ΠΑΝΕΠΙΣΤΗΜΙΟ ΠΕΙΡΑΙΩΣ

UNIVERSITY OF PIRAEUS

Μοντελοποίηση και υλοποίηση εφαρμογών με χρήση προγραμματιστικών προτύπων

Τμήμα Ψηφιακών Συστημάτων

Δικτυοκεντρικά Πληροφοριακά Συστήματα

Βένδρας Παναγιώτης

Πρόλογος

Αντικείμενο της παρούσας εργασίας είναι η διεπαφή του χρήστη με τα σχεδιαστικά πρότυπα και κυρίως η ανάπτυξη εφαρμογών με χρήση Java. Γίνεται αναλυτική παρουσίαση της τεχνολογίας των βασικών σχεδιαστικών προτύπων και της αρχιτεκτονικής Servlet. Η κατανόηση αυτών των δυο είναι σημαντική για όσους ενδιαφέρονται για τον προγραμματισμό γενικά αλλά και ειδικά για τον διαδικτυακό προγραμματισμό. Ακόμα σκοπός της εργασίας αυτής είναι να λύσει σε ένα αρκετό μεγάλο βαθμό τα ιατρικά προβλήματα που μπορεί να έχει ένας γιατρός στην παρακολούθηση της πορείας της υγείας του ασθενή του. Παρουσιάζει επίσης τρόπου σωστής σύνταξης των δομικών στοιχείων ενός προγράμματος και πως μπορεί ο προγραμματιστής να επιλύσει σωστά τα κοινά προβλήματα που παρουσιάζονται στην διαδικασία ανάπτυξης μια εφαρμογής.

Περιεχομενα

ΚΕΦΑΛΑΙΟ 1 – ΑΝΤΙΚΕΙΜΕΝΟΣΤΡΑΦΕΙΣ ΓΛΩΣΣΕΣ ΠΡΟΓΡΑΜΜΑΤΙΣΜΟΥ.

1.1 Βασικές έννοιες

1.1.1 Γενικά Στοιχεία

Τα τελευταία 25 χρόνια έχει αναπτυχθεί πολύ μια προσέγγιση στην Τεχνολογία Λογισμικού που βασίζεται στο μοντέλο των αντικειμένων. Αρχικά εμφανίστηκαν οι αντικειμενοστραφείς γλώσσες προγραμματισμού όπως η Simula [23] και η Smalltalk [24] και αργότερα η ιδέα των αντικειμένων ωρίμασε περισσότερο ώστε να δημιουργηθούν τεχνικές Αντικειμενοστραφούς Ανάλυσης και Σχεδιασμού. Τώρα πλέον θεωρείται σχεδόν βέβαιο ότι η Αντικειμενοστραφής Τεχνολογία Λογισμικού θα επικρατήσει και τα επόμενα χρόνια αναμένεται ότι θα εξελιχθεί περισσότερο.

Η Αντικειμενοστραφής Τεχνολογία Λογισμικού βασίζεται στην ιδέα των αντικειμένων με τα οποία μπορούμε να περιγράψουμε τον κόσμο και επομένως τον εκάστοτε χώρο προβλήματος που αντιμετωπίζει ένας μηχανικός λογισμικού.

Ο μεγαλύτερος στόχος της Αντικειμενοστραφούς Τεχνολογίας Λογισμικού είναι να βελτιώσει την παραγωγικότητα και να αυξήσει την επαναχρησιμοποίηση του λογισμικού, με συνέπεια το χαμηλότερο κόστος ανάπτυξης και συντήρησης.

Η επαναχρησιμοποίηση του λογισμικού επιτυγχάνεται με τον ορισμό αντικειμένων που μπορούν να χρησιμοποιηθούν κατά τρόπο όπως κάποιος οδηγός αυτοκινήτου μπορεί να κάνει χρήση των λειτουργιών ενός αυτοκινήτου (να στρίβει το τιμόνι , να πατάει φρένο, να επιταχύνει , κ.λ.π) χωρίς να γνωρίζει πως δουλεύει η μηχανή του αυτοκινήτου.

Στην περίπτωση του παραδείγματος , ένας μηχανικός λογισμικού ορίζει αντικείμενα όπως η μηχανή του αυτοκινήτου και μετά τα χρησιμοποιεί χωρίς να εστιάζει στον τρόπο λειτουργίας τους. Τα αντικείμενα μπορεί να χρησιμοποιούνται στο ίδιο πρόγραμμα αλλά και σε άλλα προγράμματα, κατά τον ίδιο τρόπο που η μηχανή ενός αυτοκινήτου μπορεί να χρησιμοποιηθεί από πολλούς τύπους αυτοκινήτων.

Ο βασικότερος τρόπος δόμησης ενός χώρου προβλήματος είναι η κατηγοριοποίηση των εννοιών σε αντικείμενα τα οποία ανήκουν σε τάξεις αντικειμένων, οι οποίες με τη σειρά τους μπορεί να ανήκουν σε κάποιες άλλες τάξεις κ.λ.π. Η διαφορά μεταξύ μιας τάξης και ενός αντικειμένου δεν είναι πάντοτε σαφής. Η τάξη αντιπροσωπεύει μία γενίκευση , ενώ ένα αντικείμενο είναι συγκεκριμένο. Ένας σαφής διαχωρισμός τάξεων και αντικειμένων μπορεί να επιτευχθεί αν όλα τα δείγματα ή στιγμιότυπα (instances) των τάξεων θεωρούνται αντικείμενα. Τα αντικείμενα κληρονομούν χαρακτηριστικά και λειτουργίες από την τάξη στην οποία ανήκουν . Η έννοια της κληρονομικότητας είναι πολύ σημαντική για την αντικειμενοστραφή τεχνολογία.

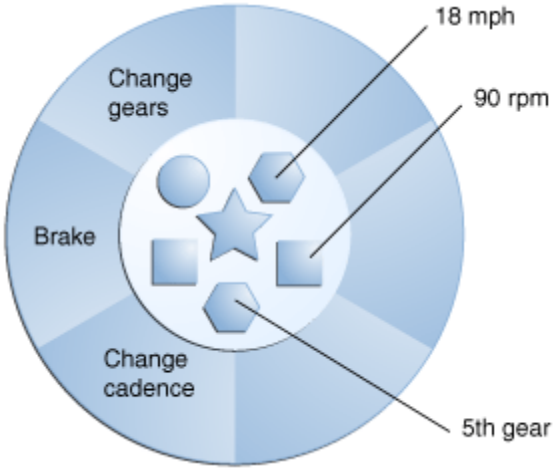
1.1.2 Βασικά στοιχεία της αντικειμενοστραφής προσέγγισης

Εκτός από τις ιεραρχίες και την κληρονομικότητα, η αντικειμενοστραφής τεχνολογία λογισμικού είναι συνδεδεμένη άμεσα με ορισμένα χαρακτηριστικά λογισμικού όπως είναι η απόκρυψη πληροφοριών και η ενθυλάκωση. Σύμφωνα με τον Parnas(1971) [25], η αρχή της απόκρυψης πληροφοριών εφαρμόζεται στο σχεδιασμό αν κάθε τμήμα κρύβει τις λεπτομέρειες υλοποίησης του από άλλα τμήματα. Στην περίπτωση της αντικειμενοστραφούς προσέγγισης η απόκρυψη πληροφοριών εφαρμόζεται στις τάξεις και τα αντικείμενα. Η κάθε τάξη ή αντικείμενο ενθυλακώνει τα στοιχεία υλοποίησης.

Το πλεονέκτημα της ενθυλάκωσης είναι ότι στην πράξη , όταν τα αντικείμενα ανταλλάσσουν υπηρεσίες παίζοντας τους ρόλους πελάτη –εξυπηρετητή(client –server), ο πελάτης δεν χρειάζεται να γνωρίζει τις εσωτερικές δομές του εξυπηρετητή , επομένως αυτές μπορούν να αλλάξουν χωρίς να χρειάζεται να γίνουν αλλαγές και στον κάθε πελάτη. Εδώ εννοείται ότι εξυπηρετητής είναι ένα αντικείμενο που παρέχει υπηρεσίες σε άλλα αντικείμενα , ενώ πελάτης είναι ένα αντικείμενο που χρησιμοποιεί τις υπηρεσίες του εξυπηρετητή.

Η ενθυλάκωση επιτυγχάνεται σε γενικές γραμμές με τη δόμηση των τάξεων σε δύο μέρη , την υλοποίηση (implementation) και τη διεπαφή (interface). Οι συναλλαγές των τάξεων πραγματοποιούνται μέσω των διεπαφών του, ενώ η υλοποίηση παραμένει συνήθως

κρυμμένη , όπως φαίνεται στο



Σχήμα 1.1 Ενθυλάκωση : Διεπαφή και υλοποίηση

Κατά τον Cox [21], η ενθυλάκωση είναι η βάση της αντικειμενοστραφούς προσέγγισης δίνοντας έμφαση στην πακετοποίηση αντί στη γραφή του κώδικα, ενώ η κληρονομικότητα χτίζεται πάνω στην ενθυλάκωση για να πραγματοποιεί την επαναχρησιμοποίηση του κώδικα.

Στα βασικά στοιχεία της αντικειμενοστραφούς προσέγγισης είναι και η αφαίρεση (abstraction) με την έννοια ότι κατά την ανάπτυξη λογισμικού η προσοχή εστιάζεται στο τι είναι ένα αντικείμενο και πως συμπεριφέρεται αντί για το πώς υλοποιείται. Η χρήση της αφαίρεσης κατά τη διάρκεια της ανάλυσης μετατοπίζει τη λήψη αποφάσεων υλοποίησης για αργότερα, όταν το πρόβλημα θα έχει κατανοηθεί καλύτερα από τους μηχανικούς λογισμικού.

Στην αντικειμενοστραφή προσέγγιση , ο μηχανικός λογισμικού πρέπει πρώτα να βρει τα αντικείμενα που συνθέτουν τον χώρο του προβλήματος και μετά να χρίσει κάποιες διαδικασίες γύρω από αυτά, σε αντίθεση με την πιο παραδοσιακή λειτουργική (functional) προσέγγιση (DeMarco 79) όπου ο χώρος του προβλήματος αναλύεται στις κύριες λειτουργίες που τον συνθέτουν.

1.1.3 Αντικειμενοστραφής γλώσσες προγραμματισμού

Ήδη από τα τέλη της δεκαετίας του 1960 αναπτύχθηκε η πρώτη αντικειμενοστραφής γλώσσα από τους Dahl, Myhrhang και Nygard, η Simula 67, η οποία βασιζόταν στην Algol 60 , αλλά περιείχε επίσης τις έννοιες της ενθυλάκωσης και της κληρονομικότητας. Η Simula ήταν η βασική επιρροή για την ανάπτυξη της Smalltalk που έγινε στο XeroxPaloAltoResearchCenter στην αρχή της δεκαετίας του 70 κατά την ίδια εποχή που έγινε και η Pascal.

Η Smalltalk θεωρείται καθαρά αντικειμενοστραφής γλώσσα διότι όλα , ακόμα και οι τύποι δεδομένων όπως οι ακέραιοι , αντιμετωπίζονται σαν τάξεις.HSmalltalk αναπτύχθηκε από τον Άλαν Κέι της εταιρείας Xerox στο πλαίσιο μίας εργασίας με στόχο τη δημιουργία ενός χρήσιμου, αλλά και εύχρηστου, προσωπικού υπολογιστή. Όταν η τελική έκδοση της Smalltalk έγινε διαθέσιμη το 1980 η έρευνα για την αντικατάσταση του δομημένου προγραμματισμού με ένα πιο σύγχρονο υπόδειγμα ήταν ήδη εν εξελίξει.

Την ίδια περίπου εποχή, και επίσης με επιρροές από τη Simula, ολοκληρωνόταν η ανάπτυξη της C++ ως μίας ισχυρής επέκτασης της δημοφιλούς γλώσσας προγραμματισμού C στην οποία είχαν "μεταμοσχευθεί" αντικειμενοστραφή χαρακτηριστικά. Η επιρροή της C++ καθ' όλη της δεκαετία του '80 ήταν καταλυτική με αποτέλεσμα τη σταδιακή κυκλοφορία αντικειμενοστραφών εκδόσεων πολλών γνωστών διαδικαστικών γλωσσών προγραμματισμού. Κατά το πρώτο ήμισυ της δεκαετίας του '90 η βαθμιαία καθιέρωση στους μικροϋπολογιστές των γραφικών διασυνδέσεων χρήστη (GUI), για την ανάπτυξη των οποίων ο ΑΠ φαινόταν ιδιαιτέρως κατάλληλος, και η επίδραση της C++ οδήγησαν στην επικράτηση της αντικειμενοστρέφειας ως βασικού προγραμματιστικού υποδείγματος.

Το 1995 η εμφάνιση της Java, μίας ιδιαίτερα επιτυχημένης, πλήρως αντικειμενοστρεφούς γλώσσας που έμοιαζε συντακτικώς με τη C/C++ και προσέφερε πρωτοποριακές για την εποχή δυνατότητες, έδωσε νέα ώθηση στον αντικειμενοστραφή προγραμματισμό. Παράλληλα εμφανίστηκαν ποικίλες άτυπες βελτιώσεις στο βασικό

προγραμματιστικό υπόδειγμα, όπως οι αντικειμενοστρεφείς γλώσσες μοντελοποίησης λογισμικού, τα σχεδιαστικά πρότυπα κλπ. Το 2001 η Microsoft εστίασε την προσοχή της στην πλατφόρμα .NET, μία ανταγωνιστική της Java πλατφόρμα ανάπτυξης και εκτέλεσης λογισμικού η οποία ήταν εξολοκλήρου προσανατολισμένη στην αντικειμενοστρέφεια.

Η αντικειμενοστραφής προσέγγιση έχει επιρροές από την τεχνητή Νοημοσύνη και ειδικά από την Αναπαράσταση της Γνώσης (Knowledge Representation) με τα λεγόμενα σημασιολογικά δίκτυα (semantic nets) και τα πλαίσια (frames).

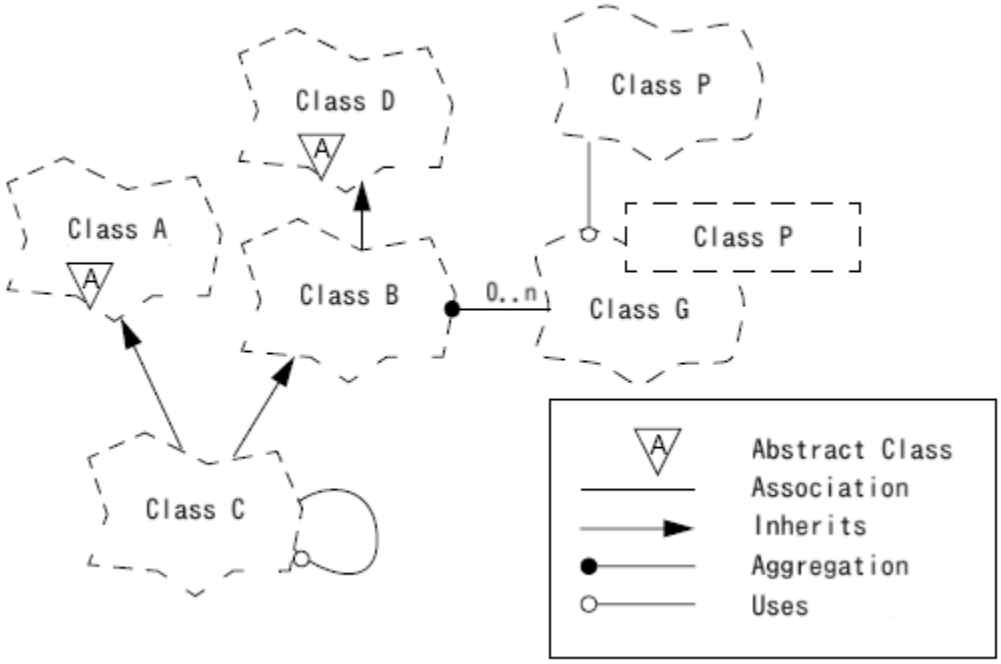
Η ιδέα των σημασιολογικών δικτύων είχε ξεκινήσει από τα τέλη της δεκαετίας του 60 και ήταν ένας τρόπος ν αναπαρασταθεί η γνώση σαν ένα σύνολο κόμβων που συμβολίζουν έννοιες και συνδέονται μεταξύ τους με βέλη.

1.2 Έννοιες αντικειμενοστραφούς προγραμματισμού

1.2.1 Αντικείμενα και τάξεις

Αντικείμενα:είναι μια έννοια, ή αφαίρεση ή πράγμα με σαφή όρια και σημασία στο χώρο του προβλήματος. Το αντικείμενο είναι μοναδικό, δηλαδή ξεχωρίζει από κάθε άλλο αντικείμενο και έχει ταυτότητα.

Τάξη αντικειμένων: είναι μια αφαίρεση που περιγράφει ένα σύνολο αντικειμένων με παρόμοιες ιδιότητες , κοινή συμπεριφορά , κοινές σχέσεις με άλλες τάξεις και κοινή σημασιολογία.



Σχήμα 1.2: Τρόπος σύνδεσης των κλάσεων

Σύμφωνα με τον Booch [26] , τα αντικείμενα καθορίζονται από τρία πράγματα :

1. Ταυτότητα (identity)
2. Κατάσταση (state)
3. Συμπεριφορά (behaviour)

Ταυτότητα (identity): είναι ο μοναδικός κωδικός που ξεχωρίζει ένα αντικείμενο από κάθε άλλο. Μερικά μέσα υλοποίησης, όπως οι Βάσεις Δεδομένων χρειάζονται τη δήλωση ενός μοναδικού κωδικού που προσδιορίζει ένα αντικείμενο , αλλά γενικά σε ένα μοντέλο αντικειμένων δεν χρειάζεται να αναφέρεται συγκεκριμένο αναγνωριστικό, παρόλο που υπονοείται η ύπαρξη του. Στις αντικειμενοστραφείς γλώσσες προγραμματισμού γίνεται αυτόματη παραγωγή αναγνωριστικών που προσδιορίζουν μοναδικά το κάθε αντικείμενο χωρίς να χρειάζεται ο προγραμματιστής να τα δηλώσει.

Γενικώς η εσωτερική ταυτότητα ενός αντικειμένου δεν πρέπει να συγχέεται με τα χαρακτηριστικά του αντικειμένου που ανταποκρίνονται στον πραγματικό κόσμο. Η εσωτερική ταυτότητα είναι ένα θέμα υλοποίησης που δεν έχει νόημα στο πεδίο του προβλήματος.

Κατάσταση: ενός αντικειμένου αποτελείται από τις ιδιότητες του αντικειμένου και τις τιμές τους. Με άλλα λόγια είναι η αναπαράσταση των πληροφοριών γύρω από το αντικείμενο. Η κατάσταση ενός αντικειμένου αλλάζει όταν αλλάζουν οι τιμές κάποιες χαρακτηριστικών. Για παράδειγμα, αν έχουμε το αντικείμενο «δωμάτιο», το οποίο έχει ένα χαρακτηριστικό που λέγεται «χρώμα» με τρέχουσα τιμή «άσπρο» και αυτή η τιμή αλλάξει σε «γαλάζιο» τότε θεωρείται ότι η κατάσταση του δωματίου έχει αλλάξει. Οι αλλαγές στην κατάσταση των αντικειμένων συνήθως γίνονται με την επίδραση μιας λειτουργίας . για παράδειγμα, το δωμάτιο άλλαξε χρώμα , από άσπρο σε γαλάζιο , διότι βάφτηκε. Δύο ή περισσότερα

αντικείμενα τα οποία έχουν ακριβώς τα ίδια χαρακτηριστικά με τις ίδιες τιμές δεν σημαίνει ότι είναι το ίδιο αντικείμενο.

Συμπεριφορά: ενός αντικειμένου αντιστοιχεί στις υπηρεσίες που παρέχει το αντικείμενο σους λεγόμενους πελάτες (clients). Η συμπεριφορά καθορίζεται από το πώς ένα αντικείμενο αντιδρά σε συμβάντα που προέρχονται από εξωτερικές πηγές πως αλληλεπιδρά ε άλλα αντικείμενα

1.2.2 Λειτουργίες και Μέθοδοι

Συνήθως οι τάξεις και τα αντικείμενα χαρακτηρίζονται από το όνομα τους , τα χαρακτηριστικά τους και τις λειτουργίες τους (operations). Μία λειτουργία είναι μία συνάρτηση ή μεταβλητή που εφαρμόζεται από ένα αντικείμενο σε κάποιο άλλο που είναι το αντικείμενο-στόχος.

Σε κάθε τάξη τα αντικείμενα έχουν τις ίδιες λειτουργίες . Η ακριβής συμπεριφορά της λειτουργίας εξαρτάται από την τάξη του στόχου της. Το αντικείμενο-στόχος γνωρίζει την τάξη του και επομένως γνωρίζει την υλοποίηση της λειτουργία. Σε γλώσσες που προέρχονται από διαδικαστικές γλώσσες , όπως η C++ που προέρχεται από τη C, μία λειτουργία είναι η ενεργοποίηση της συνάρτησης-μέλους (memberfunction) ενός αντικειμένου-στόχου από το αντικείμενο που εκτελεί τη λειτουργία.

Η μέθοδος είναι ένα όρος που χρησιμοποιείται με μικρές παραλλαγές στην αντικειμενοστρεφή βιβλιογραφία. Άλλοτε μέθοδος σημαίνει ακριβώς το ίδιο πράγμα με τη λειτουργία και άλλοτε μέθοδος σημαίνει υλοποίηση μιας λειτουργίας για μια τάξη

Μπορούμε να διακρίνουμε τέσσερα είδη λειτουργιών που μπορεί να εκτελέσει ένα αντικείμενο πάνω σε ένα άλλο:

1. **Λειτουργία Επιλογέας (Selector):** Μια λειτουργία που έχει πρόσβαση στις τιμές των χαρακτηριστικών ενός αντικειμένου, αλλά δεν τις αλλάζει. Θα μπορούσε να παρομοιαστεί

με READ_ONLY λειτουργίες.

2. **Λειτουργία Τροποποιητής (Modifier):**Μια λειτουργία που αλλάζει κάποιες τιμές χαρακτηριστικών του αντικειμένου. Θα μπορούσε να παρομοιαστεί με WRITE λειτουργίες.

3. **Λειτουργία Κατασκευαστή (Constructor):**Μια λειτουργία που δημιουργεί ένα αντικείμενο και αρχικοποιεί την κατάσταση του.

4. **Λειτουργία Καταστροφέας (Destructor):**Μία λειτουργία που καταστρέφει ένα αντικείμενο.

1.2.3 Σχέσεις τάξεων –Αντικειμένων

Οι σχέσεις των τάξεων είναι το μέσο για να δηλώνονται οι συνεργασίες μεταξύ τάξεων. Όταν δύο τάξεις συσχετίζονται μεταξύ τους, τότε σχετίζονται και τα δείγματα των τάξεων δηλαδή τα αντικείμενα, οπότε μπορούν να στείλουν μηνύματα το ένα στο άλλο. Ένα μήνυμα ξεκινά συνήθως από μία λειτουργία ενός αντικείμενου-πελάτη και κατευθύνεται σε ένα αντικείμενο-εξυπηρέτη.

Γενικά μπορούμε να διακρίνουμε τρία είδη σχέσεων τάξεων:

1. Συσχετισμοί
2. Σχέσεις-συναθροίσεις
3. Κληρονομικότητα

Οι πιο απλές σχέσεις είναι οι συσχετισμοί , οπότε κάθε άλλου είδους σχέση είναι και συσχετισμός.

Στις σχέσεις συνάθροισης έχουμε σχέση μέρους-όλου μεταξύ των τάξεων , δηλαδή μία τάξη αποτελείται από κάποιες άλλες τάξεις.

Στις σχέσεις κληρονομικότητας μία τάξη αποτελεί εξειδίκευση μιας άλλης , οπότε κληρονομεί από τη γενικευμένη τάξη τη δομή και τη συμπεριφορά της.

Η κληρονομικότητα είναι η πιο σημαντική σχέση δύο τάξεων για την αντικειμενοστραφή προσέγγιση. Σημασιολογικά, η κληρονομικότητα είναι η πιο ισχυρή σχέση απ` όλες. Μία τάξη μπορεί να συνδέεται με σχέση κληρονομικότητας με μία λεγόμενη υπέρ-τάξη , εάν η σχέση κληρονομικότητας είναι σχέση ειδίκευσης/ γενίκευσης και πολλές φορές ονομάζεται σχέση είναι (isa) ή σχέση «ένα είδος του» (a kindof).

Οι σχέσεις κληρονομικότητας είναι μεταβατικές με αποτέλεσμα να δημιουργούνται ιεραρχίες, οι οποίες αποτελούνται από πολλά επίπεδα γενικεύσεων-εξειδικεύσεων.

