Ατομική Διπλωματική Εργασία

Τίτλος Εργασίας

ASPECT ORIENTED SOFTWARE ENGINEERING

Σύστημα MHC-PMS

Μαρία Ανδρέου

ΠΑΝΕΠΙΣΤΗΜΙΟ ΚΥΠΡΟΥ

ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ

Μάιος 2013

ΠΑΝΕΠΙΣΤΗΜΙΟ ΚΥΠΡΟΥ

ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ

Aspect Oriented Software Engineering

Σύστημα MHC-PMS

Μαρία Ανδρέου

Επιβλέπων Καθηγητής

Κ. Γιώργος Παπαδόπουλος

Η Ατομική Διπλωματική Εργασία υποβλήθηκε προς μερική εκπλήρωση των απαιτήσεων απόκτησης του πτυχίου Πληροφορικής του Τμήματος Πληροφορικής του Πανεπιστημίου Κύπρου

Μάιος 2013

Ευχαριστίες

Με την εκπλήρωση της διπλωματικής μου εργασίας σηματοδοτείται το τέλος των προπτυχιακών μου σπουδών στο Πανεπιστήμιο Κύπρου στο τμήμα Πληροφορικής κλείνει ακόμα ένας σημαντικός κύκλος της ζωής μου.

Καταρχάς θα ήθελα να ευχαριστήσω θερμά πρώτα από όλους τον επιβλέπων καθηγητή μου κύριο Γιώργο Παπαδόπουλο, που μου έδωσε την ευκαιρία να ασχοληθώ με ένα από τα θέματα του καθώς επίσης και για την σωστή καθοδήγηση και στήριξη που μου πρόσφερε έτσι ώστε να καταφέρω να ανταπεξέλθω σε τυχόν δυσκολίες που αντιμετώπισα ούτος ώστε να πετύχω την ολοκλήρωση της διπλωματικής μου.

Τέλος θα ήθελα να ευχαριστήσω μέσα από τα βάθη της ψυχής μου τον πατέρα μου, την μητέρα και τον αδελφό μου για την συμπαράσταση που μου προσέφεραν όλα αυτά τα χρόνια των σπουδών μου.

Αφιερώνω λοιπόν, την εργασία μου αυτή στα πιο πάνω πρόσωπα καθώς επίσης και στο στενό μου οικογενειακό και φιλικό περιβάλλον.

Περίληψη

Η παρούσα διπλωματική εργασία αφορά κυρίως την μελέτη κατανόηση του Aspect Oriented Programming (AOP) .Αρχικά έγινε μια μελέτη σε αυτό το θέμα καθώς επίσης και μια σύγκριση όσο αφορά το Object Oriented Programming (OOP) μιας και τα μειονεκτήματα που είχε ο OOP οδήγησε στη δημιουργία του AOP. Επιπλέων έχει μελετηθεί η γλώσσα προγραμματισμού Aspect Java η οποία κάνει χρήση του AOP. Σκοπός της πιο πάνω μελέτης ήταν η κατανόηση τους σε τέτοιο βαθμό έτσι ώστε να καταφέρουμε να αναπτύξουμε τις πτυχές (aspects) του συστήματος MHCPMS.

Στην $1^{η}$ ενότητα γίνεται μια εισαγωγή όσο αφορά το κίνητρο και το σκοπό της διπλωματικής μου εργασίας. Ακολούθως στην $2^{η}$ ενότητα γίνεται μια αναφορά στις διάφορες τεχνικές του προγραμματισμού μια σύγκριση μεταξύ τους καθώς και κάποια πλεονεκτήματα και μειονεκτήματα που υπάρχουν στην κάθε μια από αυτές. Στην $3^{η}$ ενότητα γίνεται μια περιγραφή στο σύστημα (MHC-PMS) του οποίου έχουμε αναπτύξει τις διάφορες πτυχές και η ανάλυση του σε διαγράμματα UML.Στην $4^{η}$ ενότητα γίνεται μια αναφορά στην γλώσσα προγραμματισμού την οποία έχω χρησιμοποιήσει, την μεθοδολογία την οποία ακολούθησα για την ολοκλήρωση του κύκλου φάσης ανάπτυξης του συστήματος καθώς και κάποια προβλήματα που αντιμετώπισα. Τέλος στην $5^{η}$ ενότητα παρουσιάζονται τα συμπεράσματα και η πιθανή μελλοντική επέκταση.

Περιεχόμενα

Κεφάλαιο 1

Εισαγωγή

1.1 Κίνητρο Διπλωματικής Εργασίας

Είναι πλέον κοινά αποδεκτό ότι η τεχνολογία της πληροφορικής έχει αναπτυχθεί ραγδαία προς όλους τους τομείς και κυρίως την πληροφορική. Έχει καταφέρει να σημειώσει πρωτοποριακές αλλαγές όσο αφορά την ανάπτυξη νέων τεχνικών δημιουργίας προγραμμάτων και να λύσει σημαντικά προβλήματα που αντιμετωπίζει η κοινωνία μας σήμερα.

Ο συνδυασμός της τεχνολογίας με τον προγραμματισμό έχει φέρει σημαντικές αλλαγές στην καθημερινή ζωή των ανθρώπων καθώς πλέον ακόμα και η πιο μικρή εφαρμογή κάνει χρήση προγραμματισμού, από τα πιο απλά πράγματα έως τα πιο σύνθετα. Είναι λοιπόν χρήσιμο να χρησιμοποιούμε τεχνικές προγραμματισμού οι οποίες θα είναι εποικοδομητικές και ευέλικτες έτσι ώστε να μας επιφέρουν μόνο κέρδος.

Οι προηγούμενες τεχνικές προγραμματισμού που είδη υπάρχουν διέπονται από κάποια μειονεκτήματα όσο αφορά την λειτουργία τους που φέρνουν τους προγραμματιστές σε δύσκολη θέση. Έτσι η ανάγκη για κάτι το οποίο θα έχει λιγότερες συνέπειες κόστους και μνήμης στη χρήση του έχει δημιουργήσει μια σχετικά καινούργια τεχνολογία προγραμματισμού που έρχεται να βοηθήσει τους προγραμματιστές να μειώσουν όσο γίνεται το κόστος.

Ο Aspect Oriented Programming είναι μια σχετικά καινούργια τεχνολογία που δημιουργήθηκε για να καλύψει τις προαναφερθείσες ανάγκες των προγραμματιστών και να κάνει τα προγράμματα περισσότερο ευέλικτα και κατανοητά.

Κίνητρο λοιπόν της διπλωματικής αυτής εργασίας αποτελεί η μελέτη για την δυνατότητα ανάπτυξης πληροφοριακών προγραμμάτων με πρωταρχικό στόχο την απλή υλοποίηση καθώς και το να μπορούμε εύκολα και γρήγορα να εισάγουμε νέες αλλαγές στον ήδη υπάρχων κώδικα χωρίς ιδιαίτερο κόπο και χρόνο.

1.2 Σκοπός Διπλωματικής Εργασίας

Σκοπός της παρούσας διπλωματικής εργασίας αποτελεί η κατανόηση αυτής της καινούργιας τεχνικής που έχει εμφανιστεί στη ζωή μας και η ανάπτυξη του συστήματος MHC-PMS με την χρήση της τεχνικής αυτής.

Στους στόχους της εργασίας αυτής λοιπόν είναι η μελέτη του Aspect Oriented Programming η μελέτη του σε βάθος ώστε να αναλύσουμε κάποιες βασικές αρχές που τον διέπουν, καθώς και η σύγκριση του με τις είδη προϋπάρχουσες τεχνικές. Επιπλέον η κατανόηση της γλώσσας προγραμματισμού που κάνει χρήση της τεχνικής αυτής, και συγκεκριμένα της aspectJ και πως μπορούμε να αναπτύξουμε τις πτυχές του συστήματος MHC-PMS με την χρήση της γλώσσας αυτής.

Κεφάλαιο 2 Εισαγωγικές Έννοιες

2.1 Object Oriented Programming (OOP)

Στα τέλη της δεκαετίας του 70 εμφανίστηκε μια νέα τάση αντιμετώπισης προγραμματιστικών αντιλήψεων και δομών ο αντικειμενοστρεφής προγραμματισμός. Η τάση αυτή έχει γίνει η επικρατούσα κατάσταση και έχει αλλάξει ριζικά τα μέχρι πριν από λίγα χρόνια γνωστά και σταθερά σημεία αναφοράς των προγραμματιστών. Γεννήθηκε και άρχισε να αναπτύσσεται όταν πλέον ήταν φανερό ότι οι παραδοσιακές προσεγγίσεις στον προγραμματισμό δεν μπορούσαν να ανταποκριθούν στις νέες απαιτήσεις ανάπτυξης προγραμμάτων.

Η ιδέα αυτή του αντικειμενοστρεφούς προγραμματισμού έχει τις ρίζες τις σε μια απλή ιδέα. Ένα πρόγραμμα περιγράφει τις ‘ενέργειες ’ δηλαδή την επεξεργασία που εφαρμόζεται πάνω σε δεδομένα. Η βασική διαφορά του ανικειμενοστρεφούς προγραμματισμού ανάμεσα στις παραδοσιακές προγραμματιστικές τεχνικές που υπάρχουν είναι ότι η φιλοσοφία του στηρίζεται στα δεδομένα και όχι στις ‘ενέργειες’.

Ο αντικειμενοστρεφής προγραμματισμός πλέων είναι η πιο διαδεδομένη μέθοδος προγραμματισμού που χρησιμοποιείται για την ανάπτυξη συστημάτων λογισμικού. εκλαμβάνει ως πρωτεύοντα δομικά στοιχεία ενός προγράμματος τα δεδομένα, από τα οποία δημιουργούνται τα αντικείμενα αφού γίνει η κατάλληλη μορφοποίηση. Τα προγράμματα που δημιουργούνται είναι περισσότερο ευέλικτα και επαναχρησιμοποιήσιμα γεγονός που καθιστά τον αντικειμενοστρεφή προγραμματισμό μία από τις καλύτερες τεχνικές σχεδίασης. Χρησιμοποιεί την ιεραρχική σχεδίαση, τον τμηματικό προγραμματισμό και ακολουθεί τις αρχές του δομημένου προγραμματισμού.

Η κεντρική ιδέα στον αντικειμενοστρεφή προγραμματισμό είναι η κλάση, είναι μια στατική περιγραφή ενός συνόλου παρόμοιων αντικειμένων. Όλες οι δομημένες γλώσσες προγραμματισμού έχουν ενσωματωμένους τύπους δεδομένων, και μπορούμε να δηλώσουμε μεταβλητές αυτού του τύπου. Έχουν απλούς τύπους που καθορίζονται από την γλώσσα αυτή. Για παράδειγμα στη γλώσσα προγραμματισμού C έχουμε το τύπο int (integer) που είναι για την δήλωση ακέραιων αριθμών και τους τύπους float, double για δεκαδικούς. Επιπλέων μπορούμε να ορίσουμε και δικούς μας σύνθετους τύπους δεδομένων μέσω της δήλωσης STRUCT{}. Με παρόμοιο τρόπο μέσω των κλάσεων μας δίνεται η δυνατότητα σχεδίασης σύνθετων τύπων δεδομένων. Πιο γενικά μια κλάση είναι μια σύνθετη δομή δεδομένων σχεδιασμένη από τον προγραμματιστή. Περιέχει σαν μέλη της μεταβλητές-πεδία (member variables) και συναρτήσεις- μεθόδους (member function).

Παρουσιάζουμε κάποια παραδείγματα που βασίζονται στην γλώσσα προγραμματισμού JAVA, για να γίνουν πιο κατανοητά όσα έχουμε προαναφέρει.

Ο ορισμός της κλάσης γίνεται ως εξής:

```
Class myClass
        {
        Δήλωση μεθόδων και μεταβλητών
        }
```

Όπως είδη αναφέραμε μια κλάση στον αντικειμενοστρεφή προγραμματισμό είναι το αντίστοιχο των τύπων δεδομένων για τις μεταβλητές στον δομημένο προγραμματισμό. Μια κλάση αποτελεί ένα πρότυπο για την δημιουργία αντικειμένων.

Το αντικείμενο κάποιας κλάσης είναι το αντίστοιχο της μεταβλητής κάποιου τύπου δεδομένων. Με άλλα λόγια όπως στον δομημένο προγραμματισμό μια μεταβλητή είναι μια οντότητα στην μνήμη του Η/Υ που περιέχει κάποια τιμή(δεδομένο), έτσι και ένα αντικείμενο είναι μια οντότητα στη μνήμη η οποία περιέχει πεδία που λαμβάνουν συγκεκριμένες τιμές (δεδομένα) καθώς και μεθόδους. Μέσω των μεθόδων του αντικειμένου μπορούμε να επικοινωνήσουμε με το αντικείμενο και να αλλάξουμε τα δεδομένα του.

Εκτός από τις κλάσεις και τα αντικείμενα η JAVA διαθέτει και τους λεγόμενους πρωτογενείς τύπους δεδομένων: int, float, double, char. Μπορούμε να ορίσουμε μεταβλητές αυτών των τύπων με τον ίδιο τρόπο που ορίζω μεταβλητές στον δομημένο προγραμματισμό.

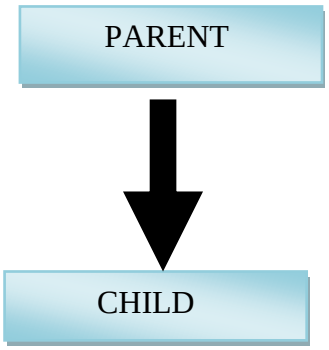
Ακολουθεί ένα μικρό παράδειγμα για να γίνει πιο κατανοητό ότι έχει προαναφερθεί πιο πάνω:

```
Class Employee{
        String name;
        String address;
        int salary;
    public void setSalary(int employeeSalary){
                salary=employeeSalary;
                                        }
    Public int getSalary() {
                Return salary;
                 }
          }
```

Παρατηρούμε την δήλωση μιας κλάσης με το όνομα Employee. Η κλάση αυτή περιλαμβάνει τα πεδία name, address, salary και τις μεθόδους setSalary() και getSalary(). Μπορούμε να δούμε ότι πριν από το όνομα της κάθε μεθόδου δηλώνουμε τον τύπο της τιμής που επιστρέφει. Μετά από το όνομα της μεθόδου δηλώνω τις παραμέτρους μέσα σε παρενθέσεις που δέχεται όταν καλείται να εκτελεστεί. Βάζουμε τον κώδικα της μεθόδου μεταξύ των αγκίστρων {}, τα οποία δηλώνουν την αρχή και το τέλος των εντολών της μεθόδου. Η δήλωση public που υπάρχει μπροστά από τις μεθόδους δίνει στο προγραμματιστή το δικαίωμα να καλέσει την μέθοδο αυτή μέσω ενός αντικειμένου που ορίζεται έξω από το σώμα της κλάσης.

Ένα σημαντικό χαρακτηριστικό των κλάσεων, που τις διαχωρίζει από τις σύνθετες δομές δεδομένων του παραδοσιακού προγραμματισμού είναι το χαρακτηριστικό της κληρονομικότητας. Όταν μια κλάση Β κληρονομεί από μια άλλη κλάση την Α τότε η κλάση Β περιέχει πεδία και μεθόδους που έχουν ορισθεί στην κλάση Α. Η κλάση Α ονομάζεται υπερκλάση της κλάσης Β, ενώ η κλάση Β ονομάζεται υποκλάση της κλάσης Α. Μια κλάση γίνεται υποκλάση μιας άλλης μέσω της δήλωσης extends.

Ακολουθεί ένα μικρό παράδειγμα για να γίνει πιο κατανοητή η κληρονομικότητα.




```
Class parent{
        int a;
        public void x(){ }

        }
Class child extends parent{
                int b;
                public void y();
                }
```

Η κλάση parent είναι υπερκλάση της κλάσης child και η κλάση child είναι υποκλάση της κλάσης parent, πράγμα που το έχουμε επιτύχει μέσω της δήλωσης child extends parent. Ένα αντικείμενο της κλάσης child μπορεί να καλέσει κάποια μέθοδο ή μεταβλητή από την κλάση parent καθώς κληρονομεί από την κλάση αυτή.

Αν κάποια κλάση όμως δεν κληρονομεί από κάποια άλλη τότε υπονοείται ότι κληρονομεί την κλάση Object η οποία προέρχεται από την Java.

Στις αντικειμενοστρεφείς γλώσσες προγραμματισμού επιτρέπεται η υπερφόρτωση. Δηλαδή ο ορισμός μιας μεθόδου με το ίδιο όνομα δύο ή περισσότερες φορές αρκεί να έχουν διαφορετικές λίστες παραμέτρων.

```
Class number  {
        int a;
        void add( int x){
                a=a+x;
                }
        void add( intx, int y){
                a=x+y;
                        }
        void add(number n){
                a=a+n.getA();
                        }
```

```
        Int getA(){
                return a;
                }
            }
number n1,n2;
n1= new number();
n2= new number();
n1.add(3);
n1.add(3,5);
n1.add(n2);
```

Βλέπουμε ότι η συνάρτηση void add() χρησιμοποιείται πολλές φορές αλλά κάθε φορά με διαφορετική λίστα παραμέτρων. Αυτή η επαναχρησιμοποίηση του ίδιου ονόματος λέγεται υπερφόρτωση του ονόματος αυτού.

Κάθε κλάση διαθέτει τουλάχιστο μια μέθοδο η οποία εκτελείται κατά την δημιουργία ενός στιγμιότυπου της. Δηλαδή κατά την δημιουργία ενός αντικειμένου της κλάσης αυτής. Αυτές οι μέθοδοι λέγονται κατασκευαστές (constructors) της κλάσης, και ορίζονται όπως ακριβώς και οι απλές μέθοδοι με την μόνη διαφορά ότι έχουν το ίδιο όνομα με την κλάση. Σκοπός τους είναι η δέσμευση μνήμης για την κατασκευή του αντικειμένου καθώς και η διενέργεια κατάλληλων αρχικοποιήσεων. Αν δεν οριστεί κανένας κατασκευαστής από τον προγραμματιστή μέσα σε μια κλάση τότε υπάρχει ο default κατασκευαστής που περιέχει η java.

Πολλές φορές θέλουμε να έχουμε κάποια πεδία ή μεθόδους μας ορατά στα αντικείμενα όλων των κλάσεων ή να θέλουμε κάποια πεδία να είναι ορατά μόνο στα αντικείμενα της τρέχουσας κλάσης και όχι στις υποκλάσεις της. Για όλες αυτές τις περιπτώσεις έχουμε τους προσδιοριστές πρόσβασης. Όπως λέει και το όνομα τους καθορίζουν ποίος θα μπορεί να χρησιμοποιεί και να επεξεργάζεται, τις μεθόδους και τα πεδία που έχουν τέτοιο προσδιοριστή. Οι προσδιοριστές αυτοί είναι:

- Τίποτα: Είναι public αλλά μόνο για τις κλάσεις που ανήκουν στο ίδιο πακέτο. Σε ένα πακέτο μπορούν να βρίσκονται κλάσεις με συναφείς λειτουργικότητες.
- Public: Το συγκεκριμένο πεδίο/μέθοδος μπορεί να χρησιμοποιηθεί από οποιονδήποτε, ακόμα και από ξένο αντικείμενο.

- Protected: Το πεδίο/μέθοδος είναι ορατό μόνο στις μεθόδους αυτής της κλάσης καθώς και τις υποκλάσεις της.
- Private: Το πεδίο/μέθοδος είναι ορατό μόνο στις μεθόδους αυτής της κλάσης αλλά όχι στις υποκλάσεις της.

