράσταση που θέλει ο πελάτης, με την εκτέλεση της μεθόδου *epilogiParastasis(int kwdParast).*Πιο κάτω φαίνεται η υλοποίηση της μεθόδου αυτής.

```
1.public void epilogiParastasis(int kwdParast) {
2.      trexParast = ergo.getParastasi(kwdParast);
3.      System.out.println("\nΕπιλέγεται η παράσταση" + kwdParast);
4.      trexParast.emfanTheseis();
5.      trexKratisi.setParastasi(trexParast);
```

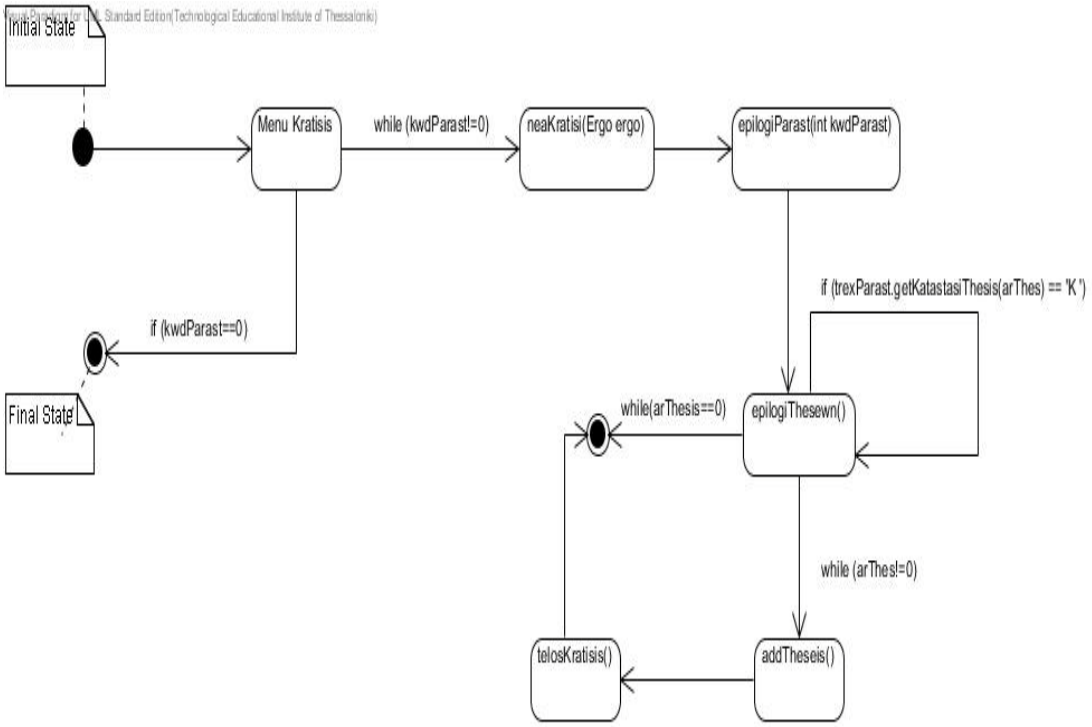
```
6.  }
```

Με την εκτέλεση της μεθόδου αυτής επιλέγεται η παράσταση και εμφανίζονται οι θέσεις της παράστασης. Στην γραμμή 4 του πιο πάνω κώδικα καλείται για να εμφανιστούν οι θέσεις της τρέχουσας παράστασης. Αν στην διαδικασία επιλογής αριθμού θέσεων επιλεχθούν μηδέν θέσεις τότε η διαδικασία ακυρώνεται. Αλλιώς γίνεται έλεγχος αν η/οι θέση/εις που κρατήθηκαν από τον πελάτη είναι διαθέσιμες ή όχι. Αν είναι διαθέσιμες, τότε γίνεται κράτηση των θέσεων και τερματίζεται η τρέχουσα κράτηση διαφορετικά ξαναεκτελείται η πιο κάτω μέθοδος μέχρι να βρεθούν διαθέσιμες θέσεις. Με την επιλογή των θέσεων εμφανίζονται τα στοιχεία της κράτησης και τερματίζεται η διαδικασία κράτησης.

```
1.  void TameioCtrl::epilogiThesewn()
2.  {
3.  int arThes;
4.  cout << "Δώσε αριθμό θέσης(0-9)=> \n" ;
5.  cin >> arThes;
6.  while (arThes!=0)
7.  {
8.  if (arThes>=-1 && arThes<10){
        a.  if (trexParast.getKatastasiThesis(arThes) == ' '){
                i.  trexParast.setKatastasiThesis(arThes, 'K');
                ii. trexKratisi.addThesiParast(trexParast.getThesiParast(arThes));
        b.  }
9.  else {
                i.  cout << "Η θέση· " << arThes << " είναι κατειλημμένη·";
10. }
```

```
                i.  cout << "Δώσε αριθμό θέσης,(0-9)=> ";
                ii. cin >> arThes;
    11. }//telos while
    12. }//telos epilogiThesewn()
```

Στο *Σχήμα 4-6* βλέπετε το ολοκληρωμένο διάγραμμα καταστάσεων για την ΠΧ-Κράτηση θέσης.

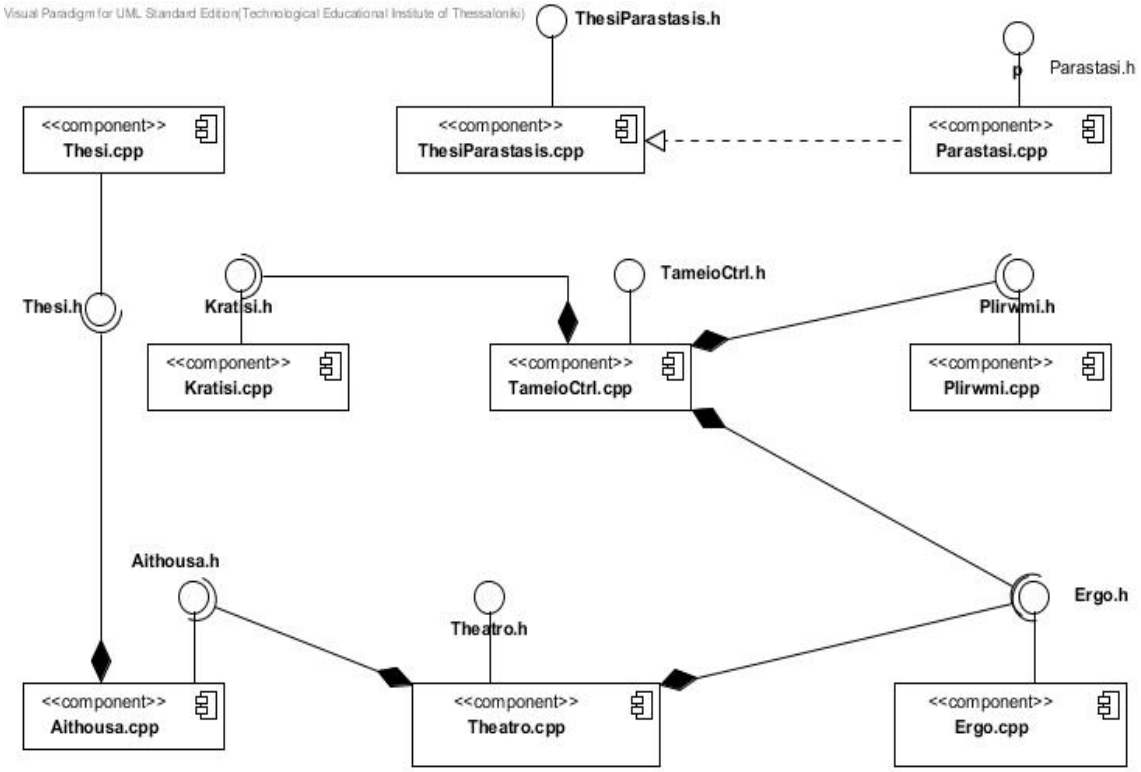


Σχήμα 4-6 *Διάγραμμα καταστάσεων*

4.7.2. Διάγραμμα συστατικών

Όπως ξέρουμε ένα συστατικό είναι μια φυσική μονάδα υλοποίησης κώδικα με σαφώς προσδιορισμένες διασυνδέσεις το οποίο αποτελεί επαναχρησιμοποιήσιμο τμήμα του συστήματος. Σε ένα αντικειμενοστρεφές σύστημα κάθε συστατικό ενσωματώνει την υλοποίηση μίας ή περισσοτέρων κλάσεων. Για παράδειγμα στη γλώσσα C++ κάθε πηγαίο αρχείο (.cpp) και κάθε αρχείο επικεφαλίδων(.h) είναι διαφορετικό συστατικό. Το εκτελέσιμο αρχείο (.exe) που παράγεται μετά την μεταγλώττιση είναι επίσης ένα

συστατικό. Παρομοίως στην γλώσσα Java κάθε αρχείο (.java) αποτελεί ένα συστατικό. Όπως φαίνεται και στην εφαρμογή διαχείρισης Κινηματογράφου κάθε αρχείο (.java) έχει γίνει συστατικό και συμβολίζεται ως ένα ορθογώνιο με δυο μικρά ορθογώνια στο ένα άκρο, όπως έχει αναφερθεί σε προηγούμενο κεφάλαιο. Στο *Σχήμα 4-7* δίνονται οι εξαρτήσεις μεταξύ των συστατικών του συστήματος. Για παράδειγμα το συστατικό *Parastasi.java* εξαρτάται από το συστατικό *ThesiParastasis.java* δηλαδή το πρώτο συστατικό δεν μπορεί να μεταγλωττιστεί προτού μεταγλωττιστεί το συστατικό



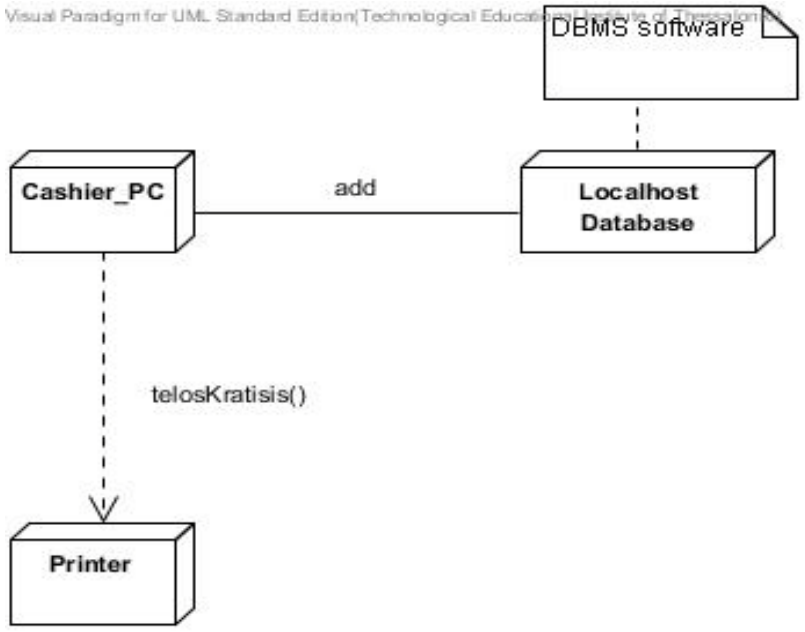
ThesiParastasis.java.

Σχήμα 4-7 Διάγραμμα Συστατικών

4.7.3. Διάγραμμα ανάπτυξης

Τα διαγράμματα ανάπτυξης *(deployment diagrams)* αναπαριστούν την αντιστοίχιση του λογισμικού σε επεξεργαστικές μονάδες-κόμβους. Μπορούν να χρησιμοποιηθούν για να δείξουν ποια συστατικά τρέχουν σε ποιους κόμβους. Ένας **κόμβος** *(node)* είναι ένα φυσικό αντικείμενο που στη γενική περίπτωση έχει τουλάχιστον μνήμη και δυνατότητα επεξεργασίας. Στην περίπτωση της εφαρμογής μας υπάρχουν τρεις κόμβοι, ένας κόμβος

είναι ο υπολογιστής που χρησιμοποιεί ο ταμίας (Cashier_PC), μια βάση αποθήκευσης των δεδομένων των κρατήσεων *(Local host Database)* και τέλος ένας εκτυπωτής που εκδίδει αποδείξεις και εισιτήρια *(Printer).*



Σχήμα 4-8 Διάγραμμα Ανάπτυξης

ΚΕΦΑΛΑΙΟ 5: ΔΙΑΧΕΙΡΙΣΗ ΤΡΑΠΕΖΙΚΩΝ ΛΟΓΑΡΙΑΣΜΩΝ

Σ' αυτό το κεφάλαιο θα παρουσιαστεί ο τρόπος με τον οποίο αλληλεπιδρά ο χρήστης του συστήματος. Στην περίπτωση μας είναι ο πελάτης μιας τράπεζας, με το σύστημα κάποιου μηχανήματος αυτόματης ανάληψης μετρητών. Η διαδικασία είναι γνωστή καθώς οι περισσότεροι έχουμε κάνει έστω μια ανάληψη ή κατάθεση χρημάτων σε κάποιο τέτοιο μηχάνημα. Για να είναι εφικτή η χρήση του ATM πρέπει πρώτα ο πελάτης να δημιουργήσει έναν λογαριασμό έτσι ώστε να καταχωρηθεί στο αρχείο της τράπεζας και να του δοθεί κάποιος κωδικός και αριθμός πελάτη.