1.3 Πλεονεκτήματα – Μειονεκτήματα Αντικειμενοστραφούς Τεχνολογίας

1.3.1 Πλεονεκτήματα

Η Αντικειμενοστρεφής Τεχνολογία δεν ακολουθεί κάποια προσέγγιση που καταργεί τα αποδεδειγμένα καλά στοιχεία της Τεχνολογίας Λογισμικού, αλλά προσθέτει ορισμένα καινούργια στα ήδη υπάρχοντα, όπως η κληρονομικότητα.

Ο λόγος που η Αντικειμενοστραφής τεχνολογία κερδίζει έδαφος είναι γιατί δίνει καλές λύσεις και προοπτικές σε ορισμένα προβλήματα του κύκλου ζωής λογισμικού, όπως για παράδειγμα τη συντηρησιμότητα

Επαναχρησιμοποιησιμότητα

Η επαναχρησιμοποιησιμότητα (reusability) είναι ίσως το πιο γνωστό και πολυσυζητημένο πλεονέκτημα της Αντικειμενοστραφούς τεχνολογίας. Κάποιες τάξεις αντικειμένων που αναπτύσσονται σε μια εφαρμογή είναι δυνατόν να χρησιμοποιηθούν και σε άλλες εφαρμογές όπου ενδεχομένως θα αποτελέσουν τις βασικές τάξεις, δηλαδή τις τάξεις – ρίζες των δέντρων κληρονομικότητας που θα οριστούν για να εξυπηρετήσουν τις ανάγκες της νέας εφαρμογής.

Το γεγονός ότι μία τάξη ενθυλακώνει τα δεδομένα και την υλοποίηση της , μπορεί να την καταστήσει μια μονάδα ανεξάρτητη που μπορεί να εμφυτευτεί σε διαφορετική εφαρμογή προκειμένου να παρέχει στοιχεία προς κληρονόμηση

Συντηρησιμότητα

Η συντηρησιμότητα (maintainability) του λογισμικού διευκολύνεται με την αντικειμενοστραφή προσέγγιση διότι τα αντικειμενοστραφή συστήματα λογισμικού είναι

επεκτάσιμα και φιλικά στις αλλαγές. Επιπλέον , αν μεγάλο κομμάτι του συστήματος χρειάζεται να ξανασχεδιαστεί αυτό δεν σημαίνει ότι όλες οι τάξεις που είχαν αρχικά οριστεί θα είναι άχρηστες , αφού μπορούν να εμφυτευτούν στο καινούριο σύστημα με ευκολία.

1.3.2 Πολυμορφισμός

Ο πολυμορφισμός είναι μία ιδιότητα κάποιων γλωσσών προγραμματισμού που συνδέεται άμεσα με τα λεγόμενα αντικειμενοστραφή χαρακτηριστικά.

Η έννοια πολυμορφισμός έχει τις ρίζες της στη θεωρία τύπων των γλωσσών προγραμματισμού. Οι γλώσσες με παραδοσιακή τυποποίηση είναι μονομορφικές διότι βασίζονται στο γεγονός ότι οι συναρτήσεις και οι διαδικασίες έχουν ένα μοναδικό τύπο. Οι περισσότερες διαδικαστικές γλώσσες είναι μονομορφικές , όπως η FORTRAN , η PASCAL, η C και η Cobol.

Αντίθετα από τον μονομορφισμό , ο πολυμορφισμός επιτρέπει σε ένα μόνο όνομα να συμβολίζει δείγματα από διαφορετικές τάξεις εφόσον υπάρχει μία κοινή υπερ-τάξη που συνδέει τις διαφορετικές τάξεις. Ο πολυμορφισμός συνδέεται με δυναμική δραστηριότητα κατά τη διάρκεια της μεταγλώττισης.

Οι Cardelli και Wegner (1985) έχουν διαπιστώσει ότι υπάρχουν αρκετά είδη πολυμορφισμού τα οποία σε μικρό ή μεγάλο βαθμό υποστηρίζονται από τις διάφορες αντικειμενοστρεφείς γλώσσες προγραμματισμού . τα πιο σημαντικά είδη πολυμορφισμού είναι τα εξής:

1. **Παραμετρικός πολυμορφισμός**

Αυτού του είδους ο πολυμορφισμός συμβαίνει όταν μία συνάρτηση μπορεί να εφαρμοστεί σε πολλά είδη τύπων.

2. **Συμπεριλαμβάνων πολυμορφισμός (inclusionpolymorphism)**

Αυτού του είδους ο πολυμορφισμός έχει γίνει για να εξυπηρετήσει την κληρονομικότητα, όπου ένα αντικείμενο μπορεί να ανήκει σε πολλές τάξεις. Υλοποιείται χρησιμοποιώντας τις συναρτήσεις virtual.

3. Αυθαίρετος πολυμορφισμός

Αυτού του είδους ο πολυμορφισμός αντιστοιχεί σε ένα μικρό σύνολο μονομορφικών συναρτήσεων , όπου το ίδιο όνομα μεταβλητής χρησιμοποιείται για να σημαίνει πολλές συναρτήσεις και τα συμφραζόμενα χρησιμοποιούνται για να καθορίσουν ποια συνάρτηση εννοείται κάθε φορά. Αυτή η περίπτωση λέγεται υπερφόρτωση (overloading).

1.3.3 Απόκρυψη πληροφοριών και ενθυλάκωση

Η απόκρυψη πληροφοριών είναι μια έννοια που συστάθηκε από τον Parnas (1971). Αυτός διατύπωσε την άποψη ότι ένα σύστημα λογισμικού πρέπει να αποτελείται από τμήματα τα οποία κρύβουν τις λεπτομέρειες υλοποίησης τους και επικοινωνούν μεταξύ τους διαμέσου καλά ορισμένων διεπαφών (interfaces).

Ειδικότερα, θα πρέπει να κρύβονται οι δύσκολες σχεδιαστικές αποφάσεις οι οποίες είναι πιθανό να αλλάξουν, και οι αλγόριθμοι. Στην αντικειμενοστραφή προσέγγιση η απόκρυψη πληροφοριών γίνεται μέσα στην αφαίρεση της τάξης.

Η απόκρυψη πληροφοριών σχετίζεται άμεσα με την έννοια της ενθυλάκωσης η οποία είναι η διαδικασία πακετοποίησης των στοιχείων μιας αφαίρεσης, είτε αυτά είναι χαρακτηριστικά , είτε λειτουργίες. Οι Britton και Parnas αποκαλούν τα ενθυλακωμένα στοιχεία «μυστικά» της αφαίρεσης.

Το κυριότερο πλεονέκτημα της ενθυλάκωσης είναι η δυνατότητα αλλαγής της

εσωτερικής δομής και λειτουργίας μιας αφαίρεσης χωρίς αυτό να επηρεάζει κανέναν από τους πελάτες της.

Κάθε τάξη έχει τρία μέρη:

1. **Το δημόσιο (public)**

Εκεί δηλώνονται τα μέλη τα οποία είναι ορατά από όλους τους πελάτες της τάξης και επομένως είναι τα λιγότερο κρυμμένα στοιχεία της τάξης.

2. **Το ιδιωτικό (private)**

Εκεί δηλώνονται τα μέλη τα οποία είναι πλήρως ενθυλακωμένα και τα οποία είναι ορατά μόνο από την ίδια την τάξη και τις τάξεις-φίλες της τάξης αυτής.

3. **Το προστατευμένο (protected)**

Εκεί δηλώνονται τα μέλη πο είναι ορατά από την ίδια την τάξη και τις υπο-τάξεις της.

Στην αντικειμενοστρεφή προσέγγιση , παρατηρείται ένας ανταγωνισμός μεταξύ της ενθυλάκωσης και της κληρονομικότητας.

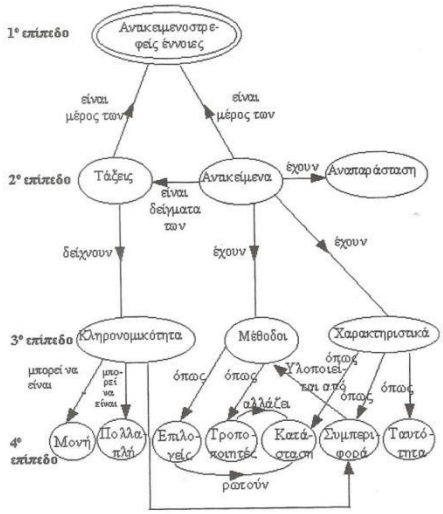
Αρχικά το πρόβλημα δίνεται από τους χρήστες στους μηχανικούς λογισμικού, οι οποίοι συνεργάζονται με τους εμπειρογνώμονες του πεδίου. Η αρχική επικοινωνία και ο ορισμός των απαιτήσεων δεν είναι απλό να γίνουν. Όταν καθοριστούν οι απαιτήσεις γίνεται η ανάλυση πεδίου, όπου θα πρέπει να αναζητηθούν οι τάξεις, τα χαρακτηριστικά, οι υπευθυνότητες , οι σχέσεις και οι ιεραρχίες.

Στην φάση της ανάλυσης , οι μηχανικοί λογισμικού ασχολούνται με τον προσδιορισμό του τι πρέπει να γίνει και όχι με το πώς θα γίνει αυτός.

Στην φάση του σχεδιασμού γίνεται η εκλέπτυνση των διαγραμμάτων τάξεων που έχουν προκύψει από την Ανάλυση, προκειμένου να συμπεριληφθούν νέες τάξεις που έχουν σχέση με την υλοποίηση και όχι με την περιγραφή του πεδίου της εφαρμογής. Τέτοιου

είδους τάξεις λέγονται εσωτερικές τάξεις ή τάξεις λογισμικού.

Τέλος στην φάση της εξέλιξης γίνεται η υλοποίηση του λογισμικού.



Σχήμα 1.3 Εννοιολογικός χάρτης για αντικειμενοστρεφείς έννοιες από τους Novak και Gowin.

1.3.4 Προβλήματα της Αντικειμενοστραφούς τεχνολογίας

Όπως σε κάθε καινούρια τεχνολογία, έχει ασκηθεί και κάποια κριτική για την αντικειμενοστραφή προσέγγιση. Το βασικότερο πρόβλημα είναι ότι αλλάζει τη φιλοσοφία ανάλυσης και σχεδιασμού, καταργώντας έτσι την παλαιότερη διαδικασία ανάπτυξης και το προϋπάρχον λογισμικό. Κατά συνέπεια , η αποκλειστική χρήση της αντικειμενοστραφούς τεχνολογίας καθίσταται δαπανηρή όταν πρέπει να αντικαταστήσει παλαιότερα συστήματα λογισμικού.

Υπάρχουν πολλοί επίσης που υποστηρίζουν ότι η αντικειμενοστραφής ανάλυση και σχεδιασμός δεν μπορούν να αποδώσουν πολύ καλά τη ροή δεδομένων σε ένα σύστημα, οπότε προτείνουν την παράλληλα χρήση Διαγραμμάτων Ροής Δεδομένων (DataFlowDiagrams) .

Οι υποστηρικτές της Αντικειμενοστραφούς τεχνολογίας Λογισμικού αντιπαραθέτουν σε αυτό το επιχείρημα ότι η αντικειμενοστραφής προσέγγιση μπορεί να αποδειχθεί ελλιπής μόνο όταν δεν υπάρχει κατάλληλα ειδικευμένο προσωπικό και τεχνολογική υποδομή που να ευνοούν την αντικειμενοστραφή ανάπτυξη.

ΚΕΦΑΛΑΙΟ 2 – ΑΝΤΙΚΕΙΜΕΝΟΣΤΡΑΦΗΣ ΓΛΩΣΣΑ ΠΡΟΓΡΑΜΜΑΤΙΣΜΟΥ JAVA

2.1 Τι είναι η JAVA

Η JAVA [27] είναι μια γλώσσα προγραμματισμού που σχεδιάστηκε από τη εταιρία Sun Microsystemes Η Java χαρακτηρίζεται από τα εξής: απλή, αντικειμενοστραφής, συμβατή με δικτυακά πρωτόκολλα, ουδέτερη της υποκείμενης αρχιτεκτονικής, φορητή ασφαλής, υψηλής απόδοσης, δυναμική, σταθερή, interpreted και multithreaded. Η Java έχει σχεδιαστεί για να υποστηρίζει δικτυακές εφαρμογές. Ένα δίκτυο, όμως, αποτελείται από ποικιλία διαφορετικών συστημάτων, με διαφορετικές CPU και λειτουργικά συστήματα. Για να μπορούν οι Java εφαρμογές να εκτελούνται παντού στο δίκτυο, το πρόγραμμα Java πρέπει να περάσει από δύο διαδικασίες ώστε να καταλήξει σε εκτελέσιμη μορφή. Πρώτα ο μεταγλωττιστής, μετατρέπει τον πηγαίο κώδικα του προγράμματος σε μία ενδιάμεση γλώσσα που καλείται Java bytecodes. Τα Java bytecodes είναι ανεξάρτητα της πλατφόρμας και με χρήση του ερμηνευτή (interpreter) κάθε bytecode εντολή μετατρέπεται σε κατάλληλη δυαδική μορφή για να τρέξει στον εκάστοτε υπολογιστή. Η μεταγλώττιση (compilation) συμβαίνει μόνο μια φορά για κάθε Java πρόγραμμα, η ερμηνεία (interpretation) γίνεται κάθε φορά που το πρόγραμμα εκτελείται.

Τα Java bytecodes μπορούμε να τα φανταστούμε σαν την γλώσσα μηχανής για την Java Virtual Machine (JVM). Κάθε Java ερμηνευτής (π.χ. ένας Web server που μπορεί να τρέχει Servlets) είναι μια λογισμική εφαρμογή του της Java Virtual Machine. Η JVM αναλαμβάνει να μετατρέψει τα bytecodes σε κατάλληλη εκτελέσιμη μορφή, ανάλογα με το υποκείμενο software και hardware.

2.2 Στοιχεία και φιλοσοφία της γλώσσας

Αφού γραφεί κάποιο πρόγραμμα σε Java, στη συνέχεια μεταγλωτίζεται μέσω του μεταγλωττιστή javac, ο οποίος παράγει έναν αριθμό από αρχεία .class (κώδικας byte ή bytecode). Ο κώδικας byte είναι η μορφή που παίρνει ο πηγαίος κώδικας της Java όταν μεταγλωττιστεί. Όταν πρόκειται να εκτελεστεί η εφαρμογή σε ένα μηχάνημα, το Java Virtual Machine που πρέπει να είναι εγκατεστημένο σε αυτό θα αναλάβει να διαβάσει τα αρχεία .class. Στη συνέχεια τα μεταφράζει σε γλώσσα μηχανής που να υποστηρίζεται από το λειτουργικό σύστημα και τον επεξεργαστή, έτσι ώστε να εκτελεστεί (αυτό συμβαίνει με την παραδοσιακή Εικονική Μηχανή (Virtual Machine). Πιο σύγχρονες εφαρμογές της εικονικής Μηχανής μπορούν και μεταγλωττίζουν εκ των προτέρων τμήματα bytecode απευθείας σε κώδικα μηχανής (εγγενή κώδικα ή native code) με αποτέλεσμα να βελτιώνεται η ταχύτητα). Χωρίς αυτό δε θα ήταν δυνατή η εκτέλεση λογισμικού γραμμένου σε Java. Η JVM είναι λογισμικό που εξαρτάται από την πλατφόρμα, δηλαδή για κάθε είδος λειτουργικού συστήματος και αρχιτεκτονικής επεξεργαστή υπάρχει διαφορετική έκδοση του. Έτσι υπάρχουν διαφορετικές JVM για Windows, Linux, Unix, Macintosh, κινητά τηλέφωνα, παιχνιδομηχανές κλπ.

Οτιδήποτε θέλει να κάνει ο προγραμματιστής (ή ο χρήστης) γίνεται μέσω της εικονικής μηχανής. Αυτό βοηθάει στο να υπάρχει μεγαλύτερη ασφάλεια στο σύστημα γιατί η εικονική μηχανή είναι υπεύθυνη για την επικοινωνία χρήστη - υπολογιστή. Ο προγραμματιστής δεν μπορεί να γράψει κώδικα ο οποίος θα έχει καταστροφικά αποτελέσματα για τον υπολογιστή γιατί η εικονική μηχανή θα τον ανιχνεύσει και δε θα επιτρέψει να εκτελεστεί. Από την άλλη μεριά ούτε ο χρήστης μπορεί να κατεβάσει «κακό» κώδικα από το δίκτυο και να τον εκτελέσει. Αυτό είναι ιδιαίτερα χρήσιμο για μεγάλα κατανεμημένα συστήματα όπου πολλοί χρήστες χρησιμοποιούν το ίδιο πρόγραμμα συγχρόνως.

2.3 Πλεονεκτήματα της γλώσσας JAVA

Η γλώσσα JAVA υπερέχει σε σχέση με τις υπόλοιπες γλώσσες προγραμματισμού σε πολλούς τομείς. Τα μοναδικά χαρακτηριστικά που την περιγράφουν είναι :

- Η ανεξαρτησία του λειτουργικού συστήματος και πλατφόρμας.
- Απλή
- Αντικειμενοστραφής
- Συμβατή με Δίκτυα
- Σταθερή
- Ασφάλεια εκτέλεσης προγραμμάτων.
- Η ευκολία εκμάθησης και εκπαίδευσης.
- Ο φυσικότερος και πιο ‘ανθρώπινος’ τρόπος έκφρασης των προβλημάτων.

Η ανεξαρτησία του λειτουργικού συστήματος την καθιστά σε μια πολύ διαδεδομένη γλώσσα προγραμματισμού. Μπορεί κάποιος να γράψει ένα πρόγραμμα και το τρέξει σε διαφορετικά λειτουργικά συστήματα και πλατφόρμες, Είναι ευρέως διαδεδομένη γλώσσα και καλύπτει σχεδόν όλες τις ανάγκες των σύγχρονων επιχειρήσεων και αγορών. Είναι εύκολη στην εκμάθηση και στην επαναχρησιμοποίηση του κώδικα. Χρησιμοποιώντας βιβλιοθήκες της JAVA μπορεί κάποιος εύκολα να είναι ασφαλείς στην εκτέλεση των προγραμμάτων. Είναι μια γλώσσα εύκολη στην εκμάθηση και έχει την δυνατότητα να λόγο της αντικειμενοστραφείς φύση της να αναπαριστά όλες τους πραγματικούς – ανθρώπινους τρόπους έκφρασης του προβλήματος με αντικείμενα.

ΚΕΦΑΛΑΙΟ 3 – ΣΧΕΔΙΑΣΤΙΚΑ ΠΡΟΤΥΠΑ

3.1 Εισαγωγή στα Design Patterns

Στην τεχνολογία λογισμικού, ένα πρότυπο σχέδιο εφ εξής design pattern [28] είναι μια γενική επαναλαμβανόμενη λύση σε προβλήματα που συμβαίνουν συνήθως στο σχεδιασμό λογισμικού. Ένα desing pattern δεν είναι ένα τελικό σχέδιο που μπορεί να μετατραπεί άμεσα σε κώδικα. Είναι μια περιγραφή ή πρότυπο για το πώς να λύσει ένα πρόβλημα που μπορεί να χρησιμοποιηθεί σε πολλές διαφορετικές καταστάσεις.

Σχεδιαστικά πρότυπα μπορούν να επιταχύνουν τη διαδικασία ανάπτυξης παρέχοντας δυνατότητες testing και χρησιμοποιούν αποδεδειγμένα πρότυπα ανάπτυξης λειτουργειών. Για τον αποτελεσματικό σχεδιασμό του λογισμικού απαιτείτε η εξέταση των θεμάτων που μπορεί να μην γίνονται ορατά άμεσα και πριν την συγγραφή του κώδικα αλλά αργότερα κατά την ανάπτυξη της εφαρμογής. Η δυνατότητα επαναχρησιμοποίησης των design patterns βοηθά στην πρόληψη προβλημάτων που μπορεί να επέλθουν κατά την ανάπτυξη της εφαρμογής και ακόμα βελτιώνει την αναγνωσιμότητα του κώδικα για προγραμματιστές και αρχιτέκτονες που θα τον διαβάσουν και είναι εξοικειωμένοι με τα αυτά.

Συχνά, οι προγραμματιστές εφαρμόζουν ορισμένες τεχνικές σχεδιασμού του λογισμικού σε διάφορα προβλήματα και μη προτυποποιημένες. Αυτές οι τεχνικές είναι δύσκολο να εφαρμοστούν σε ένα ευρύτερο φάσμα των προβλημάτων. Τα design patterns παρέχουν γενικές λύσεις, τεκμηριωμένες σε μια μορφή που δεν απαιτεί λεπτομέρειες που είναι δεμένες με ένα συγκεκριμένο πρόβλημα.

Επιπλέον, τα design patterns επιτρέπουν στους προγραμματιστές να επικοινωνούν μεταξύ τους χρησιμοποιώντας γνωστά, καλά κατανοητά ονόματα για τις αλληλεπιδράσεις του λογισμικού. Τα κοινά design patterns μπορούν να βελτιωθούν με την πάροδο του χρόνου, καθιστώντας τα πιο ισχυρά από άλλες ad-hoc σχεδιαστικές επιλογές.

3.2 Δημιουργικά πρότυπα (Creational patterns)

Στην τεχνολογία λογισμικού, τα creational patterns είναι πρότυπα σχεδιασμού που ασχολούνται με τους μηχανισμούς δημιουργίας αντικειμένου (object), προσπαθούν να δημιουργήσουν αντικείμενα με τρόπο κατάλληλο για την εκάστοτε κατάσταση. Η βασική μορφή δημιουργίας αντικείμενου θα μπορούσε να οδηγήσει σε σχεδιαστικά προβλήματα ή προσθέτοντας παραπάνω πολυπλοκότητα στο σχεδιασμό. Τα creational patterns έρχονται να λύσουν αυτό το πρόβλημα ελέγχοντας την δημιουργία του αντικειμένου. Μερικά από τα παραδείγματα των creational patterns είναι.

- Abstract Factory δημιουργεί ένα αντικείμενο από διάφορες οικογένειες κλάσεων
- Bulder Διαχωρίζει την αρχικοποίηση των αντικειμένων από την λειτουργία τους
- Factory Method Δημιουργεί ένα στιγμιότυπο ενός αντικειμένου από διάφορες παραγόμενες κλάσεις
- Object Pool Αποφύγετε η ακριβή κατανάλωση των πόρων και γίνετε η απελευθέρωση τους με την ανακύκλωση των αντικειμένων που δεν είναι πλέον σε χρήση
- Prototype Ένα πλήρως αρχικοποιημένου στιγμιότυπου ενός αντικειμένου με σκοπό να αντιγραφεί ή να κλωνοποιηθεί
- Sigleton Μια κλάση που μόνο ένα στιγμιότυπο αυτής μπορεί να υπάρχει.

3.2.1 Abstract Factory

Σκοπός

Παρέχει μια διεπαφή για τη δημιουργία οικογενειών που είναι συναφείς ή εξαρτώμενες από αντικείμενα χωρίς όμως να προσδιορίζει συγκεκριμένες κλάσεις τους. Μια ιεραρχία που ενθυλακώνει: πολλές πιθανές «πλατφόρμες», καθώς και η κατασκευή μιας σειράς αντικειμένων. Ο νέος φορέας θεωρείται επιβλαβής.

Πρόβλημα

Αν μία εφαρμογή πρέπει να είναι φορητή, υπάρχει η ανάγκη να ενσωματώσει κάποιες εξαρτήσεις. Αυτές οι εξαρτήσεις μπορεί να είναι: το σύστημα παραθύρων του λειτουργικού, βάση δεδομένων, κλπ. Πολύ συχνά, αυτή η ενθυλάκωση δεν έχει σχεδιαστεί εκ των προτέρων, και πολλές #if δηλώσεις μέσα στον κώδικα υφίστανται με διάφορες επιλογές για όλα τα πιθανά σενάρια υποστηριζόμενης πλατφόρμας. Αυτά σιγά σιγά γιγαντώνονται και γίνονται δύσκολα διαχειρίσιμα μέσα σε όλο τον κώδικα .