2.2 Πλεονεκτήματα – Μειονεκτήματα (OOP)

Ο αντικειμενοστρεφής προγραμματισμός έχει αρκετά πλεονεκτήματα. Η ερεύνα σε αυτόν αλλά και η πράξη έχει δείξει ότι οι αντικειμενοστρεφείς τεχνικές εφαρμόζονται επιτυχώς τόσο σε προβλήματα μεγάλης κλίμακας όσο και σε απλά προγράμματα. Παρέχουν μια μεθοδολογία αφαίρεσης που προσομοιάζει με τις τεχνικές που οι άνθρωποι χρησιμοποιούν για να λύνουν καθημερινά προβλήματα. Επιπλέον, οι πολύ ισχυρές και εύχρηστες γλώσσες προγραμματισμού που διατίθενται και οι οποίες υποστηρίζουν το αντικειμενοστρεφές μοντέλο, συνδέονται από ένα ολοένα και αυξανόμενο αριθμό εργαλείων και βιβλιοθηκών προσελκύοντας το ενδιαφέρον έμπειρων αλλά και νέων προγραμματιστών. Η εκμάθηση μιας τέτοιας γλώσσας είναι εύκολη και κατανοητή από τον άνθρωπο σε σχέση με τις γλώσσες μηχανής. Ακόμα είναι ανεξάρτητη από την αρχιτεκτονική του Η/Υ. Βεβαίως και ένας επιπλέων λόγος, είναι πιθανόν η αντίληψη ότι κάποιος που γνωρίζει μια γλώσσα διαδικασιακού προγραμματισμού μπορεί με μικρή προσπάθεια να προγραμματίζει σε κάποια άλλη γλώσσα αντικειμενοστρεφή προγραμματισμού. Ωστόσο, ο αντικειμενοστρεφής προγραμματισμός είναι ένας ριζοσπαστικά διαφορετικός τρόπος σκέψης για τον τρόπο με τον οποίο δομείται και μεταφέρεται η πληροφορία σε ένα σύστημα. Τα μεγαλύτερα του πλεονεκτήματα είναι ότι το πρόβλημα της επικοινωνίας μεταξύ των υπομονάδων δεν αφορά μόνο στην αναγκαιότητα συνεργασίας μεταξύ των διαφόρων τμημάτων κώδικα που θα αναπτύξουν αλλά και την συνεννόηση μεταξύ των ατόμων με πολλαπλούς ρόλους που συμμετέχουν στην ομάδα ανάπτυξης. Ο αντικειμενοστρεφής τρόπος ανάλυσης και σχεδίασης αποτελεί την ιδανική μεθοδολογία για το διαχωρισμό και την οργάνωση ενός μεγάλου έργου το οποίο θα αναπτυχθεί σε τμήματα. Η συντήρηση ενός έργου λογισμικού προφανώς δεν σχετίζεται με την κλασική έννοια της συντήρησης, καθώς το λογισμικό δεν ‘φθείρεται’ από την χρήση και τα συστατικά του δεν χρειάζονται αντικατάσταση λόγω βλάβης που παρήλθε με την πάροδο του χρόνου.

Αφού υπάρχει η κληρονομικότητα μπορούμε να αποφύγουμε την επανάληψη και την επαναχρησιμοποίηση κώδικα. Επιπλέων τα προγράμματα που δημιουργούμε είναι

επεκτάσιμα. Είναι πιο εύκολος ο προγραμματισμός, και είναι πιο εύκολη η ανάγνωση του κώδικα από τρίτους.

Εκτός από τα πλεονεκτήματα όπως κάθε τι έχει και τα μειονεκτήματα του. Παρουσιάζεται μια δυσκολία στη διαχείριση μνήμης, είναι πιο αργή η εκτέλεση του κώδικα και έχει επιπτώσεις στην επίδοση. Καθώς οι απαιτήσεις με την πάροδο του χρόνου αλλάζουν θα πρέπει το λογισμικό να είναι ευέλικτο. Δηλαδή να μπορούν εύκολα να γίνουν οι διάφορες αλλαγές που χρειάζονται με την ελάχιστη δυνατή προσπάθεια και με τις μικρότερες δυνατές συνέπειες για τις υπόλοιπες λειτουργίες του. Παρόλο που οι αντικειμενοστρεφείς γλώσσες προγραμματισμού παρέχουν την υποδομή για την ανάπτυξη ευέλικτων προγραμμάτων, η σχεδίαση ενός συστήματος είναι αυτή που εξασφαλίζει την ποιότητα αναφορικά με την εύκολη συντήρηση. Επιπλέων είναι δύσκολη η δημιουργία νέων τύπων δεδομένων.

2.3 Μετάβαση από τον OOP στον AOP

Το πέρασμα από τον OOP στον AOP έγινε κυρίως για να καλυφτούν κάποια μειονεκτήματα που παρουσίαζε **ο** OOP τα οποία προαναφέραμε πιο πάνω. Για να γίνει πιο κατανοητό αυτό ας δούμε δύο παραδείγματα στο OOP και στο AOP.

Παράδειγμα $1^{ο}$:

Έστω ότι έχουμε μια κλάση με γραφικά η οποία συμπεριλαμβάνει πολλές μεθόδους, set..(). Μετά από κάθε μέθοδο set..() αλλάζουν οι πληροφορίες με αποτέλεσμα να αλλάζουν τα ίδια τα γραφικά που θα πρέπει να εμφανιστούν στην οθόνη.

Για να γίνει αυτό όμως είναι απαραίτητη προσθήκη του επιπλέον κώδικα της display.update() μετά από κάθε μέθοδο set..(). Αυτό στην περίπτωση που ο αριθμός των μεθόδων set είναι μικρός δεν δημιουργείται πρόβλημα. Αν όμως ο αριθμός είναι μεγάλος τότε εμφανίζεται ένα πρόβλημα καθώς θα πρέπει να είμαστε τελείως σίγουροι ότι δεν έχουμε ξεχάσει να βάλουμε το κώδικα display.update() σε όλες τις μεθόδους set,καθώς επίσης και το πρόβλημα ότι η προσθήκη τέτοιου μεγάλου κώδικα σε όλες τις μεθόδους set είναι χρονοβόρα και απαιτεί περισσότερη μνήμη.

Εδώ λοιπόν κάνει την εμφάνιση του ο aspect oriented programming για να λύσει αυτά τα προβλήματα χωρίς να χρειαστεί να προσθέσει άπειρες γραμμές κώδικα, παρά μόνο ένα και μόνο aspect.

```
after():set{
        display.update();
            }
```

Με αυτό τον τρόπο δεν χρειάζεται να υποβληθούμε στην πιο πάνω χρονοβόρα διαδικασία των τόσων γραμμών κώδικα και ούτε να ανησυχούμε ότι σε κάποια μέθοδο μπορεί να έχουμε ξεχάσει να προσθέσουμε τον κώδικα. Στην ουσία εδώ με την χρίση του aspect απλά λέμε στο σύστημα ότι μετά από την εκτέλεση του pointcut set() πρέπει να εκτελέσει τον κατάλληλο κώδικα.

Το pointcut set() έχει ως εξής:

Pointcut set(): execution(*set*(*)) && this (MyGraphicsClass) && within(com.company.*);

Αυτό σημαίνει ότι μια μέθοδος είναι set pointcut αν είναι μέθοδος της κλάσης MyGraphicsClass, αν αυτή η κλάση είναι μέρος του πακέτου com.company, αν το όνομα της είναι set και ανεξάρτητα από το τι ακολουθεί, τι επιστρέφει και τι παραμέτρους παίρνει.

Παράδειγμα $2^{ο}$:

Σε αυτό το παράδειγμα μπορούμε να δούμε ένα κώδικα σε OOP καθώς και με ποιόν τρόπο μπορούμε να επέμβουμε σε αυτόν χρησιμοποιώντας τις τεχνικές του aspect oriented programming.

```
Public class MessageCommunicator
{
        Public static void deliver(string message)
        {
                System.out.println(message);
        }
        Public static void deliver(string person,string message)
        {
                System.out.println(person + “,” + message);
        }
```

```
}
Public class test
{
        Public static void main(string[] args)
        {
                MessageCommunicator.deliver("wanna learn aspectJ?");
                MessageCommunicator.deliver("yes!!, aspectJ hasfun" );
        }
}
```

Αν τρέξουμε το πιο πάνω πρόγραμμα η έξοδος που θα μας δώσει θα είναι η εξής:

```
wanna learn aspectJ?
yes!!, aspectJ hasfun
```

Έστω τώρα ότι θέλουμε να προσθέσουμε στο παράδειγμα μας την λέξη «Hello» στην αρχή κάθε απάντησης. Για να το πετύχουμε αυτό θα μπορούσαμε να τροποποιήσουμε την κλάση MessageCommunicator, αλλά θα ήταν πολύ χρονοβόρο καθώς θα πρέπει να ελέγξουμε πολλαπλά αρχεία ψάχνοντας τον σχετικό κώδικα.
Ένας πιο απλός τρόπος είναι να χρησιμοποιήσουμε ένα aspect. Για να υλοποιήσουμε αυτόν τον τρόπο μπορούμε να δηλώσουμε ένα pointcut το οποίο θα καταχωρεί όλες τις κλήσεις στο deliver καθώς και ένα advice το οποίο θα εκτελείται πριν από κάθε εκτέλεση του deliver.

```
Public aspect MannerAspect
{
        Pointcut deliverMessage() : call( *MessageCommunicator.deliver);
        Before() : deliverMessage()
        {
        System.out.print("Hello…");
        }
}
```

Αν τώρα εκτελέσουμε ξανά το παραπάνω πρόγραμμα η έξοδος θα είναι η ακόλουθη:

Hello … wanna learn aspectJ?
Hello …yes!!, aspectJ hasfun

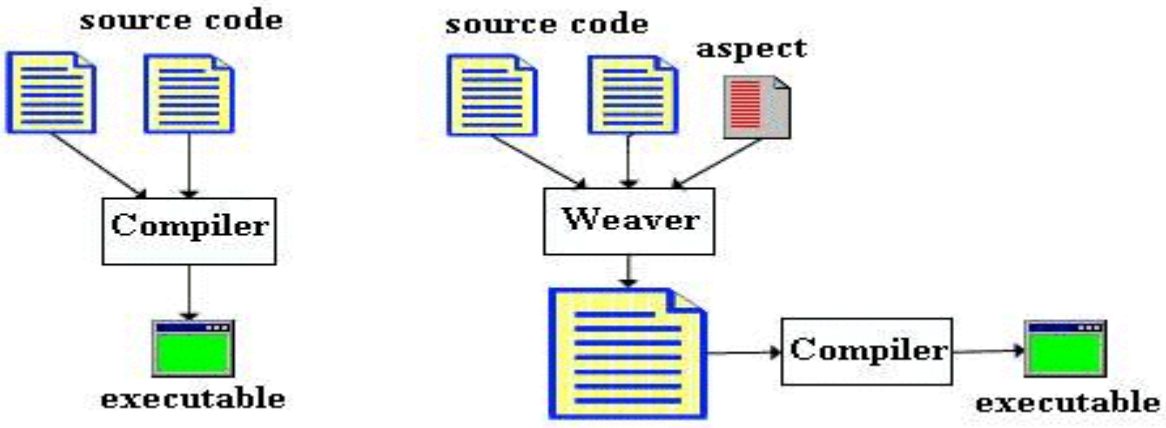
Άρα με βάση τα πιο πάνω δεν πρέπει να υπάρχει το δίλλημα ανάμεσα στους προγραμματιστές για το τι πρέπει να προτιμήσουν ανάμεσα στα δύο καθώς ο AOP δεν είναι κάτι το οποίο είναι εντελώς διαφορετικό από τον OOP και προφανώς δεν δημιουργήθηκε για να τον υπερκαλύψει, ή να τον θέσει στο περιθώριο αλλά για να χρησιμοποιηθεί σε συνδυασμό με αυτόν για να συμπληρώσει τυχών αδυναμίες του και να βελτιώσει την λειτουργικότητα του.

2.4 Aspect Oriented Programming (AOP)

Οι απαιτήσεις από τα έργα λογισμικού ολοένα και αυξάνονται, με αποτέλεσμα αυτά να γίνονται όλο και πιο πολύπλοκα. Ένα χαρακτηριστικό της πολυπλοκότητας αυτής είναι η επανάληψη λειτουργιών, όπως έλεγχοι ασφαλείας, διαχείρισης μνήμης, διαμοίραση πόρων και χειρισμός λαθών και αποτυχιών, σε όλο το εύρος μιας εφαρμογής. Τέτοιες επαναλήψεις δημιουργούν προβλήματα όταν χρειαστεί να γίνουν αλλαγές στα αντίστοιχα σημεία του κώδικα, μιας και οι ίδιες αλλαγές επιβάλλεται να γίνουν σε πολλά διαφορετικά σημεία , με αποτέλεσμα να διαφύγουν κάποιες. Προκειμένου να ξεπεραστεί το πρόβλημα αυτό, συζητούνται μεθοδολογίες βασισμένες σε ένα νέο προγραμματιστικό στοιχείο, το aspect (πτυχή) . Ο προγραμματισμός προσανατολισμένος σε πτυχές (AOP) είναι ένα μέσο προγραμματισμού λειτουργικών μονάδων ξεχωριστά από τον υπόλοιπο κώδικα που στη συνέχεια πλέκονται μαζί. Είναι δηλαδή ένα προγραμματιστικό παράδειγμα που απομονώνει τις λιγότερο σημαντικές λειτουργίες από την επιχειρησιακή λογική του κυρίως προγράμματος.
Είναι μια σχετικά νέα μεθοδολογία για το διαχωρισμό ενός προβλήματος σε μεμονωμένες μονάδες τις πτυχές. Μια πτυχή είναι μια μονάδα που «κόβει» εγκάρσια την λογική ροή μιας εφαρμογής. Έτσι συμπυκνώνει συμπεριφορές που επηρεάζουν πολλαπλές κλάσεις και ενότητες του κώδικα μας και είναι επαναχρησιμοποιήσιμες αλλά η αλλαγή τους δεν έχει απολύτως καμία επίδραση στον υπόλοιπο κώδικα. Αυτό που προσπαθεί να κάνει ο AOP είναι να προωθήσει τα επιθυμητά χαρακτηριστικά του OOP που είναι δύσκολο να εφαρμοστούν στις τρέχουσες τεχνολογίες OOP.

Με τον AOP, ξεκινάμε την υλοποίηση του έργου μας, χρησιμοποιώντας την αντικειμενοστρεφή γλώσσα μας (για παράδειγμα, Java), και στη συνέχεια απασχολούμαστε ξεχωριστά με τα θέματα που σχετίζονται στη δημιουργία επαναχρησιμοποιούμενου κώδικα που τέμνει εγκάρσια τη ροή του προγράμματος, με τη δημιουργία πτυχών. Τέλος, τόσο τα αντικείμενα όσο και οι πτυχές συνδυάζονται σε μια τελική εκτελέσιμη μορφή μέσω ενός "υφαντή πτυχών" (Aspect Weaver). Ως εκ τούτου, μια ενιαία πτυχή μπορεί να συμβάλει αποφασιστικά στην υλοποίηση μιας σειράς μεθόδων, ενοτήτων, ή αντικειμένων του λογισμικού, αυξάνοντας τόσο την δυνατότητα επαναχρησιμοποίησης όσο και της συντήρησης του κώδικα. Θα πρέπει να σημειωθεί ότι ο αρχικός κώδικας δεν χρειάζεται να έχει επίγνωση για κάθε λειτουργία που έχει προσθέσει μία πτυχή.

Το παρακάτω σχήμα εξηγεί τη διαδικασία της ύφανσης.



Για να κατανοήσουμε καλύτερα τον πτυχοστρεφή προγραμματισμό ας δούμε αναλυτικότερα κάποιες από τις βασικές έννοιες πάνω στις οποίες στηρίζεται.

- Aspect (πτυχή)**:** Όπως είπαμε και πριν, μια πτυχή είναι μια μονάδα που μπαίνει εγκάρσια στη λογική ροή μίας εφαρμογής διακόπτοντας την για να εκτελέσει μία λειτουργία. Μπορούμε να σκεφτούμε τα aspects σαν τις κλάσεις του αντικειμενοστρεφούς προγραμματισμού καθώς έχουν τύπο, μπορούν να επεκτείνουν κλάσεις αλλά και άλλα aspect. Όμως διαφέρουν από τις κλάσεις καθώς τα aspects μπορούν να περιλαμβάνουν σαν μέλη pointcuts και advice. Επιπλέων δεν περιέχουν κατασκευαστές και τέλος δεν μπορούν να κληρονομήσουν από συγκεκριμένα aspects. Πιο συγκεκριμένα τα aspects είναι μονάδες ενότητας που περιλαμβάνουν μέσω των cross cutting concerns σε άλλες μονάδες ενότητας. Καθορίζονται από τμηματικές πληροφορίες άλλων μονάδων και υπάρχουν στον σχεδιασμό αλλά και στην

εφαρμογή. Αυτό που κάνει στην ουσία είναι να αλλάζει την συμπεριφορά κάποιων κλάσεων αλλά και αντικειμένων ανεξάρτητα από την ιεραρχία κληρονομικότητας ενθυλακώνοντας συμπεριφορές μέσα σε επαναχρησιμοποιήσιμες μονάδες. Μπορούν να εφαρμοστούν σε δύο περιπτώσεις είτε κατά την διάρκεια του χρόνου εκτέλεσης, είτε κατά την διάρκεια του ελέγχου για συντακτικά λάθη.

Ας δούμε ένα παράδειγμα για να κατανοήσουμε καλύτερα τι είναι ένα aspect πως ορίζεται:

```
Aspect DisplayUpdating{

        Pointcut move(): call( void line.setP1(point)) || call (void line.setP2(point));

        After() returning: move(){

                Display.update();

                        }

                       }
```

Στο παράδειγμα αυτό ορίζουμε το aspect. Το όνομα του είναι DisplayUpdating, έχει σαν μέλη του ένα Pointcut με όνομα move και ένα advice τύπου After returning.

Το πρόβλημα που υπάρχει είναι ότι τα cross cutting concerns δεν ενθυλακώνονται κατάλληλα στις ενότητες τους με αποτέλεσμα να αυξάνεται η πολυπλοκότητα του συστήματος και να κάνει ιδιαίτερα δύσκολη την εξέλιξη του. Έτσι ο AOP προσπαθεί να λύσει αυτό το πρόβλημα επιτρέποντας στον προγραμματιστή να εκφράζει τα cross cutting concerns σε ξεχωριστές ανεξάρτητες μονάδες τα aspects.

- Join point: Σημαίνει δύο πράγματα. Γενικά για την πληροφορική σημαίνει τον έλεγχο της ροής του προγράμματος αλλά για τον AOP ένα σημείο που καθορίζει που πρέπει να εκτελεστεί ο κώδικας που περιέχει ένα advice. Είναι ένα σημείο εκτέλεσης στον βασικό κώδικα στο οποίο εφαρμόζονται τα advice που καθορίζονται σε ένα pointcut. Είναι στην ουσία σημεία συνάντηση του κώδικα του κυρίως προγράμματος με τον κώδικα των aspects.

- Pointcut: Είναι ένα σύνολο από join points. Ελέγχει εάν ένα join point ταιριάζει στον κώδικα. Για οποιαδήποτε εκτέλεση του προγράμματος τα pointcuts υποδεικνύουν ένα σύνολο από jointpoints χωρίς αυτό να σημαίνει ότι θα επιλεχθούν από ένα pointcut όλα τα joinpojnts του που υπάρχουν στον κώδικα. Μπορεί να επιλεχθούν ορισμένα η ακόμα και μόνο ένα. Έχουν την δυνατότητα να συνδυάζονται μεταξύ τους με την βοήθεια των δυαδικών τελεστών &&, || κκαι !.

  Παράδειγμα

  Pointcut move(): call(void line.setP1(point)) || call (void line.setP2(point));

  Το συγκεκριμένο Pointcut έχει όνομα move, ενώ ανάμεσα στις δύο παρενθέσεις που ακολουθούν εισάγονται οι παράμετροι του Pointcut. Επίσης παρατηρούμε ότι στο κύριως σώμα του Pointcut υπάρχει ο δυαδικός τελεστής || που σημαίνει ότι αρκεί να εκτελεστεί μια από τις δύο κλήσεις,

- Advice**:** Μια Advice είναι μια λειτουργία, μέθοδος ή διαδικασία που εφαρμόζεται σε συγκεκριμένο joinpoint του προγράμματος το οποίο πρέπει να έχει επιλεχθεί από κάποιο pointcut. Ο κώδικας της Advice πρέπει να παρεμβληθεί πριν, μετά η ακόμα και αντί του υπάρχοντος κώδικα για κάποιο σύνολο από joinpoint. Η πρακτική χρήση του Advice είναι γενικά να τροποποιηθούν ή να επεκτείνουν την συμπεριφορά λειτουργιών που δεν μπορούν εύκολα να τροποποιηθούν η να επεκταθούν. Υπάρχουν τρία είδη :

  - Before

    Εκτελείται όταν φτάσει σε ένα joinpoint πριν το πρόγραμμα το προχωρήσει στην εκτέλεση του και δεν έχει την ικανότητα να αποτρέψει την ροή εκτέλεσης προς το joinpoint:

    Before(): log() {

    System.out.println(“logging before method call”);

    }

Η λέξη Before δηλώνει το είδος Advice. Αυτό που ακολουθεί μετά από το όνομα του Advice είναι το pointcut στο οποίο αναφέρεται η Advice, στην περίπτωση μας είναι το log(). Μέσα στις αγκύλες βρίσκεται το κυρίως σώμα του Advice, δηλαδή αυτό που πρέπει να εκτελεστεί πριν από το pointcut log().

- After

  Εκτελείται ανεξάρτητα από τον τρόπο με τον οποίο τελειώνει ένα joinpoint. Χωρίζεται σε δύο κατηγορίες :

  - After returning Advice

    Η οποία εκτελείτε μετά την κανονική ολοκλήρωση του joinpoint.

  - After throwing Advice

    Η οποία εκτελείτε όταν μια μέθοδος σταματήσει εξαιτίας μιας εξαίρεσης.