Αρχικά εξηγείται η λογική και οι απαιτήσεις του συστήματος, κατόπιν εκφράζονται οι λειτουργίες του σε πρόγραμμα/κώδικα Java και τελικά γίνεται μετατροπή του κώδικα Java σε διαγράμματα UML χρησιμοποιώντας το *εργαλείο Visual Paradigm.*

5.1. Εισαγωγή

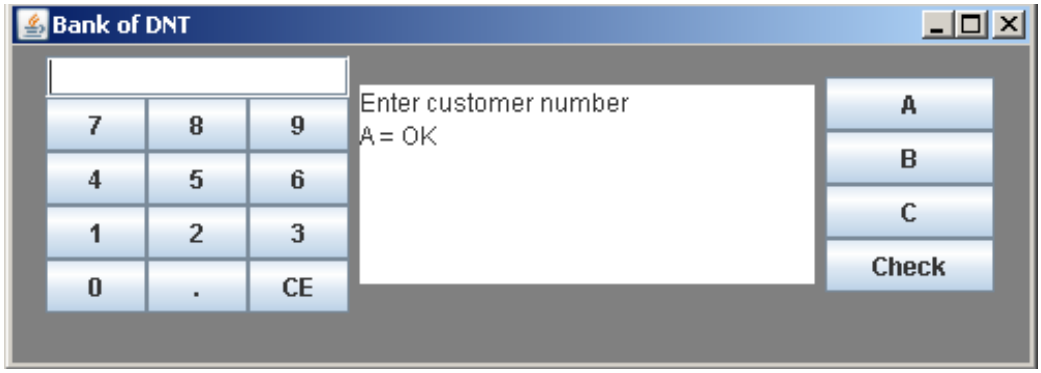
Σε ένα τραπεζικό σύστημα ο ταμίας δημιουργεί τραπεζικούς λογαριασμούς. Οι πελάτες μπορούν να πραγματοποιούν καταθέσεις και αναλήψεις χρημάτων μέσω του ταμία. Το τραπεζικό σύστημα καταγράφει κάθε δοσοληψία κατάθεσης ή ανάληψης χρημάτων και ενημερώνει το υπόλοιπο του λογαριασμού του πελάτη. Ένας πελάτης μπορεί να εξυπηρετείται και από τραπεζικά υποκαταστήματα. Ένα σύστημα διαχείρισης Τραπεζικών λογαριασμών έχει σε γενικές γραμμές τις παρακάτω απαιτήσεις:

- ✓ Εκκίνηση του συστήματος.
- ✓ Κλείσιμο του συστήματος.
- ✓ Εισαγωγή αριθμού (λογαριασμού) πελάτη.
- ✓ Εισαγωγή κωδικού πελάτη.
- ✓ Έλεγχος στοιχείων πελάτη.
- ✓ Δημιουργία ανάληψης(withdraw).
- ✓ Δημιουργία κατάθεσης(deposit).

Καταχώρηση στοιχείων πελατών

Με την εγκατάσταση του συστήματος, καταχωρούνται αρχικά σε ένα αρχείο οι πελάτες οι οποίοι έχουν ανοίξει λογαριασμό στην συγκεκριμένη τράπεζα από κάποιον υπάλληλο της τράπεζας. Για την χρήση της εφαρμογής ο κάθε πελάτης πρέπει να δώσει τα αληθή στοιχεία του στον τραπεζικό υπάλληλο, που είναι υπεύθυνος για την εισαγωγή του πελάτη στο αρχείο αποθήκευσης πελατών, τότε του δίνεται ένας μοναδικός αριθμός λογαριασμού και ένας κωδικός που του επιτρέπουν να κάνει κατάθεση, ανάληψη ή μεταφορά χρημάτων.

Ο πελάτης (χρήστης) χρησιμοποιεί την εφαρμογή όπου διαχειρίζεται τραπεζικούς λογαριασμούς για να ελέγξει τα υπόλοιπα των τραπεζικών λογαριασμών, να καταθέσει χρήματα, να κάνει ανάληψη μετρητών ή και μεταφορά κεφαλαίων. Στο πιο κάτω σχήμα βλέπετε το παράθυρο της εφαρμογής που εμφανίζεται στον τρέχον πελάτη.



Εικόνα 5-1 Παράθυρο εφαρμογής

5.2. Περιγραφή περιπτώσεων χρήσης

Από τον κώδικα που υλοποιεί την εφαρμογή διαχείρισης τραπεζικών λογαριασμών και από τις απαιτήσεις που προαναφέρθηκαν στην εισαγωγή του κεφαλαίου προκύπτει το διάγραμμα του *Σχήματος 5-1*.

Ο χειριστής του συστήματος είναι αυτός που θα εισάγει στην εφαρμογή τα στοιχεία του πελάτη. Η Π.Χ-Εισαγωγή αριθμού πελάτη ενεργοποιείται όταν ο πελάτης πατήσει το κουμπί Α και ανάλογα με την κατάσταση που βρίσκεται η εφαρμογή, εκτελεί τον κώδικα

έτσι ώστε να ολοκληρωθεί η τραπεζική συναλλαγή. Αυτό φαίνεται στο παρακάτω απόσπασμα κώδικα :

```
1. private class AButtonListener implements ActionListener
2. {
3. public void actionPerformed(ActionEvent event)
4. {
5. if (state == START_STATE){
6. setCustomerNumber();


7. }
8. else if (state == PIN_STATE){
9. selectCustomer();
10. display.setText(currentCustomer.toString()+ "\n If elements are correct press A
    or press C");
11. }
12. else if (state == ACCOUNT_STATE)
13. selectAccount(CHECKING_ACCOUNT);
14. else if (state == TRANSACT_STATE)
15. withdraw();
16. }
17. }
18. public void setCustomerNumber()
19.   {
20.     customerNumber = (int)pad.getValue();
21.     setState(PIN_STATE);
22.   }
```

```
23. public void selectCustomer()
24.   {
25.     int pin = (int)pad.getValue();
26.     currentCustomer= theBank.findCustomer(customerNumber, pin);
27.     if (currentCustomer == null)
28.       setState(START_STATE);
29.     else
30.       setState(ACCOUNT_STATE);
31.   }
```

Στην γραμμή 1 γίνεται κλήση της κλάσης *AButtonListener* η οποία υλοποιεί την κλάση *ActionListener.*

Στην γραμμή 3 καλείται η μέθοδος *actionPerformed(ActionEvent event)* αμέσως μετά το πάτημα του κουμπιού Α από τον χρήστη. Στην μέθοδο *actionPerformed(ActionEvent event)* αποθηκεύεται η ενέργεια του χρήστη στην μεταβλητή event. Η εφαρμογή χωρίζεται σε τέσσερις καταστάσεις *START_STATE, PIN_STATE, ACCOUNT_STATE* και *TRANSACT_STATE*. Αν βρίσκεται στην αρχική κατάσταση τότε καλείται η μέθοδος *setCustomerNumber()* η οποία αποθηκεύει στην μεταβλητή customerNumber οτιδήποτε έχει πληκτρολογηθεί στο πεδίο κειμένου και αλλάζει την κατάσταση από *START_STATE* σε *PIN_STATE.* Αυτό φαίνεται στις γραμμές 20,21.

Η Π.Χ- Εισαγωγή κωδικού πελάτη αρχίζει εφόσον πατηθεί το κουμπί Α και η κατάσταση της εφαρμογής βρίσκεται στο *PIN_STATE*. Στην συνέχεια στην γραμμή 8 γίνεται έλεγχος εάν ισχύουν αυτά και καλείται η μέθοδος *selectCustomer()* στην γραμμή 23 του πιο πάνω κώδικα. Στην μέθοδο αυτή αποθηκεύεται στην γραμμή 25 ο κωδικός που έχει πληκτρολογήσει ο τρέχον πελάτης στην μεταβλητή pin και στην γραμμή 26 γίνεται κλήση της μεθόδου *findCustomer(customerNumber,pin).* Η συγκεκριμένη μέθοδος βρίσκεται στην κλάση Bank και κάνει εύρεση του τρέχοντος πελάτη με τα στοιχεία που έχει δοθεί στον ταμία, δηλαδή τον αριθμό λογαριασμού και τον κωδικό πρόσβασης, και το αποτέλεσμα καταχωρείται στην μεταβλητή *currentCustomer*. Στην

γραμμή 27 γίνεται έλεγχος αν η μεταβλητή *currentCustomer* είναι άδεια δηλαδή ο πελάτης έδωσε λάθος στοιχεία κατά την πληκτρολόγηση, τότε το σύστημα επιστρέφει στην αρχική κατάσταση και ζητάει πάλι τον αριθμό πελάτη, αλλιώς αλλάζει η κατάσταση από *PIN_STATE* σε *ACCOUNT_STATE*.

Στην γραμμή 10 αφού προηγηθούν οι άλλες δυο περιπτώσεις χρήσης γίνεται έλεγχος των στοιχείων του πελάτη με αστυνομική ταυτότητα, διαβατήριο ή άδεια οδήγησης τα οποία είναι καταχωρημένα σε ένα αρχείο. Η διαδικασία αυτή αποτελεί την Π.Χ- Έλεγχος στοιχείων πελάτη. Εφόσον τα στοιχεία είναι σωστά ο πελάτης πατάει το κουμπί Α, η κατάσταση πλέον βρίσκεται στην *ACCOUNT_STATE* και ο πελάτης θα πρέπει να επιλέξει το είδος του λογαριασμού που θέλει, να κάνει ανάληψη ή κατάθεση. Αυτές οι δυο ενέργειες αποτελούν δυο ξεχωριστές περιπτώσεις χρήσης. Όταν ο πελάτης επιλέξει το είδος του λογαριασμού δηλαδή είτε Checking account είτε Saving account η κατάσταση της εφαρμογής αλλάζει από *ACCOUNT_STATE* σε *TRANSACT_STATE*.

Αν πατηθεί το κουμπί Α τότε εκτελείται η Π.Χ- Δημιουργία ανάληψης *(withdraw)* όπως φαίνεται στην γραμμή 15 του παραπάνω αποσπάσματος κώδικα και αφαιρείται από το αρχικό ποσό του χρήστη το ποσό που πληκτρολόγησε.

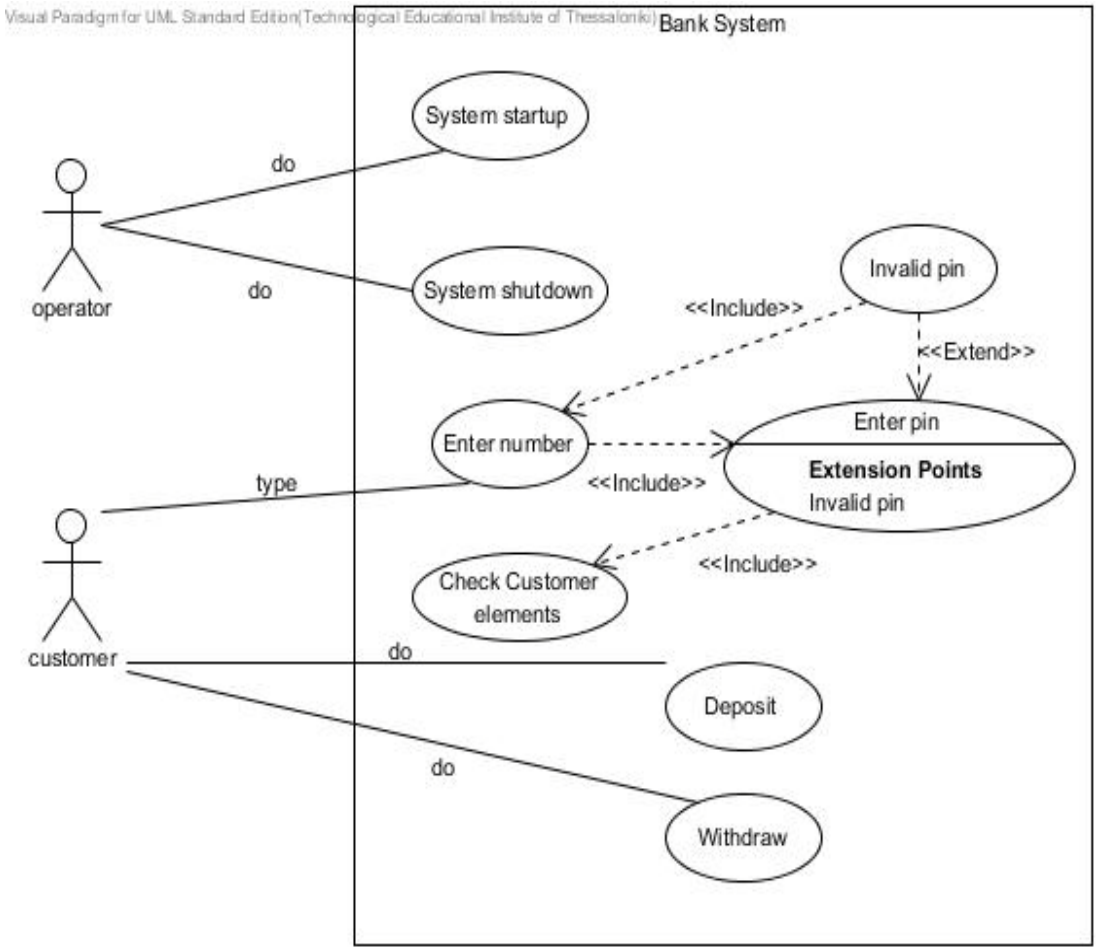
Η Π.Χ-Δημιουργία Κατάθεσης τίθεται σε λειτουργία εφόσον πατηθεί το κουμπί Β και παρουσιάζεται στο κάτω κομμάτι κώδικα:

```
1. private class BButtonListener implements ActionListener
2. {
3. public void actionPerformed(ActionEvent event)
4. {
5. if (state == ACCOUNT_STATE)
       selectAccount(SAVINGS_ACCOUNT);
6. else if (state == TRANSACT_STATE)
       deposit();
7. }
8. }
```

Εφόσον έχει πατηθεί το κουμπί Β εκτελείται η κλάση *BButtonListener* η οποία υλοποιεί την κλάση *ActionListener*. Γίνεται έλεγχος της κατάστασης εάν βρίσκεται στην

κατάσταση *ACCOUNT_STATE και* γίνεται επιλογή του λογαριασμού *SAVINGS_ACCOUNT*. Στην περίπτωση που το σύστημα βρίσκεται στην κατάσταση TRANSACT_STATE καλείται η μέθοδος *deposit()* όπου εκτελείται η Π.Χ-Δημιουργία κατάθεσης.