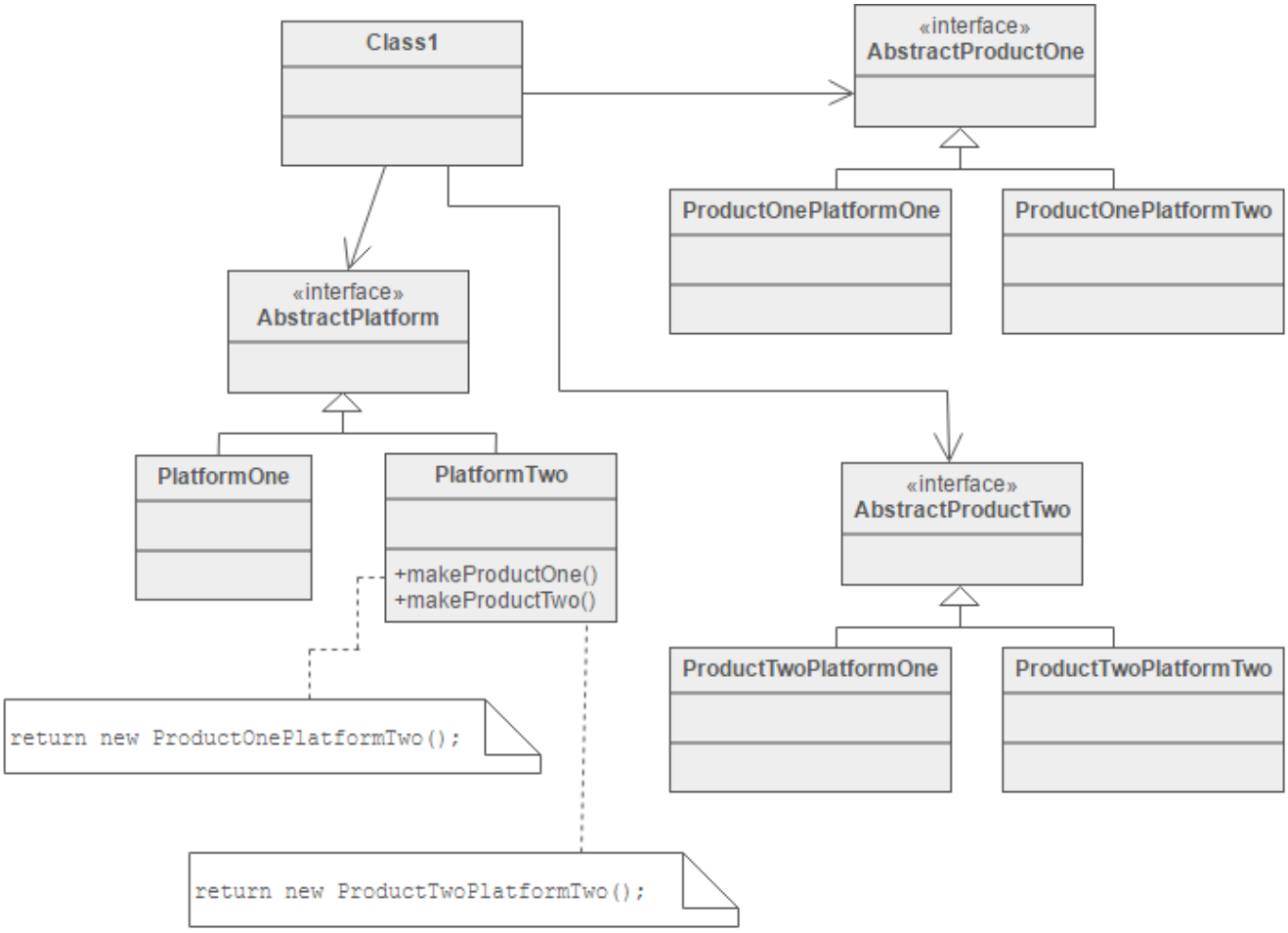
Λύση

Παρέχοντας ένα επίπεδο έμμεσου τύπου που αφαιρεί τη δημιουργία των οικογενειών των συναφών ή εξαρτώμενων αντικειμένων χωρίς να προσδιορίζεται απευθείας οι συγκεκριμένες κλάσης τους. Το Abstract Factory [29] αντικείμενο έχει την ευθύνη για την παροχή υπηρεσιών δημιουργίας για όλη της εξαρτήσεις της πλατφόρμας. Οι πελάτες δεν δημιουργούν άμεσα αντικείμενα μονο για την εκαστοτε πλατφόρμα αλλα ζητούν από το Abstract Factory για να το κάνει αυτό για αυτούς. Ο μηχανισμός αυτός καθιστά την ανταλλαγή των αντικειμένων εύκολότερη, διότι η συγκεκριμένη κλάση του Abstract Factory εμφανίζεται μόνο μία φορά στον κώδικα - όπου και τεκμηριώθηκε. Η εφαρμογή μπορεί να αντικαταστήσει ολόκληρη την οικογένεια των εξαρτήσων απλά αλλάζοντας το στιγμιότυπο αντικειμένου ένα διαφορετικό σε κάθε αλλαγή της ανάγκης. Επειδή η υπηρεσία που παρέχεται από το Factory

αντικείμενο είναι τόσο διάχυτη μέσα στον κώδικα, είναι σχεδόν αναγκαίο να υλοποιείται ως ένα Singleton.

Δομή

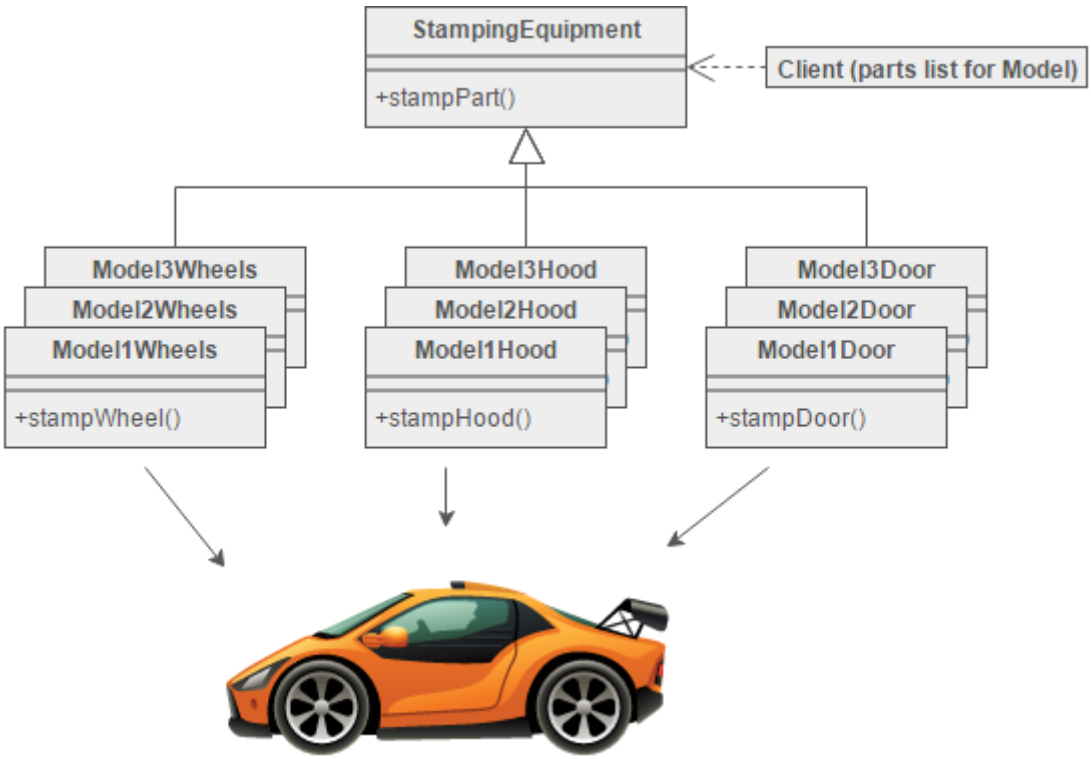
Η Abstract Factory ορίζει μια μέθοδο Factory ανά προϊόν. Κάθε Fatory μέθοδος ενθυλακώνει το νέο φορέα και τις εξαρτήσεις του για παράδειγμα η συγκεκριμένη πλατφόρμα, κατηγορίες προϊόντων κλπ. Η εφαρμογή στη συνέχεια διαμορφώνεται με ένα Factory αντικείμενο που προέρχεται από μια Factory class.



Σχήμα 3.1: Abstract Factory ορίζει μια μέθοδο Factory ανά προϊόν

Παράδειγμα

Ο σκοπός του Abstract Factory είναι να παρέχει μια διεπαφή για τη δημιουργία οικογενειών που είναι συναφείς ή εξαρτώμενες από αντικείμενα χωρίς όμως να προσδιορίζει συγκεκριμένες κλάσεις τους. Αυτό το design patter απαντάται στον εξοπλισμό που απαιτείτε για το πάτημα της λαμαρίνας που χρησιμοποιούνται στην παρασκευή των ιαπωνικών αυτοκινήτων. Ο εξοπλισμός αυτός είναι ένα Abstract Factory που δημιουργεί αυτόματα μέρη του αμαξώματος. Ο ίδιος ο μηχανισμός χρησιμοποιείται για να σφραγίσει την δεξιά πόρτα, την αριστερή πόρτα, το δεξί εμπρός φτερό, αριστερά εμπρός φτερό, καπό, κλπ για τα διάφορα μοντέλα των αυτοκινήτων. Μέσω της χρήσης των κυλίνδρων μπορεί να αλλάζει τις λειτουργίες σφράγισης και οι κατηγορίες αμαξώματος που μπορούν να πραχτούν από τη μηχανή πρεσαρίσματος μπορούν να αλλάξουν μέσα σε τρία λεπτά.



Σχήμα 3.2: Το αμάξωμα είναι προϊόν του stamping equipment

Ένα προγραμματιστικό παράδειγμα ενός Abstract Factory στην JAVA είναι η αρχικοποίηση ενός cpu ανάλογα την αρχιτεκτονική του συστήματος.

```
public abstract class CPU
{
  ...
} // class CPU

class EmberCPU extends CPU
{
  ...
} // class EmberCPU

class EmberToolkit extends AbstractFactory
{
  public CPU createCPU()
  {
    return new EmberCPU();
  } // createCPU()

  public MMU createMMU()
  {
    return new EmberMMU();
  } // createMMU()
  ...
} // class EmberFactory

public abstract class AbstractFactory
{
  private static final EmberToolkit emberToolkit = new EmberToolkit();
  private static final EnginolaToolkit enginolaToolkit = new EnginolaToolkit();
  ...

  // Returns a concrete factory object that is an instance of the
  // concrete factory class appropriate for the given architecture.
  static final AbstractFactory getFactory(int architecture)
  {
    switch (architecture)
    {
      case ENGINOLA:
        return enginolaToolkit;

      case EMBER:
        return emberToolkit;
        ...
    } // switch
    String errMsg = Integer.toString(architecture);
    throw new IllegalArgumentException(errMsg);
  } // getFactory()

  public abstract CPU createCPU();
  public abstract MMU createMMU();
```

```
  ...
} // AbstractFactory

public class Client
{
  public void doIt()
  {
    AbstractFactory af;
    af = AbstractFactory.getFactory(AbstractFactory.EMBER);
    CPU cpu = af.createCPU();
    ...
  } // doIt
} // class Client
```

Σχήμα 3.3: Χρησιμοποιόντας την Abstract Factory.

Εδώ βλέπουμε το application μπορεί να αρχικοποίηση το κατάλληλο αντικείμενο με βάση την αρχιτεκτονική του συστήματος και να έχει αρχικοποιημένο το κατάλληλο Cpu object ανάλογα την κατάσταση.

3.2.2 Builder Pattern

Σκοπός

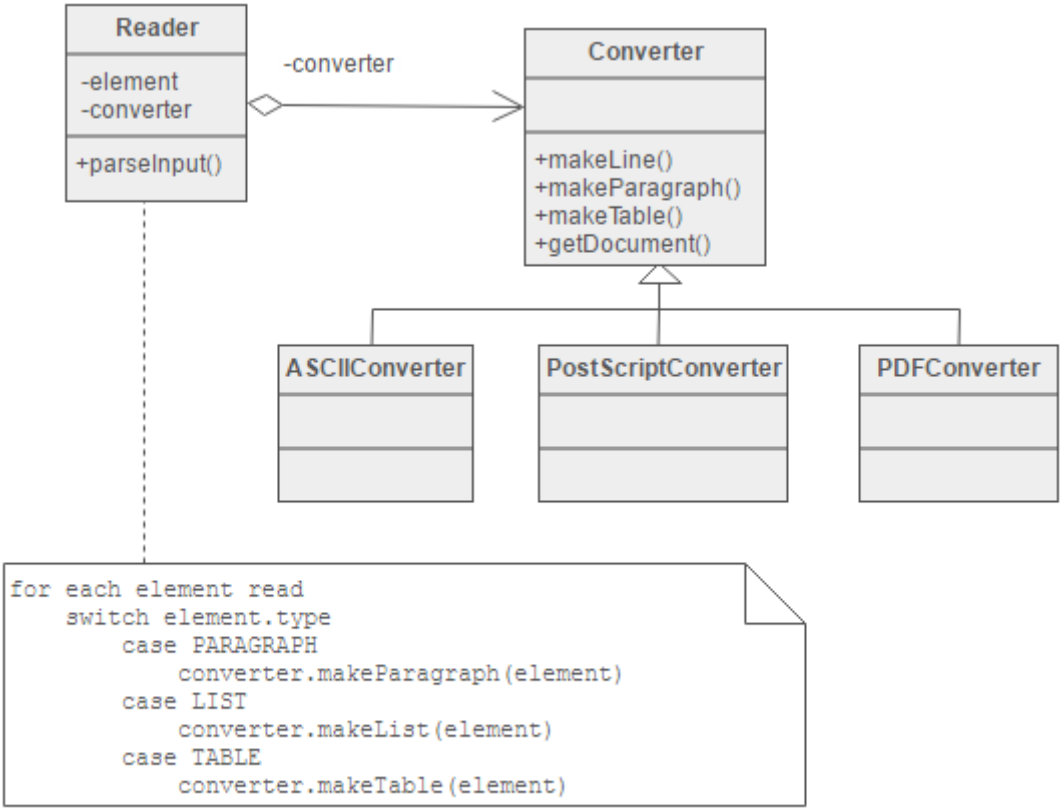
Στόχος του Builder Pattren [30] είναι να διαχωρίσει την κατασκευή ενός σύνθετου αντικειμένου από την αναπαράστασή του, έτσι ώστε η ίδια διαδικασία κατασκευής να μπορεί να δημιουργήσει διαφορετικές αναπαραστάσεις.

Πρόβλημα

Μία εφαρμογή πρέπει να δημιουργήσει τα στοιχεία ενός σύνθετου συνόλου. Οι προδιαγραφές για το σύνολο υπάρχουν στο secondary storage και μία από τις πολλές αναπαραστάσεις θα πρέπει να δημιουργηθεί στο primary storage.

Λύση

Διαχωρίζοντας τον αλγόριθμο για την ανάγνωση και την ανάλυση από ένα μέσο αποθήκευσης (π.χ. αρχεία RTF) και κατασκευάζοντας ένα αλγόριθμο που λειτουργεί για πολλαπλά αρχεία (π.χ. ASCII, TeX, widget κειμένου) μας δίνει την δυνατότητα να έχουμε σύνθετα μέσα αποθήκευσης αλλά και να μπορούμε να τα διαχειριστούμε καλύτερα. Ο "director" καλεί τα "builder" services και ταυτόχρονα διαβάζει την τα αρχεία. Τα "builder" services δημιουργούν ένα μεγάλο τμήμα αντικειμένου και κάθε φορά που καλείται ενθυλακώνει όλη την πολυπλοκότητα της ενδιάμεσης κατάστασης. Όταν το αντικείμενο έχει επεξεργαστεί, τότε ο πελάτης παίρνει το αποτέλεσμα από τα "builder" services. Με αυτό τον τρόπο παρέχετε καλύτερος έλεγχος κατά τη διάρκεια της διαδικασίας κατασκευής. Σε αντίθεση τα άλλα creational patterns που αρχικοποιούν προϊόντα σε πρώτο χρόνο, το Builder pattern κατασκευάζει το βήμα προς βήμα το αντικείμενο κάτω από τον έλεγχο του "director". Ο reader ενθυλακώνει την ανάγνωση της εισόδου.Ο Builder καθιστά δυνατή την πολυμορφική δημιουργία πολλών και σύνθετων αντικειμένων.

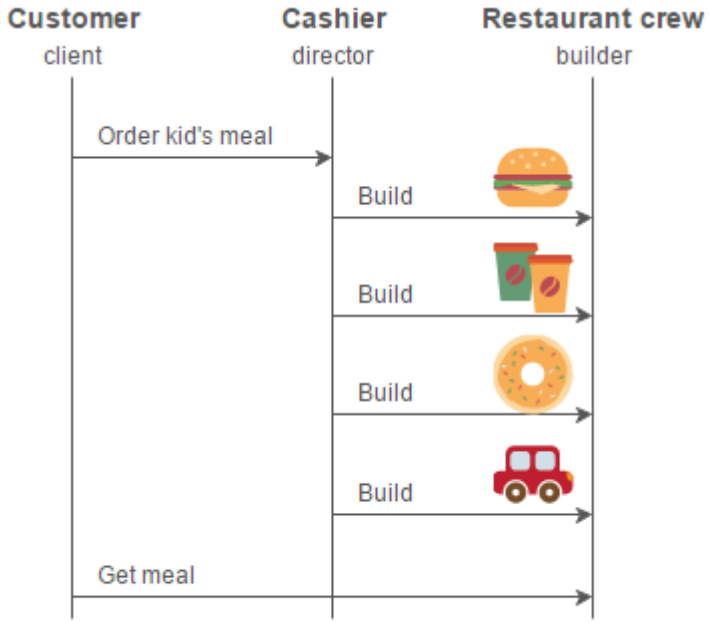


Σχήμα 3.1.1: Ο reader δεν βλέπει την αυξημένη πολυπλοκότητα

Παράδειγμα

Το Builder pattern διαχωρίζει την κατασκευή ενός σύνθετου αντικειμένου από την αναπαράστασή του, έτσι ώστε η ίδια διαδικασία κατασκευής μπορεί να δημιουργήσει σε διαφορετικές αναπαραστάσεις. Αυτό το πρότυπο χρησιμοποιείται από εστιατόρια fast food για να κατασκευάσει τα γεύματα των παιδιών. Παιδικά γεύματα αποτελούνται συνήθως από ένα κεντρικό στοιχείο, ένα δεύτερο στοιχείο, ένα ποτό, και ένα παιχνίδι (π.χ., ένα χάμπουργκερ, πατάτες, κοκ, και δεινόσαυρος παιχνίδι). Σημειώστε ότι μπορεί να υπάρξει μεταβολή στο περιεχόμενο του γεύματος των παιδιών, αλλά η διαδικασία κατασκευής είναι η ίδια. Αν ένας πελάτης παραγγέλλει ένα χάμπουργκερ, cheeseburger ή το κοτόπουλο, η διαδικασία είναι η ίδια. Ο υπάλληλος στον πάγκο κατευθύνει το πλήρωμα για να συγκεντρώσει ένα κεντρικό

στοιχείο, το δεύτερο στοιχείο, και τα παιχνιδία. Αυτά τα στοιχεία στη συνέχεια τοποθετούνται σε μια τσάντα. Το ποτό τοποθετείται σε ένα φλιτζάνι και παραμένει εκτός του σάκου. Αυτή η ίδια διαδικασία χρησιμοποιείται σε ανταγωνιστικές αλυσίδες εστιατορίων.



Σχήμα 3.1.2: Διαδικασία χτισίματος ενός παιδικού μενού

Ένα προγραμματιστικό παράδειγμα ενός Builder pattern είναι το εξής

```
class Pizza {
  private String dough = "";
  private String sauce = "";
  private String topping = "";

  public void setDough(String dough)     { this.dough = dough; }
  public void setSauce(String sauce)     { this.sauce = sauce; }
  public void setTopping(String topping) { this.topping = topping; }
}

/* "Abstract Builder" */
abstract class PizzaBuilder {
  protected Pizza pizza;

  public Pizza getPizza() { return pizza; }
```

```
  public void createNewPizzaProduct() { pizza = new Pizza(); }

  public abstract void buildDough();
  public abstract void buildSauce();
  public abstract void buildTopping();
}

/* "ConcreteBuilder" */
class HawaiianPizzaBuilder extends PizzaBuilder {
  public void buildDough()   { pizza.setDough("cross"); }
  public void buildSauce()   { pizza.setSauce("mild"); }
  public void buildTopping() { pizza.setTopping("ham+pineapple"); }
}

/* "ConcreteBuilder" */
class SpicyPizzaBuilder extends PizzaBuilder {
  public void buildDough()   { pizza.setDough("pan baked"); }
  public void buildSauce()   { pizza.setSauce("hot"); }
  public void buildTopping() { pizza.setTopping("pepperoni+salami"); }
}

/* "Director" */
class Waiter {
  private PizzaBuilder pizzaBuilder;

  public void setPizzaBuilder(PizzaBuilder pb) { pizzaBuilder = pb; }
  public Pizza getPizza() { return pizzaBuilder.getPizza(); }

  public void constructPizza() {
    pizzaBuilder.createNewPizzaProduct();
    pizzaBuilder.buildDough();
    pizzaBuilder.buildSauce();
    pizzaBuilder.buildTopping();
  }
}

/* A customer ordering a pizza. */
class BuilderExample {
  public static void main(String[] args) {
    Waiter waiter = new Waiter();
    PizzaBuilder hawaiian_pizzabuilder = new HawaiianPizzaBuilder();
    PizzaBuilder spicy_pizzabuilder = new SpicyPizzaBuilder();

    waiter.setPizzaBuilder( hawaiian_pizzabuilder );
    waiter.constructPizza();

    Pizza pizza = waiter.getPizza();
  }
}
```

Σχήμα 3.1.3 Παράδειγμα builder patter

Εδώ βλέπουμε τον σερβιτόρο – director να δίνει εντολές στον Builder για την Παρασκευή μια πίτσας. Τον σερβιτόρο – directror δεν τον ενδιαφέρει τα υλικά παρασκευής η την ζύμη τον μόνο που τον ενδιαφέρει είναι η διαδικασία και έτσι και γίνετε. Με αυτό τον τρόπο ο director είναι απαλλαγμένος από την πολυπλοκότητα του builder και έτσι αποφορτίζεται ο κώδικας από μεγάλη πολυπλοκότητα. Σε μια ποιο απλή εκδοχή αυτού του pattern είναι εφαρμοσμένη μέσα στον κώδικά μας και μπορεί να χτίζει ένα rule το οποίο στην συνέχεια θα μας δώσει την δυνατότητα να κάνουμε κάποιες συγκρίσεις στα δεδομένα μας.

```
public class RuleBuilder {


    private PatientDao patientDao;
    private RuleDao ruleDao;
    AlertDao alertDao;
    JSONObject responce;

    public JSONObject buildRules(HttpServletRequest req){
        patientDao = new PartientDaoImpl();
        ruleDao = new RuleDaoImpl();
        alertDao = new AlertDaoImpl();
        String key = req.getHeader("key");
        PatientItem patientItem = null;
        try {
            patientItem= patientDao.getPatientFromKey(key);
        } catch (Exception e) {
            e.printStackTrace();
        }
        JSONObject notAuth = new JSONObject();
        try {
            notAuth.put("message","AuthFailed");
        } catch (JSONException e) {
            e.printStackTrace();
        }
        if(patientItem != null){


            DoctorItem userDoctor = patientDao.getDoctorByPatient(patientItem);

            System.out.println("patien"+patientItem);
            System.out.println("userDoctor"+userDoctor);
            ArrayList<RuleItem> ruleItems = new ArrayList<RuleItem>();
            ruleItems = ruleDao.getAllRules();
            responce = new JSONObject();
            int count = 0;
            for(RuleItem ruleItem:ruleItems){
                boolean added = saveAlertIfExist(ruleItem,req,patientItem,userDoctor);
                if(added){
                    count++;
```

```
                }
            }

            if(count>0){

                try {
                    responce.put("message","Added");
                } catch (JSONException e) {
                    e.printStackTrace();
                }

            }else {

                try {
                    responce.put("message","Not added");
                } catch (JSONException e) {
                    e.printStackTrace();
                }
            }


            return responce;

        }else {
           return  notAuth;
        }
    }


    private boolean saveAlertIfExist(RuleItem ruleItem, HttpServletRequest req, PatientItem
userPatient, DoctorItem userDoctor) {
        String value = null;
        boolean added = false;

        try {
            value = req.getParameter(ruleItem.getRuleAttribute());
            if (value != null) {
                boolean alert = executeRule(null, null, Integer.valueOf(value),
ruleItem.getRuleThreshold(), ruleItem.getRuleStatement());
                if (alert) {
                    AlertItem alertItem = new AlertItem();
                    alertItem.setRuleId(ruleItem.getRuleId());
                    alertItem.setDoctorId(userDoctor.getUserId());
                    alertItem.setPatientId(userPatient.getPatiendId());
                    alertItem.setValue(value);
                    added = alertDao.saveAlert(alertItem);
                }
            }
        } catch (Exception e) {
            e.printStackTrace();
        }

        return added;

    }
```

```
    private boolean executeRule(String valueString, String storedValueString, int value,
int storedValue, RuleStatement ruleStatement){

        switch (ruleStatement) {
            case BIGGER:
                return value > storedValue;

            case BIGGER_EQUAL:
                return value >= storedValue;

            case SMALLER:
                return value < storedValue;

            case SMALLER_EQUAL:
                return value <= storedValue;

            case EQUAL:
                return value == storedValue;

            case EQUAL_STRING:
                return valueString.equals(storedValueString);
        }

        return false;

    }


}
```

Σχήμα 3.1.4: Παράδειγμα builder για τα rules στην εφαρμογή.