  Εκτελείται σε ένα συγκεκριμένο joinpoint μόλις το πρόγραμμα τελειώσει την εκτέλεση του.

- Around

  Αποτελεί το πιο ισχυρό είδος Advice. Μπορεί να εκτελέσει μια συμπεριφορά πριν και μετά την κλήση μίας μεθόδου και έχει την δυνατότητα να αποφασίσει αν το πρόγραμμα θα συνεχίσει με την εκτέλεση του joinpoint ή όχι.

- Cross cutting concerns: Οι Cross cutting concerns είναι δευτεροβάθμιες απαιτήσεις που μοιράζονται μεταξύ των μονάδων ενός προγράμματος. αποτελούν και οι ίδιες πτυχές του κώδικα του προγράμματος που επηρεάζουν άλλες concerns. Στις περισσότερες περιπτώσεις οι Cross cutting concerns είναι δύσκολο να διαχωριστούν από το υπόλοιπο πρόγραμμα. Αυτό έχει συνήθως ως αποτέλεσμα διπλασιασμό του κώδικα ή έναν κώδικα πολύπλοκο και μπερδεμένο. Συνεπώς με τις Cross cutting concerns υφίσταται το θέμα της συνοχής του κώδικα. Όταν μια ανησυχία διαδίδεται σε ένα μεγάλο αριθμό μεθόδων το αποτέλεσμα είναι ότι ο κώδικας που εφαρμόζει η ανησυχία να είναι διεσπαρμένος μεταξύ πολλών συστατικών του προγράμματος, κάτι που τον κάνει πιο δύσκολο

στην κατανόηση και την συντήρηση του. Όταν σε ένα πρόγραμμα οι ανησυχίες που υπάρχουν διαδίδονται σε πολλά συστατικά του προγράμματος τότε τα συστατικά αυτά καταλήγουν πλεγμένα με πολλές ανησυχίες, όπως ασφάλεια, καταγραφή κλπ. Για να αλλαχθεί κάποιο από αυτά τα συστατικά επομένως προϋποθέτει ότι έχουν κατανοηθεί πλήρως όλες οι πλεγμένες ανησυχίες. Ορισμένα παραδείγματα ανησυχιών είναι η καταγραφή (logging), η επικύρωση (validation) και η ασφάλεια (security).

- Aspect Weaving: Είναι μια μεταπρογραμματιστική λειτουργία που είναι σχεδιασμένη να παίρνει οδηγίες από τα aspects και να παράγει τον τελικό κώδικα της εφαρμογής . Συγχωνεύοντας aspects και κλάσεις δημιουργεί είτε πλεγμένες κλάσεις είτε πλεγμένο κώδικα.

- Dynamic weaving: Είναι μια από τις τεχνικές που χρησιμοποιεί ο AOP και επιτρέπει στο aspect να δεσμευτεί από το πρόγραμμα σε χρόνο εκτέλεσης. Είναι μια τεχνική παρόμοια με την τεχνική της δυναμικής σύνδεσης που εφαρμόζει ο OOP που αποτελεί ένα είδος πολυμορφισμού. Μπορεί να διακριθεί σε δύο κατηγορίες :
- Load time weaving

  Οι compiled aspects και οι κλάσεις αλληλεπιδρούν κατά την διάρκεια φόρτωσης των κλάσεων στη java virtual machine.

- Run time weaving

  Οι aspects πλέκονται με κλάσεις που έχουν ήδη φορτωθεί στην java virtual machine. Αυτό είναι ιδιαίτερα χρήσιμο όταν μια νέα ανησυχία πρόκειται να προστεθεί σε πολλές ενότητες χωρίς να δημιουργηθεί πρόβλημα στην εφαρμογή.

Για να γίνουν πιο κατανοητά όλα τα πιο πάνω ας δούμε ένα παράδειγμα:

```
public class TestClass {
        public void sayHello () {
                System.out.println (" Hello, AOP");
```

```
                                    }
        public void sayAnyThing (String s) {
                System.out.println (s);
                                                }
public static void main (String[] args) {
sayHello ();
sayAnyThing ("ok");
                                            }
                        }
```

Στο παράδειγμα μας έχουμε μια κλάση με όνομα TestClass η οποία τυπώνει το μήνυμα Hello Aop και μια συνάρτηση sayAnyThing η οποία τυπώνει ένα μήνυμα που δίνει ο χρήστης.

Ας υποθέσουμε τώρα ότι θέλουμε να κάνουμε τις εξής αλλαγές στον παραπάνω κώδικα:

Θέλουμε να τυπώσουμε ένα μήνυμα πριν και μετά από κάθε κλήση της συνάρτησης testClass.sayHello() και με κάποιον τρόπο να ελέγχουμε ότι το όρισμα της συνάρτησης testClass.sayAnyThing() είναι τουλάχιστο 3 χαρακτήρες. Σε περίπτωση που οι χαρακτήρες είναι λιγότεροι από 3 τότε ο κώδικας να τυπώνει ένα μήνυμα λάθους. Ο κώδικας τώρα γίνεται ως εξής:

```
public aspect MyAspect {
        public pointcut sayMethodCall ():  call ( public voidTestClass.say*() );
        Public pointcut sayMethodCallArg (String str):  call
                                                        (
        public void TestClass.sayAnyThing (String))  &&  args (str);
        before (): sayMethodCall() {
        System.out.println(" \n TestClass." +thisJoinPointStaticPart.getSignature().getName()
        + "start..." );
}
        After (): sayMethodCall() {
        System.out.println(" \n TestClass." +
        ThisJoinPointStaticPart.getSignature().getName() + "end...");
}
        before(String str): sayMethodCallArg(str) {
```

```
        if (str .length() < 3) {
                System.out.println ("Error: I can't say words less than 3 characters");
        return;
                        }
                                }
}
```

Στην πρώτη γραμμή ορίζεται μια aspect με το όνομα MyAspect με τον ίδιο τρόπο που ορίζεται και μια κλάση στην java. Μέσα στη κλάση testClass έχουμε ορίσει δύο joinpoints και ορίζουμε που θα γίνουν οι αλλαγές. Τα δύο joinpoints είναι η κλήση της συνάρτησης testClass.sayHello και testClass.sayAnyThing. Το pointcut Public pointcut sayMethodCall(): call (public void TestClass.say*()); Ονομάζεται sayMethodCall και διαλέγει οποιαδήποτε κλήσης της συνάρτησης testClass.sayHello. Επίσης, διαλέγει και οποιαδήποτε λέξη που ξεκινάει με «say». Τα pointcuts που χρησιμοποιούμε μας βοηθούν στον ορισμό των advice. Οι advice χρησιμοποιούνται για να καθορίσουν επιπλέον κώδικα που εκτελείται πριν, μετά ή γύρω από τα join points.

2.5 Πλεονεκτήματα – Μειονεκτήματα (AOP)

Τα πλεονεκτήματα που έχει η τεχνική του AOP είναι αρκετά. Ο διαχωρισμός των συστατικών ενός προβλήματος, όπου βοηθάει στην καλύτερη κατανόηση του λογισμικού λόγω του υψηλού επιπέδου της αφαίρεσης συνεισφέρει στην ευκολότερη ανάπτυξη αλλά και διατήρηση του προγράμματος επειδή αποτρέπεται το πλέξιμο του κώδικα και αυξάνει την δυνατότητα επαναχρησιμοποίησης τόσο για τον κώδικα του κυρίως προγράμματος όσο και για τον κώδικα των aspects. Επιπλέων μειώνεται το κόστος μελλοντικής εφαρμογής καθώς και ότι μπορεί να προσφέρει την δυνατότητα αυτόματου ελέγχου του κώδικα χωρίς να παραποιηθεί.. Επειδή ο AOP εφαρμόζει κάθε aspect σαν ξεχωριστή ενότητα, μία μεμονωμένη ενότητα δεν συνδέεται τόσο στενά με το υπόλοιπο πρόγραμμα, και προσφέρει περισσότερη επαναχρησιμοποίηση κώδικα. Προσφέρει τρόπους καθορισμού και ενθυλάκωσης των ανησυχιών σε ένα σύστημα. Τέλος μας προσφέρει συστήματα που είναι ευκολότερο να εξελιχτούν αφού ορισμένες ενότητας του κώδικα μπορεί να μην γνωρίζουν το ποιες ή πόσες ανησυχίες υπάρχουν στο σύστημα και επομένως είναι εύκολη η προσθήκη επιπλέον λειτουργικότητας με την δημιουργία καινούργιων aspects.
Όμως είναι λογικό ο AOP εκτός από τα οφέλη έχει και κάποια μειονεκτήματα. Δεν μπορεί να λύνει όλα τα προβλήματα και η ροή του προγράμματος είναι δύσκολο να ακολουθηθεί.

Επιπλέον η υποστήριξη των εργαλείων που χρησιμοποιούνται είναι σχετικά μικρή. Τα τυχόν λάθη στις ανησυχίες μπορούν να οδηγήσουν σε αποτυχία ολόκληρου του προγράμματος όπως και οι αλλαγές που μπορεί να γίνουν στους δείχτες ενός προγράμματος και δεν τις περιμένει ο χρήστης. Τέλος η χρησιμοποίηση AOP για την προσθήκη επιπλέον κώδικα μπορεί να οδηγήσει σε αναίρεση της ασφάλειας.

2.6 Σύγκριση με τις παραδοσιακές τεχνικές

Οι περισσότεροι προγραμματιστές γνωρίζουν και ξέρουν να προγραμματίζουν χρησιμοποιώντας τις παραδοσιακές τεχνικές προγραμματισμού (OOP) που είναι και η πιο ευρέως γνωστή μέθοδος προγραμματισμού. Η μέθοδος αυτή είναι πλέον διαδεδομένη με κύριο χαρακτηριστικό τον κατακερματισμό ενός προβλήματος σε αντικείμενα που χαρακτηρίζονται από ιδιότητες (μεθόδους) και δεδομένα (μεταβλητές).

Παρά το γεγονός ότι ο OOP έχει μεγάλη επιτυχία στη διαμόρφωση και υλοποίηση πολύπλοκων συστημάτων λογισμικού, έχει και τα προβλήματα του. Προβλήματα που αφορούν την διατήρηση του κώδικα, και όσο πιο μεγάλο το λογισμικό που υλοποιείται τόσο πιο δύσκολα γίνεται ο διαχωρισμός του έργου σε ενότητες (αντικείμενα), πράγμα το οποίο αποτελεί τη βάση του OOP. Για παράδειγμα μια μικρή αλλαγή σε μια επαναχρησιμοποιήσιμη ενότητα μπορεί τελικά να προκαλέσει πολλές αλλαγές σε άλλες, ανεξάρτητες, ενότητες του κώδικα. Αυτά καθώς και άλλα προβλήματα έρχεται να λύσει μια διαφορετική τεχνική ο πτυχοστρεφής (AOP) προγραμματισμός. Γεγονός που προσθέτει κάτι καινούργιο αλλά όχι για να αφανίσει τις προηγούμενες τεχνικές αλλά για να υπάρξει μαζί με αυτές και να συμπληρώσει τυχόν αδυναμίες.

Κεφάλαιο 3 Σύστημα MHCPMS

3.1 Περιγραφή συστήματος MHCPMS

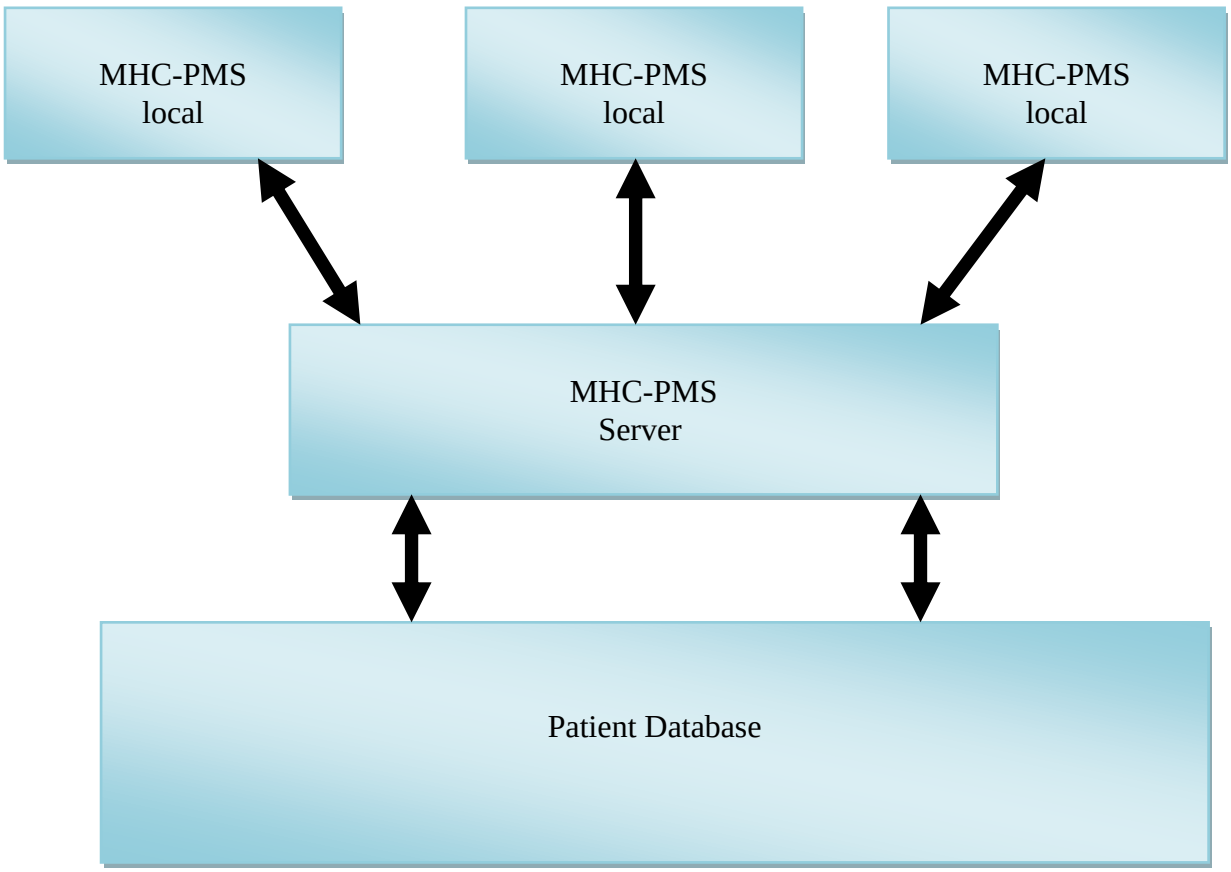
Το σύστημα MHCPMS **(**Mental Health Care Patient Management System) είναι ένα σύστημα που έχει να κάνει με την διαχείριση και την φροντίδα ασθενών με ψυχολογικά προβλήματα. Είναι μια παραλλαγή ενός είδη υπάρχον συστήματος που έχει χρησιμοποιηθεί σε αρκετές κλινικές της Αγγλίας.

Είναι ένα σύστημα το οποίο διατηρεί πληροφορίες σχετικά με το ιατρικό ιστορικό του κάθε ασθενή που πάσχει από ψυχολογικά προβλήματα και τις διάφορες θεραπείες που έχει λάβει. Οι περισσότεροι ασθενείς δεν απαιτούν κάποια πολύ εξειδικευμένη νοσοκομειακή περίθαλψη αλλά θα χρειάζεται τακτικά να παρακολουθούνται από γιατρούς οι οποίοι θα έχουν λεπτομερή γνώση των προβλημάτων τους.

Έτσι με το σύστημα MHCPMS έχουμε μια κεντρική βάση δεδομένων όπου υπάρχουν όλες οι πληροφορίες των ασθενών και τρέχει σε ένα υπολογιστή. Εκεί μπορούν να εξασφαλίσουν πρόσβαση οι γιατροί, καθώς επίσης και να χρησιμοποιηθεί από τις περιοχές που δεν έχουν ασφαλή σύνδεση δικτύου και επιπλέων επιτρέπει σε άτομα όπως νοσηλευτές που μπορεί να μην έχουν είσοδο στο σύστημα να κατεβάσουν και να χρησιμοποιήσουν τοπικά αντίγραφα των φακέλων των ασθενών όταν έχουν αποσυνδεθεί.

Το σύστημα δεν είναι ένα πλήρες ιατρικό σύστημα αρχείων άρα είναι λογικό να του λείπουν πληροφορίες που αφορούν άλλα ιατρικά προβλήματα των ασθενών, αλλά παρόλα αυτά

μπορεί να αλληλεπιδράσει και να ανταλλάξει δεδομένα με άλλα κλινικά πληροφοριακά συστήματα και να πάρει τις πληροφορίες που χρειάζεται. Όπως φαίνεται στο πιο κάτω σχήμα.



Το σύστημα αυτό έχει δύο σημαντικούς στόχους. Αρχικά να δημιουργήσει ένα αρχείο από το οποίο θα μπορούν εύκολα οι διαχειριστές των υπηρεσιών υγείας να έχουν πρόσβαση στα αρχεία αυτά και να απορρέουν πληροφορίες που χρειάζονται αξιολογώντας το σε σχέση με τους τοπικούς στόχους. Έπειτα να μπορεί να ενημερώνει το ιατρικό προσωπικό έγκαιρα για το πια θεραπεία πρέπει να ακολουθήσει για τον κάθε ασθενή, και να τον βοηθήσει σε ότι χρειαστεί καθώς ασθενείς με τέτοια προβλήματα μπορούν συχνά να αποδιοργανωθούν, να χάσουν κάποιο ραντεβού, τις συνταγές των φαρμάκων ή ακόμα και να ξεχάσουν τις οδηγίες των γιατρών και όταν τους πλησιάσει κάποιος άλλος γιατρός να του δώσουν ασαφής ή και

λανθασμένες πληροφορίες , γεγονός που μπορεί να επιφέρει κίνδυνο τόσο για την σωματική ακεραιότητα του γιατρού όσο και για τον ίδιο τον ασθενή.

Τα βασικά χαρακτηριστικά του συστήματος είναι:

- Η ατομική διαχείριση της υγείας. Οι γιατροί μπορούν να δημιουργήσουν τα αρχεία για τους ασθενείς, να επεξεργαστούν τις πληροφορίες στο σύστημα καθώς και να δουν το ιστορικό κάποιου ασθενή. Κάτι που επιπλέον μπορεί να προσφέρει το σύστημα είναι ότι μπορεί να περιλαμβάνει κάποιες περιλήψεις για το κάθε ασθενή έτσι ώστε ο γιατρός να μπορεί να μάθει γρήγορα σχετικά τα προβλήματα καθώς και τις θεραπείες που έχουν χορηγηθεί για το κάθε ασθενή.

- Η παρακολούθηση των ασθενών. Το σύστημα παρακολουθεί τακτικά τα αρχεία των ασθενών και ενημερώνει ότι νέα αλλαγή μπορεί να γίνει είτε στην αλλαγή θεραπείας είτε στη χορήγηση κάποιου νέου φαρμάκου. Επιπλέον βοηθάει στο να ανιχνευτούν πιθανών προβλήματα που μπορεί να υπάρξουν, αν για παράδειγμα κάποιος ασθενής δεν έχει δει κάποιο γιατρό για μεγάλο χρονικό διάστημα τότε μπορεί να εμφανίσει μια προειδοποίηση για να ενημερώσει τους γιατρούς.

- Διοικητική αναφορά. Το σύστημα μπορεί να παράγει μηνιαίες εκθέσεις που να δείχνει τον αριθμό των ασθενών που έλαβαν κάποια θεραπεία σε κάθε κλινική καθώς και τον αριθμό των ασθενών που έχουν επισκεφτεί την κλινική αυτή και την φαρμακευτική αγωγή που τους έχει χορηγηθεί. Όπως όμως ότι έχει να κάνει με την υγεία πρέπει να διέπεται από κάποιο ιατρικό απόρρητο έτσι και στο σύστημα αυτό πρέπει να υπάρχει ο νόμος αυτός. Δηλαδή θα πρέπει να μην υπάρχει διαρροή κάποιων δεδομένων που αφορούν τους ασθενείς, καθώς και να μην μπορεί να έχει πρόσβαση στο αρχείο κάποιος ο οποίος δεν χρειάζεται να έχει πρόσβαση. Επιπλέων υπάρχει ο νόμος ο οποίος υποχρεώνει τους γιατρούς να κρατούν υποχρεωτικά στην κλινική τους ασθενείς που πιστεύουν ότι μπορεί να αποτελέσουν κίνδυνο είτε για του ίδιους είτε για κάποιο άλλο άνθρωπο.