Όπως αναφέρθηκε στην προηγούμενη ενότητα υπάρχουν επτά περιπτώσεις χρήσης στην συγκεκριμένη εφαρμογή. Παρακάτω γίνεται ανάλυση μιας εκ των επτά περιπτώσεων χρήσης και συγκεκριμένα η Π.Χ-Εισαγωγή αριθμού πελάτη. Ο πελάτης πηγαίνει στο αυτόματο μηχάνημα ανάληψης μετρητών δίνει τον μοναδικό αριθμό λογαριασμού ο οποίος του είχε δοθεί την πρώτη φορά όταν δημιούργησε τον λογαριασμό, ο χειριστής του συστήματος πληκτρολογεί τον αριθμό, στην περίπτωση μας είναι ο ίδιος ο πελάτης στο πεδίο που εμφανίζεται στην οθόνη του. Εάν ο αριθμός είναι σωστός, τότε αλλάζει η αρχική κατάσταση στην εφαρμογή και πλέον το σύστημα περιμένει την εισαγωγή του κωδικού πελάτη. Στη περίπτωση που ο αριθμός λογαριασμού είναι λάθος, μετά την εισαγωγή του κωδικού επιστρέφει στην αρχική κατάσταση και ξαναρωτάει τον αριθμό λογαριασμού. Όταν το σύστημα βρίσκεται σε άλλη κατάσταση εκτός από την αρχική τότε επιστρέφει στην αρχική κατάσταση και ζητάει τον αριθμό λογαριασμού.



Σχήμα 5-1 Διάγραμμα περιπτώσεων χρήσης

5.3. Διάγραμμα δραστηριοτήτων

Στην προηγούμενη ενότητα αναλύθηκε μια περίπτωση χρήσης ΠΧ-Εισαγωγή αριθμού πελάτη. Με βάση αυτή την περίπτωση χρήσης ο ταμίας που είναι ο κύριος χειριστής του συστήματος εισάγει στο σύστημα τον αριθμό λογαριασμού του πελάτη, έτσι ώστε να πραγματοποιηθεί μια τραπεζική συναλλαγή. Οι απαραίτητοι έλεγχοι για την πραγματοποίηση της συναλλαγής φαίνονται μέσα από τον πιο κάτω απόσπασμα κώδικα.

```
1.  public  Transaction()
2.     {
3.  …
4.  ….
5.  ….
6.  setState(START_STATE);
7.  }// δομητής
8.
9.  public  void setState(int newState)
10. {
11. state = newState;
12. pad.clear();
13. if (state == START_STATE)
14.    display.setText("Enter customer number\nA = OK");
15. else if (state == PIN_STATE)
16.    display.setText("Enter PIN\n A = OK");
17. else if (state == ACCOUNT_STATE)
18.    display.setText("Select Account\n"
19.   + "A = Checking\nB = Savings\nC = Exit");
20. else if (state == TRANSACT_STATE)
21.    display.setText("Balance = " + currentAccount.getBalance()+ "\n Enter amount
       and select transaction\n"
22.    + "A = Withdraw\n B = Deposit\n C = Cancel");


23. }


24. private class AButtonListener implements ActionListener
25. {
26. public void actionPerformed(ActionEvent event)
27. {
28. if (state == START_STATE){
29.     setCustomerNumber();


30. }
```

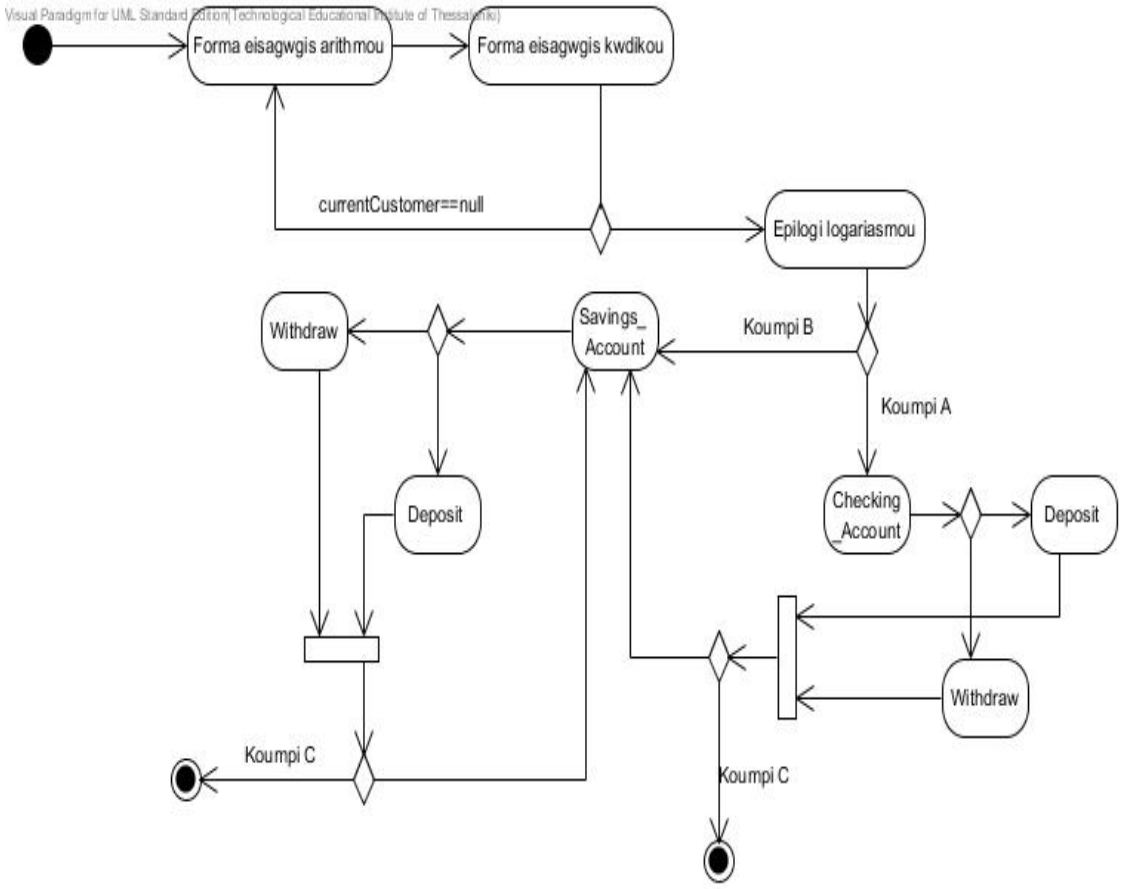
```
31. else if (state == PIN_STATE){
32.     selectCustomer();
33. }
34. else if (state == ACCOUNT_STATE)
35.     selectAccount(CHECKING_ACCOUNT);
36. else if (state == TRANSACT_STATE)
37.     withdraw();
38. }
39. }
```

Για να εισάγει ο πελάτης τον αριθμό πελάτη εμφανίζεται μια φόρμα στην οθόνη του όπου ζητάει τον αριθμό πελάτη. Όταν εκτελείται για πρώτη φορά η εφαρμογή η κατάσταση του συστήματος ορίζεται στην αρχική κατάσταση, δηλαδή ως *START_STATE*. Αυτό γίνεται μέσα στον δομητή της κλάσης *Transaction*. Άρα εφόσον η κατάσταση της εφαρμογής βρίσκεται στην αρχική μορφή η μεταβλητή *state* έχει οριστεί ως *START_STATE*. Όπως αναφέρεται παραπάνω η κατάσταση της εφαρμογής αρχικοποιείται στον δομητή της κλάσης στην γραμμή 6 και εκτελείται η μέθοδος *setState()* στην γραμμή 9 με τιμή της μεταβλητής *newState* ως *START_STATE*.

Στην γραμμή 13 γίνεται έλεγχος εάν η κατάσταση της εφαρμογής βρίσκεται στην αρχική κατάσταση, αν ναι τότε εμφανίζεται στην οθόνη του χειριστή το μήνυμα για να εισάγει τον αριθμό πελάτη. Όταν πληκτρολογήσει ο χρήστης τον αριθμό και πατήσει το κουμπί Α, καλείται η κλάση *AButtonListener* η οποία υλοποιεί στην γραμμή 24 την κλάση ActionListener. Στην συνέχεια γίνεται έλεγχος της κατάστασης της εφαρμογής εάν βρίσκεται στην αρχική και καλείται η μέθοδος *setCustomerNumber()*. Η συγκεκριμένη μέθοδος απλά ενημερώνει την μεταβλητή *customerNumber* με τον αριθμό που πληκτρολόγησε ο ταμίας και αλλάζει την κατάσταση από *START_STATE* σε *PIN_STATE* μέσω της μεθόδου *setState()*. Ορίζοντας την κατάσταση ως *PIN_STATE*, εμφανίζεται στην οθόνη του πελάτη το μήνυμα εισαγωγής του κωδικού πελάτη γραμμή 16.

Όταν ο πελάτης πληκτρολογήσει τον κωδικό του και πατήσει το κουμπί Α καλείται η μέθοδος *selectCustomer()* για να επιλεγεί ο πελάτης με τον συγκεκριμένο αριθμό λογαριασμού και κωδικό πρόσβασης. Ο έλεγχος αυτός γίνεται μέσω της μεθόδου *findCustomer(customerNumber, pin)*, που υλοποιείται στην κλάση Bank. Αν τα στοιχεία που πληκτρολόγησε ο πελάτης είναι αληθή τότε η κατάσταση αλλάζει σε *ACCOUNT_STATE* διαφορετικά επιστρέφει η εφαρμογή στην αρχική της κατάσταση.

Στη συνέχεια γίνεται επιλογή του τύπου λογαριασμού και αν θα γίνει ανάληψη ή κατάθεση τα οποία αποτελούν δυο ξεχωριστές περιπτώσεις χρήσης. Από την παραπάνω ανάλυση του κώδικα προέκυψε το *Σχήμα 5-2*.



Σχήμα 5-2 Διάγραμμα Δραστηριοτήτων

5.4. Διάγραμμα αλληλεπίδρασης

Η «Αλληλεπίδραση» ορίζει και περιγράφει την επικοινωνία ανάμεσα σε αντικείμενα ή κλάσεις. Η επικοινωνία ορίζεται με την μορφή μηνυμάτων που ανταλλάσσονται ανάμεσα στα αντικείμενα ή τις κλάσεις Τα μηνύματα μπορεί είναι κλήσεις σε συγκεκριμένες μεθόδους *(method invocations)* ή διαδικασίες που επηρεάζουν ένα αντικείμενο ή μια κλάση. Τα μηνύματα είναι μερικώς διατεταγμένα *(partially ordered)* σε σχέση με το χρόνο. Η ακολουθία των μηνυμάτων παρουσιάζεται είτε με διαγράμματα ακολουθίας που εστιάζουν στην χρονική ακολουθία των μηνυμάτων είτε με διαγράμματα επικοινωνίας που εστιάζουν στις σχέσεις μεταξύ των αντικειμένων που ανταλλάσσουν μηνύματα. Από τον κώδικα οι κλάσεις που εμπλέκονται για να εκτελεστεί η Π.Χ-Εισαγωγή αριθμού πελάτη είναι η κλάση *Pliktrologio* και η κλάση *Transaction*. Η πρώτη κλάση είναι υπεύθυνη για τα ερεθίσματα που περνά ο χρήστης μέσω του πληκτρολογίου στην εφαρμογή. Αυτό δεν μας αφορά τόσο στην περίπτωση χρήσης μας, δηλαδή πως από το πάτημα του κουμπιού εκτελούνται κάποιες μέθοδοι είναι θέμα προγραμματιστικό και όχι πρόβλημα ανάλυσης.