Εδώ βλέπουμε ότι το builder object μας κάνει όλες τις απαραίτητες ενέργειες για να συγκρίνει τα δεδομένα και πάρει την απόφαση αν πρέπει η όχι να προσθέσει το alert στο σύστημά μας.

3.2.3 Factory Method

Σκοπός

Ορίζει μια διεπαφή για τη δημιουργία ενός αντικειμένου, αλλά αφήνει τις υποκλάσεις να αποφασίσουν ποια κλάσει θα αρχηκοποιηθεί επιπρόσθετα η factory method δίνει την δυνατότητα σε μια κλάση να αποδέχεται instances από υποκλάσεις, αυτό γίνετε ορίζοντας ένα "εικονικό" constructor με αποτέλεσμα το keyword new να μην λειτουργεί.

Πρόβλημα

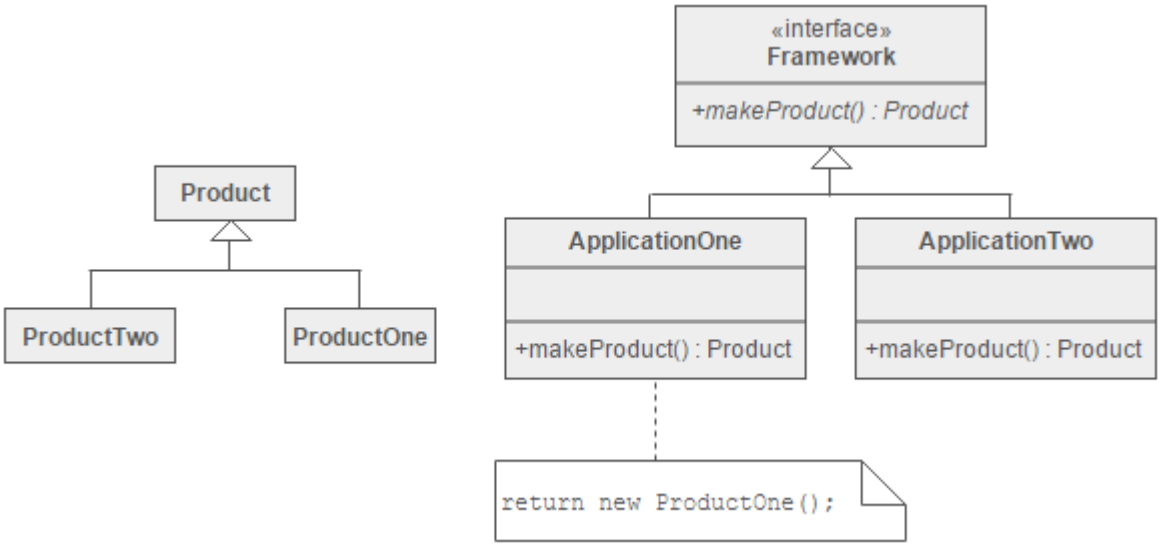
Ένα framework θα πρέπει να τυποποιήσει το αρχιτεκτονικό του μοντέλο για ένα ευρύ φάσμα εφαρμογών, αλλά και να επιτρέψει για μεμονωμένες εφαρμογές να καθορίσουν τα δικά τους αντικείμενα παρέχοντάς τους συγκεκριμενοποίηση (instantiation).

Λύση

Η Factory Method [31] μπορεί να δημιουργεί αντικείμενα, όπως η Template Method μπορεί να εφαρμόσει έναν αλγόριθμο. Μια υπερκλάση καθορίζει όλες τις τυπικές και γενικές συμπεριφορές (χρησιμοποιώντας εικονικά " placeholders"), και στη συνέχεια ενημερώνει με όλες τις λεπτομέρειες της δημιουργίας τις υποκλάσεις που παρέχονται από την εφαρμογή. Η Factory Method κάνει ένα design πιο προσαρμόσιμο και με λιγότερη πολυπλοκότητα. Άλλα design patters απαιτούν την δημιουργία πολλών νέων κλάσεων ενώ η Factory Method απαιτεί μόνο μια νέα λειτουργία. Οι άνθρωποι χρησιμοποιούν συχνά την Factory Method ως κοινό τρόπο για να αρχικοποιήσουν αντικείμενα, αλλά αυτό δεν είναι απαραίτητο εάν η κλάση που πρόκειτε να αρχικοποιηθεί δεν θα αλλάζει ποτέ.Η αρχικοποίηση λαμβάνει χώρα σε μια διαδικασία που οι υποκλάσεις μπορούν εύκολα να παρακάμφθουν (όπως μια λειτουργία προετοιμασίας συστήματος). Η Factory Method είναι παρόμοια με το Abstract Factory, αλλά χωρίς την έμφαση στις οικογένειες κλάσεων. Οι Factory Methods είναι συνήθως προσανατολισμένες για ένα framework, και στη συνέχεια ο σκοπός τους είναι να εφαρμοστούν από τον χρήστη του framework.

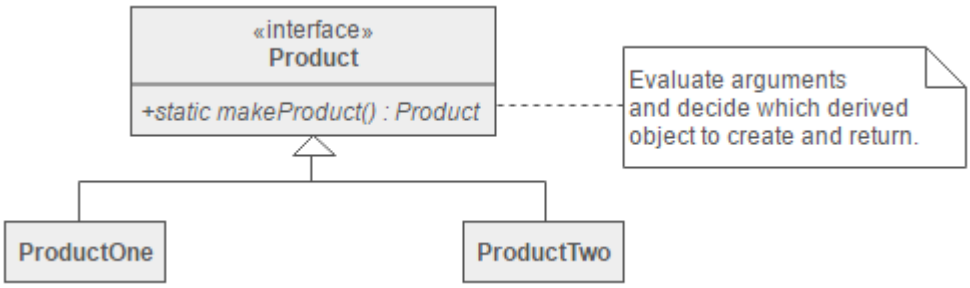
Δομή

Η εφαρμογή της Factory Method συμπίπτει σε μεγάλο βαθμό με εκείνη του Abstract Factory. Για το λόγο αυτό, η παρουσίαση στο παρόν κεφάλαιο επικεντρώνεται στην προσέγγιση που έχει γίνει δημοφιλής από τότε.



Σχήμα 3.2.1: Παράδειγμα Factory Method.

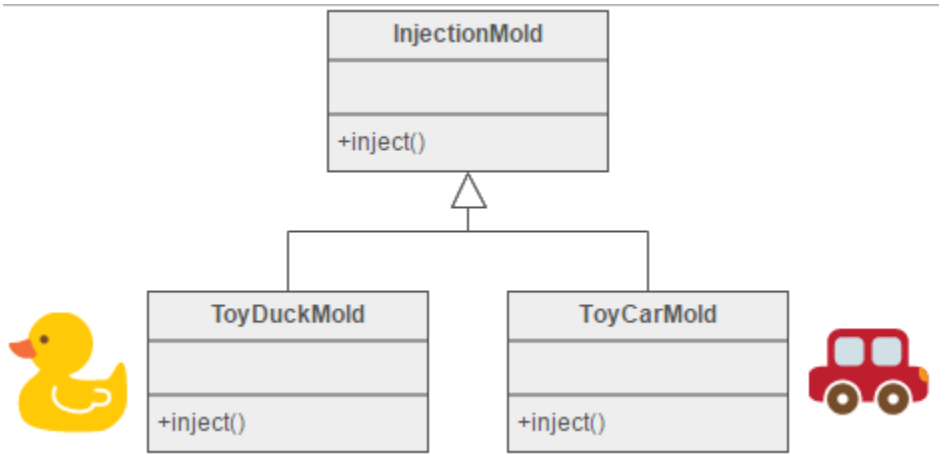
Ο αυξανόμενος και πιο δημοφιλής ορισμός της Factory Method είναι: μια στατική μέθοδο μιας κλάσεις που επιστρέφει ένα αντικείμενο αυτής της κλάσης. Σε αντίθεση με έναν constructor, το αντικείμενο επιστρέφει μπορεί να είναι ένα instance μιας υποκλάσης. Ακόμα ένα υπάρχον αντικείμενο που δημιουργήθηκε από έναν μια Factory Method θα μπορούσε να επαναχρησιμοποιηθεί, αντί ενός νέου αντικειμένου που δημιουργήθηκε από έναν constructor. Μια Factory Method μπορεί να έχει ποιο περιγραφικό όνομα σε σχέση με έναν constructor (π.χ. Color.make_RGB_color(float red, float green, float blue) and Color.make_HSB_color(float hue, float saturation, float brightness).



Σχήμα 3.2.2: Παράδειγμα δημιουργίας ενός product από factory method.

Παράδειγμα

Η Factory Method ορίζει μία διεπαφή για τη δημιουργία αντικειμένων, αλλά αφήνει τις υποκατηγορίες να αποφασίσουν ποιες κλάσεις να αρχικοποιηθούν. Οι πρέσες έγχυσης παρουσιάζουν αυτό το pattern. Κατασκευαστές πλαστικών παιχνιδιών διαχειρίζονται το πλαστικό σε μορφή σκόνης, και εγχέουν αυτή τη σκόνη σε καλούπια των επιθυμητών σχημάτων. Ο τύπος των παιχνιδιών (αυτοκίνητο, φιγούρα, κλπ) καθορίζεται από το καλούπι.



Σχήμα 3.2.3: Παράδειγμα κατασκευής παιχνιδιών από λιωμένο πλαστικό.

Ένα προγραμματιστικό παράδειγμα είναι το εξής

```
public interface ImageReader {
    public DecodedImage getDecodedImage();
}

public class GifReader implements ImageReader {
    public GifReader( InputStream in ) {
        // check that it's a gif, throw exception if it's not, then if it is decode it.
    }

    public DecodedImage getDecodedImage() {
       return decodedImage;
    }
}

public class JpegReader implements ImageReader {
    //...
}
```

Σχήμα 3.2.4: Παράδειγμα ενός image reader.

3.2.3 Object Pool

Σκοπός

Σκοπός του object pool [16] pattern είναι μπορεί να προσφέρει μια σημαντική αύξηση της απόδοσης είναι πιο αποτελεσματικό σε περιπτώσεις όπου το κόστος της αρχικοποίησης μιας κλάσης είναι υψηλό για παράδειγμα, ο ρυθμός δημιουργίας μιας κλάσης είναι υψηλή, και ο αριθμός των instantiations που είναι σε χρήση οποιαδήποτε στιγμή είναι χαμηλή.

Πρόβλημα

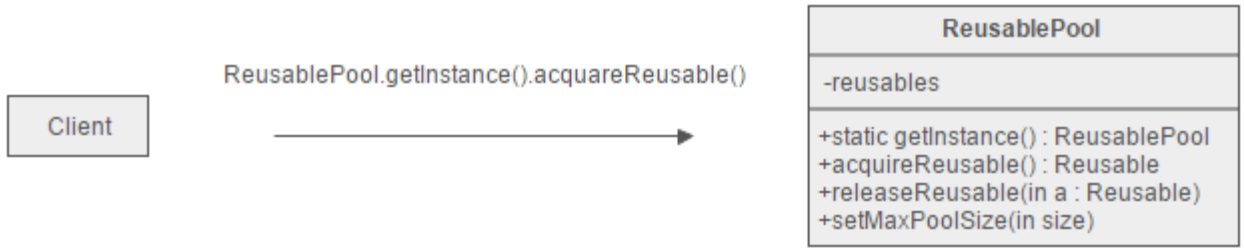
Το object pool [32] (η αλλιώς γνωστά ως resource pools) χρησιμοποιούνται για τη διαχείριση της προσωρινής μνήμης. Ένας πελάτης που έχει πρόσβαση σε ένα object pool μπορεί να αποφύγει τη δημιουργία νέων αντικειμένων από αυτόν απλά ζητώντας από το object pool να του δοθεί ένα που είναι ήδη αρχικοποιημένο. Σε γενικές γραμμές το object pool θα είναι μια αυξανόμενη «πισίνα», δηλαδή η ίδια η «πισίνα» θα δημιουργήσει νέα αντικείμενα όσο είναι άδεια και θα γεμίζει, ή μπορούμε να έχουμε μια «πισίνα», η οποία περιορίζει τον αριθμό των αντικειμένων που δημιουργούνται. Είναι λογικό να κρατήσουμε όλα τα επαναχρησιμοποιούμενα αντικείμενα που δεν είναι τώρα σε χρήση στο ίδιο object pool, έτσι ώστε να μπορούν να διέπονται από την ίδια πολιτική. Για να επιτευχθεί αυτό, η κλάση Object Pool πρέπει να σχεδιαστεί για να είναι μια κλάση singleton.

Λύση

Το object pool [16] αφήνει τους χρήστες να χρησιμοποιήσουν αντικείμενα από την «πισίνα». Όταν τα αντικείμενα αυτά δεν χρειάζονται ποια επιστρέφουν στην «πισίνα» να ξαναχρησι-μοποιηθούν. Ωστόσο δεν θέλουμε μια διαδικασία να περιμένει ένα συγκεκριμένο αντι-κείμενο να αρχικοποιηθεί οπότε η «πισίνα» κρατάει ήδη αρχικοποιημένα αντικείμενα μέσα της. Βέβαια αυτό την καθιστά να έχει υψηλό κόστος πόρων και για αυτό το λόγο θα πρέπει αν έχει έναν μηχανισμό κάθαρσης αντικειμένων.

Δομή

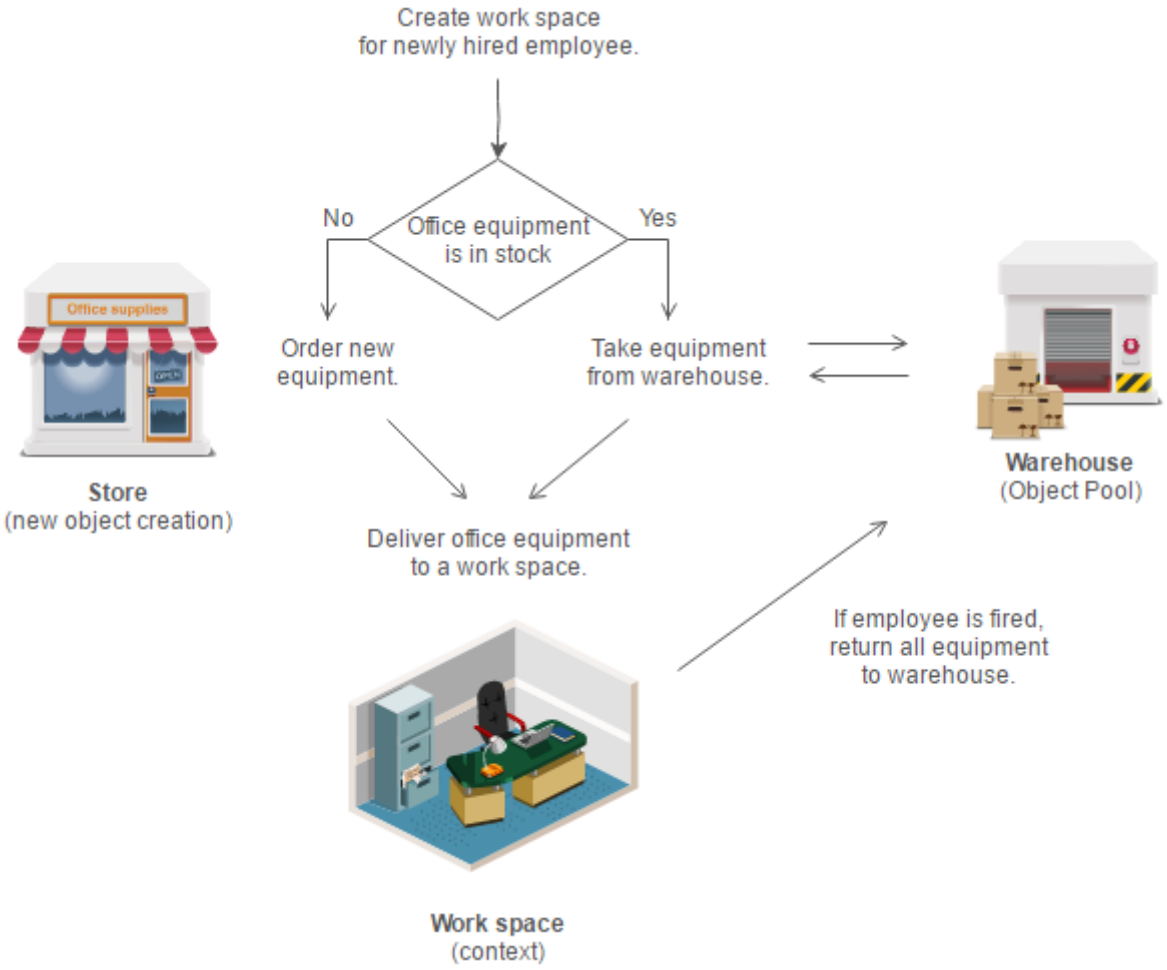
Η βασική ιδέα για των object pools είναι ότι αν ένα instance μιας κλασης μπορεί να επαναχρησιμοποιηθεί τότε αποφεύγουμε να την ξανά αρχικοποιήσουμε όταν την ξανά χρησι-μοποιήμοποιουμε.



Σχήμα 3.3.1: Ο χρήστης ζητάει από το pool ένα instance.

Παράδειγμα

Το object pool pattern μπορεί να εφαρμοστεί σε μια αποθήκη ενός γραφείου. Όταν ένας καινούργιος υπάλληλος προσληφθεί ο προϊστάμενος πρέπει να προετοιμάσει ένα νέο γραφείο για αυτόν. Κοιτάζει στην αποθήκη αν υπάρχει ο απαραίτητος εξοπλισμός γραφείου. Αν ναι τότε τον χρησιμοποιεί. Αν όχι τον παραγγέλνει από την Amazon. Τώρα που στην περίπτωση ο υπάλληλος απολυθεί η παραιτηθεί ο εξοπλισμός αυτός μεταφέρετε στην αποθήκη όπου και φυλάσσετε για την επόμενη χρήση του.



Σχήμα 3.3.2: Διαδικασία άντλησης υλικού γραφείου.

Ένα προγραμματιστικό παράδειγμα αυτού είναι σε JAVA είναι ο jdbc [33] connector που μας δίνει την δυνατότητα να έχουμε όποτε το χρειαστούμε ένα connection object για να κάνουμε σύνδεση σε μια πηγή δεδομένων.

```
// ObjectPool Class

public abstract class ObjectPool<T> {
  private long expirationTime;
```

```
private Hashtable<T, Long> locked, unlocked;

public ObjectPool() {
  expirationTime = 30000; // 30 seconds
  locked = new Hashtable<T, Long>();
  unlocked = new Hashtable<T, Long>();
}

protected abstract T create();

public abstract boolean validate(T o);

public abstract void expire(T o);

public synchronized T checkOut() {
  long now = System.currentTimeMillis();
  T t;
  if (unlocked.size() > 0) {
    Enumeration<T> e = unlocked.keys();
    while (e.hasMoreElements()) {
      t = e.nextElement();
      if ((now - unlocked.get(t)) > expirationTime) {
        // object has expired
        unlocked.remove(t);
        expire(t);
        t = null;
      } else {
        if (validate(t)) {
          unlocked.remove(t);
          locked.put(t, now);
          return (t);
        } else {
          // object failed validation
          unlocked.remove(t);
          expire(t);
          t = null;
        }
      }
    }
  }
  // no objects available, create a new one
  t = create();
  locked.put(t, now);
  return (t);
}

public synchronized void checkIn(T t) {
  locked.remove(t);
  unlocked.put(t, System.currentTimeMillis());
}
```

```
}

//The three remaining methods are abstract
//and therefore must be implemented by the subclass

public class JDBCConnectionPool extends ObjectPool<Connection> {

  private String dsn, usr, pwd;

  public JDBCConnectionPool(String driver, String dsn, String usr, String pwd) {
    super();
    try {
      Class.forName(driver).newInstance();
    } catch (Exception e) {
      e.printStackTrace();
    }
    this.dsn = dsn;
    this.usr = usr;
    this.pwd = pwd;
  }

  @Override
  protected Connection create() {
    try {
      return (DriverManager.getConnection(dsn, usr, pwd));
    } catch (SQLException e) {
      e.printStackTrace();
      return (null);
    }
  }

  @Override
  public void expire(Connection o) {
    try {
      ((Connection) o).close();
    } catch (SQLException e) {
      e.printStackTrace();
    }
  }

  @Override
  public boolean validate(Connection o) {
    try {
      return (!((Connection) o).isClosed());
    } catch (SQLException e) {
      e.printStackTrace();
      return (false);
    }
  }
}
```

Σχήμα 3.3.4: JDBC pool creation.

Ο JDBCConnection Pool δίνει την δυνατότητα στην εφαρμογή να δανειστεί ένα database αντικείμενο και να το επιστρέψει μόλις τελειώσει.

```
public class Main {
  public static void main(String args[]) {
    // Do something...
    ...

    // Create the ConnectionPool:
    JDBCConnectionPool pool = new JDBCConnectionPool(
      "org.hsqldb.jdbcDriver", "jdbc:hsqldb://localhost/mydb",
      "sa", "secret");

    // Get a connection:
    Connection con = pool.checkOut();

    // Use the connection
    ...

    // Return the connection:
    pool.checkIn(con);

  }
}
```

Σχήμα 3.3.5: Πως να χρησιμοποιήσουμε τον JDBC .

3.2.4 Prototype

Σκοπός

Καθορίζει τα είδη των αντικειμένων που θα δημιουργήσει χρησιμοποιώντας ένα prototypical instance . Τα αντικείμενα αυτά θα δημιουργηθούν αντιγράφοντας το prototype. Επιλέγετε ένα instance μιας κλάσης για να γίνει δημιουργός όλων των μελλοντικών instance

που θα παραχθούν.

Πρόβλημα

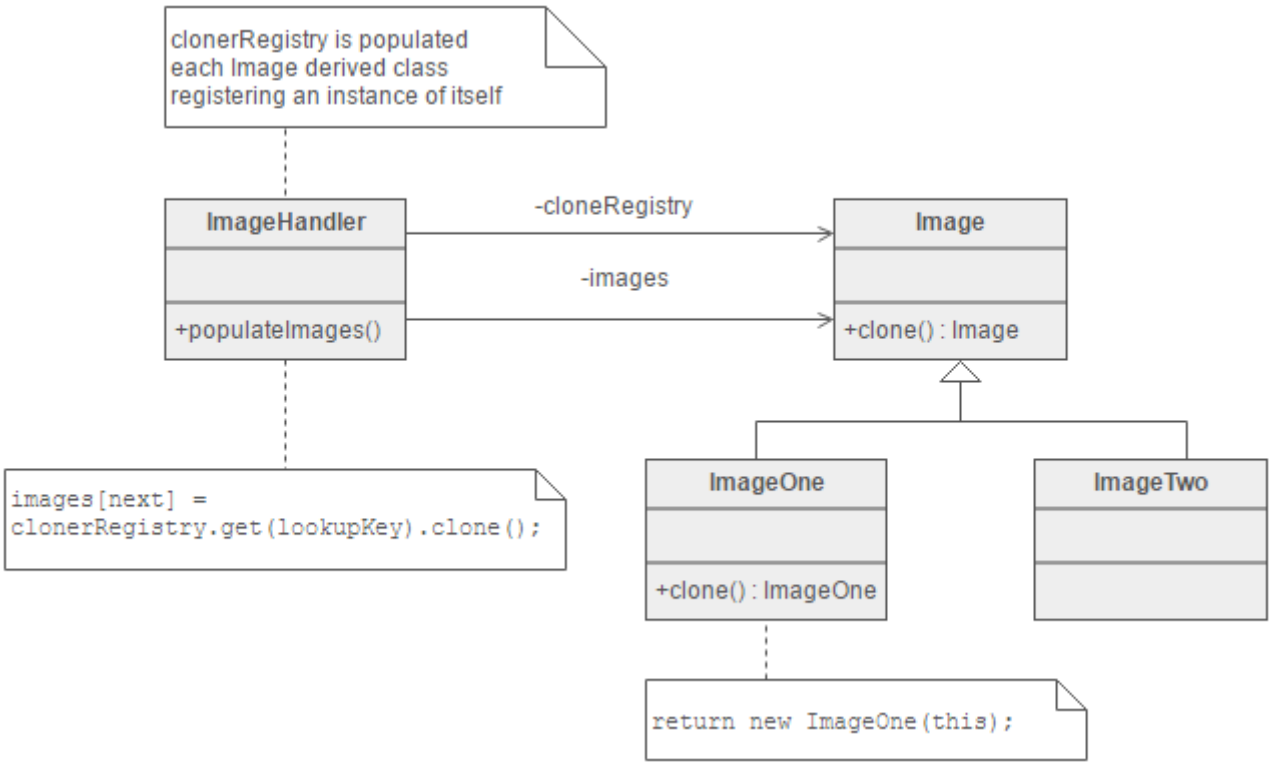
Για την δημιουργία ενός object πρέπει να γραφεί μέσα στον κώδικα η κλάση του και να αρχικοποιηθεί με το keyword new.