3.2 Λειτουργικές απαιτήσεις

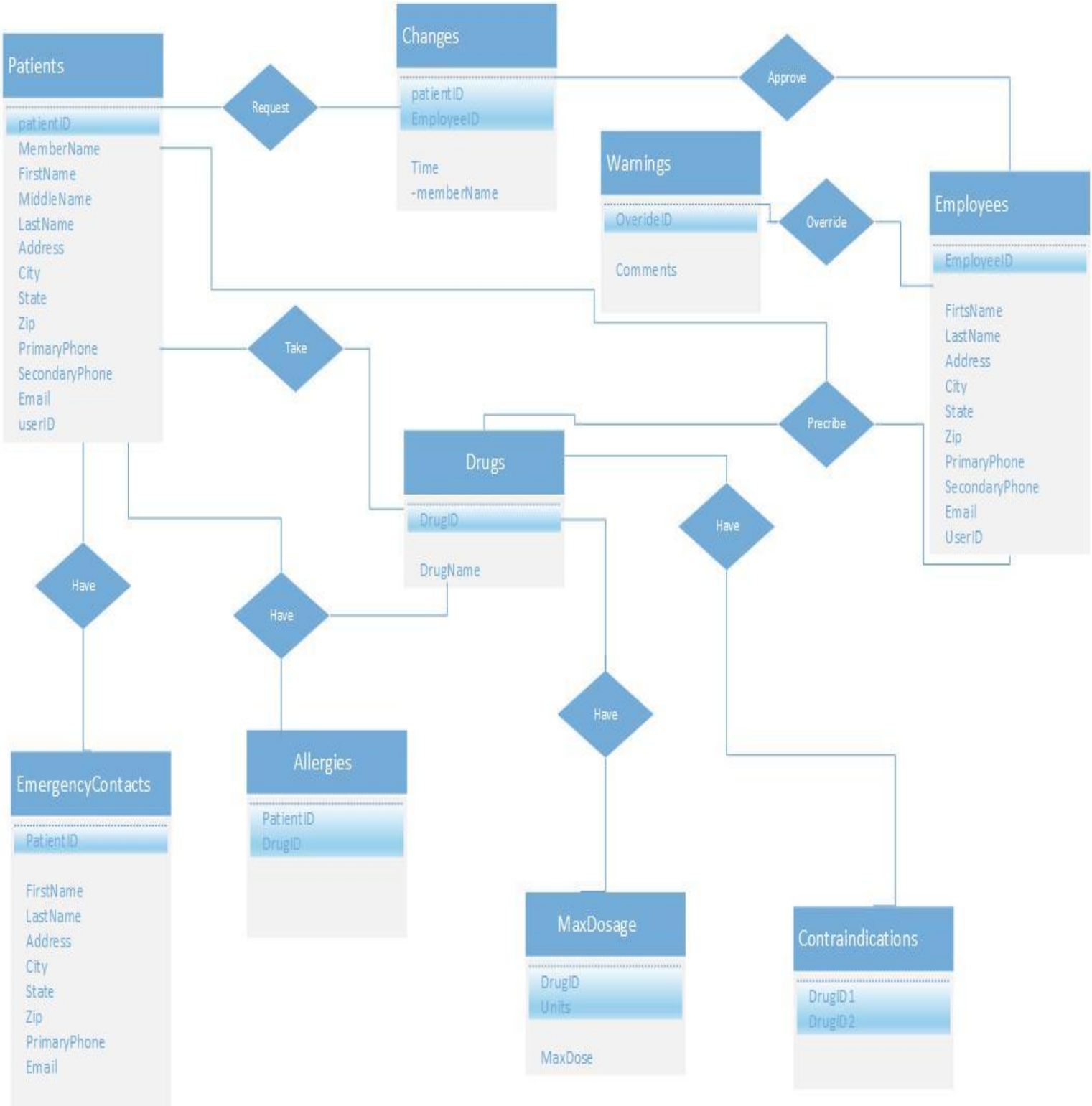
Οι απαιτήσεις του συστήματος χωρίζονται σε λειτουργικές και μη λειτουργικές. Πιο κάτω παρουσιάζονται όλες οι απαιτήσεις:

1. Οι χρήστες του συστήματος θα πρέπει να είναι σε θέση να μπορούν να δημιουργήσουν και να τροποποιήσουν τις πληροφορίες στους φακέλους των ασθενών.
2. Οι χρήστες θα πρέπει να είναι σε θέση να ανακτήσουν τα δεδομένα από το σύστημα. Για παράδειγμα ο ασθενής να μπορεί να ζητήσει την προσωπική του κατάσταση.
3. Το σύστημα θα πρέπει να μπορεί να παράγει περιοδικές εκθέσεις για την αναθεώρηση της διαχείρισης.
4. Το σύστημα θα πρέπει να μπορεί να παράγει προειδοποιήσεις/ειδοποιήσεις.
5. Το σύστημα θα πρέπει να έχει ένα χαρακτηριστικό για να μπορεί να χειριστεί το θάνατο ενός ασθενή.
6. Οι ασθενείς θα πρέπει να είναι σε θέση να ζητούν προσωπικές πληροφορίες και να ζητούν αλλαγές.
7. Το σύστημα θα υποστηρίζει λειτουργίες εκτύπωσης για τις συνταγές φαρμάκων ή εκθέσεις.
8. Το σύστημα θα πρέπει να επιτρέπει την αξιολόγηση του κινδύνου, όταν η ταξινόμηση κινδύνου ενός ασθενή (χαμηλή/μεσαία/υψηλή) αλλάζει από το ένα επίπεδο στο άλλο.

3.3 Μη λειτουργικές απαιτήσεις

1. Το σύστημα θα πρέπει να περιορίσει την πρόσβαση σε εξουσιοδοτημένους χρήστες και να μην επιτρέψουμε σε αυτούς τους χρήστες να έχουν πρόσβαση σε πληροφορίες έξω από το ρόλο τους.
2. Οι λειτουργικές δαπάνες οι οποίες μπορεί να περιλαμβάνουν κόστος συντήρησης και λειτουργίας θα πρέπει να είναι λογικά.
3. Το σύστημα θα πρέπει να είναι φιλικό προς το χρήστη.
4. Το σύστημα πρέπει να λαμβάνει υπόψη ότι ο ασθενής μπορεί να προκαλέσει βλάβη είτε στον εαυτό του είτε σε κάποιο άλλο.
5. Το σύστημα πρέπει να έχει αξιοπιστία.

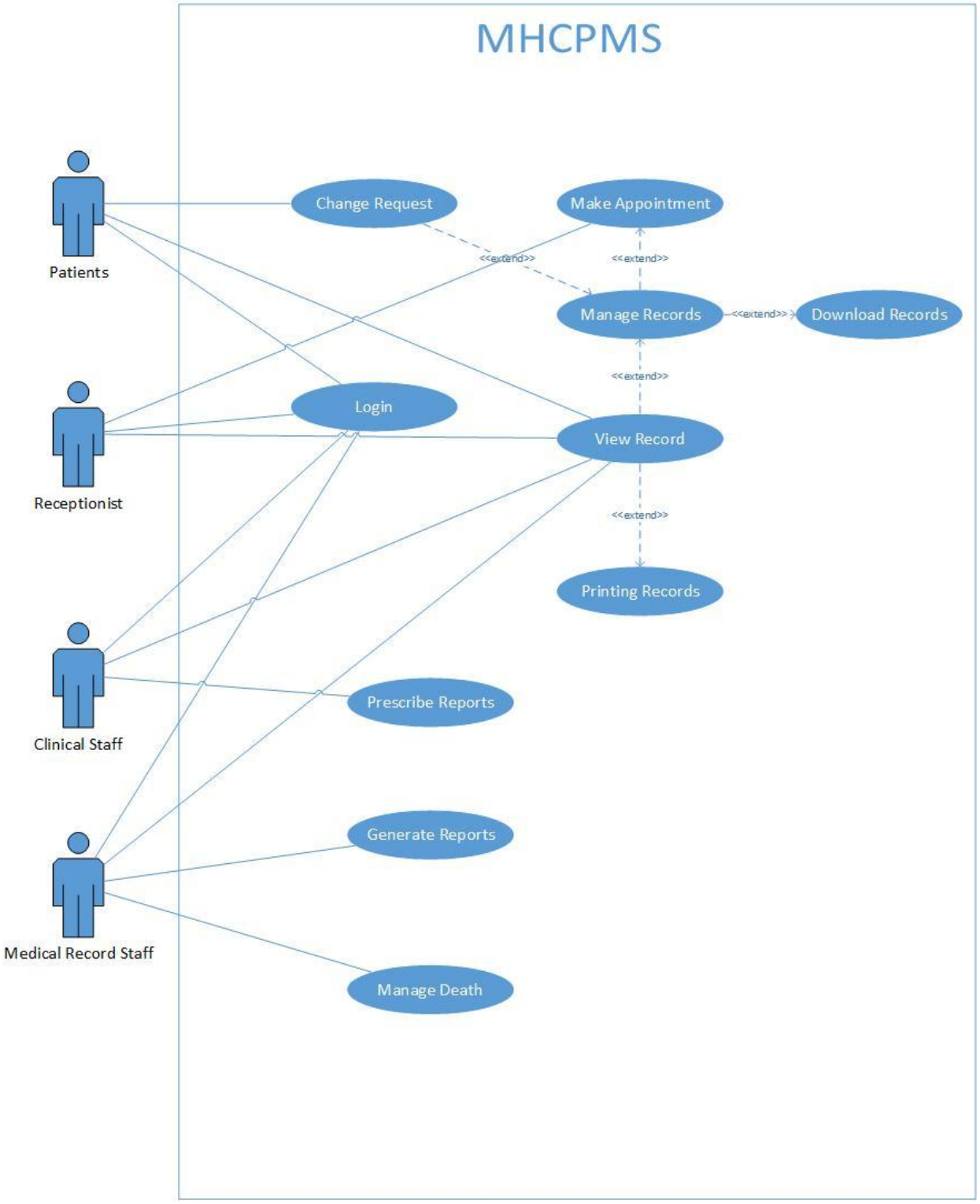
3.4 Διαγράμματα UML



3.4.1 Διάγραμμα σχέσεων οντοτήτων

Περιπτώσεις χρήσης

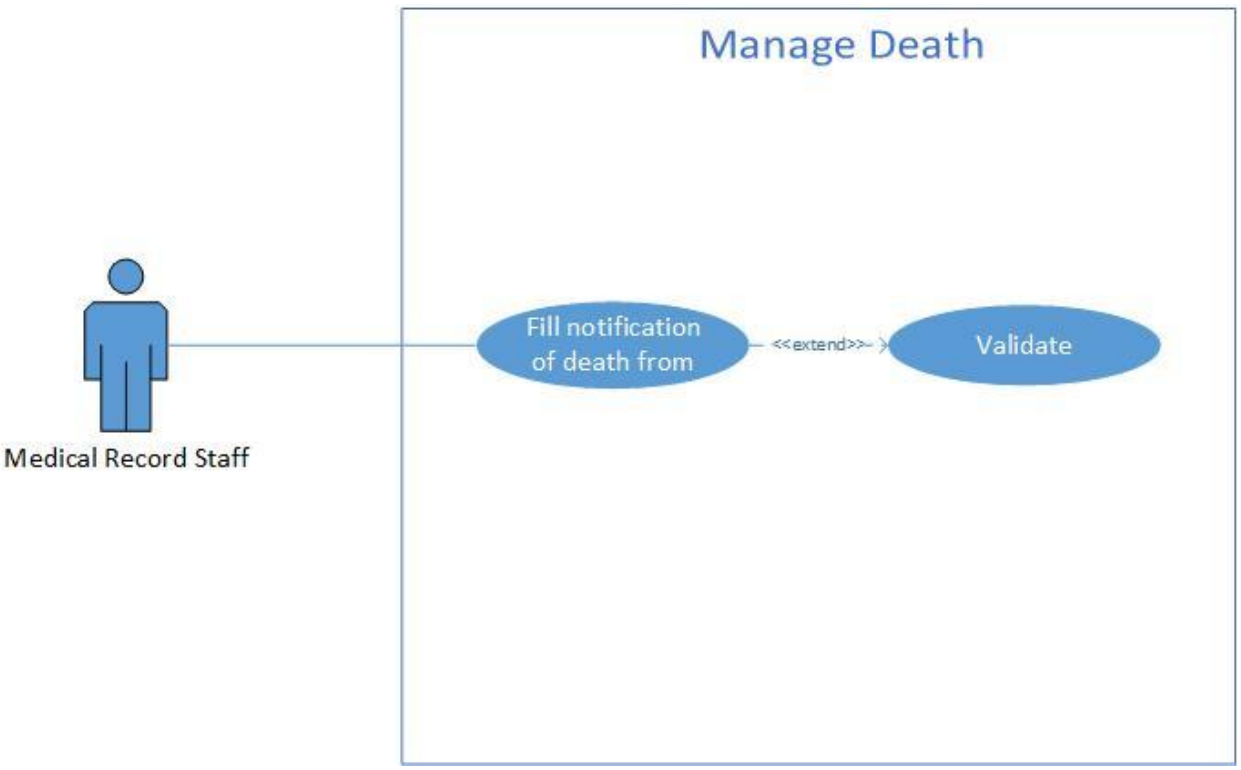
MHC-PMS Υπόθεση Χρήστη



Actors: ασθενείς, ρεσεψιονίστ, κλινική, προσωπικό, ιατρικό προσωπικό
Ο κάθε actor θα πρέπει να συνδεθεί για να αλληλεπιδράσει με το σύστημα MHC-PMS.

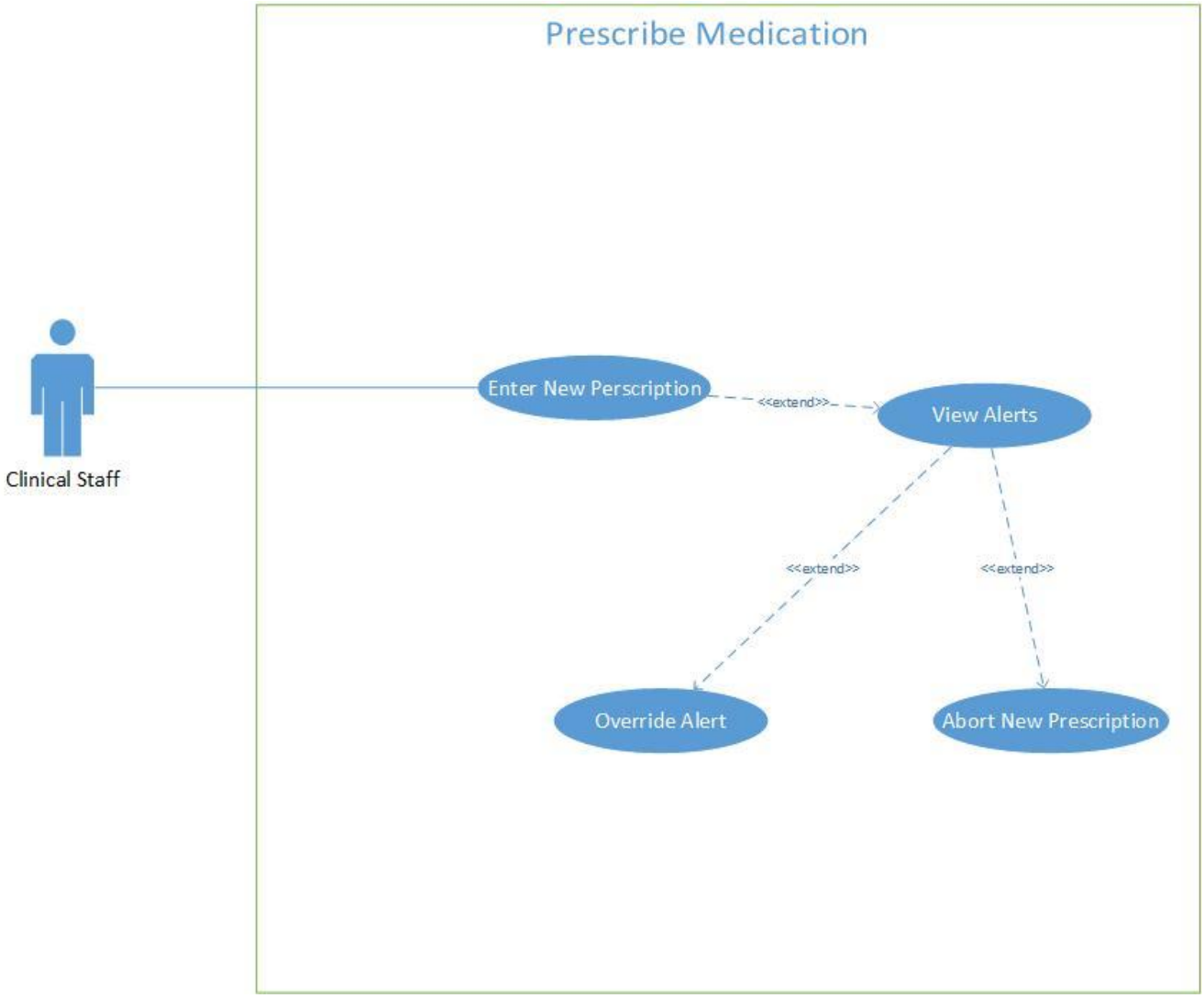
Περιγραφή: Οι ασθενείς μπορούν να δουν το δικό τους ιατρικό ιστορικό και μπορούν να ζητήσουν αλλαγές σε αυτά τα αρχεία. Οι ρεσεψιονίστ μπορούν να κλείσουν ένα ραντεβού σε ένα ασθενή καθώς επίσης να δει και το ιστορικό των ραντεβού του ασθενή. Το προσωπικό της κλινικής μπορεί να δημιουργήσει και να διαχειριστεί τα αρχεία των ασθενών. Μπορούν επίσης να κατεβάσουν αυτά τα αρχεία σε ένα προσωπικό υπολογιστή, και να συνταγογραφήσουν φάρμακα. Το ιατρικό προσωπικό μπορεί να κάνει εγγραφές, να δημιουργήσουν αναφορές στο αίτημα της διοίκησης και μπορεί επίσης να κινήσεί τη διαδικασία αρχειοθέτησης που έχει να κάνει με το θάνατο του ασθενούς. Όλοι οι φορείς που έχουν πρόσβαση και μπορούν να δουν τα αρχεία έχουν την ικανότητα να τις εκτυπώσουν.

Περίπτωση χρήσης 1: Ο θάνατος του ασθενή



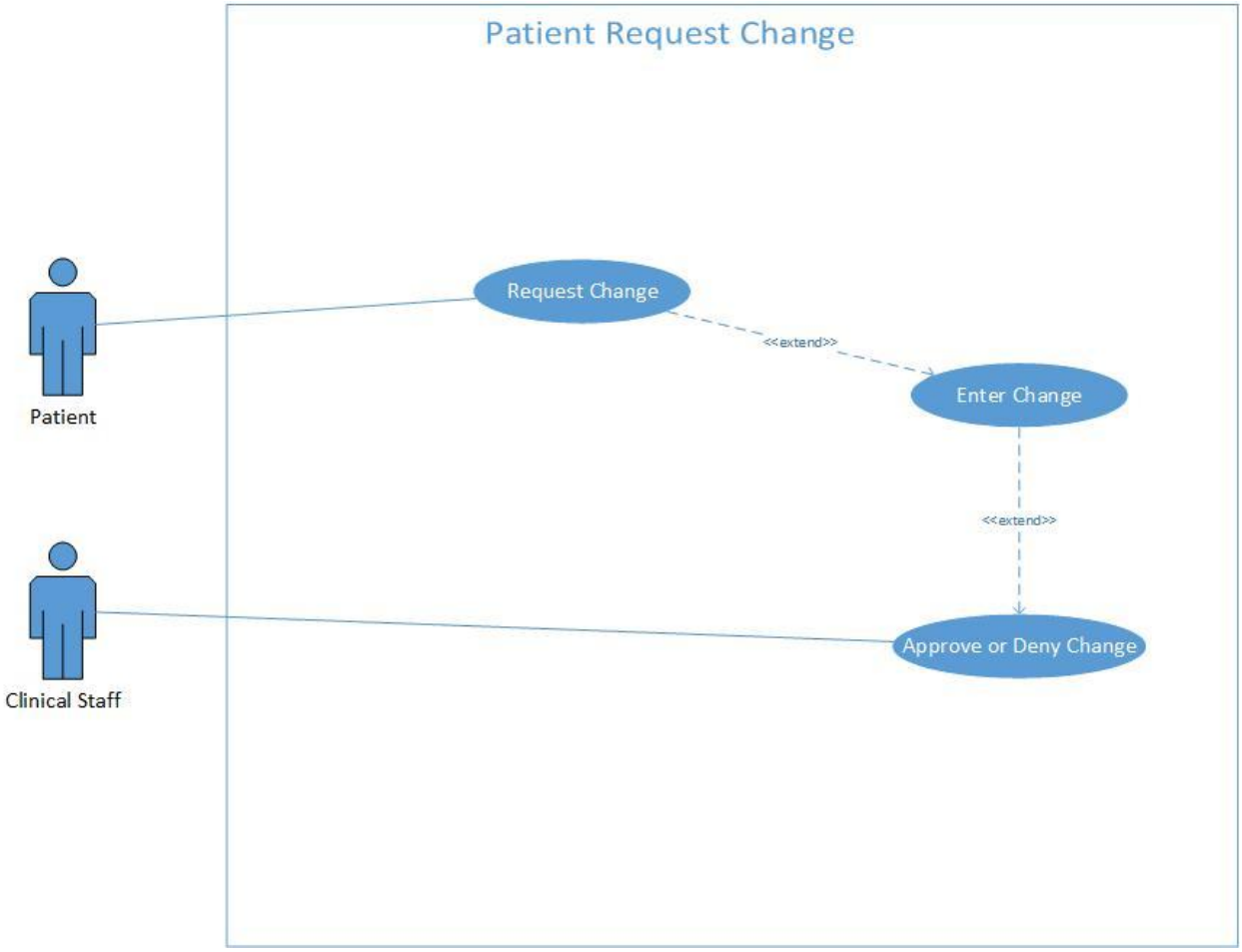
Περιγραφή: Όταν μια κλινική λάβει ειδοποίηση ότι ο ασθενής έχει πεθάνει, ένα μέλος του ιατρικού προσωπικού πρέπει να συμπληρώσει μια φόρμα με τον αριθμό ταυτότητας του ασθενούς. Το σύστημα θα ανακτήσει το όνομα του ασθενούς και θα δώσει στο χρήστη τη δυνατότητα να επικυρώσει ότι είναι ο σωστός ασθενής. Μόλις ο χρήστης επιβεβαιώνει ότι ο σωστός ασθενής εισήχθη, τότε το σύστημα θα αποτρέψει οποιαδήποτε μελλοντική τροποποίηση του ραντεβού, συνταγή του ασθενούς και ιατρικά αρχεία.