Ας δούμε τον κώδικα στην κλάση *Transaction* που είναι υπεύθυνος για την περίπτωση χρήσης στην οποία αναφερόμαστε.

```
1.  public  void setState(int newState)
2.  {
3.  state = newState;
4.  pad.clear();
5.
6.  if (state == START_STATE)
7.  display.setText("Enter customer bank account number\n A = OK");
8.  else if (state == PIN_STATE)
9.  display.setText("Enter PIN\n A = OK");
10. else if (state == ACCOUNT_STATE)
11. display.setText("Select Account\n"
12. + "A = Checking\nB = Savings\n C = Exit");
```

```
13. else if (state == TRANSACT_STATE)
14. display.setText("Balance = " + currentAccount.getBalance()+ "\n Enter amount and
    select transaction\n"
15. + "A = Withdraw\n B = Deposit\n C = Cancel");
16. }
```

```
17. private class AButtonListener implements ActionListener
18. {
19. public void actionPerformed(ActionEvent event)
20. {
21. if (state == START_STATE){
22. setCustomerNumber();
23. }
24. else if (state == PIN_STATE){
25. selectCustomer();
26. }
27. else if (state == ACCOUNT_STATE)
28. selectAccount(CHECKING_ACCOUNT);
29. else if (state == TRANSACT_STATE)
30. withdraw();
31. }
32. }
33. public void setCustomerNumber()
34. {
35. customerNumber = (int)pad.getValue();
36. setState(PIN_STATE);
    }c void setCustomerNumber()
1.  {
2.  customerNumber = (int)pad.getValue();
3.  setState(PIN_STATE);
4.  }
```

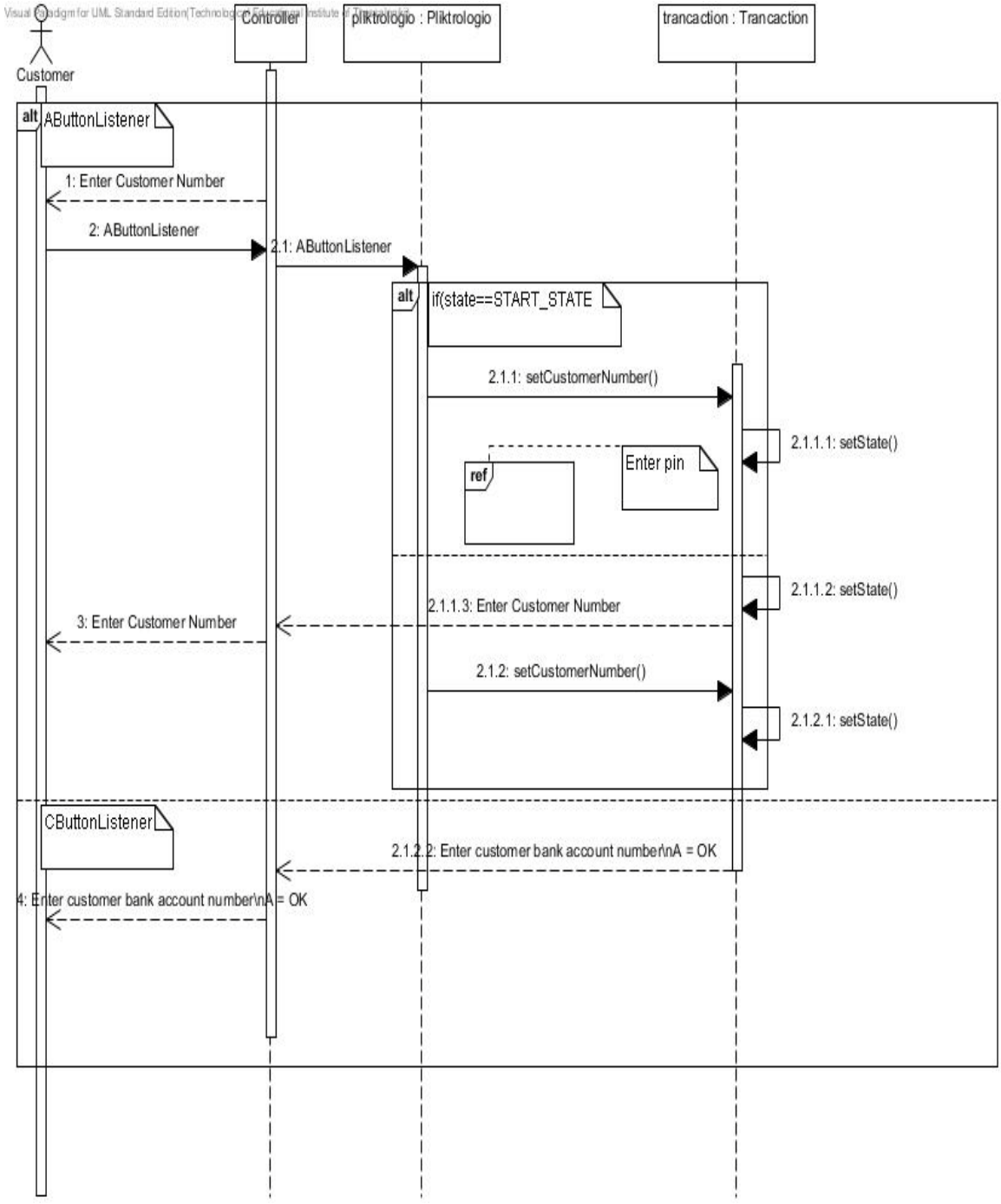
Όταν γίνεται εκκίνηση του συστήματος από τον διαχειριστή τότε από τον δομητή της κλάσης *Transaction* καλείται η μέθοδος *setState(int newState);* Η μέθοδος αυτή δίνει τιμή στην μεταβλητή *newState* ως *START_STATE,* δηλαδή ορίζει την κατάσταση της εφαρμογής ως αρχική κατάσταση και εκτελεί οποιαδήποτε γραμμή κώδικα βρίσκεται σε αυτήν την εναλλακτική περίπτωση όπως φαίνεται στις γραμμές 5-6.

Στην γραμμή 5 κάνει έλεγχο σε ποιά κατάσταση βρίσκεται και στη γραμμή 6 εμφανίζει στην οθόνη το μήνυμα για εισαγωγή του αριθμού πελάτη. Ο πελάτης πληκτρολογεί τον αριθμό λογαριασμού και πατάει το κουμπί Α. Το σύστημα μέσω της κλάσης *Pliktrologio* δέχεται το ερέθισμα ότι έχει πιεστεί κάποιο κουμπί. Στην γραμμή 16 αφού έχει πατηθεί το κουμπί εκτελείται αμέσως η κλάση. Στην συνέχεια γίνεται έλεγχος εάν η κατάσταση είναι η αρχική, δηλαδή *state=START_STATE*. Όπως φαίνεται στο *Σχήμα 5-4* το ορθογώνιο κουτί με την ετικέτα *alt* συμβολίζει τις εναλλακτικές ροές. Αν η κατάσταση είναι η αρχική τότε εκτελείται η μέθοδος *setCustomerNumber()* στην γραμμή 32 του κώδικα και μήνυμα 2.1.1 στο διάγραμμα που ακολουθεί. Αφού εκτελεστεί η μέθοδος αλλάζει η κατάσταση της εφαρμογής από αρχική σε κατάσταση PIN_STATE και καλείται η περίπτωση χρήσης Π.Χ-Εισαγωγή κωδικού. Αυτό γίνεται για τον λόγο ότι γίνεται έλεγχος για την ορθότητα του κωδικού και του αριθμού λογαριασμού ταυτόχρονα για εξοικονόμηση χρόνου. Αν ο αριθμός είναι λάθος τότε εμφανίζεται στην οθόνη το μήνυμα ότι ο χρήστης πρέπει να εισάγει αριθμό λογαριασμού στο *Σχήμα 5-4* και μήνυμα 2.1.1.3. Αυτό οφείλεται στην αλλαγή της κατάστασης από *PIN_STATE* σε *START_STATE* στο διάγραμμα μήνυμα 2.1.1.2. Ο έλεγχος για το αν υπάρχει πελάτης με τα στοιχεία που πέρασε ο τρέχον χρήστης γίνεται στην μέθοδο selectCustomer(); όπως φαίνεται στην γραμμή 3 όπου καλείται η μέθοδος findCustomer(customerNumber, pin) της κλάσης Bank με τα στοιχεία που έδωσε ο τρέχον χρήστης του παρακάτω αποσπάσματος κώδικα.

```
1. public void selectCustomer(){
2. int pin = (int)pad.getValue();
3. currentCustomer=
   theBank.findCustomer(customerNumber, pin);

4. if (currentCustomer == null)
5. setState(START_STATE);
6. else {
7. setState(ACCOUNT_STATE);
8. }
```

```
9.  }
```

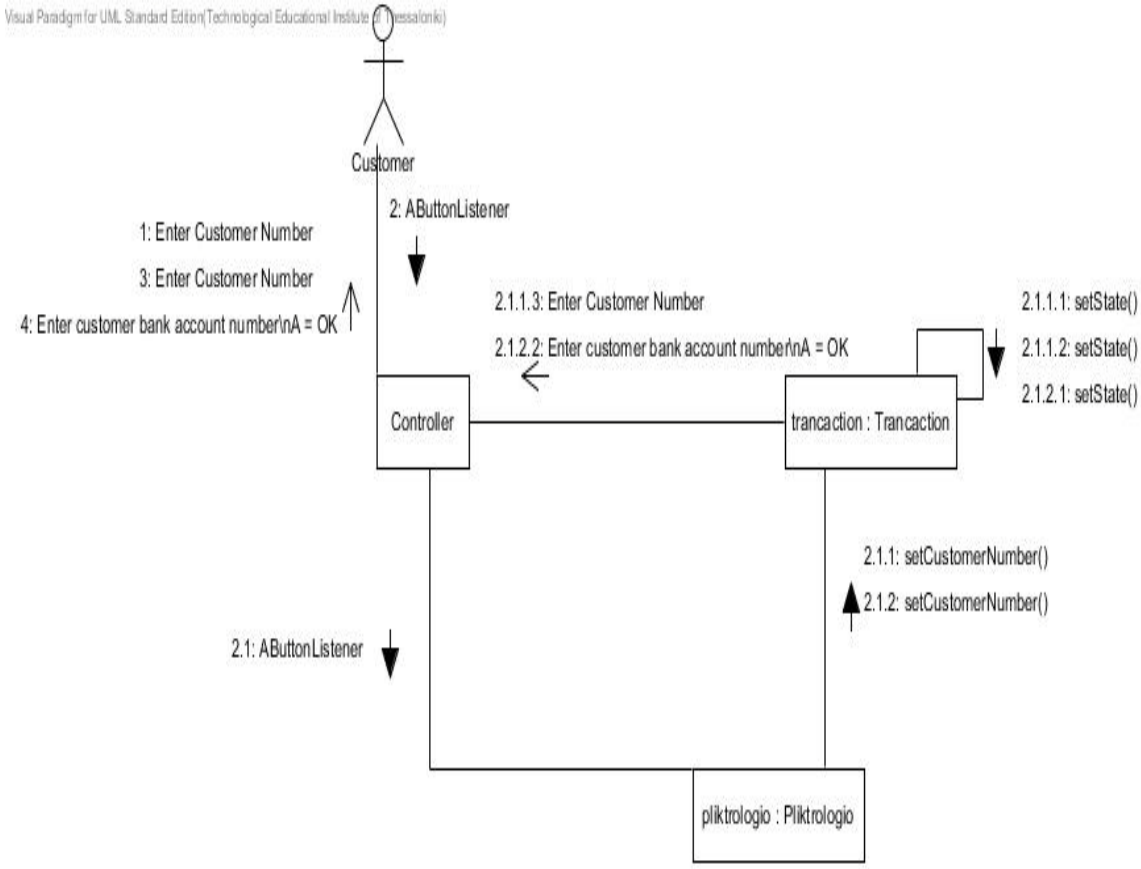
Αν ο πελάτης πατήσει το κουμπί C τότε επιστρέφει μήνυμα στην οθόνη ότι πρέπει να πληκτρολογήσει τον αριθμό πελάτη και να πατήσει το κουμπί Α στο *Σχήμα 5-3* και μήνυμα 2.1.2.2 στο διάγραμμα. Πιο κάτω δίνεται το ολοκληρωμένο διάγραμμα ακολουθίας για την Π.Χ-Εισαγωγή αριθμού πελάτη.