Λύση

Δηλώνουμε μια βασική κλάση που εμπεριέχει μια εικονική μέθοδο «clone» όπου και διατηρεί ένα αποθετήριο όλων στον κλάσεων που είναι συσχετισμένες με την βασική κλάση και μπορούν να είναι «clonable». Οποιαδήποτε λοιπόν κλάση θέλει να έχει ένα πολυμορφικό constructor και να συσχετιστεί με την βασική κλάση για να γινει «clonable» πρέπει να κανει τα εξής. Να προέρχεται από την βασική κλάση, να κάνει εγγραφή στο «prototypical instance» και να υλοποιήσει την μέθοδο «clone». Στην συνέχεια ο πελάτης αντί να αρχικοποιήσει την κλάση χρησιμοποιώντας το keword new καλεί την «clone» μέθοδο παρέχοντάς της τις κατάλληλες πληροφορίες για το τη είδους κλάση θα αρχικοποιήσει, Με την σειρά της η μέθοδος θα του επιστρέψει το κατάλληλο αρχικοποιημένο αντικείμενο με όλες τις απαραίτητες ιδιότητες που είναι συσχετισμένες με την βασική κλάση.

Δομή

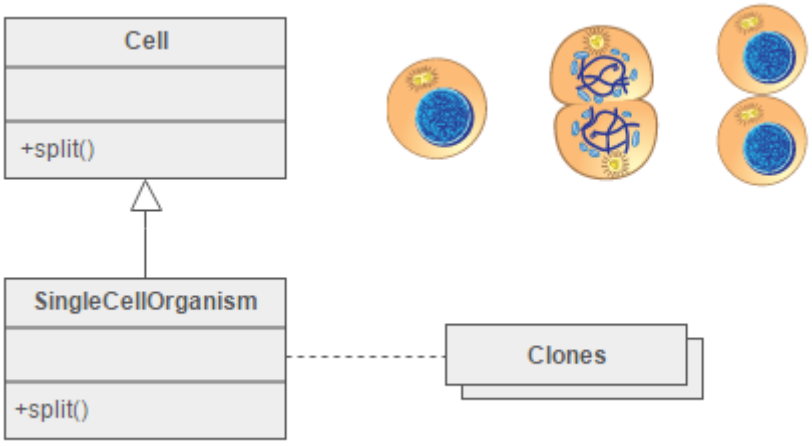
Το Factory γνωρίζει πως θα βρει το σωστό Prototype, και κάθε προϊόν γνωρίζει πως να δημιουργεί καινούργια instances από αυτό.



Σχήμα 3.4.1: Εύρεση του σωστού prototype για να διαχειριστούμε μια εικόνα.

Παράδειγμα

Το prototype pattern [33] καθορίζει το είδος των αντικειμένων που θα δημιουργήσει χρησιμοποιώντας το prototypical instance. Η προτυποποίηση των καινούργιων προϊόντων γίνετε πριν μπουν στην παραγωγή. Σε αυτό το παράδειγμα το πρωτότυπο είναι παθητικό και δεν συμμετέχει στην κλωνοποίηση του. Η μιτωτική διαίρεση ενός κυττάρου καταλήγει σε δυο πανομοιότυπα κύτταρα που είναι ένα παράδειγμα prototype που παίζει ενεργό ρόλο στην αντιγραφή. Όταν ένα κύτταρο διαιρείτε σε δύο κύτταρα με πανομοιότυπο γονιδίωμα δηλαδή με άλλα λόγια κλωνοποιεί τον εαυτό του.



Σχήμα 3.4.2: Διαδικασία μίτωσης ενός κυττάρου και πως αναπαριστάτε.

Ένα προγραμματιστικό παράδειγμα σε JAVA είναι:

```
interface Prototype {
  Object clone();
  String getName();
}
// 1. The clone() contract
interface Command {
  void execute();
}

class PrototypesModule {
  // 2. "registry" of prototypical objs
  private static Prototype[] prototypes = new Prototype[9];
  private static int total = 0;

  // Adds a feature to the Prototype attribute of the PrototypesModule class
  // obj  The feature to be added to the Prototype attribute
  public static void addPrototype( Prototype obj ) {
    prototypes[total++] = obj;
  }

  public static Object findAndClone( String name ) {
    // 4. The "virtual ctor"
    for ( int i = 0; i < total; i++ ) {
      if ( prototypes[i].getName().equals( name ) ) {
        return prototypes[i].clone();
      }
```

```
    }
    System.out.println( name + " not found" );
    return null;
  }
}

// 5. Sign-up for the clone() contract.
// Each class calls "new" on itself FOR the client.
class This implements Prototype, Command {
  public Object clone() {
    return new This();
  }
  public String getName() {
    return "This";
  }
  public void execute() {
    System.out.println( "This: execute" );
  }
}

class That implements Prototype, Command {
  public Object clone() {
    return new That();
  }
  public String getName() {
    return "That";
  }
  public void execute() {
    System.out.println( "That: execute" );
  }
}

class TheOther implements Prototype, Command {
  public Object clone() {
    return new TheOther();
  }
  public String getName() {
    return "TheOther";
  }
  public void execute() {
    System.out.println( "TheOther: execute" );
  }
}


public class PrototypeDemo {
  // 3. Populate the "registry"
  public static void initializePrototypes() {
    PrototypesModule.addPrototype( new This() );
    PrototypesModule.addPrototype( new That() );
    PrototypesModule.addPrototype( new TheOther() );
  }
  public static void main( String[] args ) {
    initializePrototypes();
```

```
    Object[] objects = new Object[9];
    int total = 0;

    // 6. Client does not use "new"
    for (int i=0; i < args.length; i++) {
      objects[total] = PrototypesModule.findAndClone( args[i] );
      if (objects[total] != null) total++;
    }
    for (int i=0; i < total; i++) {
      ((Command)objects[i]).execute();
    }
  }
}
```

Σχήμα 3.4.3: Χρησιμοποιώντας το prototype σε JAVA.

Εδώ βλέπουμε αρχικά τα αντικείμενα να δηλώνονται κατά το initialization και στην συνέχεια να κλωνοποιούνται μέσα στην εφαρμογή για περεταίρω επεξεργασία. Ας πούμε για παράδειγμα θα μπορούσε να ήταν κάποια προϊόντα της ιδίας ομάδας. Πχ θα μπορούσε να ήταν μακαρόνια σε ένα σούπερ μάρκετ. Τα μακαρόνια θα προθέτονταν στον Prototype κατά το initialization και θα μπορούσαν να κλωνοποιηθούν σαν μακαρόνια μισκο μπαριλα κλπ.

3.2.5 Singleton

Σκοπός

Ενθυλακώνει μια κλάση και δημιούργει ένα instance αυτής μέσω ενός σημείου η μεθόδου που είναι ευρέως γνωστό μέσα στον κώδικα. Ακόμα ενθυλακώνει την λογική "just-in-time initialization" ή "initialization on first use".

Πρόβλημα

Μία εφαρμογή έχει την αναγκη μόνο ενός instance από ένα αντικείμενο και επιπρόσθετα να είναι ευρέως γνωστό και προσπελάσιμο αυτό το αντικείμενο μέσα στον κώδικα.

Λύση

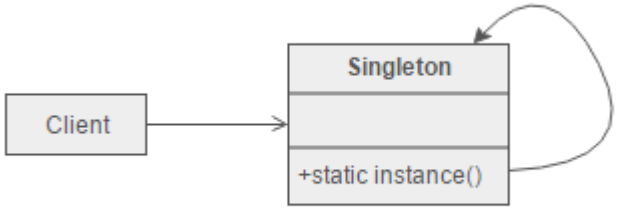
Κάνουμε την κλάση να έχει μόνο ένα αντικείμενο το οποίο θα είναι υπεύθυνο για την δημιουργία του την προσπέλαση του. Δημιουργούμε το αντικείμενο σαν private static data member. Παρέχουμε στην εφαρμογή μια μέθοδο που ενθυλακώνει όλη την λειτουργικότητα της αρχικοποίησης και ταυτόχρονα επιτρέπει στην εφαρμογή την προσπέλαση όλων των μεθόδων του. Η εφαρμογή καλεί αυτή την μέθοδο οπότε χρειάζεται να χρησιμοποιήσει αυτό το αντικείμενο χωρίς να το αρχικοποήσει από με το keyword new. Κάθε φορά που την καλεί χρησιμοποιεί δεν αρχικοποιήτε καινούργιο αντικείμενο άλλα το ήδη υπάρχον. Οπότε με αυτό τον τρόπο καταφέρνουμε να έχουμε πάντα ένα αντικείμενο αυτού. Το singleton pattern θα πρέπει να το χρησιμοποιούμε όταν υφίστανται τα παρακάτω κριτήρια:

- Τα δικαιώματα του μοναδικού instance δεν μπορούν να μεταφερθούν στην εφαρμογή.
- Το initialization γίνετε σε ξεχωριστό χρόνο από την εφαρμογή.
- Τα δικαιώματα του instance δεν μπορούν αν είναι global.

Αν η ιδιοκτησία του singleton [16] για το πότε και με ποιο τρόπο θα γίνει η αρχικοποίηση δεν είναι πρόβλημα τότε η επιλογή του singleton δεν είναι και η καλύτερη. Το singleton δεν

μπορεί να υποστηρίζει υποκλάσεις οπότε αν υπάρχει τέτοια πρόθεση το singleton πάλι δεν το επιλέγουμε. Ακόμα το διαγράψουμε ένα singleton δεν δημιουργεί σχεδιαστικό πρόβλημα.

Δομή



Σχήμα 3.5.1: Ο χρήστης ζητάει ένα και μόνο singleton.

Φτιάχνουμε μια κλάση και κάνουμε μια στατική μέθοδο «initialization on first use». Το μοναδικό instance είναι μια ιδιότητα της κλάσης και για αυτό το λόγο είναι και μοναδικά το instance. Στην συνέχεια δημιουργούμε μια δημόσια στατική μέθοδο που επιστρέφει αυτό το instance.

Σχήμα 3.5.2: Οι δυο μέθοδοι ενός singleton.

Παράδειγμα

Το singleton είναι ένα σχετικά εύκολο pattern και δεν απαιτεί ιδιαίτερη πολυπλοκότητα. Στην JAVA μπορεί να υλοποιηθεί ως εξής:

```
public class Singleton {
  // Private constructor prevents instantiation from other classes
  private Singleton() {}

  /**
   * SingletonHolder is loaded on the first execution of Singleton.getInstance()
   * or the first access to SingletonHolder.INSTANCE, not before.
   */
  private static class SingletonHolder {
    private static final Singleton INSTANCE = new Singleton();
  }

  public static Singleton getInstance() {
    return SingletonHolder.INSTANCE;
  }
}
```

Σχήμα 3.5.3: Ένα τυπικό παράδειγμα singleton.

Εδώ βλέπουμε ότι μπορεί το singleton να αρχικοποιηθεί κατά την εκτέλεση της getInstance() μόνο μια φορά. Στην εφαρμογή μας χρησιμοποιούμε το singleton pattern για να έχουμε ένα instance του connection. Στην περίπτωση αυτή είναι ένα ο χρήστης μας είναι το dao μας που ζητάει από το singleton το οποίο το ζητάει από το object pool του JNDI.

```
public class ConnectionInstance implements DataBaseProperties{
    private static Connection con;


    public static void newInstance() {
        MysqlDataSource mysqlDS = new MysqlDataSource();
        mysqlDS.setURL(MYSQL_DB_URL);
        mysqlDS.setUser(MYSQL_DB_USERNAME);
        mysqlDS.setPassword(MYSQL_DB_PASSWORD);
        try {
            con = mysqlDS.getConnection();
        } catch (SQLException e) {
            e.printStackTrace();
        }

    }

    public static Connection getConnection(){
```

```
            return con;
        }


}
```

Σχήμα 3.5.4: Δίνοντας μόνο ένα connection μέσα από το ConnectionInstance singleton.

3.3 Δομικά πρότυπα (Structural patterns)

Στην Τεχνολογία Λογισμικού, τα Μελέτη Structural patterns είναι Design Patterns, που διευκολύνουν το σχεδιασμό, υλοποιώντας έναν απλό τρόπο για να συνδεθούν οι σχέσεις των οντοτήτων μεταξύ τους. Μερικά από αυτά είναι:

- Adapter
  Διασυνδέει τα interfaces από διαφορετικές κλάσεις
- Decorator
  Δίνει αρμοδιότητες σε αντικείμενα με δυναμικό τρόπο.
- Proxy
  Ένα αντικείμενο που εκπροσωπεί ένα άλλο
- Bridge
  Διαχωρίζει ένα αντικείμενο από την εφαρμογή του.

3.3.1 Adapter

Σκοπός

Ο σκοπός του adapter pattern [16] είναι να μετατρέπει μια διεπαφή μιας κλάσης σε μία άλλη διεπαφή που η εφαρμογή χρειάζεται. «Τυλίγει» ουσιαστικά μια υπάρχουσα κλάση και προσφέρει στο σύστημα μια διεπαφή για αυτήν.

Πρόβλημα

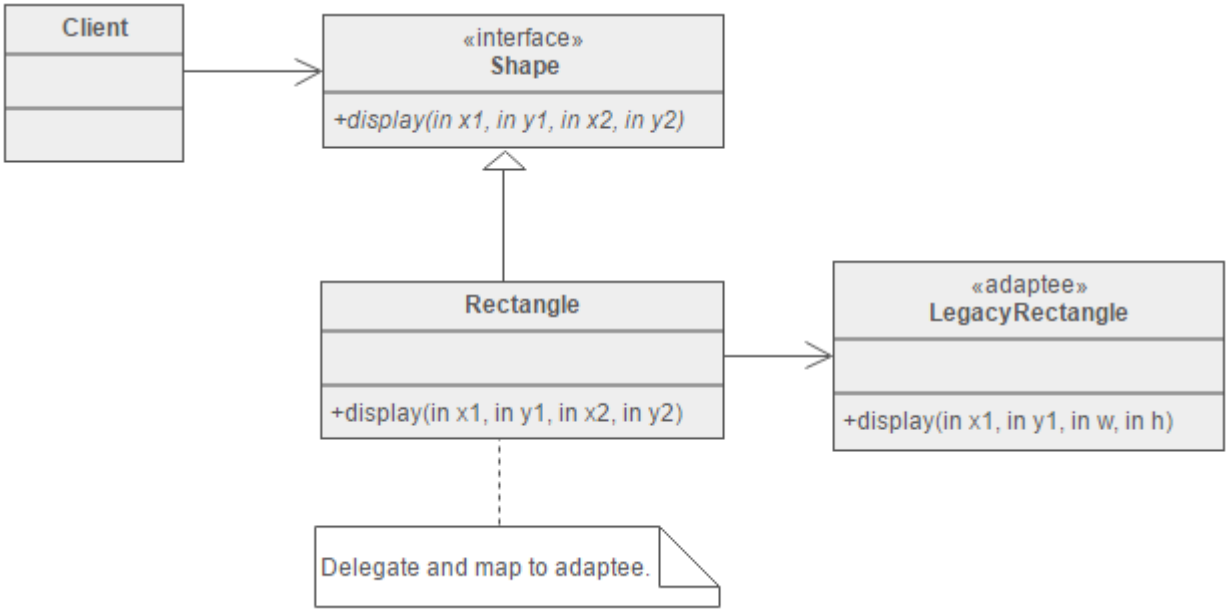
Ένα component έχει μια λειτουργικότητα η οποία είναι αναγκαστική και δεν μπορεί να αλλάξει. Θέλουμε να το χρησιμοποιήσουμε ξανά και ξανά αλλά δεν συμβατό με την υπάρχουσα φιλοσοφία του συστήματος μας.

Λύση

Η επαναχρησιμοποίηση πραγμάτων που έχουν υλοποιηθεί στο παρελθόν και δεν έχουν σχεδιαστεί από εμάς είναι πάντα οδυνηρή. Ένας από τους λόγους που δυσκολευόμαστε για αυτό είναι να παντρέψουμε κάτι καινούργιο με κάτι που έχει γραφεί από πριν. Υπάρχει πάντα κάτι δεν πάει καλά μεταξύ της παλαιάς και της νέας τεχνολογίας. Είναι σαν το πρόβλημα της ενσωμάτωσης μιας νέας πρίζας τριών ακροδεκτών σε μία παλιά πρίζα δισχιδής χρειαζόμαστε ένα είδους adapter ή ενδιάμεσων μέσων για να μπορέσουμε να τα ενώσουμε. Ο adapter πρόκειται να δημιουργήσει ένα ενδιάμεσο μέσο που μεταφράζει η απεικονίζει το παλιό στοιχείο και το κάνει συμβατό με το νέο σύστημα. Το σύστημα καλεί τις μεθόδους του adapter και αυτές μεταφράζονται σε κλήσεις στο παλιό στοιχείο που χρησιμοποιούμε. Αυτό μπορούμε να το πετύχουμε είτε με κληρονομικότητα (inheritance) είτε με συνάθροιση (aggregation).

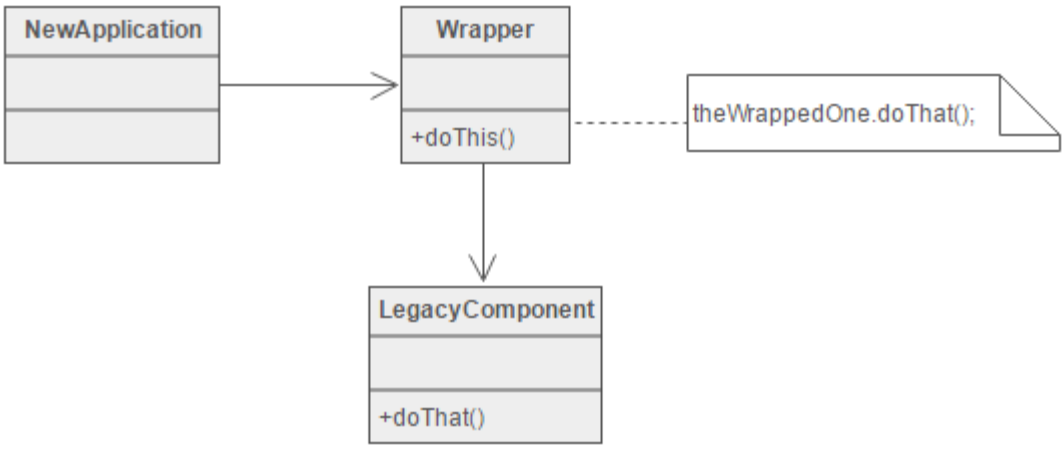
Δομή

Παρακάτω, η μέθοδος εμφάνισης ενός ορθογώνιου display() αναμένει να λάβει "x, y, w, h" παραμέτρους. Αλλά η εφαρμογή θέλει να περάσει "πάνω αριστερά x και y" και "κάτω δεξιά x και y". Αυτή η ασυμφωνία μπορεί να συμβιβαστεί με την προσθήκη ενός επιπλέον επιπέδου indirection – δηλαδή ενός adapter.



Σχήμα 3.6.1: Η προσαρμογή ενός ενδιάμεσου adapter

Ο προσαρμογέας θα μπορούσε επίσης να θεωρηθεί ως ένα «wrapper».



Σχήμα 3.6.2: Από την σκοπιά του adapter ως wrapper

Ο adapter επιτρέπει διαφορετικές ασύμβατες κλάσεις με το σύστημα να εργαστούν από κοινού με τη μετατροπή της διεπαφής τους σε μια διεπαφή που αναμένεται από το σύστημα. Ένα προγραμματιστικό παράδειγμα σε JAVA που φαίνεται η παλιά κλάση πως ενσωματώνεται με την χρήση του adapter.

```
/* The OLD */
class SquarePeg {
   private double width;
   public SquarePeg( double w )       { width = w; }
   public double getWidth()           { return width; }
   public void   setWidth( double w ) { width = w; }
}

/* The NEW */
class RoundHole {
   private int radius;
   public RoundHole( int r ) {
      radius = r;
      System.out.println( "RoundHole: max SquarePeg is " + r * Math.sqrt(2) );
   }
   public int getRadius() { return radius; }
}

// Design a "wrapper" class that can "impedance match" the old to the new
class SquarePegAdapter {

   // The adapter/wrapper class "has a" instance of the legacy class
   private SquarePeg sp;

   public SquarePegAdapter( double w ) { sp = new SquarePeg( w ); }

   // Identify the desired interface
   public void makeFit( RoundHole rh ) {
      // The adapter/wrapper class delegates to the legacy object
      double amount = sp.getWidth() - rh.getRadius() * Math.sqrt(2);
      System.out.println( "reducing SquarePeg " + sp.getWidth() + " by " + ((amount < 0) ? 0 :
amount) + " amount" );
      if (amount > 0) {
         sp.setWidth( sp.getWidth() - amount );
         System.out.println( "   width is now " + sp.getWidth() );
      }
   }
}
```

```
class AdapterDemoSquarePeg {
   public static void main( String[] args ) {
      RoundHole        rh = new RoundHole( 5 );
      SquarePegAdapter spa;

      for (int i=6; i < 10; i++) {
         spa = new SquarePegAdapter( (double) i );
         // The client uses (is coupled to) the new interface
         spa.makeFit( rh );
      }
   }
}
```

Σχήμα 3.6.3: Ένα παράδειγμα σε JAVA προσαρμογής παραθύρων

Το παραπάνω μας δίνει το αποτέλεσμα:

```
RoundHole: max SquarePeg is 7.0710678118
reducing SquarePeg 6.0 by 0.0 amount
reducing SquarePeg 7.0 by 0.0 amount
reducing SquarePeg 8.0 by 0.9289321881345245 amount
   width is now 7.0710678118654755
reducing SquarePeg 9.0 by 1.9289321881345245 amount
   width is now 7.0710678118654755
```

Σχήμα 3.6.4: Αποτέλεσμα παραδείγματος

3.3.2 Decorator

Σκοπός

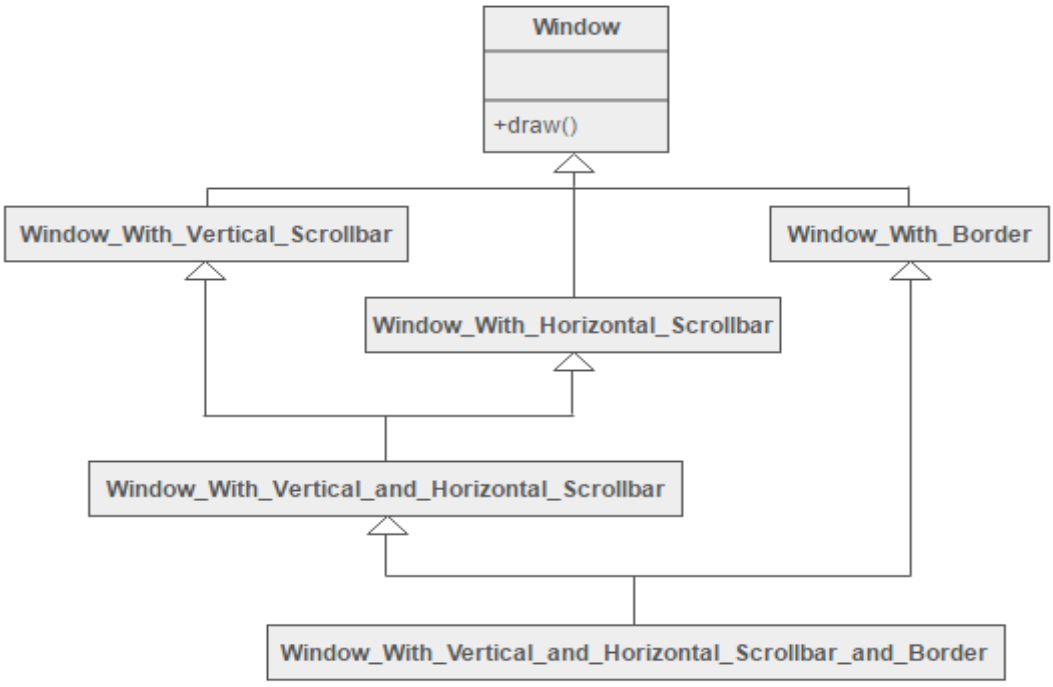
Το decorator pattern [16] μας δίνει την δυνατότητα να δώσουμε σε ένα αντικείμενο επιπλέων λειτουργικότητες. Είναι μία εναλλακτική λύση της κληρονομικότητας και δίνει επεκτασιμότητα στο αντικείμενο. Είναι σαν να τυλίγουμε ένα δώρο να το βάζουμε σε ένα κουτί και μετά να τυλίγουμε το κουτί.