Περίπτωση χρήσης 2: Δημιουργία συνταγής



Περιγραφή: Όταν ένας γιατρός επιθυμεί να συνταγογραφήσει φάρμακα, θα εισέλθουν στο σύστημα η ταυτότητα του ασθενούς και το φάρμακο που πρέπει να του χορηγηθεί. Το σύστημα θα ανακτήσει έναν κατάλογο όλων των επί του παρόντος φαρμάκων που λαμβάνονται από τον ασθενή από την βάση δεδομένων καθώς επίσης και πιθανές αλλεργίες που μπορεί να έχει ο ασθενής στο συγκεκριμένο φάρμακο. Αν υπάρχει κάποιο πρόβλημα με κάποιο φάρμακο, τότε ο γιατρός θα λάβει μια προειδοποίηση. Όπως και να έχει ο χρήστης θα λάβει μια επικύρωση είτε ότι η συνταγή δεν παρουσιάζει κανένα γνωστό κίνδυνο για τον ασθενή, ή μια προειδοποίηση ότι η συνταγή δεν συνιστάται. Ο γιατρός έχει την δυνατότητα να παρακάμψει την προειδοποίηση καθώς επίσης και να ακυρώσει την συνταγή. Εάν η συνταγή πρόκειται να χορηγηθεί, θα εκτυπωθεί από το σύστημα.

Περίπτωση χρήσης 3: Αίτηση αλλαγής πληροφοριών ασθενή

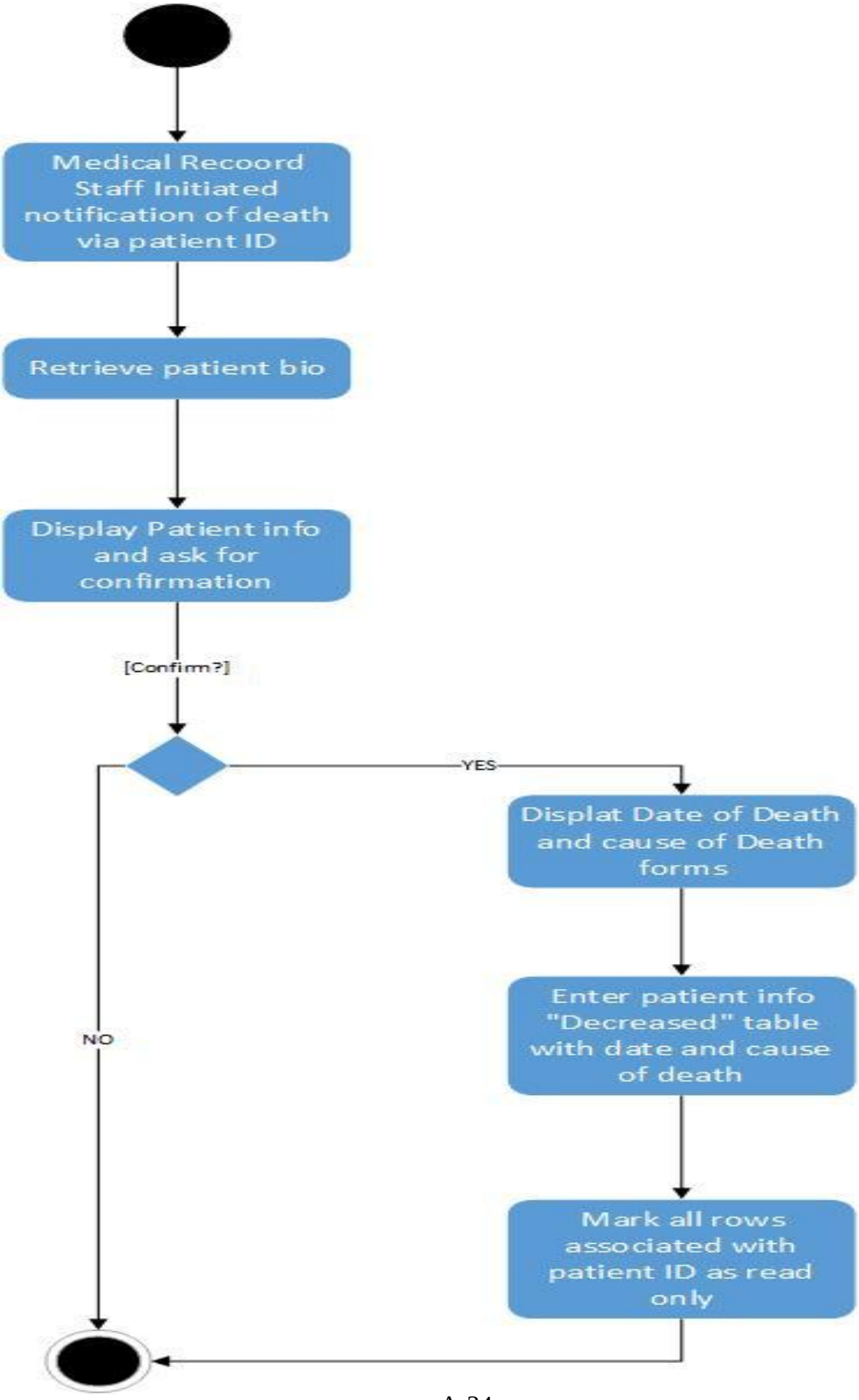


Περιγραφή: εάν ένας ασθενής επιθυμεί να αλλάξει ένα τμήμα του ιατρικού φακέλου του, θα συμπληρώσει μια φόρμα που δείχνει ποια στοιχεία θέλει να αλλάξει. Το σύστημα θα παρέχει ένα πεδίο εισαγωγής για τις νέες τιμές. Μόλις ο χρήστης, υποβάλλει την παρούσα μορφή, μια ειδοποίηση μέσω ηλεκτρονικού ταχυδρομείου θα σταλεί στην γιατρό του ασθενούς, ζητώντας από τον κλινικό ιατρό να αποδεχθεί ή να απορρίψει αυτές τις αλλαγές. Ο γιατρός θα συνδεθεί και θα έχει τη δυνατότητα να αποδεχθεί ή να απορρίψει κάθε επιμέρους τροποποίηση που ζητήθηκε από τον ασθενή. Ανάλογα με την απόκριση, η βάση δεδομένων είτε μπορεί να ενημερωθεί είτε όχι. Ο ασθενής θα πρέπει να ενημερώνεται από το γιατρό μέσω ενός ηλεκτρονικού μηνύματος.

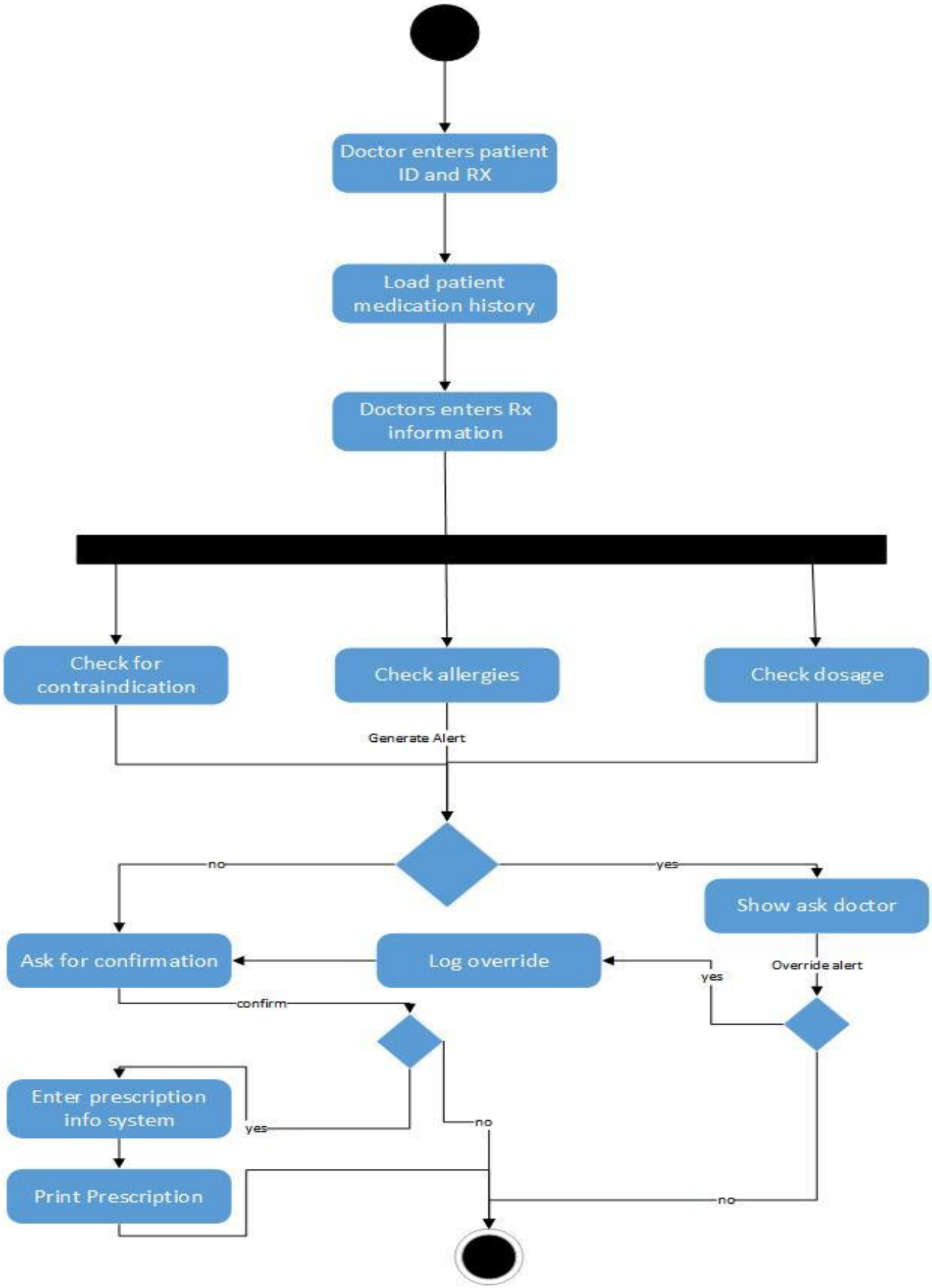
Οι περιγραφές αυτές αναφέρονται και στα υπόλοιπα διαγράμματα απλά φαίνονται πιο αναλυτικά.