Σχήμα 5-3 Διάγραμμα Ακολουθίας Π.Χ-Εισαγωγή αριθμού πελάτη

Σε ένα διάγραμμα συνεργασίας τα αντικείμενα απεικονίζονται με τις γραμμές συσχετίσεων των κλάσεων τους να τα ενώνουν, δηλαδή απεικονίζονται οι στατικές συνδέσεις μεταξύ των αντικειμένων. Ενώ τα διαγράμματα ακολουθίας απεικονίζουν κυρίως τη χρονική ροή των μηνυμάτων σε ένα σενάριο μιας περίπτωσης χρήσης, τα

διαγράμματα συνεργασίας χρησιμοποιούνται για να παρουσιάσουν τις σχέσεις μεταξύ αντικειμένων. Δεν υπάρχει συγκεκριμένη μορφή (τα αντικείμενα μπορούν να εμφανίζονται σε οποιοδήποτε σημείο του διαγράμματος) ενώ για να απεικονιστεί η ακολουθία των μηνυμάτων που ανταλλάσσονται χρησιμοποιείται αρίθμηση. Τα διαγράμματα ακολουθίας και συνεργασίας θεωρούνται συμπληρωματικά καθώς περιέχουν τις ίδιες πληροφορίες αλλά κάθε ένα δίνει μια διαφορετική οπτική γωνία (σε πολλά εργαλεία το ένα είδος διαγράμματος παράγεται αυτόματα από το άλλο). Το διάγραμμα συνεργασίας που προκύπτει από το πιο πάνω διάγραμμα ακολουθίας δίνεται στο *Σχήμα 5-4*που ακολουθεί.



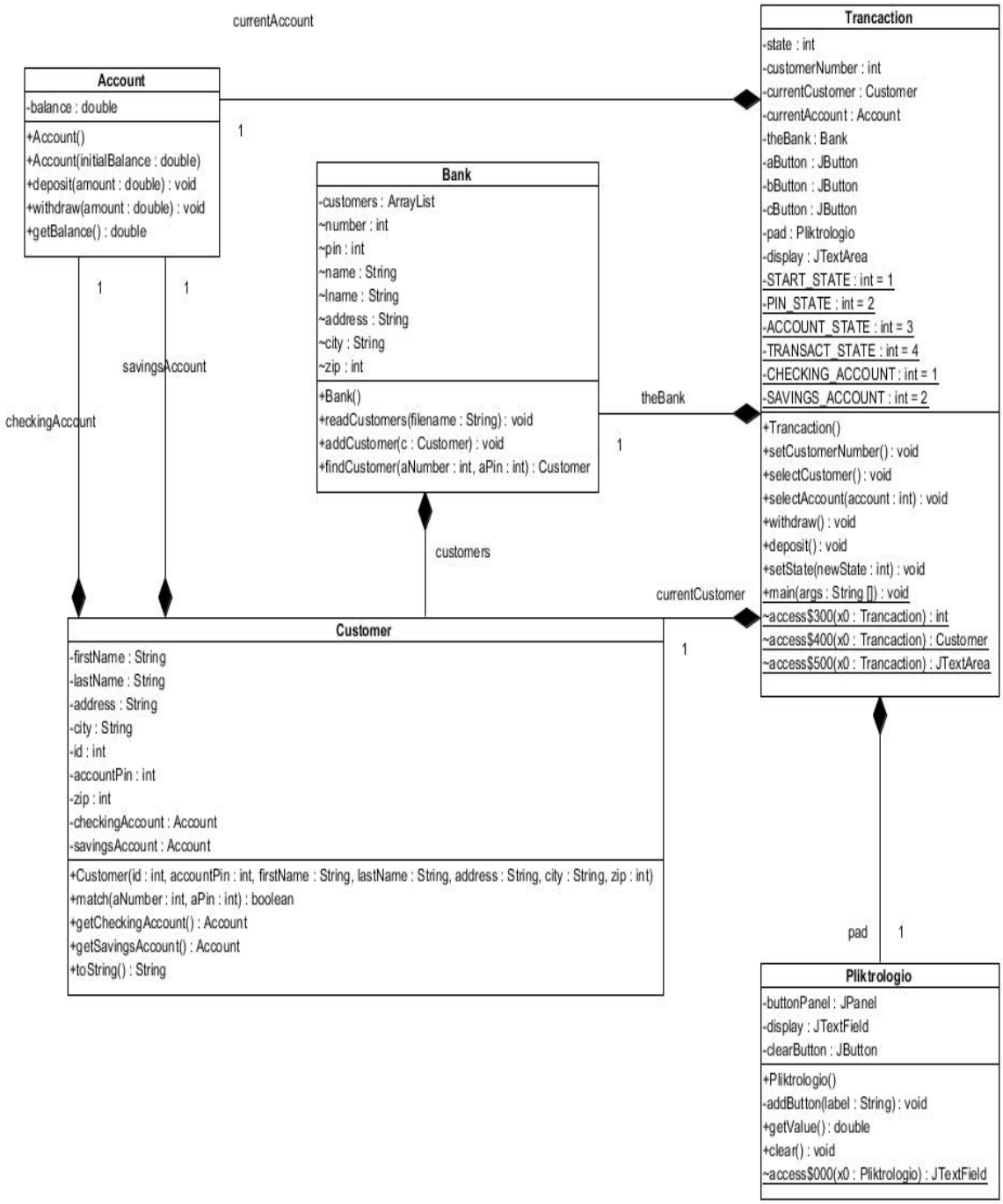
Σχήμα 5-4 Διάγραμμα Συνεργασίας

5.5. Το στατικό μοντέλο του συστήματος

Όπως έχει αναφερθεί, το Στατικό Μοντέλο Δομής *(Structural Model ή Object Model)* παρουσιάζει και περιγράφει τη στατική δομή του συστήματος και των υποσυστημάτων σαν ένα σύνολο από κλάσεις, αντικείμενα, οντότητες και σχέσεις ανάμεσα σε αυτά. Για τη δημιουργία του στατικού Μοντέλου Δομής ενός συστήματος χρησιμοποιούμε διαγράμματα που έχουν να κάνουν με την λεπτομερή στατική σχεδιαστική δομή του συστήματος όπως είναι τα Διαγράμματα Κλάσεων *(Class Diagrams).*

5.5.1. Διάγραμμα κλάσεων του συστήματος

Στην εφαρμογή διαχείρισης τραπεζικών λογαριασμών το σύστημα αποτελείται από πέντε κλάσεις με την κάθε κλάση να περιέχει αντικείμενα, ιδιότητες και σχέσεις με άλλες κλάσεις. Αν όλα αυτά τα στοιχεία τα βάλουμε σε ένα διάγραμμα τότε δημιουργούμε το διάγραμμα κλάσης του συστήματος που μας δίνει μια γενική άποψη για το σύστημα που αναλύουμε. Στο *Σχήμα 5-5* που ακολουθεί δίνεται το ολοκληρωμένο διάγραμμα κλάσεων για την εφαρμογή διαχείρισης τραπεζικών λογαριασμών.



Σχήμα 5-5 Διάγραμμα κλάσεων

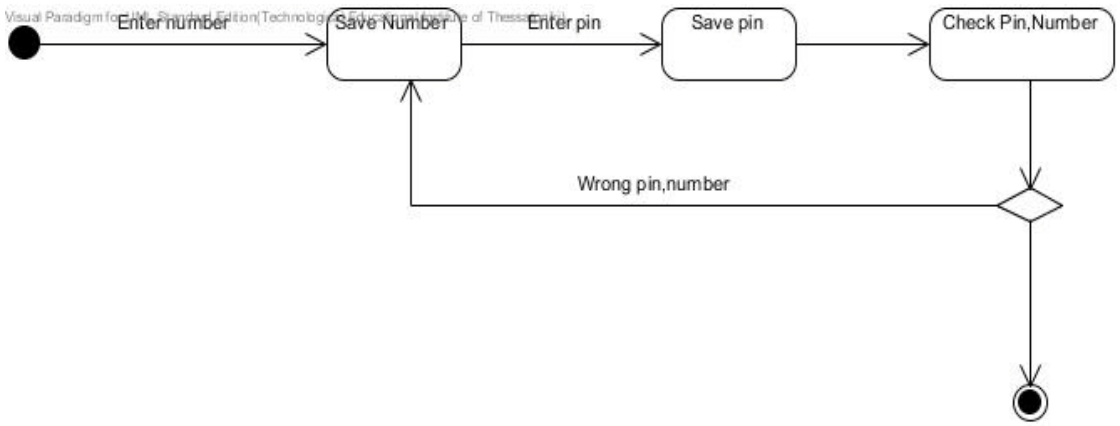
5.6. Το δυναμικό μοντέλο του συστήματος

Τα δυναμικά μοντέλα απεικονίζουν την δυναμική συμπεριφορά του συστήματος, παραδείγματος χάριν πώς αποκρίνεται στις ενέργειες των χρηστών ή σε άλλα εξωτερικά ερεθίσματα και πώς διαμορφώνεται η εσωτερική του κατάσταση κατά τη λειτουργία του. Το δυναμικό μοντέλο αποτελείται από τα διαγράμματα καταστάσεων, συστατικών και ανάπτυξης.

5.6.1. Διάγραμμα καταστάσεων

Στην *Π.Χ-Εισαγωγή αριθμού πελάτη*, ο πελάτης πληκτρολογεί τον αριθμό και στη συνέχεια αλλάζει η κατάσταση της εφαρμογής από αρχική *START_STATE* σε κατάσταση *PIN_STATE*. Έπειτα ο χρήστης του συστήματος πληκτρολογεί τον κωδικό πρόσβασης του και αυτόματα γίνεται έλεγχος των στοιχείων που έδωσε ο πελάτης. Εάν είναι σωστά τελειώνει η περίπτωση χρήσης «Εισαγωγή αριθμού πελάτη». Στην περίπτωση που ο αριθμός είναι λάθος, τότε η εφαρμογή επιστρέφει στην αρχική κατάσταση και ζητάει πάλι να δώσει ο ταμίας τον αριθμό πελάτη για να γίνει ξανά ο έλεγχος στοιχείων.

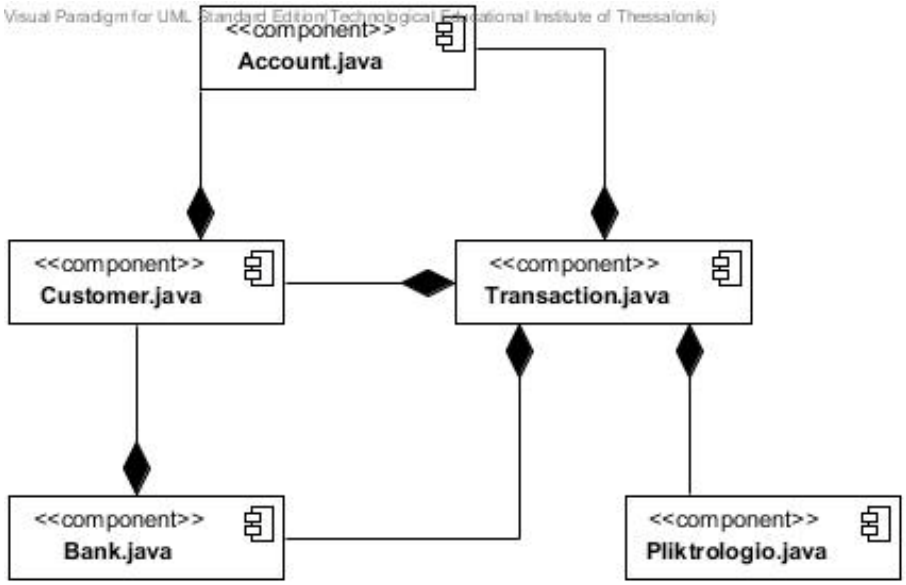
Στο *Σχήμα 5-7*, περιγράφονται οι διαφορετικές καταστάσεις ενός συστατικού σε ένα σύστημα, ο κύκλος ζωής των στιγμιότυπων των κλάσεων και η εκτέλεση μίας πράξης ενός στιγμιότυπου μίας κλάσης. Στο πιο κάτω διάγραμμα φαίνεται πως αλλάζει από την μια κατάσταση στην άλλη, δηλαδή εκεί που ο χρήστης πληκτρολογεί τον αριθμό πελάτη αμέσως μετά πάει σε άλλη κατάσταση εισαγωγής του κωδικού και στην συνέχεια σε έλεγχο των στοιχείων του πελάτη. Το συγκεκριμένο διάγραμμα αφορά μόνο την περίπτωση χρήσης «Εισαγωγή αριθμού πελάτη», αν και περιέχει και την περίπτωση χρήσης «Εισαγωγή κωδικού» για τον λόγο ότι ο έλεγχος του κωδικού και του αριθμού πελάτη γίνεται ταυτόχρονα για να μην γίνεται ο έλεγχος δυο φορές.