Πρόβλημα

Θέλουμε να προσθέσουμε λειτουργικότητα σε ένα αντικείμενο κατά την εκτέλεση του. Το να του την δώσουμε με κληρονομικότητα δεν είναι εφικτό γιατί πρέπει να κληρονομήσουμε όλη την κλάση από πριν και δεν γίνετε αυτό στην στο χρόνο της εκτέλεσης του αντικειμένου.

Λύση

Ας υποθέσουμε ότι εργάζεστε σε μία οθόνη χρήστη και θέλετε να υποστηρίξετε την προσθήκη των συνόρων της οθόνης καθώς και τις γραμμές κύλισης του παράθυρου. Θα μπορούσατε να ορίσετε μια ιεραρχία κληρονομικότητας, όπως



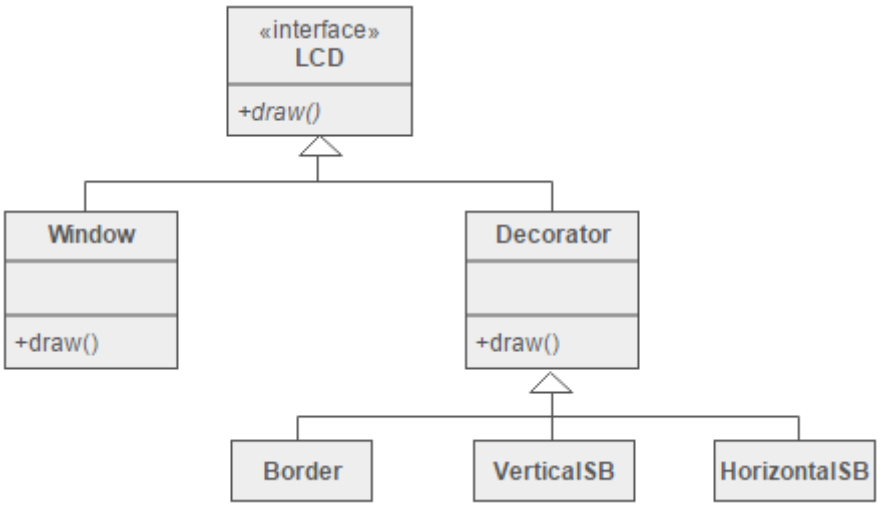
Σχήμα 3.7.1: Διαδικασία διακόσμησης παραθύρου ιεραρχία

Αλλά το decorator pattern δίνει στον πελάτη τη δυνατότητα να καθορίσει ο ίδιος, τι συνδυασμούς - χαρακτηριστικά επιθυμεί.

```
Widget* aWidget = new BorderDecorator(
  new HorizontalScrollBarDecorator(
    new VerticalScrollBarDecorator(
      new Window( 80, 24 ))));
aWidget->draw();
```

Σχήμα 3.7.2: Παράδειγμα διακόσμησης παραθύρου προγραμματιστικά

Αυτή η ευελιξία μπορεί να επιτευχθεί με το ακόλουθο σχέδιο



Σχήμα 3.7.3: Αποτέλεσμα οθόνης LCD και πως γράφει ο decoratore

Ένα άλλο παράδειγμα επικαλυπτόμενων (cascading) ιδιοτήτων που έχουν συνδυαστεί για να παράξουν ένα προσαρμοσμένο στο χρήστη αντικείμενο είναι ως εξής

```
Stream* aStream = new CompressingStream(
  new ASCII7Stream(
    new FileStream("fileName.dat")));
aStream->putString( "Hello world" );
```

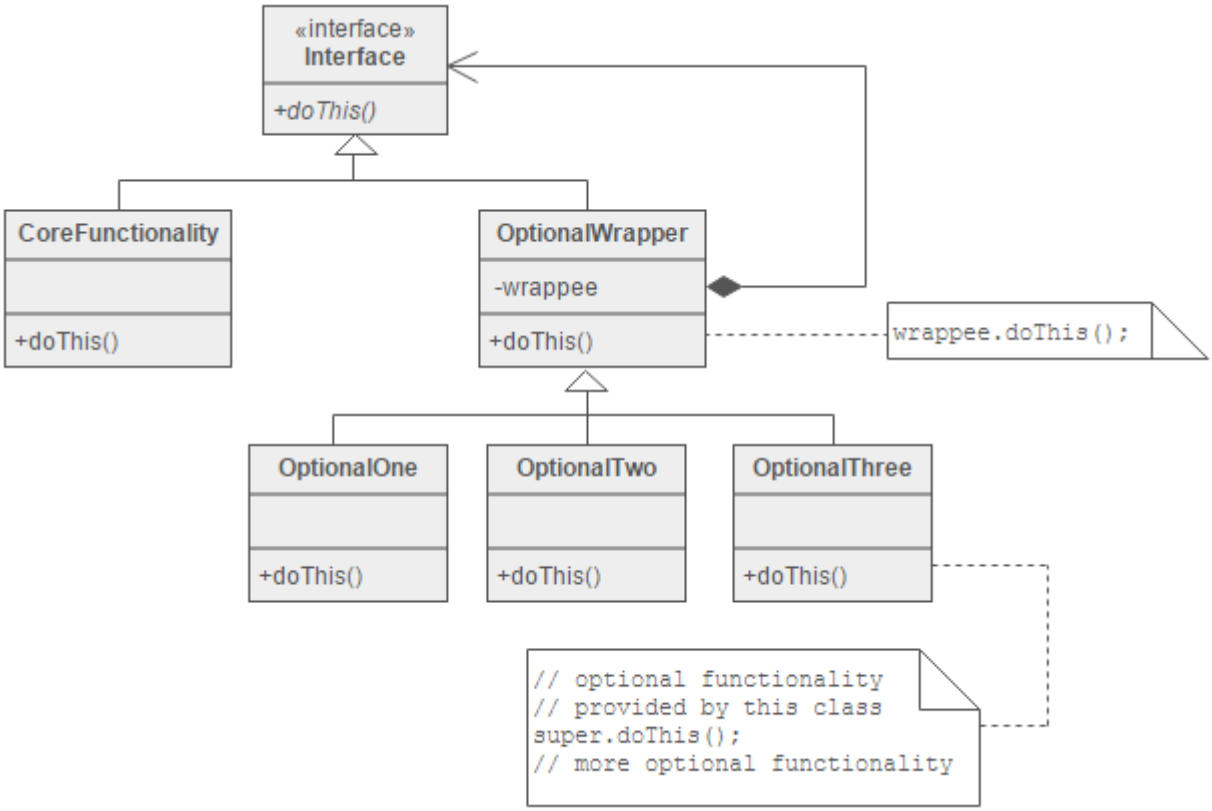
Σχήμα 3.7.4: Δίαβασμα ενός φακέλου από το σύστημα

Η λύση σε αυτή την κατηγορία των προβλημάτων περιλαμβάνει την ενθυλάκωση του αρχικού αντικειμένου μέσα σε μία διεπαφή τύπου wrapper. Να σημειώσουμε ότι αυτό το pattern δίνει ευθύνες σε ένα αντικείμενο και όχι μεθόδους στην διεπαφή του αντικειμένου. Η διεπαφή που φαίνεται στον χρήστη πρέπει να παραμένει σταθερή όσο προσδιορίζονται διαδοχικά layers.

Επίσης, να σημειώσουμε ότι η δομή του πυρήνα του αντικειμένου έχει «κρυφτεί» στο εσωτερικό ενός decorator object. Οπότε το να έχουμε άμεση πρόσβαση στον πυρήνα του αντικείμενου είναι ένα πρόβλημα.

Δομή

Ο πελάτης ενδιαφέρετε για την CoreFunctionality.doThis(). Ο πελάτης μπορεί να ενδιαφέρετε ή και όχι για την OptionalOne.doThis() και OptionalTwo.doThis(). Κάθε μια από αυτές τις κλάσης έχουν ανατεθεί στην Decorator base κλάση και η κλάση αυτή αντιπροσωπεύεται από το wrapped αντικείμενο.

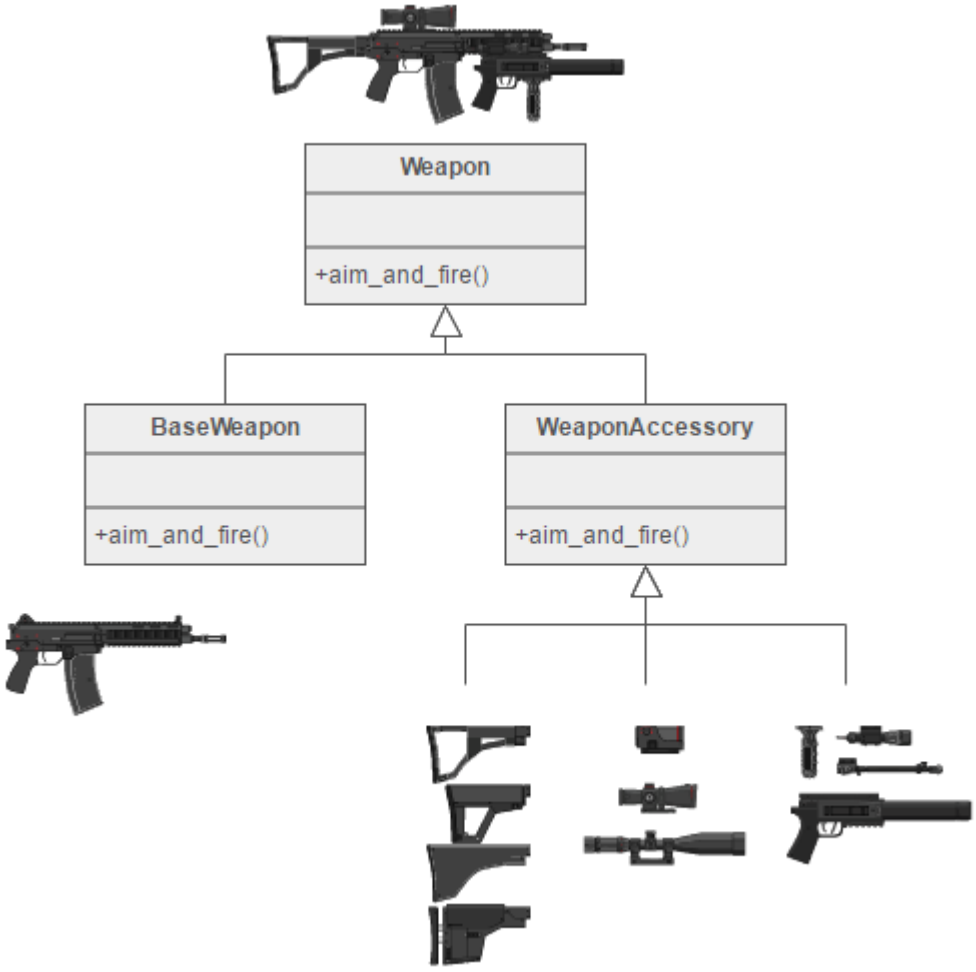


Σχήμα 3.7.5: Αντιπροσώπευση του dothis σε διάφορα αντικείμενα

Παράδειγμα

Το decorator pattern μας δίνει την δυνατότητα να δώσουμε σε ένα αντικείμενο

επιπλέων λειτουργικότητες. Τα στολίδια που προστίθενται σε ένα πεύκο ή ένα έλατο είναι παραδείγματα decorator. Φώτα, γιρλάντα, καραμέλες, διακοσμητικά από γυαλί, κλπ, μπορούν να προστεθούν σε ένα δέντρο για να του δώσουν μια εορταστική εμφάνιση. Τα στολίδια δεν αλλάζουν το δέντρο το οποίο είναι ένα χριστουγεννιάτικο δέντρο, ανεξάρτητα από τα στολίδια που θα χρησιμοποιηθούν. Ως παράδειγμα επιπρόσθετης λειτουργικότητας είναι ότι η προσθήκη των φώτων επιτρέπει σε κάποιον να «ανάψει» ένα χριστουγεννιάτικο δέντρο. Ένα άλλο παράδειγμα μπορεί να είναι το όπλο ενός δολοφόνου είναι από μόνο του φονικό όπλο. Αλλά μπορούμε να του προσθέσουμε διόπτρα σιγαστήρα κλπ. και μετά να γίνει ποιο φονικό δηλαδή να αυξηθεί η ακρίβειά του και να γίνει ποιο αθόρυβο.



Σχήμα 3.7.6: Το όπλο του εκτελεστή και οι βελτιώσεις του

Ένα προγραμματιστικό παράδειγμα σε JAVA είναι:

```
public class DecoratorBefore {

static class A { public void doIt() { System.out.print( 'A' ); } }

static class AwithX extends A {
   public  void doIt() { super.doIt();  doX(); }
   private void doX()  { System.out.print( 'X' ); }
}

static class AwithY extends A {
   public void doIt() { super.doIt();  doY(); }
   public void doY()  { System.out.print( 'Y' ); }
}

static class AwithZ extends A {
   public void doIt() { super.doIt();  doZ(); }
   public void doZ()  { System.out.print( 'Z' ); }
}

static class AwithXY extends AwithX {
   private AwithY obj = new AwithY();
   public void doIt() {
      super.doIt();
      obj.doY();
}  }

static class AwithXYZ extends AwithX {
   private AwithY obj1 = new AwithY();
   private AwithZ obj2 = new AwithZ();
   public void doIt() {
      super.doIt();
      obj1.doY();
      obj2.doZ();
}  }

public static void main( String[] args ) {
   A[] array = { new AwithX(), new AwithXY(), new AwithXYZ() };
   for (int i=0; i < array.length; i++) {
      array[i].doIt();
      System.out.print( "  " );
}  }  }
```

Σχήμα 3.7.7: Παράδειγμα Decorator στην JAVA

Το οποίο επιστρέφει

```
AX  AXY  AXYZ
```

Σχήμα 3.7.7: Αποτέλεσμα παραδείγματος

3.3.3 Proxy

Σκοπός

Proxy pattern [16] παρέχει έναν placeholder σε ένα αντικείμενο για τον έλεγχο της πρόσβασης σε αυτό. Δηλαδή χρησιμοποιεί ένα στοιχείο επικοινωνίας, ελέγχου και έξυπνης διαχείρισης. Ακόμα προσθέτει ένα wrapper στο αντικείμενο αυτό για την προστασία του και αποτρέπει την αδικαιολόγητη πολυπλοκότητα.

Πρόβλημα

Θα θέλαμε να υποστηρίξουμε αντικείμενα τα οποία απαιτούν πολλούς πόρους και για αυτό το λόγο θα πρέπει να τα αρχικοποιήσουμε όταν τα ζητήσει ο χρήστης.

Λύση

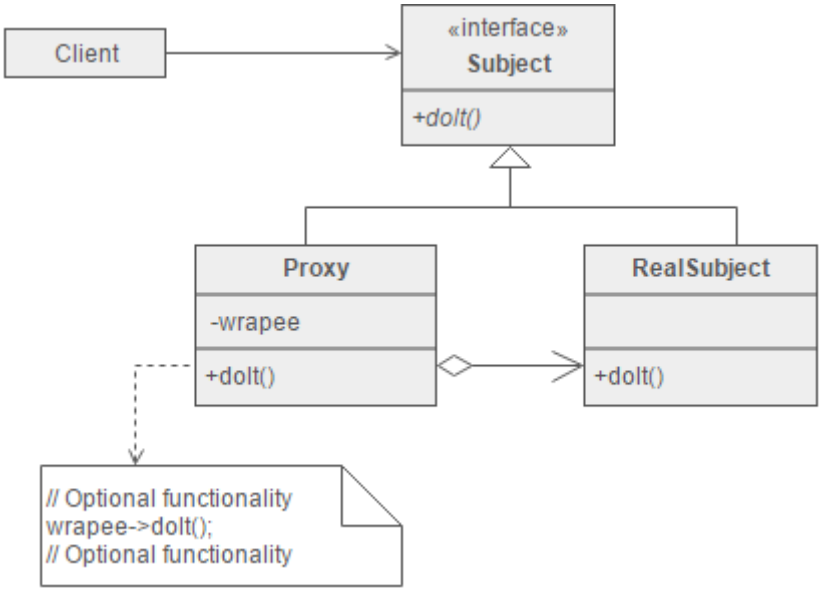
Σχεδιάζουμε ένα αντικείμενο proxy δηλαδή ένα αντικείμενο το οποίο αρχικοποιεί ένα άλλο αντικείμενο που απαιτεί πολλούς πόρους κατά την πρώτη απαίτηση του χρήστη. Στην συνέχεια κάθε φορά που ο χρήστης το καλεί τότε το proxy αντικείμενο θυμάται την ταυτότητα του πραγματικού αντικειμένου και την προωθεί εκεί. Υπάρχουν 3 συνήθεις κατάστάσεις στις οποίες το proxy pattern λειτουργεί.

1. Όταν έχουμε αντικείμενα που απαιτούν πολλούς πόρους για να δημιουργηθούν. Το

πραγματικό αντικείμενο δημιουργείται μόνο όταν ένας πελάτης κάνει για πρώτη φορά αίτημα στο αντικείμενο.
2. Ένας απομακρυσμένος proxy παρέχει τοπική αντιπροσώπευση σε ένα αντικείμενο με διαφορετική διεύθυνση. Αυτό επιβάλει το RPC και το CORBA.
3. Ένα proxy μπορεί να παρέχει ασφάλεια στο αντικείμενο που προωθεί το αίτημα του χρήστη. Το proxy αντικείμενο ελέγχει αν έχει τα απαραίτητα δικαιώματα ο πελάτης για να προωθήσει την αίτηση του στο πραγματικό αντικείμενο.

Δομή

Το αντικείμενο proxy μπαίνει μπροστά στο πραγματικό αντικείμενο και το πραγματικό αντικείμενο παίρνει τις αιτήσεις από το proxy. Ο πελάτης δεν μπορεί να καταλάβει την διαφορά.

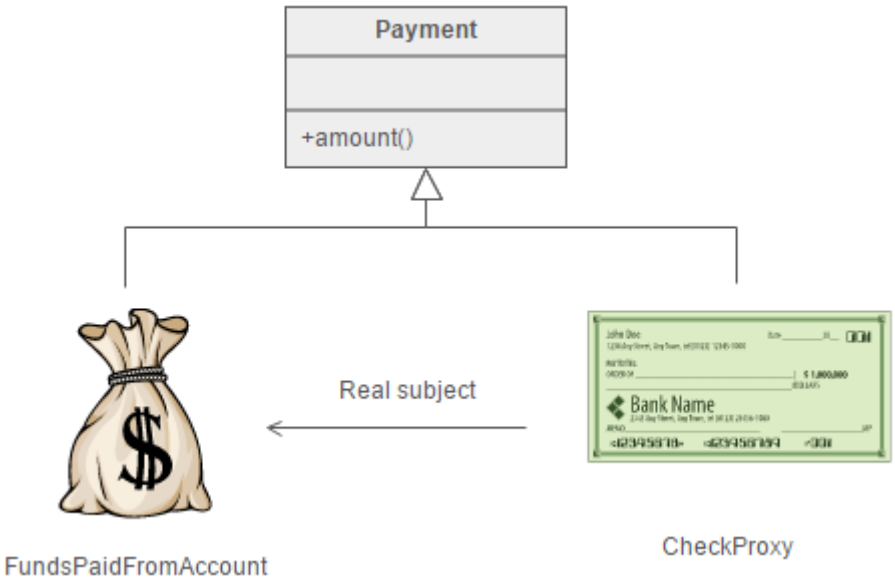


Σχήμα 3.8.1: Πελάτης ζητάει από τον proxy μία λειτουργία

Παράδειγμα

Μία τραπεζική επιταγή ή μια κατάθεση στην τράπεζα μπορεί να θεωρηθεί ως ένα proxy pattern. Μια επιταγή μπορεί να χρησιμοποιηθεί για μια συναλλαγή αντί για λεφτά όπως

και ένα έναυσμα σε ένα τραπεζικό λογαριασμό μπορούν να κάνουν συναλλαγές όπως τα λεφτά.



Σχήμα 3.8.2: Το proxy στην νομισματική λειτουργία

Παρακάτω παραθέτω ένα προγραμματιστικό παράδειγμα ενός proxy pattern

```
import java.io.*;  import java.net.*;

// 5. To support plug-compatibility between
// the wrapper and the target, create an interface
interface SocketInterface {
  String readLine();
  void  writeLine( String str );
  void  dispose();
}

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

    // 3. The client deals with the wrapper
    SocketInterface socket = new SocketProxy( "127.0.0.1", 8189,
      args[0].equals("first") ? true : false );
```

```
    String  str = null;
    boolean skip = true;
    while (true) {
      if (args[0].equals("second")  &&  skip) {
        skip = ! skip;
      }
      else {
        str = socket.readLine();
        System.out.println( "Receive - " + str );  // java ProxyDemo first
        if (str.equals("quit")) break;             // Receive - 123 456
      }                                            // Send ---- 234 567
      System.out.print( "Send ---- " );            // Receive - 345 678
      str = Read.aString();                        //
      socket.writeLine( str );                     // java ProxyDemo second
      if (str.equals("quit")) break;               // Send ---- 123 456
    }                                              // Receive - 234 567
    socket.dispose();                              // Send ---- 345 678
  }
}

class SocketProxy implements SocketInterface {
  // 1. Create a "wrapper" for a remote,
  // or expensive, or sensitive target
  private Socket      socket;
  private BufferedReader in;
  private PrintWriter   out;

  public SocketProxy( String host, int port, boolean wait ) {
    try {
      if (wait) {
        // 2. Encapsulate the complexity/overhead of the target in the wrapper
        ServerSocket server = new ServerSocket( port );
        socket = server.accept();
      } else
        socket = new Socket( host, port );
        in  = new BufferedReader( new InputStreamReader(
                                      socket.getInputStream()));
        out = new PrintWriter( socket.getOutputStream(), true );
      } catch( IOException e ) {
        e.printStackTrace();
      }
  }
  public String readLine() {
    String str = null;
    try {
      str = in.readLine();
    } catch( IOException e ) {
      e.printStackTrace();
    }
    return str;
```

```
  }
  public void writeLine( String str ) {
    // 4. The wrapper delegates to the target
    out.println( str );
  }
  public void dispose() {
    try {
      socket.close();
    } catch( IOException e ) {
      e.printStackTrace();
    }
  }
}
```

Σχήμα 3.8.3: Proxy pattern σε JAVA

3.3.4 Bridge

Σκοπός

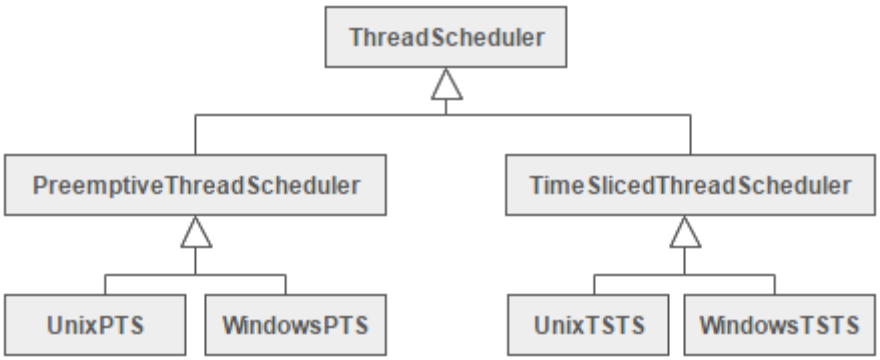
Αποσυνδέσει το abstraction του αντικειμένου από την εφαρμογή του. Δημοσιεύει τις διεπαφές του μέσα από μια ιεραρχία κληρονομικότητας.

Πρόβλημα

Η εφαρμογή γίνετε ποιο δύστροπη όταν χρησιμοποιούμε πολλές υποκλάσεις μιας βασικής κλάσεις και δίνουμε διαφορετικές μεθόδους στις υποκλάσεις αυτές. Αυτό κλειδώνει στη σύνδεση μεταξύ της διεπαφής και της εφαρμογής των αντικειμένων κατά το χρόνο της μεταγλώττισης.

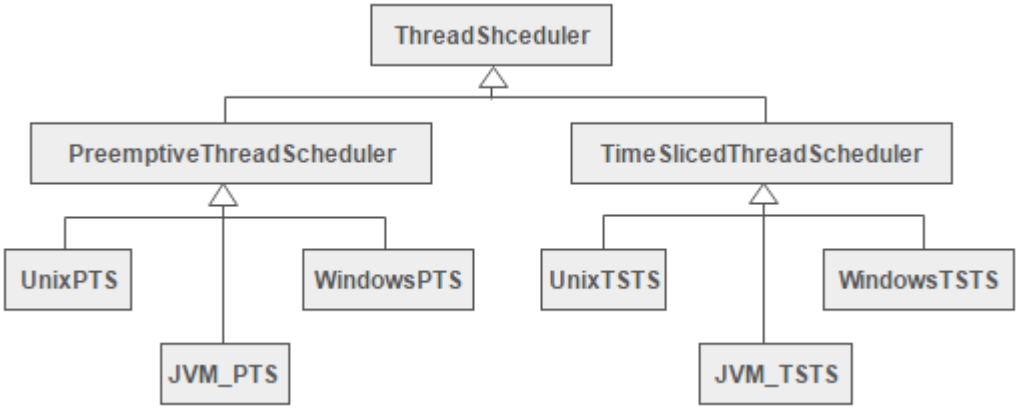
Δομή

Ας υποθέσουμε ότι έχουμε ένα domain που εκτελεί threads σε συγκεκριμένο χρόνο.