3.4.2 Διαγράμματα δραστηριοτήτων

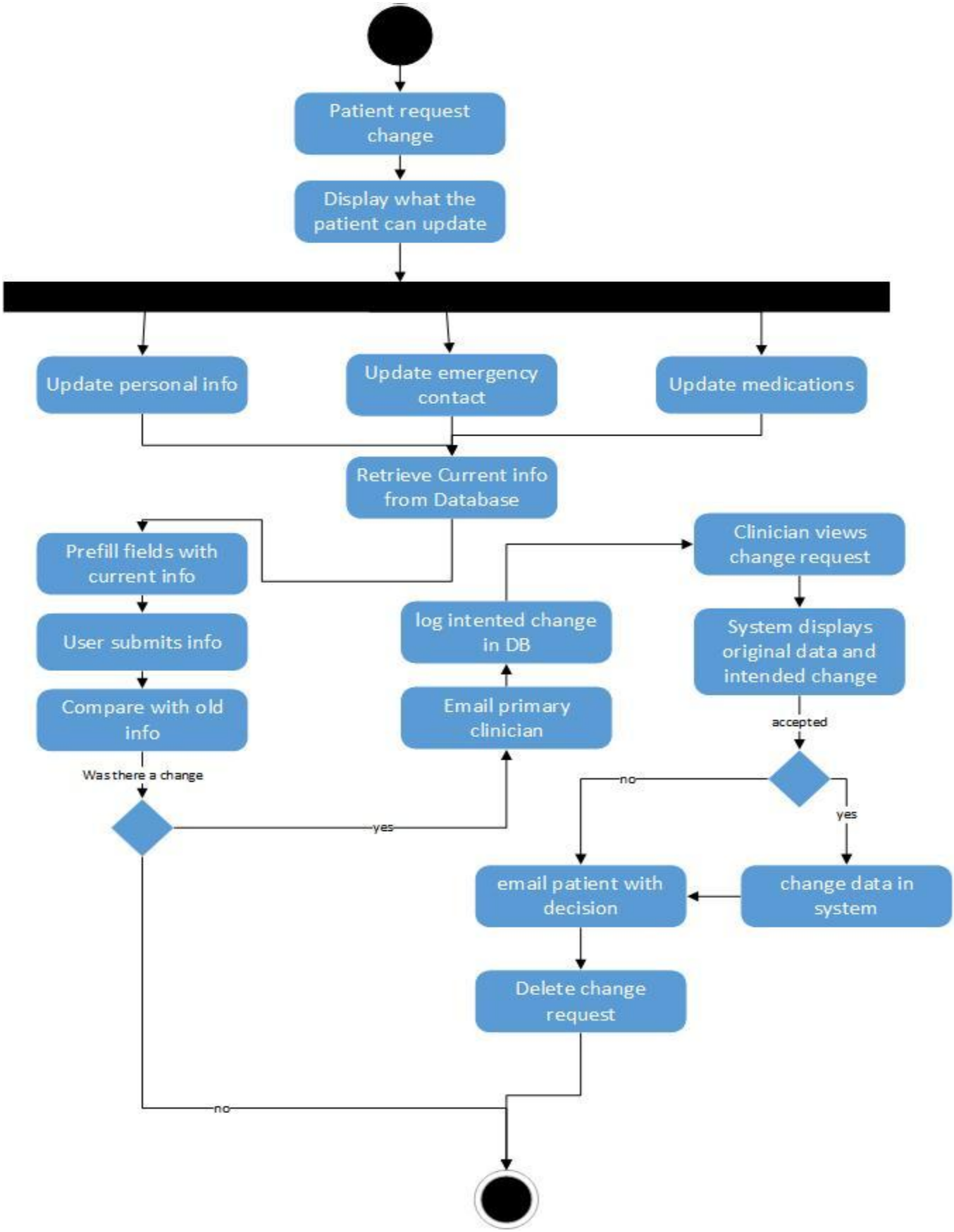
Περίπτωση 1: Ο θάνατος του ασθενή



Περίπτωση 2 : Δημιουργία Συνταγής

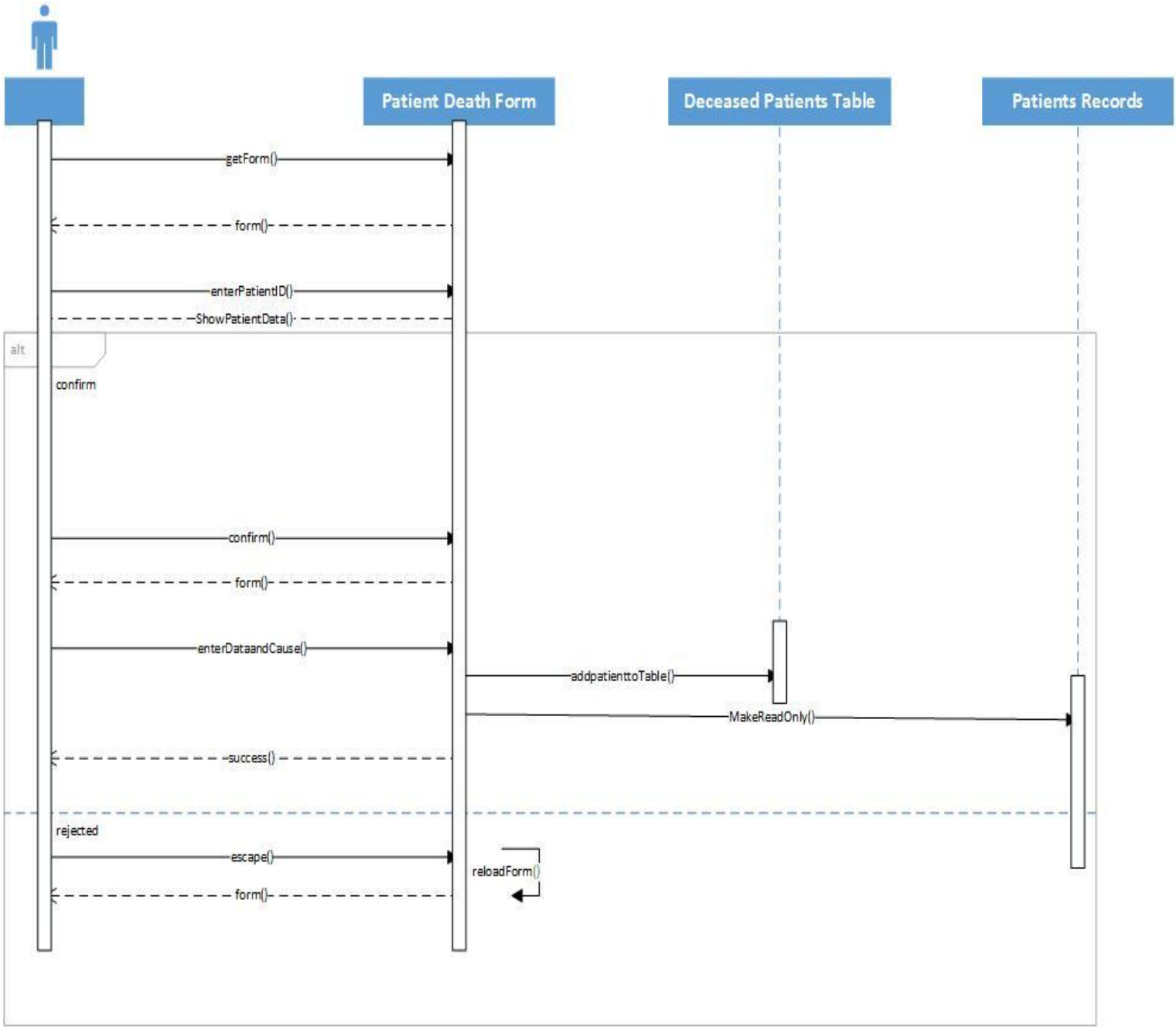


Περίπτωση 3: Αίτηση αλλαγής πληροφοριών ασθενή

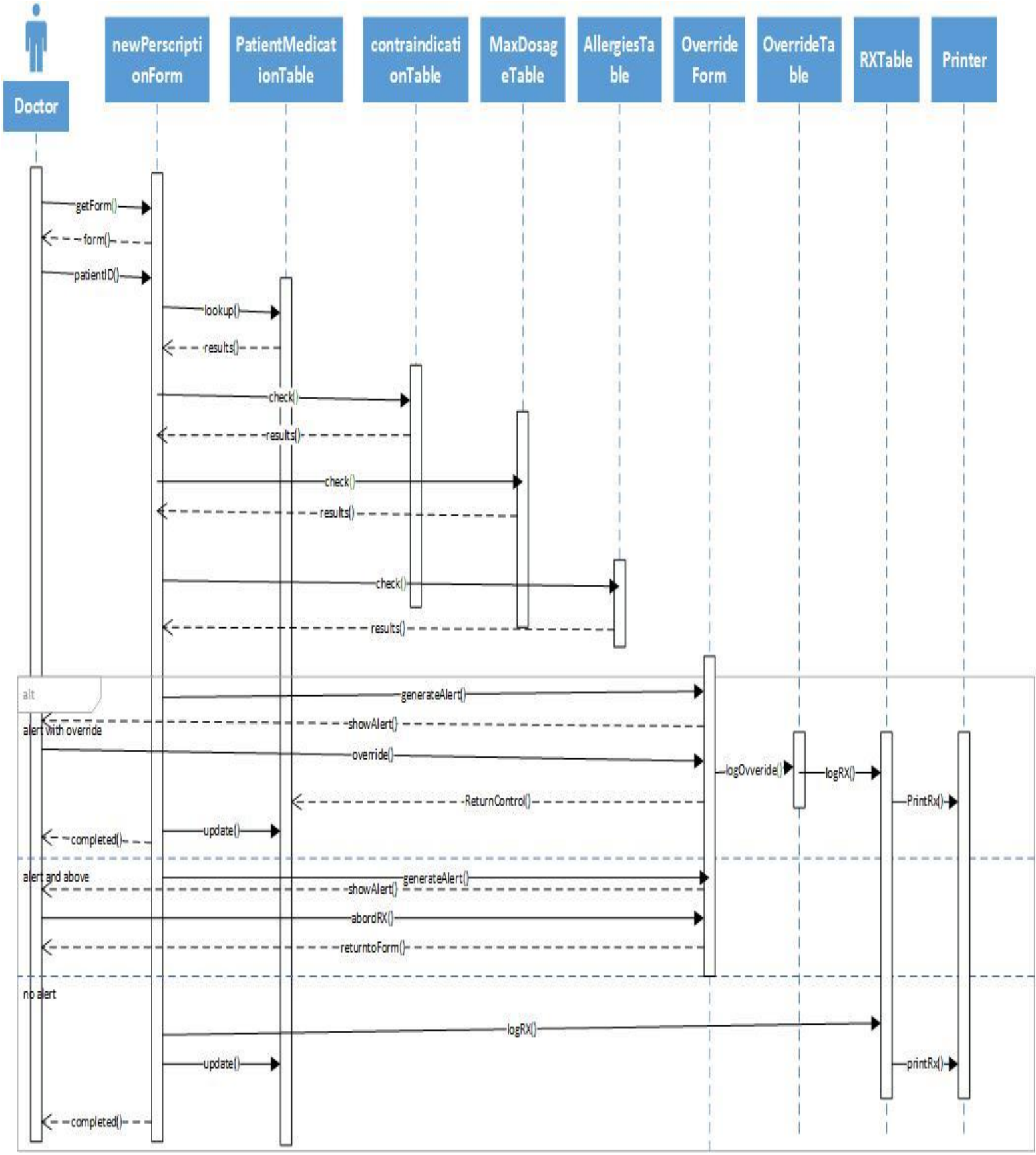


3.4.3 Διαγράμματα ακολουθίας

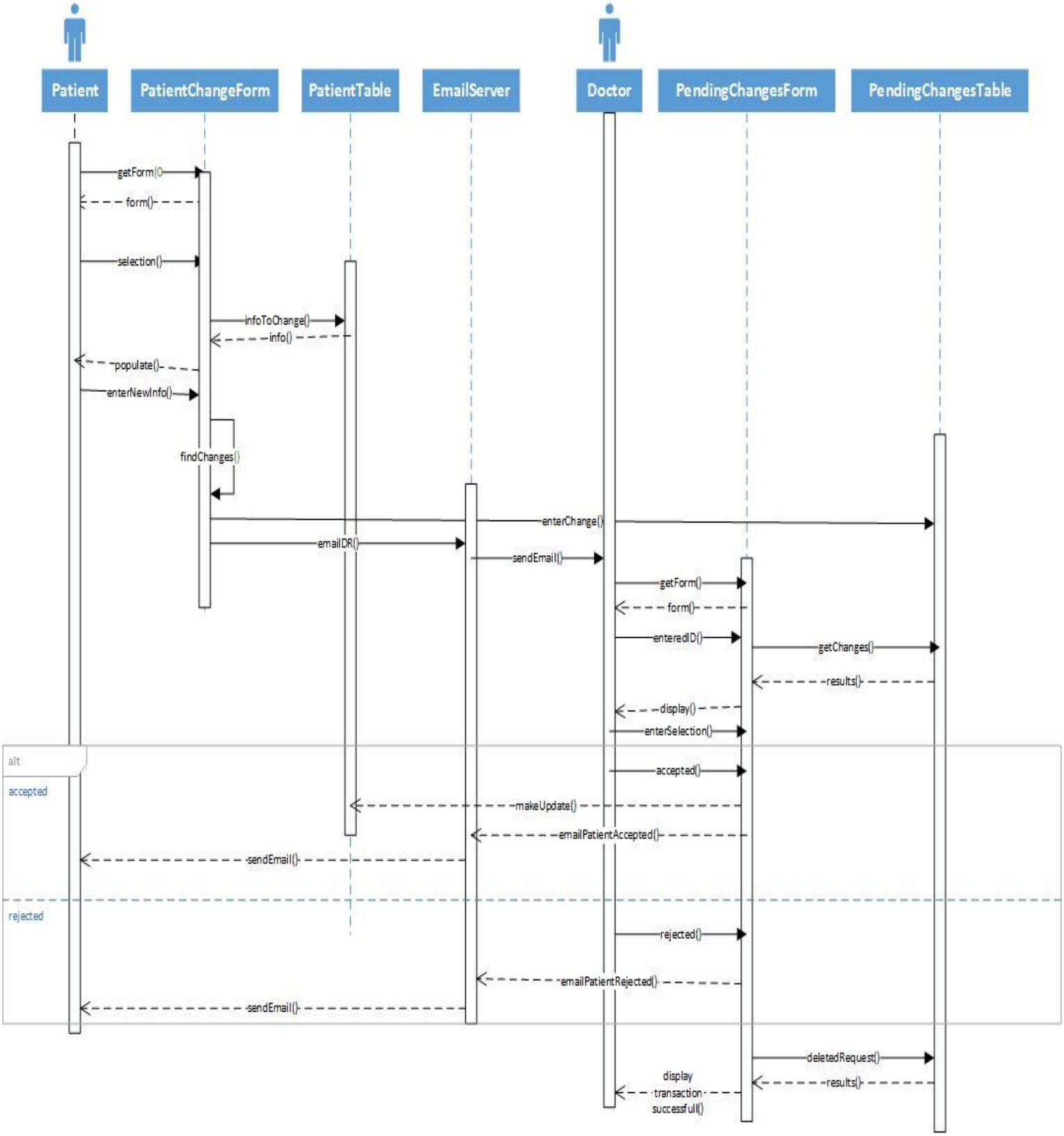
<u>Περίπτωση 1:</u> Ο Θάνατος του ασθενή



<u>Περίπτωση 2:</u> Δημιουργία συνταγής



<u>Περίπτωση 3:</u> Αίτηση αλλαγής πληροφοριών ασθενή



3.5 Υποθέσεις

Για την υλοποίηση του πιο πάνω συστήματος και την δημιουργία των UMLδιαγραμμάτων έχουν γίνει κάποιες υποθέσεις οι οποίες παρουσιάζονται πιο κάτω:

Ο θάνατος του ασθενή

Για την έκδοση της φόρμας επιβεβαίωσης του θανάτου του ασθενή θα πρέπει να επιβεβαιώνουν και να ενημερώνουν την βάση δεδομένων.
Προστέθηκε λειτουργία για την πρόληψη ενημέρωσης του φακέλου του ασθενή αφού ο ασθενής πεθάνει.

Δημιουργία συνταγής

Για την έκδοση της φόρμας για να εισάγουν μια νέα συνταγή, πρέπει να συμπεριλαμβάνουν τον έλεγχο ασφαλείας για τις ενδείξεις σε αλλεργίες και υπερδοσολογία.

Αίτηση αλλαγής πληροφοριών ασθενή

Για την έκδοση της φόρμας εντύπου αίτησης αλλαγής διαθέτει χώρο για να εισάγουν οι ασθενείς τις αλλαγές στα προσωπικά τους στοιχεία, αλλά δεν επικοινωνεί με τη βάση δεδομένων.

Προστέθηκε υποστήριξη για επικοινωνία έκτακτης ανάγκης και αλλαγές φαρμάκων

3.6 Γενική επεξήγηση

Το σύστημα MHCPMS είναι βασισμένο σε ένα πραγματικό σύστημα που είδη χρησιμοποιείται σε κάποια νοσοκομεία. Οι απαιτήσεις που έχει κάποιος από αυτό το σύστημα είναι η ενημέρωση και η εύκολη διαχείριση των ασθενών με ψυχολογικά προβλήματα καθώς ένας άνθρωπος σε αυτή την κατάσταση οφείλει να είναι υπό παρακολούθηση και φροντίδα.

Οι στόχοι του συστήματος αυτού είναι να μπορεί να παρέχει πληροφορίες και να επιτρέπει στους διαχειριστές υπηρεσιών υγείας να μπορούν να αξιολογήσουν την κατάσταση που επικρατεί στην κλινική καθώς επίσης και να δίνει την παροχή στο ιατρικό προσωπικό να μπορεί να παίρνει πληροφορίες από όπου και αν βρίσκεται ο ασθενής. Για παράδειγμα ένας ασθενής δεν θα είναι υποχρεωμένος να προβαίνει στο ίδιο νοσοκομείο πάντα για την θεραπεία του. Ένας ασθενής μπορεί να ξεχάσει σε πιο νοσοκομείο θα έπρεπε να πάει καθώς και το ραντεβού του, όπως και τον ιατρικό του φάκελο. Άρα θα πρέπει να υπάρχουν καταχωρημένες όλες οι πληροφορίες καθώς και ο ιατρικός φάκελος του κάθε ασθενή στο σύστημα έτσι ώστε οι χρήστες του συστήματος στους οποίους ανήκουν (οι γιατροί, οι νοσηλευτές, οι επισκέπτες υγείας) να μπορούν να ανακτήσουν οποιαδήποτε πληροφορία επιθυμούν.

Παρόλα αυτά όμως το σύστημα μας θα πρέπει να είναι ασφαλισμένο με το ιατρικό απόρρητο το οποίο απαιτεί μυστικότητα και εμπιστευτικότητα. Το σύστημα αυτό δεν είναι ένα πλήρες ιατρικό σύστημα αρχείων με τα ιστορικά που αφορούν όλους τους τομής υγείας του ασθενή αλλά αποκλειστικά τον τομέα της ψυχικής υγείας.

Concerns

Οι ανησυχίες τους συστήματος έχουν να κάνουν με τις προϋποθέσεις που χρειάζεται το σύστημα για να λειτουργά σωστά και να ορίζονται τυχών ασάφειες που μπορεί να έχει το σύστημα. Αντιπροσωπεύουν τους στόχους του συστήματος. Οι κύριες ανησυχίες των MHC-PMS είναι η ευχρηστία δηλαδή το σύστημα πρέπει να χρησιμοποιείται από όλους τους χρήστες και να είναι εύκολο στη χρήση, ασφάλεια, μυστικότητα και προστασία της ιδιωτικής ζωής των ασθενών.

ViewPoints

Οι απόψεις του συστήματος έχουν να κάνουν με διάφορες απόψεις που υπάρχουν στο σύστημα που αφορούν τις διάφορες προδιαγραφές. Το σύστημα αυτό διέπετε από τέσσερις βασικές απόψεις το κλινικό προσωπικό το οποίο αλληλεπιδρά με το σύστημα άμεσα, η γραμματέας η οποία καθορίζει τα διάφορα ραντεβού, τα ιατρικά αρχεία καθώς και η υπηρεσία διαχείρισης της υγείας.

Κεφάλαιο 4

Μεθοδολογία που χρησιμοποιήθηκε

4.1 Η γλώσσα προγραμματισμού και η πλατφόρμα που χρησιμοποιήθηκε

Υπάρχουν πολλές και διάφορες γλώσσες προγραμματισμού που μπορεί κάποιος να χρησιμοποιήσει προκειμένου να προγραμματίσει με την τεχνική Aspect Oriented Programming. Αναφέρω κάποιες από αυτές ενδεικτικά:

AspectJ

AspectC++

AspectC

Caesar

AspectWerkz

LogicAJ

XWeaver

Από τις πιο πάνω γλώσσες, η πιο διαδεδομένη και αυτή που χρησιμοποιείται πιο πολύ για την εφαρμογή των μεθόδων του AOP είναι η AspectJ. Η AspectJ ήρθε στο προσκήνιο από τον Grefor Kiczales το 2000 στο Xeroc PARC. Αποτελεί μια γενικού σκοπού aspect-oriented επέκταση της Java και είναι συμβατή με την πλατφόρμα της Java.

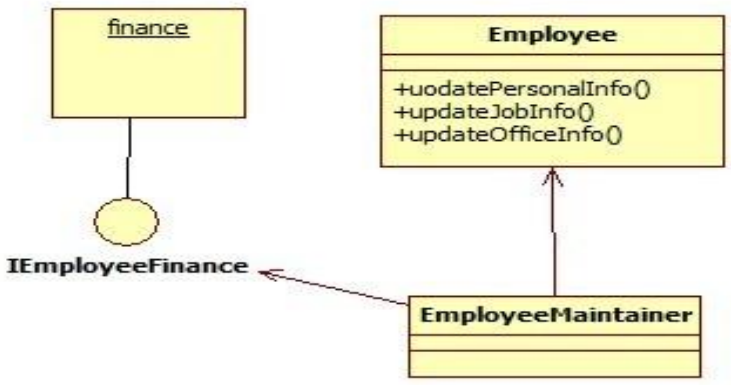
Το κυριότερο πλεονέκτημα της είναι ότι, ακόμα και αν δεν υπάρχει το σχετικό υπόβαθρο, μπορεί εύκολα κάποιος να την μάθει αλλά και να την χρησιμοποιήσει.
Διατίθεται ελεύθερα στο διαδίκτυο στο OpenSource. Ο compiler που χρησιμοποιεί η AspectJ ονομάζεται ajc. Ο ajc συνδυάζει μαζί τον κώδικα java μαζί με την AspectJ και ελέγχει για συντακτικά λάθη. Αυξάνει τον έλεγχο σύνταξης, δέχεται τον bytecode και παράγει αποδοτικό κώδικα. Σκοπός της AspectJ είναι να κάνει την τεχνολογία του Aspect Oriented Programming διαθέσιμη σε ένα ευρύ φάσμα προγραμματιστών για να δημιουργηθεί και να υποστηριχθεί μια κοινότητα χρηστών της AspectJ.

4.2 Προγραμματίζοντας με την AspecJ (παραδείγματα)

Για να εξηγηθεί καλύτερα η γλώσσα προγραμματισμού AspecJ και να γίνουν κατανοητά οι μεθόδους και οι τεχνικές που έχουν περιγραφεί στο κεφάλαιο 2 ας δούμε κάποια παραδείγματα.

Παράδειγμα 1ο

Αρχικά ας μελετήσουμε το παράδειγμα το οποίο έχει να κάνει με ένα σύστημα χρηματοδότησης ενός υπαλλήλου.



Αρχικά ορίζουμε δύο poincuts, ένα για τα joinpoints της κλάσης Employee και ένα για το συστατικό της IEmployeeFinance, ως εξής :

pointcut employeeUpdates(Employee e) :Call(public void Employee.update*Info()) && target(e);

pointcut employeeFinanceUpdates(Employee e): call(public void update*Info(Employee)) && args(e);

Το πρώτο poincut περιγράφει όλα τα joinpoints όπου καλούμε μία μέθοδο ενός αντικειμένου Employee και περιγράφει αυτήν την μέθοδο όπως καθορίζεται στην κλάση Employee μέσω του προσδιοριστή στόχων.
Το δεύτερο poincut περιγράφει όλα τα joinpoints όπου υπάρχει μια κλήση σε οποιαδήποτε μέθοδο που αρχίζει με update, τελειώνει με info και έχει ένα όρισμα τύπου Employee.
Αυτά τα δύο poincuts μαζί περιγράφουν όλα τα joinpoints για τα οποία ενδιαφερόμαστε. Το επόμενο βήμα είναι να γράψουμε τον κώδικα καταγραφής ο οποίος είναι παρόμοιος με οποιαδήποτε άλλη μέθοδο της java, με την διαφορά ότι βρίσκεται σε ένα νέο τύπο που λέγεται aspect. Η aspect είναι ο μηχανισμός που χρησιμοποιούμε για να ενθυλακώσουμε τον κώδικα που σχετίζεται με μια συγκεκριμένη ανησυχία (concern). Η εφαρμογή της aspect για την καταγραφή των αλλαγών των υπαλλήλων έχει ως εξής:

```
public aspect EmployeeChangeLogger {
        pointcut employeeUpdates(Employee e) : call( public void Employee.update*Info())
        && target(e);
        pointcut employeeFinanceUpdates(Employee e) : call( public void
        update*Info(Employee)) && args(e);
        after(Employee e) returning : employeeUpdates(e)|| employeeFinanceUpdates(e) {
                System.out.println("\t>Employee : " +e.getName() + " has had a change ");
                System.out.println("\t>Changed by " + thisJoinPoint.getSignature());
                                                                                }
                                        }
```

Η δομή μιας aspect είναι παρόμοια με αυτήν μίας κλάσης σε java. Μετά από τα poincuts ακολουθεί ένα κομμάτι κώδικα το οποίο είναι παρόμοιο με μία μέθοδο σε κώδικα java. Το κομμάτι αυτό του κώδικα ονομάζεται advice. Υπάρχουν πολλές παραλλαγές που μπορούμε να χρησιμοποιήσουμε για να προσαρμόσουμε τις advice μας. Στο παράδειγμα μας επιλέγουμε να εκτελέσουμε την καταγραφή αμέσως μόλις τελειώσει η λειτουργία ενημέρωσης.
Παρατηρούμε επίσης ότι έχουμε συνδυάσει δύο poincuts μετά την άνω και κάτω τελεία στην αρχή της advice που συνδέονται μεταξύ τους με το λογικό ή. Αυτό μπορούμε να το κάνουμε επειδή και τα δύο έχουν μια παράμετρο Employee. Αμέσως μετά πρέπει να ελέγξουμε τον

κώδικα μας για τυχόν συντακτικά λάθη και να τον ενσωματώσουμε στο υπάρχον σύστημα. Στο παράδειγμα μας εκτός από τις κλάσεις Employee και EmployeeFinance έχουμε επίσης και μια συνάρτηση main:

```
public static void main(String[ ] args) {
        Employee e = new Employee("Chris Smith");
        e.updateJobInfo();
        e.updateOfficeInfo();
        EmployeeFinance.updateFederalTaxInfo(e);
        EmployeeFinance.updateStateTaxInfo(e); }
```

Ο κώδικας αυτός τρέχει μια χαρά χωρίς την εφαρμογή AOP. Όλες οι μέθοδοι στο παράδειγμα μας περιέχουν απλά μια print. Αν τρέξουμε λοιπόν, αυτόν τον κώδικα θα μας δώσει το εξής :

```
Updating job information
Updating office information
Updating federal tax information
Updating state tax information
```

Προκειμένου να ενσωματωθεί η aspect μας στο σύστημα, προσθέτουμε τον κώδικα της aspect και κάνουμε built με τον μεταγλωτιστή της aspectJ τον ajc. Μόλις τρέξουμε τώρα τον κώδικα με τον μεταγλωττιστή της aspectJ, θα πάρουμε την ακόλουθη έξοδο:

```
Updating job information
>Employee : Chris Smith has had a change
>Changed by void employee.Employee.updateJobInfo()
Updating office information
>Employee : Chris Smith has had a change
>Changed by void employee.Employee.updateOfficeInfo()
Updating federal tax information
>Employee : Chris Smith has had a change
>Changed by void
employee.EmployeeFinance.updateFederalTaxInfo(Emplo
yee)
```

Updating state tax information
>Employee : Chris Smith has had a change
>Changed by void
employee.EmployeeFinance.updateStateTaxInfo(Employee)

Παράδειγμα 2ο

Το παράδειγμα αυτό έχει να κάνει με το χρώμα ενός αυτοκινήτου. Θέλουμε να τυπώνουμε το χρώμα του αυτοκινήτου κάθε φορά που θα αλλάζει αλλά με την προϋπόθεση ότι δεν θα πειράζουμε τον κώδικα της κλάσης car. Έχουμε δύο απλές μεθόδους getColor και setColor. Αρχικά το πρώτο πράγμα που έχουμε να κάνουμε είναι να δηλώσουμε μια κλάση car ως εξής:

```
Public class Car {
        private String color;
        Public Stringget Color(){
        return color;
                                }
        public void setColor(String color){
        this.color=color;
                                        }
                }
```

Αυτό που χρειαζόμαστε τώρα είναι να πούμε με κάποιο τρόπο στον μεταγλωττιστή να εκτελεί κάποιον πρόσθετο κώδικα κάθε φορά που καλείται η μέθοδος setColor. Για να γίνει αυτό, ο μεταγλωττιστής θα πρέπει να ξέρει σε ποιο ακριβώς σημείο της εκτέλεσης του προγράμματος πρέπει να παρέμβει για να εισάγει τον πρόσθετο κώδικα. Το σημείο αυτό είναι το pointcut και η σύνταξη του είναι η εξής :

Pointcut setColorCall(): call(public void
Car.setColor(String));

Το pointcut αυτό λέει στον μεταγλωττιστή να δημιουργήσει ένα καινούργιο pointcut με όνομα SetColorPointCut κάθε φορά που καλείται η μέθοδος setColor με μια παράμετρο τύπου String. Στη συνέχεια προσθέτουμε μια advice τύπου before ή οποία λέει στον μεταγλωττιστή να τυπώσει ένα μήνυμα πριν την εκτέλεση του pointcut:

```
before():setColorCall(){
System.out.println("The car color has changed!");
}
```

Όλα τα παραπάνω τα τοποθετούμε μέσα σε μια aspect την οποία ονομάζουμε CarAspect:

```
Public aspect CarAspect
{
        pointcut setColorCall():call(public void
                Car.setColor(String));
        before():setColorCall(){
                System.out.println(”The car color has changed!”);
                                }
}
```

Ονομάζουμε το αρχείο αυτό carAspect.aj που σημαίνει ότι πρόκειται για μια aspect της γλώσσας aspectJ αντί για μια κανονική κλάση της γλώσσας java.
Δημιουργούμε επίσης μια κλάση carDriver η οποία θα καλεί την κλάση car για να διαπιστώνει ότι όλα λειτουργούν κανονικά :

```
Public class CarDriver
{
public static void main(Stringargs[]){
Car mycar=newCar();
mycar.setColor("blue");
}
}
```

Για να εκτελέσουμε το πιο πάνω παράδειγμα θα πρέπει να είμαστε σίγουροι ότι έχουμε εγκαταστήσει σωστά τα εργαλεία της aspecJ και να εκτελέσουμε την εντολή :

ajc CarAspect.aj CarDriver.java Car.java

στην γραμμή εντολών του λειτουργικού συστήματος. Ένα πάμε και εκτελέσουμε java carDriver, τοτε η έξοδος που θα βγάλει το σύστημα πρέπει να είναι η ακόλουθη :

The car color has changed!