Σχήμα 5-6 Διάγραμμα καταστάσεων Π.Χ-Εισαγωγή αριθμού πελάτη

5.6.2. Διάγραμμα συστατικών

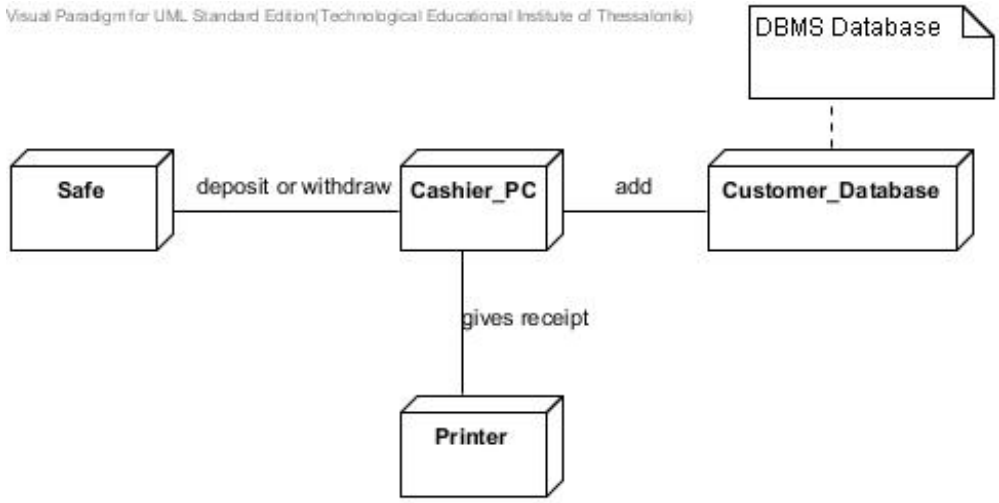
Όπως αναφέρθηκε στο δεύτερο κεφάλαιο τα διαγράμματα συστατικών περιέχουν τα συστατικά του λογισμικού, δηλαδή τα τμήματα του κώδικα και την δομή των αρχείων του κώδικα. Κάθε αρχείο της μορφής .java αποτελεί συστατικό του συστήματος και συμβολίζεται σχηματικά με ένα ορθογώνιο παραλληλόγραμμο. Στην εφαρμογή διαχείρισης τραπεζικών λογαριασμών έχουμε πέντε αρχεία .java άρα πέντε συστατικά που συνεργάζονται μεταξύ τους και υλοποιούν το σύστημα μας. Στο *Σχήμα 5-7* φαίνονται οι κλάσεις του συστήματος και πως η κάθε μια επικοινωνεί με τις άλλες κλάσεις.



Σχήμα 5-7 Διάγραμμα Συστατικών

5.6.3. Διάγραμμα ανάπτυξης

Τα διαγράμματα ανάπτυξης *(deployment diagrams)* αναπαριστούν την αντιστοίχιση του λογισμικού σε επεξεργαστικές μονάδες–κόμβους. Μπορούν να χρησιμοποιηθούν για να δείξουν ποια συστατικά τρέχουν σε ποιους κόμβους. Ένας κόμβους είναι ένα φυσικό αντικείμενο που στη γενική περίπτωση έχει τουλάχιστον μνήμη και δυνατότητα επεξεργασίας. Στην περίπτωση της εφαρμογής μας υπάρχουν τέσσερεις κόμβοι, ένας κόμβος είναι ο υπολογιστής που χρησιμοποιεί ο ταμίας *(Cashier_PC),* μια βάση αποθήκευσης των στοιχείων του κάθε πελάτη *(Local host Database),* ένα χρηματοκιβώτιο *(Safe)* για αποθήκευση ή ανάληψη χρημάτων και τέλος ένας εκτυπωτής *(Printer)* που εκδίδει αποδείξεις ανάλογα με το είδος της τραπεζικής συναλλαγής και του ποσού.

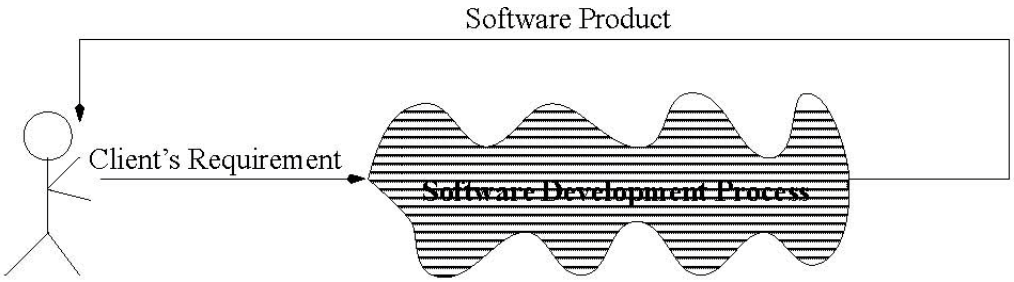


Σχήμα 5-8 Διάγραμμα *Ανάπτυξης*

ΕΠΙΛΟΓΟΣ

Συμπεράσματα

Σε έργα λογισμικού μεγάλης κλίμακας, στα οποία συμμετέχουν δεκάδες προγραμματιστές και παράγονται μπορεί και εκατομμύρια γραμμές πηγαίου κώδικα, είναι πολύ σημαντικό να ακολουθείται μια καλά σχεδιασμένη διαδικασία ανάπτυξης. Σε γενικές γραμμές η πειθαρχημένη διαδικασία ορίζει ποιος θα κάνει τι, πότε και πως για να μπορέσει να φτάσει στον συγκεκριμένο στόχο. Η διαδικασία ανάπτυξης λογισμικού περιγράφεται συχνά ως μια σχέση που αποτελείται από ένα σύνολο δραστηριοτήτων τα οποία απαιτούνται για να μετατρέψουν τις απαιτήσεις του χρήστη σε ένα σύστημα λογισμικού. Στην γενική μορφή, η διαδικασία ανάπτυξης μπορεί να απεικονιστεί όπως φαίνεται στο *Σχήμα 6-1*.



Εικόνα 6-1 Διαδικασία ανάπτυξης λογισμικού

Τον πρώτο καιρό τα έργα λογισμικού δεν ακολουθούσαν καμία διαδικασία ανάπτυξης, απλά ο προγραμματιστής συζητούσε την κατάσταση με τους πιθανούς χρήστες και μετά άρχιζε να γράφει κώδικα. Αυτό θα μπορούσε να λειτουργήσει σε έργα μικρής κλίμακας. Αργότερα καθώς τα προγράμματα γίνονταν όλο και πιο μεγάλα, η διαδικασία ανάπτυξης χωρίστηκε σε αρκετές φάσης, οι οποίες εκτελούνταν στην σειρά. Η κάθε φάση χρησιμοποιούσε διαφορετικές ομάδες εργαζομένων και είχε το δικό της όνομα όπως ανάλυση, σχεδιασμός, κωδικοποίηση και τελική φάση.

Ο αντικειμενοστρεφής προγραμματισμός δημιουργήθηκε για την επίλυση των προβλημάτων που είναι άρρηκτα συνδεδεμένα με την ανάπτυξη μεγάλων προγραμμάτων. Ο αντικειμενοστρεφής προγραμματισμός βοηθά στη διαδικασία του σχεδιασμού, επειδή τα αντικείμενα του προγράμματος αντιστοιχούν με αντικείμενα στον κόσμο του χρήστη.

Ωστόσο ο αντικειμενοστρεφής προγραμματισμός δεν μας λέει από μόνος του τι πρέπει να κάνει το πρόγραμμα έρχεται στο προσκήνιο μόνο όταν έχουν καθοριστεί σαφώς οι στόχοι του έργου. Χρειαζόμαστε μια αρχική φάση η οποία να εστιάζεται στους χρήστες του προγράμματος και να συλλαμβάνει τις ανάγκες του. Αφού επιτευχθεί αυτό, μπορούμε να μεταφράσουμε τις πληροφορίες σε μια αντικειμενοστραφή σχεδίαση προγράμματος. Για να διαπιστώσουμε τι πραγματικά έχει ανάγκη ο χρήστης και στο τέλος να παραδοθεί το προϊόν λογισμικού και να αφήσει ικανοποιημένους τους χρήστες, πρέπει να τηρηθεί μια σειρά φάσεων μέχρι την ολοκλήρωση του έργου. Οι φάσεις αυτές αποτελούν την ενοποιημένη διεργασία η οποία διασφαλίζει μια πειθαρχημένη ανάθεση καθηκόντων και αρμοδιοτήτων μέσα σε ένα οργανισμό ανάπτυξης.

Η Ενοποιημένη διεργασία χωρίζεται σε τέσσερις φάσεις :

- Σύλληψη/ Έναρξη
- Επεξεργασία
- Κατασκευή
- Μετάβαση

Στην φάση της έναρξης μαζεύονται πληροφορίες για τις απαιτήσεις του έργου, τα κύρια χαρακτηριστικά και τους περιορισμούς. Στην φάση της επεξεργασίας σχεδιάζεται η βασική αρχιτεκτονική του συστήματος. Εδώ είναι που προσδιορίζονται οι ανάγκες των χρηστών. Η φάση της κατασκευής περιλαμβάνει τον σχεδιασμό του λογισμικού και την πραγματική γραφή του πηγαίου κώδικα. Στην φάση της μετάβασης παραδίδεται το σύστημα στους χρήστες για έλεγχο και τελική ανάπτυξη.

Σε γενικές γραμμές για την δημιουργία κάποιου έργου λογισμικού χρειάζεται η συλλογή των απαιτήσεων από τους χρήστες, η δημιουργία εικονικού μοντέλου του συστήματος με την βοήθεια της ενοποιημένης γλώσσας μοντελοποίησης (UML). Στην συνέχεια με βάση το εικονικό μοντέλο, οι προγραμματιστές αρχίζουν την αντικειμενοστρεφή σχεδίαση του συστήματος, δηλαδή την δημιουργία του πηγαίου κώδικα. Τέλος γίνεται παράδοση στην κοινότητα των χρηστών για δοκιμή.

ΒΙΒΛΙΟΓΡΑΦΙΑ

Χατζηγεωργίου, Α. (2005), Αντικειμενοστρεφής Σχεδίαση: UML, Αρχές, Πρότυπα και Ευρετικοί Κανόνες, Εκδόσεις Κλειδάριθμος.

Γερογιάννης Β., Κακαρόντζας Γ, Καμέας Α., Σταμέλος Γ. και Φιτσιλής Π., (2006), Αντικειμενοστραφής Ανάπτυξη Λογισμικού με τη UML, Εκδόσεις Κλειδάριθμος,

Lafore R., (2005), Αντικειμενοστρεφής προγραμματισμός με τη C++, Εκδόσεις Κλειδάριθμος

Πηγές στο Διαδίκτυο

- http://www.worldlingo.com/ma/enwiki/el/Object-oriented_programming#Introduction
- http://el.wikipedia.org/
- http://www.google.gr/search?q=%EF%83%98%09.http%3A%2F%2Fwww.businessanalystfaq.com%2Fwhatis-uml-explain.htm&ie=utf-8&oe=utf-8&aq=t&rls=org.mozilla:en-US:official&client=firefox-a
- http://www.dmst.aueb.gr/louridas/lectures/dais/uml/index.html
- http://www.tutorialspoint.com/uml/uml_component_diagram.htm
- http://www.visual-paradigm.com/support/documents/vpumluserguide.jsp
- http://www.kodejava.org/examples/15.html
- http://docs.kde.org/stable/en/kdesdk/umbrello/index.html
- http://www.uml-diagrams.org/examples/bank-atm-example.html

ΠΑΡΑΡΤΗΜΑ

Κώδικας Διαχείρισης Συστήματος Κινηματογράφου

Aithousa.cpp

```
#ifndef __Aithousa_CPP
#define __Aithousa_CPP
#include "Aithousa.hpp"

Aithousa::Aithousa(int arThes)
{
	for (int i=0; i<arThes; i++)
		theseis.push_back(new Thesi(i));
}

std::vector Aithousa::getTheseis()
{
	return theseis;
}

void Aithousa::emfanTheseis()
{
	for (int i=0; i<theseis.size(); i++)
		cout << theseis.at(i);
}
#endif
```

Aithousa.hpp

```
#ifndef __Aithousa_HPP
#define __Aithousa_HPP
#include "Theatro.hpp"
#include "Thesi.hpp"
#include <iostream>
#include <string>
#include <vector>

class Aithousa
{

private:
	std::string onoma;
	std::vector<Thesi*> theseis;

public:
	Aithousa(int arThes);

	std::vector<Thesi*> getTheseis();
```

```
	virtual void emfanTheseis();

private:
	bool initialized;
	void InitializeInstanceFields()
	{
		if ( ! initialized)
		{
			onoma = "T. Karezi";
			theseis = std::vector(10);

			initialized = true;
		}
	}
};
#include "Aithousa.cpp"
#endif
```

Ergo.cpp

```
#ifndef __Ergo_CPP
#define __Ergo_CPP

#include "Ergo.hpp"
#include <iterator>
Ergo::Ergo(std::string titlos)
{
	InitializeInstanceFields();
	this->titlos = titlos;
}

void Ergo::setParastasi(Parastasi *parast)
{
	parastaseis.push_back(parast);
}

void Ergo::emfanParast()
{
	cout << " Εμφανιση παραστασης: \n";
	std::vector<Parastasi*>::const_iterator itr = parastaseis.begin();
	while (itr != parastaseis.end())
	{
		Parastasi *parast = static_cast<Parastasi*>(*itr);
		cout << parast;
		itr++;
	}
}

Parastasi *Ergo::getParastasi(int kwdikos)
{
	std::vector<Parastasi*>::const_iterator itr = parastaseis.begin();
```

```
	while (itr != parastaseis.end())
	{
		Parastasi *parast = static_cast<Parastasi*>(*itr);
		if (parast->getKwdikos()==kwdikos)
			return parast;
		itr++;
	}
	return 0;
}

std::string Ergo::getTitlos()
{
	return titlos;
}
#endif
```