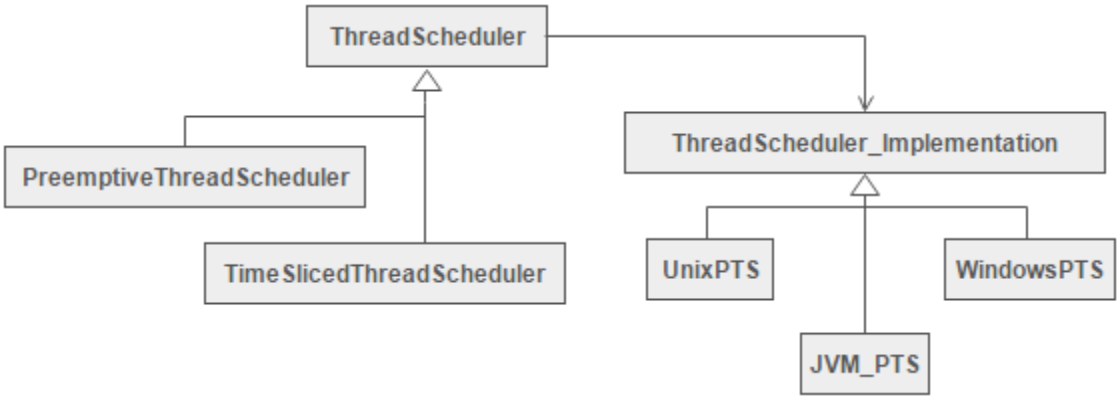
Σχήμα 3.9.1: Domain που περιέχει πολλά λειτουργικά και εκτελεί threads

Έχουμε 2 ειδών thread scheduler και 2 ειδών λειτουργικά συστήματα. Προσεγγίζοντας έναν τρόπο που να είναι που να είναι ποιο ειδικός για τα λειτουργικά συστήματα πρέπει να δηλώσουμε μια κλάση για το κάθε σύστημα. Αν βάλουμε μια πλατφόρμα όπως είναι ένα java virtual machine η ιεραρχία μας θα αλλάξει ως εξής.



Σχήμα 3.9.2: Προσθέτοντας την πλατφόρμα JVM

Τώρα αν είμαι 3 τύπου thread scheduler και 4 ειδών πλατφόρμες ή αν είχαμε 5 ειδών thread scheduler και 10 ειδών πλατφόρμες; Τότε ο αριθμός των κλάσεων που πρέπει να οριστούν είναι πολύ μεγαλύτερος. Το bridge patter λοιπόν έρχεται να μας λύσει τα χέρια και προτείνοντας αυτή την εκθετικά αυξανόμενη κληρονομικότητα που δημιουργείτε να την απλοποιήσει σε 2 ορθογωνικές ιεραρχίες. Μία για platform-independent abstractions και την άλλη για platform-dependent implementations.



Σχήμα 3.9.3: Ο bridge κάνει την λειτουργία platform independent.

Αποσυνδέουμε την διεπαφή και την εφαρμογή του στοιχείου σε μια ορθογώνια ιεραρχία. Η interface κλάση περιέχει έναν pointer για την abstract κλάση implementation. Αυτός ο pointer αρχικοποιείται στην implementation κλάση που είναι παιδί της abstract implementation κλάσης. Αλλά όλες τα interactions από την interface κλάση προς την implementation κλάση είναι περιορισμένα από την λειτουργικότητα της abstract implementation κλάση. Ο χρήστης αλληλοεπιδρά με την interface κλάση και όλες αυτές οι αλληλεπιδράσεις μεταφράζονται ως αλληλεπιδράσεις με την implementation κλάση. Χρησιμοποιούμε το bridge pattern όταν:

- Όταν θέλουμε κατά την εκτέλεση της εφαρμογής να συνδέσουμε μια διεπαφή
- Όταν θέλουμε να μοιράσουμε μία διεπαφή σε πολλά αντικείμενα
- Όταν θέλουμε να κρύψουμε κάποιες λεπτομέρειες από τον χρήστη.

Το bridge pattern είναι ένας μηχανισμός που ενθυλακώνει το implementation μιας κλάσης μέσα σε μία interface κλάση. Το πρώτο μπορούμε να το παρομοιάσουμε με το σώμα και το δεύτερο με την λαβή του σώματος. Η λαβή είναι τελικά αυτό που θα δει ο χρήστης αλλά όλη η δουλειά γίνετε από το σώμα αυτής της λαβής. Αυτό το κάνουμε για να κρύψουμε ουσιαστικά

όλη αυτή την πολυπλοκότητα από το χρήστη και να μπορεί ο χρήστης να έχει πρόσβαση σε αυτή με απλούς τρόπους.

Και ένα προγραμματιστικό παράδειγμα αυτού είναι:

```
class Node {
    public int  value;
    public Node prev, next;
    public Node( int i ) { value = i; }
}

class StackArray {
    private int[] items = new int[12];
    private int   total = -1;
    public void push( int i ) { if ( ! isFull()) items[++total] = i; }
    public boolean isEmpty()  { return total == -1; }
    public boolean isFull()   { return total == 11; }
    public int top() {
        if (isEmpty()) return -1;
        return items[total];
    }
    public int pop() {
        if (isEmpty()) return -1;
        return items[total--];
}   }

class StackList {
    private Node last;
    public void push( int i ) {
        if (last == null)
            last = new Node( i );
        else {
            last.next = new Node( i );
            last.next.prev = last;
            last = last.next;
    }   }
    public boolean isEmpty() { return last == null; }
    public boolean isFull()  { return false; }
    public int top() {
        if (isEmpty()) return -1;
        return last.value;
    }
    public int pop() {
        if (isEmpty()) return -1;
        int ret = last.value;
        last = last.prev;
        return ret;
}   }

class StackFIFO extends StackArray {
    private StackArray temp = new StackArray();
```

```
    public int pop() {
        while ( ! isEmpty())
            temp.push( super.pop() );
        int ret =  temp.pop();
        while ( ! temp.isEmpty())
            push( temp.pop() );
        return ret;
}  }

class StackHanoi extends StackArray {
    private int totalRejected = 0;
    public int reportRejected()   { return totalRejected; }
    public void push( int in ) {
        if ( ! isEmpty()  &&  in > top())
            totalRejected++;
        else super.push( in );
}   }

class BridgeDisc {
    public static void main( String[] args ) {
        StackArray[] stacks = { new StackArray(), new StackFIFO(), new StackHanoi() };
        StackList    stack2 = new StackList();
        for (int i=1, num; i < 15; i++) {
            stacks[0].push( i );
            stack2.push( i );
            stacks[1].push( i );
        }
        java.util.Random rn = new java.util.Random();
        for (int i=1, num; i < 15; i++)
            stacks[2].push( rn.nextInt(20) );
        while ( ! stacks[0].isEmpty())
            System.out.print( stacks[0].pop() + "  " );
        System.out.println();
        while ( ! stack2.isEmpty())
            System.out.print( stack2.pop() + "  " );
        System.out.println();
        while ( ! stacks[1].isEmpty())
            System.out.print( stacks[1].pop() + "  " );
        System.out.println();
        while ( ! stacks[2].isEmpty())
            System.out.print( stacks[2].pop() + "  " );
        System.out.println();
        System.out.println( "total rejected is "
            + ((StackHanoi)stacks[2]).reportRejected() );
}   }
```

Σχήμα 3.9.4: Bridge παράδειγμα σε JAVA.

Και το αποτέλεσμα του παραπάνω είναι:

```
12  11  10  9  8  7  6  5  4  3  2  1
14  13  12  11  10  9  8  7  6  5  4  3  2  1
1  2  3  4  5  6  7  8  9  10  11  12
0  0  1  12  16  18
total rejected is 8
```

Σχήμα 3.9.4: Το αποτέλεσμα του παραδείγματος.

4 Κεφαλαιο Εφαρμογη SOLID Principles

Σε αυτό το κεφάλαιο θα αναφερθούμε στα SOLID Principles και θα περιγράψουμε κάθε ένα ξεχωριστά με πολλαπλά παραδείγματα.

Τα SOLID Principles [34] είναι 5 αρχές για δομημένο και καθαρό αντικειμενοστραφή προγραμματισμό. Έγιναν δημοφιλή από τον Robert Cecil Martin ή αλλιώς Uncle Bob όπως τον ξέρουν οι περισσότεροι στη βιομηχανία της ανάπτυξης λογισμικού. Όταν ένα άτομο ή μια προγραμματιστική ομάδα δουλεύει σε ένα αντικειμενοστρεφές σύστημα όπου η διαχείριση dependencies καθίσταται δύσκολη, ο κώδικας καταλήγει να είναι δύσκολος στη συγγραφή, στη διαχείριση, στη επεκτασιμότητα και κυρίως στο τεστάρισμα. Σαν αποτέλεσμα όλων των παραπάνω μειονεκτημάτων καθίσταται εξαιρετικά δύσκολη η περαιτέρω επέκταση του συστήματος. Το αποτέλεσμα είναι τα λεγόμενα Legacy systems. Εάν κάποιος ακολουθήσει όσο πιο πιστά μπορεί τις αρχές SOLID τότε μπορεί να πετύχει εύκολα διαχειρίσιμο κώδικα που είναι πολύ εύκολος στο τεστάρισμα και εξαιρετικά επεκτάσιμος.

Η πρώτη αρχή είναι η λεγόμενη Single Responsibility Principle. Αυτή η αρχή ορίζει ότι μια κλάση θα πρέπει να έχει μόνο ένα λόγο να αλλάξει. Δηλαδή, ο σχεδιασμός της κλάσης θα πρέπει να γίνει με τέτοιο τρόπο ώστε να εξυπηρετεί ένα και μόνο ένα συγκεκριμένο σκοπό. Βέβαια αυτό δεν σημαίνει ότι θα πρέπει να έχει μόνο μια μέθοδο αλλά ότι κάθε πεδίο και κάθε μέθοδος της κλάσης θα πρέπει να εξυπηρετούν το μοναδικό της σκοπό.

Όταν εφαρμόζεται η αρχή αυτή, οι κλάσεις καταλήγουν να είναι πιο μικρές σε μέγεθος και πιο καθαρές από μεριά ευθυνών με αποτέλεσμα και ο κώδικας σε κάθε μέθοδο να είναι πιο κατανοητός. Επίσης, άλλο ένα πλεονέκτημα του να έχουμε μικρές και καθαρές κλάσεις είναι η ελαχιστοποίηση πιθανών λογικών σφαλμάτων στο κώδικα μας κάνοντας το σύστημα πιο ρωμαλέο και ανθεκτικό.

Το παρακάτω παράδειγμα απεικονίζει μια Java κλάση που παραβιάζει αυτή την αρχή. Με μικρά και απλά βήματα θα την μετασχηματίσουμε.

```
public class OxygenMeter {
    public double oxygenSaturation { get; set; }

    public void readOxygenLevel()
    {
        MeterStream ms = new MeterStream("O2")
        int raw = ms.ReadByte();
        OxygenSaturation = (double)raw / 255 * 100;
    }

    public boolean oxygenLow()
    {
        return oxygenSaturation <= 75;
    }

    public void showLowOxygenAlert()
    {
        System.out.println("Oxygen low ({0:F1}%d)", oxygenSaturation);
    }
}
```

Σχήμα 4.1: JAVA κλάση που παραβιάζει την Single Responsibility Principle.

Η παραπάνω κλάση επικοινωνεί με μια συσκευή που ελέγχει το επίπεδο οξυγόνου στο νερό. Η κλάση περιέχει μια μέθοδο readOxygenLevel() που ανακτά μια τιμή από μια ροή δεδομένων που προέρχεται από την εξωτερική συσκευή ελέγχου του οξυγόνου. Στη συνέχεια μετασχηματίζει τη τιμή σε ένα ποσοστό και το αποθηκεύει στο πεδίο oxygenSaturation. Η δεύτερη μέθοδος ελέγχει αν το οξυγόνο είναι χαμηλό όταν η συγκέντρωση οξυγόνου δεν είναι μεγαλύτερη από 75%. Η τελευταία μέθοδος εμφανίζει ένα προειδοποιητικό μήνυμα με τη τρέχουσα τιμή της συγκέντρωσης του οξυγόνου.

Η παραπάνω κλάση έχει περισσότερους από ένα λόγους να αλλάξει και μάλιστα 3 λόγους. Πρώτον, αν η εξωτερική συσκευή μέτρησης της συγκέντρωσης του οξυγόνου αλλάξει τότε η μέθοδος readOxygenLevel() θα πρέπει να τροποποιηθεί. Δεύτερον, αν ο τρόπος που καθορίζεται η χαμηλή περιεκτικότητα οξυγόνου αλλάξει και περιλαμβάνει για παράδειγμα και πιθανή θερμοκρασία τότε η μέθοδος oxygenLow() θα πρέπει να αλλάξει. Τρίτον, αν αποφασίσουμε αργότερα ότι απαιτείται πιο εξεζητημένη ειδοποίηση αντί να εκτυπώνουμε ένα μήνυμα στη κονσόλα τότε θα πρέπει να αλλάξει και η τρίτη μέθοδος.

Refactoring

Για να τροποποιήσουμε τη κλάση και να παραμείνουμε πιστοί στην αρχή Single Responsibility τότε θα πρέπει να διαιρέσουμε τη κλάση σε τρείς διαφορετικές κλάσεις.

1. Η πρώτη κλάση θα είναι η OxygenMeter. Η κλάση αυτή διατηρεί το πεδίο oxygenSaturation και τη μέθοδο readOxygenLavel(). Οι υπόλοιπες μέθοδοι θα χωριστούν στις υπόλοιπες 2 κλάσεις ώστε ο μοναδικός λόγος να αλλάξει η κλάση OxygenMeter θα είναι να αποφασίσουμε να αλλάξουμε το τρόπο επικοινωνίας με την εξωτερική συσκευή μέτρησης οξυγόνου.
2. Η δεύτερη κλάση ονομάζεται OxygenSaturationChecker. Η κλάση αυτή περιέχει μια μοναδική μέθοδο που συγκρίνει το τρέχων επίπεδο οξυγόνου με το κατώτατο όριο. Ο μοναδικός λόγος να αλλάξει αυτή η κλάση είναι η αλλαγή του τρόπου εξαγωγής συμπεράσματος σχετικά με το επίπεδο του οξυγόνου.
3. Η τρίτη κλάση ονόμαζεται OxygenAlerter. Η κλάση αυτή απεικονίζει ένα μήνυμα στη κονσόλα του χρήστη σχετικά με τρέχων επίπεδο συγκέντρωσης του οξυγόνου. Ο μοναδικός λόγος να αλλάξει αυτή η κλάση είναι να αλλάξει ο τρόπος που απεικονίζεται το τρέχων επίπεδο οξυγόνου.

```
public class OxygenMeter {
    public double oxygenSaturation { get; set; }

    public void readOxygenLevel() {
        MeterStream ms = new MeterStream("O2")

        int raw = ms.ReadByte();
        OxygenSaturation = (double)raw / 255 * 100;
    }
}
```

Σχήμα 4.2: Η πρώτη κλάση OxygenMeter.

```
public class OxygenSaturationChecker {
    public boolean oxygenLow(OxygenMeter meter) {
        return meter.oxygenSaturation <= 75;
    }
}
```

Σχήμα 4.3: Η δεύτερη κλάση OxygenSaturationChecker.

```
public class OxygenAlerter {
    public void showLowOxygenAlert(OxygenMeter meter) {
        println("Oxygen low ({0:F1}%)", meter.OxygenSaturation);
    }
}
```

Σχήμα 4.4: Η τρίτη κλάση OxygenAlerter.

Η επόμενη αρχή ονομάζεται Open Closed Principle. Η αρχή αυτή ορίζει ότι μια κλάση θα πρέπει να είναι «ανοικτή» για επέκταση αλλά «κλειστή» για τροποποίηση. Αυτό σημαίνει ότι θα πρέπει μια κλάση να σχεδιαστεί με τέτοιο τρόπο ώστε να είναι εύκολο να προστεθεί νέα λειτουργικότητα αλλά θα πρέπει να απαγορεύει να τροποποιηθεί αυτή η κλάση εκτός από τη περίπτωση που θα διορθωθεί κάποιο λογικό σφάλμα.

Στην αρχή αυτές οι δυο έννοιες φαντάζουν αντικρουόμενες αλλά στη πραγματικότητα αυτή η αρχή γίνεται εφικτή με τη χρήση abstractions στο κώδικα μέσω Abstract classes ή Interfaces. Με αυτό το τρόπο νέα λειτουργικότητα μπορεί να υλοποιηθεί με τη δημιουργία νέων κλάσεων που είτε κληρονομούν απο μια Abstract class ή υλοποιούν ένα Interface.

Τα πλεονεκτήματα που προσφέρει η αρχή αυτή είναι οτι περιορίζει μια κλάση να τροποποιηθεί και να επιφέρει λογικά σφάλματα αφού έχει επικυρωθεί η λειτουργικότητα της με tests και quality assurance. Επίσης, με τη χρήση abstractions μπορούμε να εστιάσουμε πιο εύκολα στο πραγματικό business πρόβλημα και να προωθήσουμε πιο εύκολα το πολυμορφισμό.

Για να εξηγήσουμε την αρχή στη πράξη θα παρουσιάσουμε ένα παράδειγμα από δυο κλάσεις που παραβιάζουν τη αρχή και θα δείξουμε μέσα από απλά βήματα πως θα τις μετασχηματίσουμε.

```
public class Logger
{
    public void log(String message, LogType logType)
    {
        switch (logType)
        {
            case LogType.Console:
                println(message);
                break;

            case LogType.File:
                // Code to send message to printer
                break;
        }
    }
}


public enum LogType
{
    Console,
    File
}
```

Σχήμα 4.5: Η Logger που εκτυπώνει το μήνυμα.

Οι παραπάνω κλάσεις χρησιμεύουν προκειμένου να εκτυπώσουν ένα μήνυμα. Συγκεκριμένα η κλάση Logger έχει μια μέθοδο log που δέχεται το μήνυμα που θα εκτυπωθεί καθώς και το μέσο στο οποίο θα εκτυπωθεί το μήνυμα. Το switch χρησιμεύει στο να φιλτράρει το μέσο στο οποίο θα εκτυπωθεί το μήνυμα ανάλογα με τη τιμή του enum LogType.

Σύμφωνα με το παραπάνω σχεδιασμό αν επιθυμούσαμε να εισάγουμε ένα νέο τρόπο εκτύπωσης του μηνύματος, όπως για παράδειγμα να στέλνουμε το μήνυμα σε ουρά αναμονής για περαιτέρω επεξεργασία ή να τα αποθηκεύαμε σε βάση δεδομένων, δεν θα μπορούσαμε χωρίς να τροποποιήσουμε τον υπάρχον κώδικα. Πρώτον, θα χρειαζόμασταν να προσθέσουμε ένα καινούργιο enum constant και δεύτερον να προσθέσουμε άλλο ένα case στο switch statement.

Refactoring

Προκείμενου να τροποποιήσουμε τη κλάση Logger για να μην παραβιάζει το Open Closed Principle θα πρέπει να αφαιρέσουμε το enum LogType καθώς περιορίζει τους τρόπους με τους οποίους μπορούμε να εκτυπώσουμε ένα μήνυμα. Επιπλέον, θα σταματήσουμε να περνάμε το τρόπο με τον οποίο θα εκτυπωθεί το μήνυμα σαν παράμετρο στη μέθοδο log της κλάσης Logger, αλλά θα δημιουργήσουμε ξεχωριστές κλάσεις για κάθε διαφορετικό τρόπο εκτύπωσης του μηνύματος. Τέλος, θα έχουμε δυο κλάσεις ονόματι ConsoleLogger και PrinterLogger. Περαιτέρω τρόποι εκτύπωσης του μηνύματος θα μπορέσουν να προστεθούν πολύ εύκολα με τη δημιουργία ξεχωριστών κλάσεων.

```
public class Logger
{
    MessageLogger messageLogger;

    public Logger(MessageLogger messageLogger)
    {
        this.messageLogger = messageLogger;
    }

    public void log(String message)
    {
        this.messageLogger.log(message);
    }
}


public interface MessageLogger
{
    void log(String message);
}


public class ConsoleLogger implements MessageLogger
{
    public void log(string message)
    {
        println(message);
    }
}


public class PrinterLogger implements MessageLogger
{
    public void log(String message)
    {
        // Code to send message to printer
    }
}
```

Σχήμα 4.6: Η logger που έχει γίνει refactored.

Η κλάση Logger εξακολουθεί να έχει την ευθύνη να εκτυπώσει το μήνυμα, αλλά πλέον δεν χρειάζεται να γνωρίζει ποιον αλγόριθμο θα διαλέξει προκειμένου να το εκτυπώσει. Αντιθέτως, δέχεται ως παράμετρο στην constructor ένα instance του Interface MessageLogger που περιέχει τη μέθοδο log(). Οι κλάσεις PrinterLogger και ConsoleLogger υλοποιούν το interface MessageLogger και τη μέθοδο log(). Το μόνο που χρειάζεται πλέον να γνωρίζει η κλάση Logger είναι να να εκτυπώσει το μήνυμα και όχι το τρόπο που θα το κάνει. Με αυτό το τρόπο η κλάση Logger καθώς και οι υπόλοιπες κλάσεις δεν χρειάζεται να αλλάξουν και δεν παραβιάζουν ούτε το Open Closed Principle αλλά ούτε και το Single Responsibility Principle.

Η επόμενη αρχή ονομάζεται Interface Segregation Principle. Η αρχή αυτή περιγράφει ότι ορισμένες κλάσεις έχουν interfaces που δεν είναι συνεκτικά, δηλαδή περιλαμβάνουν πολλά groups από μεθόδους όπου κάθε group χρησιμοποιείται από διαφορετικούς clients. Ιδανικά, κάθε κλάση θα πρέπει να έχει συνεκτικά interfaces.

Η κύρια έννοια αυτής της αρχής είναι ότι οι clients δεν θα πρέπει να αναγκάζονται να εξαρτώνται απο interfaces που δεν χρειάζονται ή δεν πρέπει να γνωρίζουν. Η αρχή αυτή προωθεί την ιδέα των πολλών μικρών σε μέγεθος και ευθύνη interfaces και όπου κάθε κλάση υλοποιεί ένα ή περισσότερα από αυτά τα μικρά interfaces. Πιο μικρά interfaces είναι πιο εύκολα στη δημιουργία, στη συντήρηση, στην επέκταση και στην επαναχρησιμοποιησιμότητα.

Το παρακάτω παράδειγμα παρουσιάζει τρεις κλάσεις που παραβιάζουν την αρχή. Η κλάση Contact περιέχει τέσσερα απλά πεδία επικοινωνίας για μια επαφή. Η κλάση Emailer αποστέλλει ένα email σε ένα Contact παίρνοντας ως παραμέτρους στη μέθοδο sendMessage(), την επαφή, το κείμενο του email και το θέμα του μηνύματος. Η κλάση Dialer πραγματοποιεί μια τηλεφωνική κλήση στο άτομο που αναπαριστά η κλάση Contact.

```
public class Contact
{
    public String name { get; set; }
    public String address { get; set; }
    public String emailAddress { get; set; }
    public String telephone { get; set; }
}


public class Emailer
{
    public void sendMessage(Contact contact, String subject, String body)
    {
        // Code to send email, using contact's email address and name
    }
}


public class Dialler
{
    public void makeCall(Contact contact)
    {
        // Code to dial telephone number of contact
    }
}
```

Σχήμα 4.6: Η contact emailer και η dialer παραβιάζοντας την Interface Segregation Principle.

Η κλάση Emailer είναι client της κλάσης Contact και χρησιμοποιεί μόνο το όνομα και την ηλεκτρονική διεύθυνση του Contact αλλά παρόλα αυτά έχει πρόσβαση σε όλα τα πεδία της κλάσης Contact. Αντίστοιχα και η κλάση Dialer χρειάζεται μόνο το τηλέφωνο του Contact αλλά παρόλα αυτά έχει πρόσβαση σε όλα τα πεδία.

Refactoring

Για να μπορέσουμε να μετασχηματίσουμε το κώδικα ώστε να πάψει να παραβιάζει την αρχή θα πρέπει να μπορέσουμε να κρύψουμε τα μη απαραίτητα πεδία της κλάσης Client απο κάθε Client.