4.3 Οδηγίες για την χρήση της aspect στο eclipse

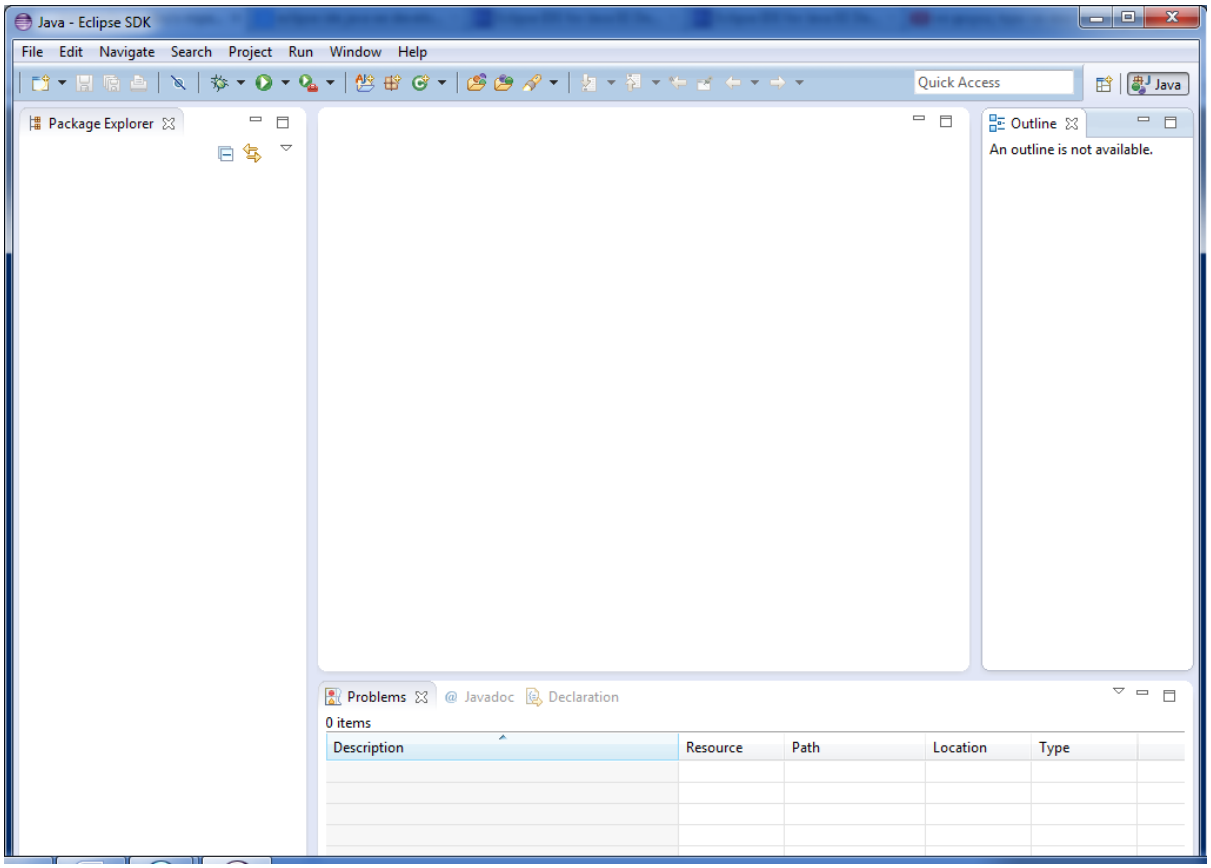
Στην αναζήτηση μου για το πώς μπορώ να χρησιμοποιήσω την aspectJ στο Eclipse δυσκολεύτηκα αρκετά με την εγκατάσταση, για το λόγω αυτό θα σας εξηγήσω τον τρόπο με τον οποίο μπορούμε να ενσωματώσουμε ένα πρόγραμμα γραμμένο σε κώδικα java που επιδεικνύει τον AOP μέσω του Eclipse.

1. **Απόκτηση και εκτέλεση του Eclipse IDE**

Αρχικά πρέπει να κατεβάσουμε το Eclipse από την σελίδα www.eclipse.org. Από την σελίδα αυτή επιλέγουμε να κατεβάσουμε το αρχείο Eclipse IDE for Java EE Developers.

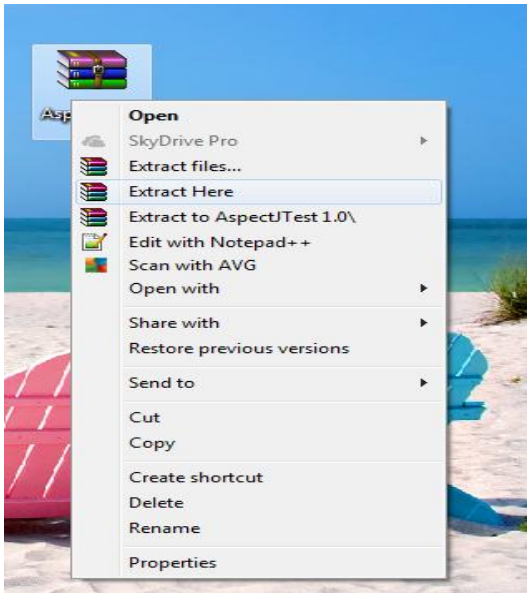
2.Εκτέλεση του Eclipse

Αφού κατεβάσουμε το κατάλληλο πακέτο στην συνέχει ανοίγουμε από τον φάκελο το εκτελέσιμο αρχείο. Στη συνέχεια θα μας εμφανίσει ένα παράθυρο για το που θέλουμε να αποθηκεύσουμε αυτά που θα κάνει το Eclipse. Βασικά να καθορίσουμε το workspace. Μπορούμε να αφήσουμε το καθορισμένο μονοπάτι ή να δώσουμε το μονοπάτι της δικής μας επιλογής. Ακολούθως θα ανοίξει η κεντρική οθόνη του Eclipse



3.Ένα απλό παράδειγμα AOP

Ας προσπαθήσουμε να εκτελέσουμε ένα απλό παράδειγμα AOP με την χρήση της AspectJ. Είναι μια απλή εφαρμογή η οποία περιλαμβάνει δύο aspects. Την AspectJAspect και την AspectJAnnotation. Αρχικά για να το πετύχουμε αυτό θα πρέπει να κάνουμε extract τον κώδικα δείγμα.

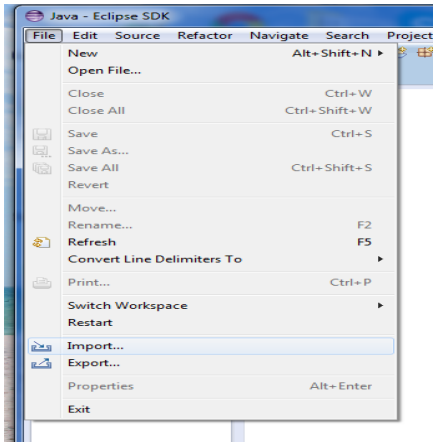


Ακολούθως αντιγράφουμε τον φάκελο με τον κώδικα μέσα στο workspace του Eclipse.

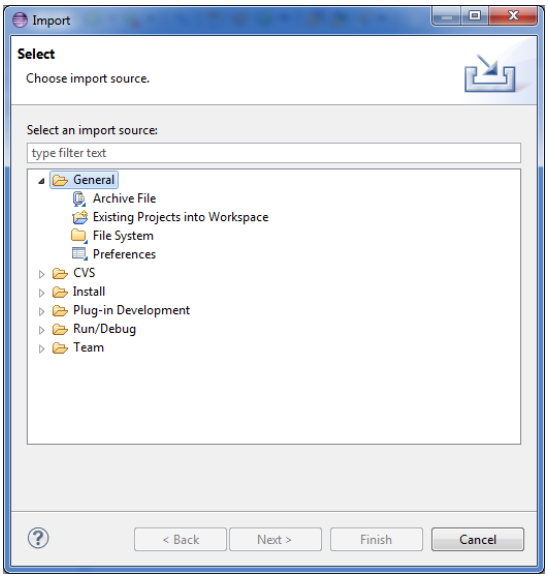
4.Εισαγωγή του κώδικα μέσα στο Eclipse

Από την γραμμή του μενού επιλέγουμε :

file ⟶ import όπως βλέπουμε στην εικόνα 1 ⟶ επιλέξτε general στην εικόνα 2 ⟶ ExistingProjects into Workspace ⟶ next



Εικόνα 1



Εικόνα 2

Στο επόμενο παράθυρο που εμφανίζεται πατάμε το κουμπί Browse για να επιλέξουμε την διαδρομή του φακέλου που βρίσκεται ο κώδικας μας.

5.Εργαλεία ανάπτυξης της AspecJ

Αν το Eclipse δεν έχει το AJDE τότε θα πρέπει να κατεβάσετε το αντίστοιχο plug-in μέσω του Eclipse. Για να το κάνετε αυτό πρέπει να είστε σίγουροι ότι θα εισάγεται στο Eclipse το κατάλληλο link που στη περίπτωση μας είναι το http://www.eclipse.org/ajdt/downloads/. Από το μενού επιλέγουμε το help και ακολούθως το install New Software και στο παράθυρο που θα εμφανιστεί αντιγράψτε το link στο πεδίο work with και πατήστε το κουμπί add, εμφανίζεται ένα καινούργιο παράθυρο στο οποίο επιλέγουμε όλα τα διαθέσιμα Plug-ins, επιλέγουμε finish και περιμένουμε μέχρι να εγκατασταθούν στο eclipse.

4.4 Μεθοδολογία

Για την μεθοδολογία του συστήματος έπρεπε να μελετήσω σαν αρχικό στάδιο τι είναι το Aspect Oriented Programming. Να κατανοήσω τις βασικές έννοιες, και τους όρους στο τομέα αυτό καθώς επίσης και την γλώσσα προγραμματισμού η οποία καλύπτει τον τομέα αυτό. Την γλώσσα προγραμματισμού aspecJava. Επιπλέον μετέπειτα αφού είχα κατανοήσει τις ορολογίες και είχα δει πως προγραμματίζεις με την χρήση της γλώσσας αυτής η οποία δεν διαφέρει και πολύ από τις αντικειμενοστραφείς γλώσσες προγραμματισμού, έπρεπε να μελετήσω το σύστημα MHC-PMS έτσι ώστε να καταλάβω πως λειτουργεί, τι απαιτήσεις έχει και με ποιο τρόπο χρησιμοποιώντας aspects μπορούσε να υλοποιηθεί πιο εύκολα, μειώνοντας κόπο και χρόνο. Μελετώντας τις απαιτήσεις του συστήματος εντοπίζουμε τις κοινές απαιτήσεις οι οποίες θα μπορούσαν να αναπτυχθούν με την χρήση των aspects. Έτσι καταλήγουμε στο συμπέρασμα ότι μπορούμε να χρησιμοποιήσουμε 3 aspect στο σύστημα αυτό:

1) Έλεγχος θανάτου, δεν επιτρέπετε καμία δράση σε ένα ασθενή εάν θεωρηθεί νεκρός
2) Ενημέρωση του ιατρού σε περίπτωση που αλλάξει ένα ιατρικό αρχείο
3) Έλεγχος για την διαχείριση των συστημάτων των χρηστών
   τα οποία και θα περιγράψουμε πιο κάτω.

4.4.1 Έλεγχος θανάτου

Το aspect αυτό εκτελείτε για κάθε λειτουργία που είναι μαρκαρισμένη με το @CheckDeath. Όταν εκτελεστεί αυτή η συνάρτηση παίρνουμε τον ασθενή από την παράμετρο που περάστηκε στη συνάρτηση και εγείρουμε ένα RuntimeException αν ο ασθενής είναι πεθαμένος. Υποθέτουμε ότι η εφαρμογή ελέγχει για τους θανάτους των ασθενών πριν επιτρέψει σε ένα χρήστη να τροποποιήσει τις εγγραφές του. Αυτός είναι και ο λόγος που εγείρουμε το RuntimeException αντί απλά να το επιστρέφουμε αφού θεωρείται προγραμματιστικό λάθος. Αν ο ασθενής δεν είναι πεθαμένος προχωρούμε με το κάλεσμα της μεθόδου.

4.4.2 Ενημέρωση του ιατρού

Οι μέθοδοι που μαρκάρονται με το @informDoctor, στέλνουν email στους ασθενείς του γιατρού μετά από κάθε εκτέλεση της μεθόδου. Ελέγχουμε αν το κάλεσμα της μεθόδου δεν έγειρε exception και προχωρούμε στέλλοντας με email τον ασθενή σαν παράμετρο στην συνάρτηση email Manager. Η παράμετρος ασθενής ανακτάται από τις παραμέτρους που περάστηκαν στην μέθοδο.

4.4.3 Έλεγχος Ρόλου

Το aspect αυτό είναι υπεύθυνο για την εξουσιοδότηση του χρήστη. Θα αρνηθεί την εκτέλεση της μεθόδου αν ο χρήστης δεν έχει τις απαιτούμενες παραμέτρους. Για να μπορεί κάποιος να καλέσει το aspect αυτό πρέπει να μαρκάρουμε την μέθοδο με το @RequiresRole και να περάσουμε το ρόλο σαν string. Όταν η μέθοδος εκτελεστεί το aspect ελέγχει αν ο χρήστης έχει το ρόλο που χρειάζεται και προχωρά αλλιώς βγάζει exception. Ο λόγος που βγάζουμε exception είναι αντί να επιστρέφουμε χωρίς λάθος που θεωρείται προγραμματιστικό λάθος.

4.5 Προβλήματα που αντιμετώπισα

Τα προβλήματα που αντιμετώπισα ήταν αρκετά καθώς θα έπρεπε να μελετήσω κάτι καινούργιο που ακόμα δεν έχει αρκετές πληροφορίες διαθέσιμες και τα βιβλία που υπάρχουν για αυτό το θέμα δεν είναι αρκετά. Επιπλέων αντιμετώπισα το πρόβλημα ότι η πλατφόρμα που χρησιμοποίησα για να προγραμματίσω ήταν το Eclipse και μου πήρε αρκετό χρόνο να καταφέρω να ενσωματώσω ότι χρειαζόταν για να μπορεί να τρέχει με aspects. Τέλος εφόσον δεν είχα ολόκληρο το πηγαίο κώδικα έπρεπε να αναπτύξω και ένα κομμάτι του κώδικα ολόκληρου του συστήματος προκειμένου να μπορώ να υλοποιήσω τα aspects.

4.6Σύγκριση ανάπτυξης συστήματος με τη χρήση aspect και χωρίς

Με την χρήση των aspects καταφέραμε να έχουμε καθαρό αρθρωτό διατηρήσιμο κώδικα. Αν αναπτύσσαμε το σύστημα χωρίς την χρήση των aspects θα έπρεπε να καλούσαμε κάθε μέθοδο που θέλαμε για να γίνεται κάποιος έλεγχος. Ενώ με αυτό τον τρόπο το μόνο πράγμα που έχουμε να κάνουμε είναι να μαρκάρουμε κάθε μέθοδο με την κατάλληλη ένδειξη. Επίσης αν βρεθεί ένα σφάλμα το μόνο πράγμα που πρέπει να κάνουμε είναι να διορθώσουμε το λάθος στο aspect αντί να διασχίζουμε το κώδικα και έτσι και εξοικονομούμε χρόνο και κόστος παρά να ψάχνουμε σε όλο το κώδικα να βρούμε το λάθος.

Κεφάλαιο 5

Συμπεράσματα

5.1 Συμπεράσματα και συνεισφορά

Ο Aspect Oriented Programming είναι μια σχετικά καινούργια τεχνολογία η οποία μέρα με την μέρα γίνεται ολοένα και πιο δημοφιλής και έχει μεγάλες πιθανότητες να γίνει μια τεχνολογία που θα λαμβάνεται σοβαρά υπόψη.
Καθώς η τεχνολογία αυτή ωριμάζει, αυξάνεται και η παροχή τα εργαλεία και η διαθέσιμη βιβλιογραφία προσφέροντας στους ενδιαφερόμενους περισσότερες πληροφορίες και καλύτερη τεχνική υποστήριξη. Επιπλέον συμπεραίνουμε ότι ο σκοπός του Aspect Oriented Programming είναι η αύξηση της διαμόρφωσιμότητας χωρίς να αντικαθιστά τα υπάρχοντα παραδείγματα προγραμματισμού αλλά συνεργαζόμενος μαζί τους για να βελτιώσει την έκφραση και την λειτουργικότητα τους.
Ο Aspect Oriented Programming είναι σχετικά απλός αλλά σε πολλές περιπτώσεις μπορεί να είναι αρκετά πολύπλοκος αλλά παραμένει η πιο απλή από τις προσφερόμενες εναλλακτικές. Η δυσκολία του Aspect Oriented Programming έγκειται στο γεγονός ότι ασχολείται και προσφέρει λύσεις σε πολύπλοκα προβλήματα. Παρόλα αυτά , ακόμα και αν δεν υπάρχει το κατάλληλο γνωστικό υπόβαθρο, με λίγες ώρες μελέτη μπορεί κάποιος να κατανοήσει τον τρόπο με τον οποίο συντάσσονται.
Θα πρέπει να γίνει κατανοητό ότι ο Aspect Oriented Programming δεν είναι μια τεχνική προγραμματισμού που επιλύει όλα τα προβλήματα που μέχρι τώρα παραμένουν άλυτα αλλά μία τεχνική η οποία στοχεύει σε ένα συγκεκριμένο πρόβλημα, την ρύθμιση των cross cutting concerns.
Ο Aspect Oriented Programming μας βοηθάει στην επίλυση δύο σημαντικών προβλημάτων που παρουσιάζονται με τον αντικειμενοστρεφή προγραμματισμό. Του προβλήματα του

διεσπαρμένου και του πλεγμένου κώδικα. Τα προβλήματα του δεύτερου τα οποία λύνει ο Aspect Oriented Programming είναι τα εξής :
Το πρόβλημα του περιττού κώδικα όπου μπορεί να παρουσιάζεται το ίδιο κομμάτι κώδικα σε πολλά σημεία μέσα σε ένα πρόγραμμα.

- Μη- ρητή δομή ενός προγράμματος όπου το μεγαλύτερο μέρος του πλεγμένου κώδικα δεν είναι ξεκάθαρο.
- Δυσκολία αλλαγής του κώδικα, όπου για να αλλαχθεί ο πλεγμένος κώδικας θα πρέπει να εντοπιστούν όλα τα σημεία του προγράμματος που υπάρχουν και όταν βρεθεί θα πρέπει να αλλαχθεί με προσοχή για να μην προκληθεί κάποια ζημία στο πρόγραμμα.