Ergo.hpp

```
#ifndef __Ergo_HPP
#define __Ergo_HPP
#include <iostream>
#include "Parastasi.hpp"
#include <string>
#include <vector>
#include <iterator>

class Ergo
{

private:
	std::string titlos;
	std::vector<Parastasi*> parastaseis;

public:
	Ergo(std::string titlos);

	virtual void setParastasi(Parastasi *parast);

	virtual void emfanParast();

	virtual Parastasi *getParastasi(int kwdikos);

	virtual std::string getTitlos();

private:
	bool initialized;
	void InitializeInstanceFields()
	{
		if ( ! initialized)
		{
			parastaseis = std::vector(3);
```

```
                        initialized = true;
                }
        }
};
#include "Ergo.cpp"

#endif
```

Kratisi.cpp

```
#ifndef __Kratisi_CPP
#define __Kratisi_CPP

#include "Kratisi.hpp"
#include <iterator>

Kratisi::Kratisi(int arithmoskratisis)
{

        this.kwdikos =arithmoskratisis;
}

void Kratisi::setParastasi(Parastasi *parast)
{
        this.parast = parast;
}

void Kratisi::addThesiParast(ThesiParastasis *thesiParast)
{
        this.theseisKrat.push_back(thesiParast);
}

void Kratisi::emfanKratisis()
{
        cout << "\nΑριθμός κράτησης: " << kwdikos << "  " << parast);
        cout << " Θέσεις κράτησης:";
        std::vector<ThesiParastasis*>::const_iterator itr = theseisKrat.begin();
        while (itr != theseisKrat.end())
        {
                ThesiParastasis *thesParast = static_cast<ThesiParastasis*>(*itr);
                puts(thesParast);
                itr++;
        }
}
#endif
```

Kratisi.hpp

```
#ifndef __kratisi_HPP
#define __Kratisi_HPP
#include "Parastasi.hpp"
#include "ThesiParastasis.hpp"
#include <iostream>
```

```
#include <vector>
#include <iterator>

class Kratisi
{

private:
	int kwdikos;
	Parastasi *parast;
	std::vector<ThesiParastasis*> theseisKrat;

public:
	Kratisi(int arithmoskratisis);

	virtual void setParastasi(Parastasi *parast);

	virtual void addThesiParast(ThesiParastasis *thesiParast);

	virtual void emfanKratisis();

private:
	bool initialized;
	void InitializeInstanceFields()
	{
		if ( ! initialized)
		{
			theseisKrat = std::vector(10);

			initialized = true;
		}
	}
};
#include "Keratisi.cpp"

#endif
```

Parastasi.cpp

```
#ifndef __Parastasi_CPP
#define __Parastasi_CPP
#include "Parastasi.hpp"

Parastasi::Parastasi(int kwdikos, std::string wra, std::vector theseis)
{
	this.kwdikos = kwdikos;
	this.wra = wra;
	theseisParast = std::vector(theseis.size());
	for (int i=0; i<theseis.size(); i++)
	{
		Thesi *thesi = static_cast<Thesi*>(theseis.at(i));
		ThesiParastasis *thesiParast = new ThesiParastasis(thesi);
		theseisParast.push_back(thesiParast);
```

```
	}
}

void Parastasi::emfanTheseis()
{
	cout << "Διαθεσιμεσ θεσεις παραστασης \n";
	for (int i=0; i<theseisParast.size(); i++)
	{
		cout << theseisParast.at(i);
	}
}

int Parastasi::getKwdikos()
{
	return kwdikos;
}

ThesiParastasis *Parastasi::getThesiParast(int t)
{
	return theseisParast.at(t);
}

wchar_t Parastasi::getKatastasiThesis(int t)
{
	ThesiParastasis *thesiParast = theseisParast.at(t);
	return thesiParast.getKatastasi();
}

void Parastasi::setKatastasiThesis(int t, wchar_t k)
{
	ThesiParastasis *thesiParast = theseisParast.at(t);
	thesiParast.setKatastasi(k);
}

std::string Parastasi::ToString()
{
	cout << "Παρασταση·:" <<  kwdikos <<  "  " << "Ωρα:" << wra << "  ";
}
#endif
```

Parastasi.hpp

```
#ifndef __Parastasi_HPP
#define __Parastasi_HPP
#include "ThesiParastasis.hpp"
#include "Thesi.hpp"
#include <iostream>
#include <string>
#include <vector>
using namespace std;

class Parastasi
```

```
{

private:
        int kwdikos;
        std::string wra;
        std::vector<ThesiParastasis*> theseisParast;

public:
        Parastasi(int kwdikos, std::string wra, std::vector theseis);

        virtual void emfanTheseis();

        virtual int getKwdikos();

        virtual ThesiParastasis *getThesiParast(int t);

        virtual wchar_t getKatastasiThesis(int t);

        virtual void setKatastasiThesis(int t, wchar_t k);

        virtual std::string ToString();
};
#include "Parastasi.cpp"

#endif
```

Plirwmi.cpp

```
#ifndef __Plirwmi_CPP
#define __Plirwmi_CPP

#include "Plirwmi.hpp"

Plirwmi::Plirwmi(int poso)
{
        timi=poso;
        katastasi = ' ';
        }

int Plirwmi::getTimi()
{
          return timi;
}

void Plirwmi::setPistwtiki(int i)
{
        katastasi = 'P';
        }

void Plirwmi::setMetrita(int i)
{
         katastasi = 'M';
```

```
}

wchar_t Plirwmi::getKatastasi()
{
	return katastasi;
}

void Plirwmi::setKatastasi(wchar_t k)
{
	katastasi = k;
}

std::string Plirwmi::ToString()
{
	cout << "Συνολικο ποσο : " << timi;
}
#endif
```

Plirwmi.hpp

```
#ifndef __Plirwmi_HPP
#define __Plirwmi_HPP
#include <iostream>
#include <string>

 /*
 * @author Nektarios
 */
class Plirwmi
{
 private:
	int timi;
	wchar_t katastasi;
 public:
	Plirwmi(int poso);
	virtual int getTimi();
	virtual void setPistwtiki(int i);

	virtual void setMetrita(int i);

	virtual wchar_t getKatastasi();

	virtual void setKatastasi(wchar_t k);
	virtual std::string ToString();

	};
#include "Plirwmi.cpp"

#endif
```

TameioCTRL.cpp

```
#ifndef __SHAPE_CPP
#define __SHAPE_CPP

#include "TameioCtrl.hpp"

TameioCtrl::TameioCtrl()
{

}

void TameioCtrl::setErgo(Ergo *ergo)
{
	this.ergo=ergo;
}

void TameioCtrl::neaKratisi(Ergo *ergo)
{
	 trexKratisi= new Kratisi(arithmosKratisis++);
}

void TameioCtrl::epilogiParast(int kwdParast)
{
	trexParast = ergo->getParastasi(kwdParast);
   cout << "Επιλεγεται η παρασταση \n" << kwdParast;
	trexParast.emfanTheseis();
	trexKratisi.setParastasi(trexParast);
 }

void TameioCtrl::epilogiThesewn()
{
	int arThes;
	cout << "Δωσε αριθμο θεσης(0-9)=> \n" ;
	cin >> arThes;
	while (arThes!=0)
	{
		if (arThes>=-1 && arThes<10)
		{
			if (trexParast.getKatastasiThesis(arThes) == ' ')
			{
				trexParast.setKatastasiThesis(arThes, 'K');
				trexKratisi.addThesiParast(trexParast.getThesiParast(arThes));
			}
			else
				cout << "Η θεση· " << arThes << " ειναι κατειλλημενη·";
		}
		cout << "Δωσε αριθμο θεσης,(0-9)=> ";
		cin >> arThes;
	}
}
```

```
void TameioCtrl::telosKratisis()
{
	kratiseis.push_back(trexKratisi);
	cout << "---------- Τελος κρατησης ----------";
}

void TameioCtrl::emfanKratiseis()
{
	cout << "---------- Εμφανιση Κρατησεων ----------";
	std::vector<Kratisi*>::const_iterator itr = kratiseis.begin();
	while (itr != kratiseis.end())
	{
		Kratisi *kratisi = static_cast<Kratisi*>(*itr);
		kratisi.emfanKratisis();
		itr++;
	}
}

void TameioCtrl::emfaniseiPlirwmis()
{

	trexPlirwmi.toString();

}
#endif
```

TameioCTRL.hpp

```
#ifndef __TameioCtrl_HPP
#define __TameioCtrl_HPP
#include "Ergo.hpp"
#include "Kratisi.hpp"
#include "Parastasi.hpp"
#include "Plirwmi.hpp"
#include <iostream>
#include <vector>

class TameioCtrl
{

private:

	static int arithmosKratisis = 1;
	Ergo *ergo;
	Kratisi *trexKratisi;
	std::vector<Kratisi*> kratiseis;
	Parastasi *trexParast;
	Plirwmi *trexPlirwmi;

public:
	TameioCtrl();
```

```
        virtual void setErgo(Ergo *ergo);

        virtual void neaKratisi(Ergo *ergo);

        virtual void epilogiParast(int kwdParast);

        virtual void epilogiThesewn();

        virtual void telosKratisis();

        virtual void emfanKratiseis();
        virtual void emfaniseiPlirwmis();


private:
        bool initialized;
        void InitializeInstanceFields()
        {
                if ( ! initialized)
                {
                        kratiseis = std::vector(12);

                        initialized = true;
                }
        }
};
#include "TameioCtrl.cpp"

#endif
```

Tameio.cpp

```
#ifndef __Theatro_CPP
#define __Theatro_CPP
#include "Theatro.hpp"

Theatro::Theatro(std::string epwn)
{
        this.epwnymia = epwn;
        this.aith = new Aithousa(10);
}

void Theatro::setErgo(Ergo *ergo)
{
        this.ergo = ergo;
        cout << "Θεατρο: \n" << epwnymia;
        cout << " Εργο: ]n " << ergo->getTitlos();
}

std::vector Theatro::getTheseis()
{
```

```
	cout << aith.getTheseis();
}

void Theatro::emfanAithousa()
{
	aith.emfanTheseis();
}
#endif
```

Tameio.hpp

```
#ifndef __Theatro_HPP
#define __Theatro_HPP
#include "Aithousa.hpp"
#include "Ergo.hpp"
#include <iostream>
#include <string>
#include <vector>

class Theatro
{

private:
	std::string epwnymia;
	Aithousa *aith;
	Ergo *ergo;


public:
	Theatro(std::string epwn);

	virtual void setErgo(Ergo *ergo);

	virtual std::vector getTheseis();

	virtual void emfanAithousa();


};
#include "Theatro.cpp"

#endif
```

Thesi.cpp

```
#ifndef __Thesi_CPP
#define __Thesi_CPP

#include "Thesi.hpp"

Thesi::Thesi(int ar)
{
```

```
	this->arithmos = ar;
}

int Thesi::getArithmos()
{
	return arithmos;
}

std::string Thesi::ToString()
{
	cout << "Αριθμος θεσης " << arithmos;
}
#endif
```

Thesi.hpp

```
#ifndef __Thesi_HPP
#define __Thesi_HPP
#include <iostream>
#include <string>

class Thesi
{

private:
	int arithmos;

public:
	Thesi(int ar);

	virtual int getArithmos();

	virtual std::string ToString();
};
#include "Thesi.cpp"

#endif
```

ThesiParastasis.cpp

```
#ifndef __ThesiParastasis_CPP
#define __ThesiParastasis_CPP

#include "ThesiParastasis.hpp"

ThesiParastasis::ThesiParastasis(Thesi *thesi)
{
	this.thesi = thesi;
	this.katastasi = ' ';
}

void ThesiParastasis::setDiathesimi(int i)
```

```
{
	katastasi = ' ';
}

void ThesiParastasis::setKatillimeni(int i)
{
	katastasi = 'K';
}

wchar_t ThesiParastasis::getKatastasi()
{
	return katastasi;
}

void ThesiParastasis::setKatastasi(wchar_t k)
{
	katastasi = k;
}

int ThesiParastasis::getArithmos()
{
	return thesi.getArithmos();
}

std::string ThesiParastasis::ToString()
{
	cout << "    Αριθμος θεσης,: " << getArithmos() << "  [" << katastasi << "]";
}
#endif
```

ThesiParastasis.hpp

```
#ifndef __ThesiParastasis_HPP
#define __ThesiParastasis_HPP
#include "Thesi.hpp"
#include <iostream>
#include <string>

class ThesiParastasis
{

private:
	Thesi *thesi;
	wchar_t katastasi;

public:
	ThesiParastasis(Thesi *thesi);

	virtual void setDiathesimi(int i);

	virtual void setKatillimeni(int i);
```

```
        virtual wchar_t getKatastasi();

        virtual void setKatastasi(wchar_t k);

        virtual int getArithmos();

        virtual std::string ToString();
};
#include "ThesiParastasis.cpp"

#endif
```