Επιπλέον με βάση την ανάπτυξη του συστήματος MHCPMS με την χρήση Aspect Oriented Programming βλέπουμε ότι είναι πιο εύκολο και γρήγορο σε περίπτωση που χρειαστεί να γίνει μια αλλαγή σε κάποιο σημείο στο κώδικα να γίνει πιο εύκολα εν αντίθεση με την χρήση Object Oriented Programming. Βλέπουμε ότι το σύστημα MHCPMS ικανοποιεί σε βάθος τους στόχους του αφού υπάρχει ασφάλεια της ιδιωτικής ζωής των ασθενών καθώς κάθε πληροφορία για τον ασθενή διέπεται από το ιατρικό απόρρητο στο οποίο κανένας δεν μπορεί να έχει πρόσβαση εκτός από τα εξουσιοδοτημένα άτομα.

5.2 Μελλοντική επέκταση

Μια μελλοντική εργασία θα μπορούσε να είναι η μελέτη και η χρήση aspect και σε άλλα συστήματα που αφορούν και άλλους τομείς στη ζωή μας και όχι μόνο στο τομέα της υγείας. Επιπλέον θα μπορούσε να γίνει μια πλήρης ανάπτυξη του συστήματος MHCPMS με βάση την ανάλυση που έχει γίνει πιο πάνω και να γίνει χρήση των πιο πάνω aspects καθώς εγώ στην παρούσα διπλωματική έχω αναπτύξει τα διάφορα aspects με ένα κομμάτι από τον κυρίως κώδικα για να δω αν λειτουργούν τα aspects. Δεν έχω υλοποιήσει όλο το σύστημα σε βάθος, άρα αυτό θα μπορούσε να αφεθεί για μελλοντική επέκταση. Επιπλέον θα μπορούσε το σύστημα MHCPMS να αναπτυχθεί σε ένα πλήρες σύστημα που να μην περιορίζεται αποκλειστικά στην υποστήριξη της ψυχικής υγείας, αλλά και άλλα πιθανά προβλήματα που μπορεί να έχει ο συγκεκριμένος ασθενής που θα είναι καταχωρημένος στο σύστημα.

Βιβλιογραφία

[1] Ian Sommerville, "Software Engineering " 9th Edition, March 20 chapter21 pp. 555- 589.

[2] Ericj Gamma "Object-Oriented Analysis and Design with Applications", 1994.

[3] Erich Gamma, Richard Helm, Ralph Jonson, John Vlissides, "Design Patterns : Elements of Reusable Object-Oriented Software".

[4] Homepage available at : http://en.wikipedia.org/wiki/Aspect-oriented_programming.

[5] Homepage available at : http://www.eclipse.org/ajdt/downloads/.

[6] Gradecki, J. D. and Lezeiki, N. (2003). Mastering AspectJ: Aspect-Oriented Programming in Java.New York: John Wiley & Sons.

.

[7] Robert E, Filman,Tzilla Elrad, Siobhán Clarke, Mehmet Akşit, "Aspect –Oriented Software Development",6 October 2004.

[8] Ivan Kiselev "Aspect Oriented Programming with AspectJ".

[9] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes,Jean-Marc Loingtier, John Irwin, "Aspect Oriented Programming", June 1997.

[10] Notes from Jignesh Patel "Aspect-Oriented Software Development(AOSD)" .

[11] AJDT: aspectJ Development Tool available at: http://eclipse.org/ajdt/downloads/

[12] Ian Sommerville, "Software Engineering " 9th Edition MHC-PMS Case Study available at:http://www.cs.standrews.ac.uk/~ifs/Books/SE9/CaseStudies/MHCPMS/SupportingDocs/MHCPMSCaseStudy.pdf

[13] Ian Sommerville, "An Integrated Approach to Dependability Requirements Engineering" available at : http://www.cs.st-andrews.ac.uk/~ifs/Books/SE9/CaseStudies/MHCPMS/SupportingDocs/DependabilityReqEngineering.pdf

Παράρτημα Α

Παρουσιάζεται ο κώδικας για το πρόγραμμα MHCPMS

Entities

Patient Entitie

```
package com.maria.aop.entities;

import java.util.List;
import javax.persistence.CascadeType;
import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.FetchType;
import javax.persistence.ManyToOne;
import javax.persistence.OneToMany;
import javax.persistence.SequenceGenerator;
import javax.persistence.Table;
import javax.validation.constraints.NotNull;

/*Database entity for patients*/
@Entity
@Table(name = "patients")
@SequenceGenerator(name="SEQ",sequenceName="PATIENT_ENTITY_SEQUENCE_N
AME")
public class PatientEntity extends UserEntity {
	private static final long serialVersionUID = 3606171968320349988L;
	/*Flag whether patient is dead*/
	private Boolean dead;

	/*Link to the assigned doctor*/
	private DoctorEntity doctor;
```

```
	/*A list of prescriptions*/


private List<PrescriptionEntity> perscriptions;

	@NotNull
	@Column(nullable=false)

	public Boolean getDead() {
		return dead;
	}

	public void setDead(Boolean dead) {
		this.dead = dead;
	}

	@ManyToOne(fetch=FetchType.LAZY,optional=false)
	public DoctorEntity getDoctor() {
		return doctor;
	}

	public void setDoctor(DoctorEntity doctor) {
		this.doctor = doctor;
	}

	@OneToMany(fetch=FetchType.LAZY,mappedBy="patient",cascade=CascadeType.
ALL)
	public List<PrescriptionEntity> getPerscriptions() {
		return perscriptions;
	}

	public void setPerscriptions(List<PrescriptionEntity> perscriptions) {
		this.perscriptions = perscriptions;
	}
```

```
}
```

Οντότητα Γιατρός

```
package com.maria.aop.entities;

import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.SequenceGenerator;
import javax.persistence.Table;

import javax.
validation.constraints.NotNull;

/*Database entity for doctor*/
@Entity
@Table(name = "doctors")
@SequenceGenerator(name="SEQ",sequenceName="DOCTOR_ENTITY_SEQUENCE_N
AME")
public class DoctorEntity {
	private String email;

	@NotNull
	@Column(nullable=false,unique=true)
	public String getEmail() {
		return email;
	}

	public void setEmail(String email) {
		this.email = email;
	}
}
```

User Entity

```
package com.maria.aop.entities;

import java.util.List;
import javax.persistence.CollectionTable;
```

```
import javax.persistence.Column;
import javax.persistence.ElementCollection;
import javax.persistence.EnumType;
import javax.persistence.Enumerated;
import javax.persistence.FetchType;
import javax.persistence.JoinColumn;
import javax.persistence.MappedSuperclass;
import javax.persistence.UniqueConstraint;
import javax.validation.constraints.NotNull;
import com.maria.aop.enums.UserRole;


/*Database entry for uses that are able to login to the system*/
@SuppressWarnings("serial")
@MappedSuperclass
public abstract class UserEntity extends GenericPersonEntity {
	private String username;
	private String password;

	/*A list of roles this user has*/
	private List<UserRole> roles;

	@NotNull
	@Column(nullable=false)
	public String getUsername() {
		return username;
	}
	public void setUsername(String username) {
		this.username = username;
	}

	@NotNull
	@Column(nullable=false)
	public String getPassword() {
		return password;
	}
```

```
	public void setPassword(String password) {
		this.password = password;
	}



	@Enumerated(EnumType.ORDINAL)
	@ElementCollection(fetch = FetchType.LAZY)
	@Column(name = "role")
	@CollectionTable(name = "user_roles", joinColumns = @JoinColumn(name =
"user_id"), uniqueConstraints = @UniqueConstraint(columnNames = { "user_id", "role" }))
	public List<UserRole> getRoles() {
		return roles;
	}

	public void setRoles(List<UserRole> roles) {
		this.roles = roles;
	}



}
```

Doctor Entity

```
package com.maria.aop.entities;

import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.FetchType;
import javax.persistence.ManyToOne;
import javax.persistence.SequenceGenerator;
import javax.persistence.Table;
import javax.validation.constraints.Min;
import javax.validation.constraints.NotNull;

/*Database entry for prescriptions*/
@Entity
@Table(name = "perscriptions")
```

```
@SequenceGenerator(name="SEQ",sequenceName="PERSCRIPTION_ENTITY_SEQUEN
CE_NAME")
public class PrescriptionEntity {
	/*The dose for that prescription*/
	private Long dose;

	/*The patient that will take this prescription*/
	private PatientEntity patient;

	@Min(1)
	@NotNull
	@Column(nullable=false)
	public Long getDose() {
		return dose;
	}

	public void setDose(Long dose) {
		this.dose = dose;
	}

	@ManyToOne(fetch=FetchType.LAZY,optional=false)
	public PatientEntity getPatient() {
		return patient;
	}

	public void setPatient(PatientEntity patient) {
		this.patient = patient;
	}


}
```

Generic Entity

```
package com.maria.aop.entities;

import java.io.Serializable;
import javax.persistence.Column;
```

```
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.MappedSuperclass;

/*Generic entity for all database objects.
 * This entity ads an id  hashcode and equal moethods to all database entities
 */
@SuppressWarnings("serial")
@MappedSuperclass
public abstract class GenericEntity implements Serializable
{
	/*The entities id (Also primary key for table)*/
	private Long id = null;

	@Id
	@GeneratedValue(strategy = GenerationType.SEQUENCE, generator="SEQ")
	@Column(name = "id", updatable = false, nullable = false)
	public Long getId() {
		return this.id;
	}

	protected void setId(final Long id) {
		this.id = id;
	}


	@Override
	public int hashCode() {
		final int prime = 31;
		int result = 1;
		result = prime * result + ((id == null) ? 0 : id.hashCode());
		return result;
	}

	@Override
```

```
	public boolean equals(Object obj) {
		if (this == obj)
			return true;
		if (obj == null)
			return false;
		if (getClass() != obj.getClass())
			return false;
		GenericEntity other = (GenericEntity) obj;
		if (id == null) {
			if (other.id != null)
				return false;
		} else if (!id.equals(other.id))
			return false;
		return true;
	}


}
```

Generic Person Entity

```
package com.maria.aop.entities;

import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.Inheritance;
import javax.persistence.InheritanceType;
import javax.persistence.SequenceGenerator;
import javax.persistence.Table;
import javax.validation.constraints.NotNull;



/*
 * Entity to keep all common information of persons (Doctors,Patients etc)
 */
@SuppressWarnings("serial")
@Entity
@Table(name = "persons")
```

```
@Inheritance(strategy = InheritanceType.JOINED)
@SequenceGenerator(name="SEQ",sequenceName="PERSON_ENTITY_SEQUENCE_NA
ME",allocationSize=0)
public abstract class GenericPersonEntity extends GenericEntity
{
	private String firstName;
	private String lastName;

	public GenericPersonEntity() {
		super();
	}

	@NotNull
	@Column(nullable = false)
	public String getFirstName() {
		return firstName;
	}

	public void setFirstName(String firstName) {
		this.firstName = firstName;
	}

	@NotNull
	@Column(nullable = false)
	public String getLastName() {
		return lastName;
	}

	public void setLastName(String lastName) {
		this.lastName = lastName;
	}
}
```

Aspects

Aspect CheckDeathMethodAdvice

```
package com.maria.aop.aspects;
```

```
import org.apache.tapestry5.plastic.MethodAdvice;
import org.apache.tapestry5.plastic.MethodInvocation;
import com.maria.aop.annotations.CheckDeath;
import com.maria.aop.entities.PatientEntity;

/**
 * The aspect that does the actual death check
 */
public class CheckDeathMethodAdvice implements MethodAdvice {

	private final CheckDeath annotation;

	public CheckDeathMethodAdvice(CheckDeath annotation) {
		this.annotation = annotation;
	}

	public void advise(final MethodInvocation invocation) {
		/*Get the patient entity from the parameters*/
		Final PatientEntity patient = (PatientEntity)invocation.getParameter(annotation.patientParamIndex());
		/*Check if patient is dead and throw an exception.Else proceed with the method invocation*/
		if(patient.getDead()) {
			throw new RuntimeException(String.format("Patient %s %s is dead and it cannot be modified",patient.getFirstName(),patient.getLastName()));
		}
		invocation.proceed();
	}

}
```

Aspect InformDoctorMethodAdvice

```
package com.maria.aop.aspects;

import org.apache.tapestry5.plastic.MethodAdvice;
import org.apache.tapestry5.plastic.MethodInvocation;
```

```
import com.maria.aop.annotations.InformDoctor;
import com.maria.aop.entities.PatientEntity;
import com.maria.aop.managers.EmailsManager;

/**
 * The aspect that does the actual doctor informing
 */
public class InformDoctorMethodAdvice implements MethodAdvice {

	private final InformDoctor annotation;
	private final EmailsManager emailsManager;

	public InformDoctorMethodAdvice(InformDoctor annotation,EmailsManager emailsManager) {
		this.annotation = annotation;
		this.emailsManager = emailsManager;
	}

	public void advise(final MethodInvocation invocation) {
		/*Get the patient entity from the parameters*/
		final PatientEntity patient = (PatientEntity)invocation.getParameter(annotation.patientParamIndex());
		/*Process with the invocation of the method*/
		invocation.proceed();

		/*If the invocation did not throw an exception.Which means everything went ok.Continue
		 * with informing the doctor.*/
		if(!invocation.didThrowCheckedException()) {
			emailsManager.emailDoctor(patient, annotation.emailTemplate());
		}
	}

}
```

Aspect RequiresRoleMethodAdvice

```
package com.maria.aop.aspects;

import org.apache.tapestry5.plastic.MethodAdvice;
import org.apache.tapestry5.plastic.MethodInvocation;

import com.maria.aop.annotations.RequiresRole;
import com.maria.aop.entities.UserEntity;


/**
 * The aspect that does the actual role checking
 */
public class RequiresRoleMethodAdvice implements MethodAdvice {

	private final RequiresRole annotation;

	public RequiresRoleMethodAdvice(RequiresRole annotation) {
		this.annotation = annotation;
	}

	public void advise(MethodInvocation invocation) {
		/*Get the user entity from the parameters*/
		final UserEntity user = (UserEntity)invocation.getParameter(annotation.userParamIndex());

		/*Check if user does not have the role and throw an exception.Else proceed with the method invocation*/
		if(!user.getRoles().contains(annotation.requiredRole())) {
			throw new RuntimeException(String.format("User %s %s does not have role %s",user.getFirstName(),user.getLastName(),annotation.requiredRole().toString()));
		}
		invocation.proceed();
	}
```

```
}
```

Emails Manager

```
package com.maria.aop.managers;

import com.maria.aop.entities.PatientEntity;

/**
 * General interface to work with emails
 */
public interface EmailsManager {
	/**
	 * Function to inform a doctor via email
	 *
	 * @param patient The patient entity to get the doctor from
	 * @param emailTemplate The email template to use
	 */
	void emailDoctor(PatientEntity patient,String emailTemplate);
}
```

UserRole

```
package com.maria.aop.enums;

/*Enumeration that contains all the roles a UserEntity can have*/
public enum UserRole {
	ADMIN,
	ADD_PERSCRIPTIONS,
	EDIT_PATIENT
}
```

PatientsDAO

```
package com.maria.aop.dao;
import com.maria.aop.annotations.CheckDeath;
import com.maria.aop.annotations.InformDoctor;
import com.maria.aop.annotations.RequiresRole;
import com.maria.aop.entities.DoctorEntity;
import com.maria.aop.entities.PatientEntity;
import com.maria.aop.enums.UserRole;
/**
```

```
 * General interface to work with patients.
 */
public interface PatientsDAO {

	/**


	 * Function to update patient profile.
	 * <p/>
	 * Method is advised in order to check if a patient is dead and don't allow modifications
	 * and also check that the doctor that performed the change has the correct role
	 * to modify the profile
	 *
	 * @param patient The patient entity with the changes profile
	 * @param doctor The doctor that did change the profile
	 */
	@CheckDeath
	@RequiresRole(requiredRole=UserRole.EDIT_PATIENT,userParamIndex=1)
	void updatePatient(PatientEntity patient,DoctorEntity doctor);


	/**
	 * Function that lets patients to update their profile.
	 * <p/>
	 * The function does not save the changes to the database directly but
	 * temporary saves them and informs the managing doctor in order to approve and proceed with
	 * the database update
	 * <p/>
	 * Method is advised in order to check if a patient is dead and don't allow modifications
	 * and also inform the doctor of the changed profile
	 *
	 * @param patient The patient entity with the changes profile
	 */
```

```
        @CheckDeath
        @InformDoctor
        void updatePatient(PatientEntity patient);
}
```

PrescriptionsDAO

```
package com.maria.aop.dao;

import com.maria.aop.annotations.RequiresRole;
import com.maria.aop.entities.DoctorEntity;
import com.maria.aop.entities.PrescriptionEntity;
import com.maria.aop.enums.UserRole;

/**
 * Generic interface to work with prescriptions
 */
public interface PrescriptionsDAO {

        /**
         * Function to add a prescription to a specific patient by a specific doctor
         * <p/>
         * Method is advised in order to check that the doctor that performed the change
         * has the correct role to modify the profile
         *
         * @param perscription The prescription entity of the patient
         * @param doctor The doctor that created the prescription
         */
        @RequiresRole(requiredRole=UserRole.ADD_PERSCRIPTIONS,userParamIndex=2
)
        void addPerscription(PrescriptionEntity perscription,DoctorEntity doctor);
}
```

CheckDeath

```
package com.maria.aop.annotations;
```

```
import java.lang.annotation.Documented;
import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

/**
 * A marker annotation that is added on methods that we want to check of patient is dead
 */
@Documented
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
public @interface CheckDeath {
	/**
	 * Parameter index of the patient entity
	 */
	int patientParamIndex() default 0;
}
```

InformDoctor

```
package com.maria.aop.annotations;

import java.lang.annotation.Documented;
import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

/**
 * A marker annotation that is added on methods that we want to inform the assigned patients
doctor
 */
@Documented
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
public @interface InformDoctor {
	/**
```

```
         * Parameter index of the patient entity
         */
        int patientParamIndex() default 0;



        /**
         * Template to use for email.We might have different templates.
         * For example on for informing the doctors for changes
         * and another for prescriptions etc
         */
        String emailTemplate() default "defaultTemplate.ftl";
}
```

RequiresRole

```
package com.maria.aop.annotations;

import java.lang.annotation.Documented;
import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

import com.maria.aop.enums.UserRole;



/**
 * A marker annotation that is added on methods that we want to check if user has a specific permission
 */
@Documented
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
public @interface RequiresRole {

        /**
```

```
        * Parameter index of the user entity
        */
        int userParamIndex() default 0;


        /**
        * The actual role we want to check that user entity has
        */
        UserRole requiredRole() default UserRole.ADMIN;

}
```

Main

```
package com.maria.aop.main;
import java.lang.reflect.Method;
import org.apache.tapestry5.ioc.MethodAdviceReceiver;
import org.apache.tapestry5.ioc.annotations.Advise;
import org.apache.tapestry5.ioc.annotations.Order;
import com.maria.aop.annotations.CheckDeath;
import com.maria.aop.annotations.InformDoctor;
import com.maria.aop.annotations.RequiresRole;
import com.maria.aop.aspects.CheckDeathMethodAdvice;
import com.maria.aop.aspects.InformDoctorMethodAdvice;
import com.maria.aop.aspects.RequiresRoleMethodAdvice;
import com.maria.aop.managers.EmailsManager;


/*This class contains static method for applying the aspects*/
public final class Startup {

        /*Apply the CheckDeath aspect*/
        @Advise
        public static void adviceCheckDeath(MethodAdviceReceiver receiver) {
                for (final Method m : receiver.getInterface().getMethods()) {
                        final CheckDeath annotation = m.getAnnotation(CheckDeath.class);
                        if(annotation != null) {
```

```
                receiver.adviseMethod(m, new CheckDeathMethodAdvice(annotation));
            }
        }


}

    /*Apply the Inform Doctor aspect*/
    @Advise
    public static void adviceInformDoctor(MethodAdviceReceiver receiver,final EmailsManager emailsManager) {
        for (final Method m : receiver.getInterface().getMethods()) {
            final InformDoctor annotation = m.getAnnotation(InformDoctor.class);
            if(annotation != null) {
                receiver.adviseMethod(m, new InformDoctorMethodAdvice(annotation,emailsManager));
            }
        }
    }
    /*Apply the RequiresRole.This aspect should come always before all other aspects*/
    @Advise
    @Order("before:*")
    public static void adviceRequiresRolesDoctor(MethodAdviceReceiver receiver) {
        for (final Method m : receiver.getInterface().getMethods()) {
            final RequiresRole annotation = m.getAnnotation(RequiresRole.class);
            if(annotation != null) {
                receiver.adviseMethod(m,new RequiresRoleMethodAdvice(annotation));
            }
        }
    }


}
```