Κώδικας Διαχείρισης Τραπεζικών Λογαριασμών

Bank.java

```
import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;
import java.util.ArrayList;
import java.util.StringTokenizer;



public class Bank{

        private ArrayList customers;
    int number;
    int pin;
    String name;
    String lname;
    String address;
    String city;
    int zip;

  // Domitis opou periexei mioa lista me pelates
   public Bank()
   {
     customers = new ArrayList();
   }

   /*
    Methodos opou diavazei to id pelati kai ton kwdiko
    */

   public void readCustomers(String filename)
     throws IOException
   {
     BufferedReader in = new BufferedReader(new FileReader(filename));
```

```
    boolean done = false;
    while (!done)
    {
      String inputLine = in.readLine();
      if (inputLine == null) done = true;
      else
      {
        StringTokenizer tokenizer = new StringTokenizer(inputLine,"-");
        while (tokenizer.hasMoreTokens()) {
           number = Integer.parseInt(tokenizer.nextToken());
           pin = Integer.parseInt(tokenizer.nextToken());
           name= tokenizer.nextToken();
           lname= tokenizer.nextToken();
           address= tokenizer.nextToken();
           city= tokenizer.nextToken();
           zip = Integer.parseInt(tokenizer.nextToken());
         }

        Customer c = new Customer(number,pin,name,lname,address,city,zip);
        addCustomer(c);
      }
    }
    in.close();
  }

  /**
    Prosthiki pelati
  */
  public void addCustomer(Customer c)
  {
    customers.add(c);
  }

  /**
    Euresi pelati me ton kwdiko id
    kai to kwdiko logariasmou
  */
  public Customer findCustomer(int aNumber, int aPin){
          // Sarwsei tis listas pelatwn
    for (int i = 0; i < customers.size(); i++){
          /*kathe fora pernei ton i pelati diladi kathos trexei
           o vrogxos for pernei ton pelati stin thesi 0 meta ton pelati 1
           mexrito telos tis listas*/
      Customer c = (Customer)customers.get(i);
      if (c.match(aNumber, aPin))// an teriazoun to id kai to Pin me ton sugkekrimeno pelati
        return c;// epistrefei ton pelati
    }
    return null;//alliws den epistrefei tipota
  }
}
```

Account.java

```
public class Account{
        //Metavlites klasis

        private double balance;

        /* Default domitis
         Arxikopoii to balance=0
         */

  public Account(){
                balance = 0;
  }

  /*
        Domitis  opou dinei mia arxiki  timi sto upoloipo
  */
  public Account(double initialBalance){
                balance = initialBalance;
  }

  /*
   Methodos katathesis opou prosthetei sto proigoumeno
   poso katatheseis to poso amount
  */

  public void deposit(double amount)
  {  balance = balance + amount;
  }

  /**
    Methodos analipseis opou aferei apo to uparxon upoloipo
    to poso amount
  */
  public void withdraw(double amount)
  {  balance = balance - amount;
  }

  /**
    Epistrefei to upoloipo tou logariasmou
  */
  public double getBalance(){
        return balance;
  }
}
```

Customers.java

```
/**
 * @(#)Customer.java
```

```
 *
 *
 * @author Nektarios
 * @version 1.00 2010/12/15
 */


public class Customer{

	// Metavlites Customer

		private String firstName,lastName,address,city;
		private int id,accountPin,zip;
		private Account checkingAccount;
	  private Account savingsAccount;

	//Domitis opou arxikopoiei tis metavlites tou customer

		public Customer(int id,int accountPin,String firstName,String lastName, String address, String city,int zip){
					this.id=id;
					this.accountPin=accountPin;
					this.firstName=firstName;
					this.lastName=lastName;
					this.address=address;
					this.city =city;
					this.zip=zip;
					checkingAccount=new Account();
			savingsAccount=new Account();
		}
	//Methodos elegxou teriasmatos tou kwdikou pelati kai tou Pin logariasmou

	public boolean match(int aNumber, int aPin){
    return id == aNumber && accountPin == aPin;
  }

 //Methodos pou pernei to upolipo tou logariasmou

  public Account getCheckingAccount(){
    return checkingAccount;
  }

  public Account getSavingsAccount(){
    return savingsAccount;
  }

		// Methodos toString opou emfanizei ta stoixeia tou customer
   @Override
		public String toString(){
					String str;
					str="Customer Number: " + id + "\n";
```

```
                    str +="Customer Name: " + firstName +"\n";
                    str +="Customer Surname: " + lastName +"\n";
                                str +="Customer Address: " +address + "\n";
                                str +="Customer City: " + city + "  T.K:  " + zip + "\n";

                                return(str);
                }

}
```

Pliktrologio.java

```
import java.awt.BorderLayout;
import java.awt.GridLayout;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import javax.swing.JButton;
import javax.swing.JPanel;
import javax.swing.JTextField;

/**
  Dimiourgia pliktrologiou opou o xristis dinei ta stoixeia tou gia na
   exei prosvasei ston logariasmo tou

*/
public class Pliktrologio extends JPanel{

        //Metavlites tis klasis
        private JPanel buttonPanel;
        private JButton clearButton;
        private JTextField display;

  /*
    Domitis pliktrologiou :Dimiourgia tou Panel
  */
  public Pliktrologio()
  {
    setLayout(new BorderLayout());

    // Prosthiiki tou pediou display sto panel

    display = new JTextField();
    add(display, "North");

    // Dimiourgia Panel opou tha periexei to pliktrologio
    buttonPanel = new JPanel();
    buttonPanel.setLayout(new GridLayout(4, 3));


    // Prosthiki twn koumpiwn

    addButton("7");
```

```
    addButton("8");
    addButton("9");
    addButton("4");
    addButton("5");
    addButton("6");
    addButton("1");
    addButton("2");
    addButton("3");
    addButton("0");
    addButton(".");

    // Prosthiki tou koumpiou clear

    clearButton = new JButton("CE");
    buttonPanel.add(clearButton);

    clearButton.addActionListener(
      new ActionListener()  {
        public void actionPerformed( ActionEvent evt )
        {
          clear();
        }
                    }
         );
    add(buttonPanel, "Center");
  }

  /*
    prosthiki tou koumpiou "." sto buttonPanel
  */
  private void addButton(final String label)
  {
    class DigitButtonListener implements ActionListener
    {
      public void actionPerformed(ActionEvent event)
      {

        // elegxos an exei patithei 2 fores to koumpi "."
        if (label.equals(".")
          && display.getText().indexOf(".") != -1)
          return;// An exei patithei tote den epistrefete tipota

        // Alliws ginete enwsei tou String pou uparxei sto display me to String label="."
        display.setText(display.getText() + label);
      }
    }

    JButton button = new JButton(label);
    buttonPanel.add(button);// Prosthki tou koumpiou sto buttonPanel
    ActionListener listener = new DigitButtonListener();
    button.addActionListener(listener);
```

```
  }

  /*I methodos auti epistrefei ton arithmo pou exei pliktrologisi  o xristis
  */
  public double getValue()
  {
    return Double.parseDouble(display.getText());
  }

  //Methodos katharismou tou display

  public void clear()
  {
    display.setText("");
  }
}
```

Transaction.java

```
import java.awt.Color;
import java.awt.Container;
import java.awt.FlowLayout;
import java.awt.GridLayout;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.io.IOException;
import java.util.logging.Level;
import java.util.logging.Logger;
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JOptionPane;
import javax.swing.JPanel;
import javax.swing.JTextArea;
import javax.swing.text.Utilities;

/**@author Nektarios
   A frame displaying the components of an ATM
*/
class Transaction extends JFrame
{


  private int state;
  private int customerNumber;
  private Customer currentCustomer;
  private Account currentAccount;
  private Bank theBank;

  private JButton aButton;
  private JButton bButton;
  private JButton cButton;
  private JButton dButton;
```

```
private Pliktrologio pad;
private JTextArea display;
int account=1;

private static final int START_STATE = 1;
private static final int PIN_STATE = 2;
private static final int ACCOUNT_STATE = 3;
private static final int TRANSACT_STATE = 4;

private static final int CHECKING_ACCOUNT = 1;
private static final int SAVINGS_ACCOUNT = 2;

/**
   Constructs the user interface of the ATM application.
*/
public  Transaction()
{
   // initialize bank and customers

   theBank = new Bank();
   try
   {
      theBank.readCustomers("customers.txt");
   }
   catch(IOException e)
   {
      JOptionPane.showMessageDialog(null,
         "Error opening accounts file.");
   }

   // construct components

   pad = new Pliktrologio();

   display = new JTextArea(6, 20);

   aButton = new JButton("  A  ");
   aButton.addActionListener(new AButtonListener());

   bButton = new JButton("  B  ");
   bButton.addActionListener(new BButtonListener());

   cButton = new JButton("  C  ");
   cButton.addActionListener(new CButtonListener());

    dButton = new JButton("  Check  ");
    dButton.addActionListener(new DButtonListener());

   // add components to content pane
```

```
    JPanel buttonPanel = new JPanel();
    buttonPanel.setLayout(new GridLayout(4, 1));
    buttonPanel.add(aButton);
    buttonPanel.add(bButton);
    buttonPanel.add(cButton);
    buttonPanel.add(dButton);



    Container contentPane = getContentPane();
    contentPane.setLayout(new FlowLayout());
    contentPane.add(pad);
    contentPane.add(display);
    contentPane.add(buttonPanel);
    contentPane.setBackground(Color.gray);


    setState(START_STATE);
}

/**
    Sets the current customer number to the keypad value
    and sets state to PIN.
*/
public void setCustomerNumber()
{
    customerNumber = (int)pad.getValue();
    setState(PIN_STATE);
}

/**
    Gets PIN from keypad, finds customer in bank.
    If found sets state to ACCOUNT, else to START.
*/
public void selectCustomer()
{
    int pin = (int)pad.getValue();
    currentCustomer= theBank.findCustomer(customerNumber, pin);

    if (currentCustomer == null)
      setState(START_STATE);
    else{
       setState(ACCOUNT_STATE);


     }
}

/**
    Sets current account to checking or savings. Sets
    state to TRANSACT
```

```
    @param account one of CHECKING_ACCOUNT or SAVINGS_ACCOUNT
  */
  public void selectAccount(int account)
  {
    if (account == CHECKING_ACCOUNT)
      currentAccount = currentCustomer.getCheckingAccount();
    else
      currentAccount = currentCustomer.getSavingsAccount();
    setState(TRANSACT_STATE);
  }

  /**
    Withdraws amount typed in keypad from current account.
    Sets state to ACCOUNT.
  */
  public void withdraw()
  {
    currentAccount.withdraw(pad.getValue());
    setState(ACCOUNT_STATE);
  }

  /**
    Deposits amount typed in keypad to current account.
    Sets state to ACCOUNT.
  */
  public void deposit()
  {
    currentAccount.deposit(pad.getValue());
    setState(ACCOUNT_STATE);
  }

  /**
    Sets state and updates display message.
    @param state the next state
  */
   @Override
  public  void setState(int newState)
  {
    state = newState;
    pad.clear();
    if (state == START_STATE)
      display.setText("Enter customer bank account number\nA = OK");
    else if (state == PIN_STATE)
      display.setText("Enter PIN\n A = OK");
    else if (state == ACCOUNT_STATE)
      display.setText("Select Account\n"
        + "A = Checking\nB = Savings\nC = Exit");
    else if (state == TRANSACT_STATE)
       display.setText("Balance = " + currentAccount.getBalance()+ "\nEnter amount and select
transaction\n"
         + "A = Withdraw\nB = Deposit\nC = Cancel");
```

```
}

private class AButtonListener implements ActionListener
{
  public void actionPerformed(ActionEvent event)
  {
    if (state == START_STATE){
      setCustomerNumber();

     }
    else if (state == PIN_STATE){
      selectCustomer();


    }
    else if (state == ACCOUNT_STATE)
      selectAccount(CHECKING_ACCOUNT);
    else if (state == TRANSACT_STATE)
      withdraw();
  }
}

private class BButtonListener implements ActionListener
{
  public void actionPerformed(ActionEvent event)
  {
    if (state == ACCOUNT_STATE)
      selectAccount(SAVINGS_ACCOUNT);
    else if (state == TRANSACT_STATE)
      deposit();
  }
}

private class CButtonListener implements ActionListener
{

  public void actionPerformed(ActionEvent event)
  {
    if (state == ACCOUNT_STATE)
      setState(START_STATE);
    else if (state == TRANSACT_STATE)
      setState(ACCOUNT_STATE);
    else
       display.setText("Enter customer bank account number\nA = OK");
   }
}

private class DButtonListener implements ActionListener
{
```

```
    public void actionPerformed(ActionEvent event)
    {

       if(account==1){
        display.setText(currentCustomer.toString()+ "\n If elements are correct press Check or
press C");
         ++ account;
       }
       else{
          --account;
          if (state == ACCOUNT_STATE)
            setState(ACCOUNT_STATE);
          else if (state == TRANSACT_STATE)
            setState(ACCOUNT_STATE);
       }
     }
  }

  public static void main(String[] args)
  {
    JFrame frame = new  Transaction();
    frame.setTitle("Bank of DNT");
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    frame.pack();
    frame.setSize(550, 180);
    frame.show();
  }

}
```