Για αυτό το λόγο θα δημιουργήσουμε δυο interfaces, το πρώτο ονόματι Emailable που καθορίζει τα πεδία name και emailAddress και το δεύτερο Diallable που καθορίζει μόνο το telephone πεδίο.

Η κλάση Emailer τροποποιείται και αντί για μια παράμετρο Contact δέχεται μια παράμετρο Emailable. Αντίστοιχα και η κλάση Dialler δέχεται μια παράμετρο Diallable. Τώρα πλέον και οι δυο κλάσεις αλληλεπιδρούν με interfaces που περιέχουν όσο το δυνατόν λιγότερη πληροφορία και απολύτως απαραίτητη για τις κλάσεις Emailer και Dialler.

Με μικρότερα interfaces γίνεται αρκετά εύκολο να προστεθεί μια νέα κλάση για παράδειγμα MobileEngineer που έχει πεδία name, vehicle, telephone. Το τελευταίο πεδίο το κληρονομεί απο το interface Diallable.

```
public interface Emailable
{
    String name { get; set; }
    String emailAddress { get; set; }
}


public interface Diallable
{
    String telephone { get; set; }
}


public class Contact implements Emailable, Diallable
{
    public String name { get; set; }
    public String address { get; set; }
    public String emailAddress { get; set; }
    public String telephone { get; set; }
}


public class MobileEngineer implements Diallable
{
    public String name { get; set; }
    public String vehicle { get; set; }
    public String telephone { get; set; }
}


public class Emailer
{
    public void sendMessage(Emailable target, String subject, String body)
    {
        // Code to send email, using target's email address and name
    }
}


public class Dialler
{
    public void makeCall(Diallable target)
    {
        // Code to dial telephone number of target
    }
}
```

Σχήμα 4.7: Η contact emailer και η dialer μετασχηματισμένες σύμφωνα με την Interface Segregation Principle.

5 ΚΕΦΑΛΑΙΟ ΣΧΕΤΙΚΕΣ ΤΕΧΝΟΛΟΓΙΕΣ

Η πρώτη τεχνολογία που θα περιγράψω είναι τα servlets. Είναι ένα είδος προγράμματος που μπορεί να εκτελεστεί όταν ζητηθεί από έναν browser.Τα servlets είναι γραμμένα σε γλωσσα προγραμματισμού JAVA, φορτώνονται στον διακομιστή Web και εκτελούνται όταν μία κατάλ-ληλη εντολή HTTP που ζητά την εκτέλεση τους λαμβάνεται από τον διακομιστή. Τα servlets έχουν ορισμένα σημαντικά χαρακτηριστικά:

1. Μπορούν να εκτελεστούν χωρίς αλλαγές σε διάφορους τύπους διακομιστών.
2. Από την στιγμή που τα servlets είναι γραμμένα σε Java έχουν πρόσβαση σε διάφορα εργαλεία της Java όπως η CORBA, RMI, εργαλεία ασφαλείας της Java και εργαλεία σύνδεσης βάσεων δεδομένων.
3. Τα servlets είναι εγκατεστημένα στην μνήμη. Με τον προγραμματισμό CGI σε γλώσσες όπως η Perl, κάθε φορά που ένα καινούργιο αίτημα προωθείται από τον Web browser μια καινούργια επεξεργασία πρέπει να ξεκινήσει και να τερματιστεί με την λήξη της σύνδεσης και το κατάλληλο πρόγραμμα πρέπει να φορτωθεί και να ξεφορτωθεί από τη μνήμη. Αυτό σημαίνει ότι το φορτίο μπορεί να είναι μεγάλο για τον διακομιστή Web.
4. Από την στιγμή που τα servlets βασίζονται στην Java περιλαμβάνουν όλα τα χαρακτηριστικά ασφαλείας της γλώσσας.
5. Τα servlets μπορούν να διατηρήσουν την κατάσταση τους, να διατηρήσουν κάποιο είδος μνήμης, ανάμεσα στα αιτήματα, κάτι που, στα πρώτα βήματα του Παγκόσμιου Ιστού, ήταν δύσκολο. Αυτό σημαίνει ότι τα servlets μπορούν να θυμηθούν δεδομένα και λεπτομέρειες ενός προηγούμενου αιτήματος.
6. Το αντικειμενοστραφές μοντέλο της Java βοηθά τον κώδικα της να είναι πιο κομψός από κώδικα γραμμένο σε άλλες γλώσσες όπως η C και η Perl. Άν εξετάσετε τα προγράμματα Perl CGI θα δείτε, συχνά, ότι είναι τεράστια, μονοκόμματα και δυσνόητα. Η προσανατολισμένη στ' αντικείμενα φύση της Java παρέχει επίσης και την δυνατότητα μεγαλύτερης επαναχρησιμοποίησης κώδικα.
7. Τα servlets, επειδή ενσωματώνονται καλά με τις τεχνολογίες ασφάλειας που σχετίζονται με την Java, μπορούν να διαμορφωθούν ώστε να παρέχουν υψηλά επίπεδα ασφαλείας.

Πριν εξετάσουμε τα servlets πιο λεπτομερώς αξίζει να θυμηθούμε το πως οι διακομιστές Web επεξεργάζονται τα αιτήματα. Ένας browser θα εκδώσει ένα αίτημα σε HTTP, για

παράδειγμα ζητώντας μία ιστοσελίδα. Αυτό αποστέλλεται στον διακομιστή Web που ερμηνεύει το αίτημα και εκτελεί κώδικα, ο οποίος επιστρέφει μία σελίδα HTML. Η επεξεργασία που εκτελείται από τον διακομιστή εξαρτάται από την λειτουργικότητα της ιστοσελίδας που ζητήθηκε. Για παράδειγμα, ο κώδικας που εκτελείται μπορεί:

- Να επιστρέψει μία ιστοσελίδα χωρίς να την αλλάξει καθόλου.
- Να επιστρέψει μία ιστοσελίδα αφού την μορφοποιήσει, για παράδειγμα να εισάγει κάποιο δυναμικό περιεχόμενο όπως τιμές μετοχών και εμπορευμάτων.
- Να επεξεργαστεί μία φόρμα, να συνδεθεί με μία βάση δεδομένων ώστε να ανακτήσει δεδομένα και στην συνέχεια να κατασκευάσει μία σελίδα βασιζόμενη σ' αυτές τις λεπτομέρειες.

Σε αυτή την εφαρμογή έχουμε χρησιμοποιήσει την τεχνολογία της java 8 EE servlet 3 που ουσιαστικά είναι ένας mvc (model and view controller).

5.1 Μέθοδοι HTTP αίτησης

Κάθε αίτηση μπορεί να χρησιμοποιήσει μία από τις πολλές μεθόδους που καθορίζονται στα HTTP πρότυπα.

Μέθοδος Περιγραφή

GET Είναι η πιο απλή και η πιο συχνά χρησιμοποιούμενη μέθοδος. Αυτή απλώς παραλαμβάνει τα δεδομένα που αναγνωρίζονται από την URL. Αν η URL αναφέρεται σε ένα κομμάτι κώδικα (CGI, servlet ή άλλο), επιστρέφει τα δεδομένα που δημιουργούνται από αυτό.HEAD Εκτελεί την ίδια λειτουργία με την GET, αλλά η HEAD επιστρέφει μόνο τις επικεφαλίδες χωρίς το κυρίως κείμενο.

POST Όπως η GET, η POST χρησιμοποιείται και αυτή πολύ συχνά. Τυπικά χρησιμοποιείται σε φόρμες HTML. Μεταφέρει μια ομάδα δεδομένων στον διακομιστή στο κυρίως μέρος της αίτησης. OPTIONS Η μέθοδος αυτή χρησιμοποιείται για να ρωτήσει έναν διακομιστή σχετικά με τις δυνατότητες που παρέχει. Τα ερωτήματα μπορεί να είναι γενικά ή συγκεκριμένα για ένα καθορισμένο πόρο. PUT Η μέθοδος PUT συμπληρώνει την GET και αποθηκεύει το κυρίως μέρος του αιτήματος στη θέση που καθορίζεται από την URL. Είναι παρόμοια με την

λειτουργία PUT του FTP. DELETE Διαγράφει ένα αρχείο από τον server. Το αρχείο προς διαγραφή καθορίζεται στο URI τμήμα της αίτησης. TRACE Χρησιμοποιείται για να παρακολουθεί την διαδρομή μιας αίτησης διαμέσου του τείχους προστασίας (firewall) και πολλαπλών διακομιστών αντιπροσώπων (proxy server). Η TRACE είναι χρήσιμη για αποσφαλμάτωση περίπλοκων προβλημάτων δικτύου και είναι παρόμοια με το εργαλείο traceroot.

5.1.1 HTTP Απαντήσεις

Όπως και οι αιτήσεις, έτσι και οι απαντήσεις αποτελούνται από τρία μέρη:

• Πρωτόκολλο—Status code—Περιγραφή

• Επικεφαλίδες

• Κυρίως μέρος

Το παρακάτω είναι παράδειγμα HTTP απάντησης:

GET /patient/api?heartpump=20 HTTP/1.1
Host: localhost:8080
key: 24792

HTTP/1.1 200
X-Powered-By: Panagis
Server: NothingSpecial
Content-Type: application/json;charset=UTF-8
Content-Length: 19
Date: Tue, 20 Dec 2016 15:47:45 GMT

{"message":"Added"}

Η πρώτη γραμμή της επικεφαλίδας της απάντησης είναι όμοια με την πρώτη γραμμή της επικεφαλίδας της αίτησης. Ενημερώνει ότι το πρωτόκολλο που χρησιμοποιείται είναι το HTTP 1.1, η αίτηση είναι πετυχημένη (200 = επιτυχία) και όλα είναι σωστά. Οι επικεφαλίδες περιέχουν χρήσιμες πληροφορίες όπως και αυτές της αίτησης. Το κυρίως μέρος είναι το ίδιο το HTML περιεχόμενο της απάντησης. Επικεφαλίδες και κυρίως μέρος χωρίζονται από μια ακολουθία

CRLF.

5.2 Model and view controller

Το Model–view–controller [35] (σε συντομογραφία αναφέρεται ως MVC) είναι ένα μοντέλο αρχιτεκτονικής λογισμικού το οποίο χρησιμοποιείται για την δημιουργία περιβαλλόντων αλληλεπίδρασης χρήστη. Στο μοντέλο αυτό η εφαρμογή διαιρείται σε τρία διασυνδεδεμένα μέρη ώστε να διαχωριστεί η παρουσίαση της πληροφορίας στον χρήση από την μορφή που έχει αποθηκευτεί στο σύστημα. Το κύριο μέρος του μοντέλου είναι το αντικείμενο Model το οποίο διαχειρίζεται την ανάκτηση/αποθήκευση των δεδομένων στο σύστημα. Το αντικείμενο View χρησιμοποιείται μόνο για να παρουσιάζεται η πληροφορία στον χρήστη (π.χ. με γραφικό τρόπο). Το τρίτο μέρος είναι ο Controller ο οποίος δέχεται την είσοδο και στέλνει εντολές στο αντικείμενο Model και στο View. Στο δικό μας παράδειγμα θα δούμε διεξοδικά και τα 3 αυτά μέρει που θα τα αναλύσουμε.

5.2.1 Λειτουργία

Η εφαρμογή μας έχει controllers που αποτελεί το βασικό δομικό κομμάτι για την λειτουργία της. Αρχικά πρέπει να δούμε τα request που πρέπει να εξυπηρετούνται από τους controllers δηλαδή την βασική λογική της εφαρμογή μας που ουσιαστικά είναι servlets. Αρχικά παρατηρούμε το πρόβλημα του authentication στην περίπτωση μας θα πρέπει να έχουμε δύο μορφές authentication. Η εφαρμογή μας ξεκινά στην αρχική οθόνη που ουσιαστικά αναμένει από το χρήστη στοιχεία για το authentication. Οπότε δημιουργούμε ένα LoginController που θα μπορεί να κάνει αυτήν την δουλειά. Εδώ θα πρέπει να αναφέρουμε ότι για τη ορθή λειτουργεία του authenticated χρήστη μέσα στην εφαρμογή μας θα χρησιμοποιήσουμε ένα σύστημα session που παρέχετε από το servlet api. Το servlet API υποστηρίζει μία κομψή και βολική λύση για το πρόβλημα του εντοπισμού διαρκών δεδομένων. Παρέχει μία κλάση HttpSession [36], που είναι μία φόρμα συσχετιστικής μνήμης, η οποία εντοπίζει τα δεδομένα που σχετίζονται με ένα session, όπου session είναι η σειρά των διαδράσεων ενός browser μ' έναν ιστοχώρο πριν την απομάκρυνση. Ένα servlet μπορεί να δημιουργήσει ένα αντικείμενο το οποίο θα περιγράφεται απ' αυτήν την κλάση δια μέσου

της μεθόδου getSession που βρίσκεται στην κλάση HttpServletRequest. Οπότε στην περίπτωσή μας δημιουργούμε έναν controller ως εξής:

```
@WebServlet(name = "/login", urlPatterns = "/login")
public class LoginController extends HttpServlet {
```

Σχήμα 5.1: Δήλωση ενός controller.

Οπότε βλέπουμε ότι με το annotation @WebServlet δηλώνουμε το όνομα και το urlPattern το οποίο θα «ακούει» το servlet αυτό. Αυτό με την σειρά του αντικαθιστά τον παλιό τρόπο που δηλώνεις τα servlet στον web descriptor αρχείο και το σύστημα καταλαβαίνει ότι πρόκειται για έναν controller. Αρχικά στον controller εμφανίζουμε ένα login screen για να μπορεί ο χρήστης να δώσει τα απαραίτητα στοιχεία για να μπορέσει να ξεκινήσει ένα session. Έτσι κάνουμε forward το request σε ένα jsp το οποίο περιέχει με την σειρά του τον html κώδικα που θα εμφανίσει.

```
if (userRole == null) {
    req.getRequestDispatcher("login.jsp").forward(req, resp);
}
```

Σχήμα 5.2: Προώθηση του αιτήματος σε ένα view.

Σε αυτό το σημείο θα πρέπει να αναφερθούμε στο jsp [37]. Το μοντέλο που υιοθετήθηκε από τους σχεδιαστές της τεχνολογίας servlet ήταν αυτό της διαδικασίας ελέγχου του πρωτοκόλλου εντολών, όπως το GET, της αποκωδικοποίησής τους, της εκτελέσεως οποιασδήποτε επεξεργασίας και της αποστολής μιας ιστοσελίδας HTML πίσω στον browser που έδωσε την εντολή. Η διαδικασία αυτή μπορεί να είναι πολύ βαρετή, ειδικά ο προγραμματισμός που σχετίζεται με την δημιουργία και αποστολή απαντητικών σελίδων HTML. Η Java Server Pages (JSP) είναι μία τεχνολογία δυναμικών σελίδων, που παρέχει εργαλεία που κάνουν την ανάπτυξη προγραμμάτων διακομιστών μία πιο ευχάριστη υπόθεση. Η ιδέα πίσω από την Java Server Pages είναι ότι ο κώδικας (κώδικας Java) είναι τοποθετημένος μέσα στην ιστοσελίδα κατά τέτοιο τρόπο που να μεταβάλλει δυναμικά την HTML της σελίδας. Ένα παράδειγμα δίνεται παρακάτω:

```
<% for (int j = 0; j<100;j++)
{
if (j%2==0) %>
```

```
   <P>The value of the even integer is <%= j %>
<% }%>
```

Σχήμα 5.3: Παράδειγμα jsp με JAVA κώδικα.

Αυτό δημιουργεί μία σελίδα HTML [38] που θα εμφανίσει τους πρώτους 50 ίσους αριθμούς. Εδώ ο κώδικας Java μπερδεύεται με την HTML. Ο κώδικας Java ορίζεται από τους χαρακτήρες <% and %>. Έξω από τα άγκιστρα είναι η HTML, για παράδειγμα το επίθεμα <P> της παραγράφου. Το αποτέλεσμα του παραπάνω κώδικα είναι η παραγωγή του HTML κώδικα

```
<P> The value of the even integer is 0
<P> The value of the even integer is 2
<P> The value of the even integer is 4
<P> The value of the even integer is 6
<P> The value of the even integer is 8
```

Σχήμα 5.4: Το αποτέλεσμα του παραδείγματος

Ο τρόπος με τον οποίο ένας JSP-ευαίσθητος διακομιστής επεξεργάζεται ένα JSP κείμενο είναι να εξάγει τον κώδικα Java, να δημιουργήσει ένα servlet και μετά να επιστρέψει την σελίδα που αντιστοιχεί στην HTML που έχει αυξηθεί από την HTML που παρήγαγε το servlet, για παράδειγμα τις γραμμές που αρχίζουν με το <P>.
Στην ουσία οτιδήποτε μπορεί να επιτευχθεί με την προγραμματισμό των servlet μπορεί να επιτευχθεί με την χρήση της JSP, για παράδειγμα ο κώδικας Java μπορεί να δηλώνει μεταβλητές εύρους σελίδων, παραμέτρους πρόσβασης, αιτήσεις πρόσβασης και αντικείμενα απάντησης και μπορεί να αποκτήσει πρόσβαση στο αντικείμενο που σχετίζεται με το αίτημα. Το jsp είναι μια τεχνολογία για την παροχή εμφάνισης μέσω των servlet. Είναι μικρά προγράμματα που εξειδικεύονται στον να σερβίρουν HTML κωδικα και να την αλλάζουν πριν αυτή φτάσει στον χρήστη. Οπότε μέσω ενός jsp καταφέρνουμε και παρέχουμε στον χρήστη μια φόρμα που μπορεί να στείλει πίσω σε εμάς δεδομένα για την δημιουργία ενός session. Γίνετε ένα post request και ο χρήστης μας στέλνει τα δεδομένα στον ίδιο controller που αρχικά είχε δημιουργηθεί το request. Τελικός γίνετε και το authentication και εμπλουτίζεται ένα session object . Εμπλουτίζουμε ένα session ανάλογα για το τι είδους user πρόκειται να συνδεθεί. Οπότε κάνουμε override την μέθοδο doPost και αναζητούμε μέσα στο μοντέλο μας για τον user αυτό. Ο mvc βρίσκει τον controller που απαντάει στο request /login/user μέσω του annotation

```
@WebServlet(name = "Api", urlPatterns = "/patient/api")
```

```
public class PatientApiController extends HttpServlet{
```

Σχήμα 5.5: Δήλωση του δυναμικού api.

και εκτελεί την doPost μέθοδο του αντίστοιχου controller. Σε αυτό το σημείο πρέπει να πούμε ότι ο εκτελείτε η αντίστοιχη μέθοδος σε αντιστοίχιση με το request. Δηλαδή αν το request μας είναι τύπου HTTP GET request τότε ο η μέθοδος doGet θα είναι αυτή που θα εκτελεστεί αν είναι HTTP Post τότε η doPost είναι αυτή που θα εκτελεστεί και ούτω κάθε εξής. Οπότε κάνοντας override την doPost μιας και το request μας θα είναι ένα post request εμπλουτίζουμε το session object με το pojo object () του ευρεθέντος χρήστη. Αυτό μας δίνει την δυνατότητα να γνωρίζουμε αν αυτός ο χρήστης είναι authenticated και στα επόμενα request που θα κάνει για να μπορέσουμε να του παρέχουμε το κατάλληλο content.

```
@Override
protected void doPost(HttpServletRequest request, HttpServletResponse resp)
throws ServletException, IOException {
```

```
String userName = request.getParameter("username");
String password = request.getParameter("password");
UserItem userItem = userDao.getUserByUsernameAndPassWord(userName,
password);

        String responceRedirect = "";
        if (userItem != null) {
            UserRole userRole = userItem.getUserRole();
            request.getSession().invalidate();
            request.getSession().setAttribute("username", userItem);
```

Σχήμα 5.6: Προώθηση του αιτήματος σε ένα view και καλώντας την override doPost().

Στην συνέχεια καλείτε να δώσει content στον χρήστη ανάλογα την φύση του χρήστη που συνδέθηκε. Οπότε αν ο χρήστης είναι ένας γιατρός τότε θα πρέπει να δει τον content των ασθενών του, αν ο χρήστης είναι διαχειριστής τότε θα πρέπει να μπορεί να τροποποιήσει η να εισάγει κάποια κριτήρια. Για αυτό το λόγο θα χρειαστούμε να ανακατευθύνουμε τον χρήστη στην αντίστοιχη οθόνη. Είναι λειτουργεία του controller και αυτή. Οπότε στην δια

μέθοδο κάνουμε έναν έλεγχο με τον παρακάτω τρόπο:

```
switch (userRole) {
    case ADMIN:
        responceRedirect = "/admin";
        break;
    case DOCTOR:
        responceRedirect = "/doctor";
        break;
    case EMPTY:
        responceRedirect = "index.jsp";
        break;
}
```

Σχήμα 5.7: Έλεγχος του routing.

Οπότε ανακατευθύνουμε τον admin user στον admin controller και τον doctor στον doctor controller.

Αναλύοντας παραπάνω τις διαδικασίες βλέπουμε άλλους 2 controller:

Τον doctor controller που δίνει την δυνατότητα στον χρήστη να βλέπει τους ασθενείς του σε μια λίστα καθώς και να προσθέτει καινούργιους ασθενείς. Αυτός ο controller ακούει στο path /doctor και εμπλουτίζει με δεδομένα και προωθεί το ερώτημα με σε ένα jsp για την εμφάνιση των δεδομένων.

```
@WebServlet( name = "doctor", urlPatterns = "/doctor")
public class DoctorController extends HttpServlet {
```

Οπότε για την εμφάνιση των δεδομένων κάνουμε override την doGet:

```
@Override
protected void doGet(HttpServletRequest req, HttpServletResponse resp)
throws ServletException, IOException {

    UserItem userItem = (UserItem)
req.getSession().getAttribute("username");
    if(userItem.getUserRole()!= UserRole.DOCTOR) {
        RequestDispatcher requestDispatcher =
req.getRequestDispatcher("login.jsp");
        requestDispatcher.forward(req, resp);
```

```
            return;
        }

        DoctorItem doctorItem = doctorDao.getDoctorItemByUser(userItem);
        req.setAttribute("doctorName",userItem.getUserName());
        req.setAttribute("doctorType",doctorItem.getDoctorType());
        RequestDispatcher requestDispatcher =
req.getRequestDispatcher("doctor.jsp");
        requestDispatcher.forward(req, resp);

}
```

Σχήμα 5.7: Ανακατεύθυνση σύμφωνα με τον χρήστη.

Εδώ βλέπουμε ότι χρησιμοποιούμε το request object για να του περάσουμε κάποια δεδομένα που θα είναι χρήσιμα στο χρήστη όπως είναι το όνομα του χρήστη - γιατρού για αυτή την περίπτωση και καθώς και τον τύπο του γιατρού για παράδειγμα καρδιολόγος ορθοπεδικός κλπ. Στην συνέχεια προωθούμε το request αυτό σε ένα jsp για την παρουσίαση των δεδομένων. Αυτό είναι ένα παράδειγμα για το πως μπορεί να ξεδιπλωθεί όλη εφαρμογή μας και να υλοποιηθεί μέχρι τέλους.

Βιβλιογραφία

1. Alexander, Christopher, et al. *A pattern language*. Gustavo Gili, 1977.
2. Shaw, Mary, and David Garlan. *Software architecture: perspectives on an emerging discipline*. Vol. 1. Englewood Cliffs: Prentice Hall, 1996.
3. Fowler, Martin. *Patterns of enterprise application architecture*. Addison-Wesley Longman Publishing Co., Inc., 2002.
4. Erl, Thomas. *Soa: principles of service design*. Prentice Hall Press, 2007.
5. Deterding, Sebastian, et al. "From game design elements to gamefulness: defining gamification." *Proceedings of the 15th international academic MindTrek conference: Envisioning future media environments*. ACM, 2011.
6. Tou, Julius T., and Rafael C. Gonzalez. "Pattern recognition principles." (1974).
7. Wooldridge, Michael, Nicholas R. Jennings, and David Kinny. "The Gaia methodology for agent-oriented analysis and design." *Autonomous Agents and multi-agent systems* 3.3 (2000): 285-312.
8. van Der Aalst, Wil MP, et al. "Workflow patterns." *Distributed and parallel databases* 14.1 (2003): 5-51.
9. Buschmann, Frank, Kelvin Henney, and Douglas Schimdt. *Pattern-oriented Software Architecture: On Patterns and Pattern Language*. Vol. 5. John Wiley & Sons, 2007.
10. Kitano, Hiroaki. "Systems biology: a brief overview." *Science* 295.5560 (2002): 1662-1664.
11. Lea, Douglas. *Concurrent programming in Java: design principles and patterns*. Addison-Wesley Professional, 2000.
12. Alur, Deepak, et al. *Core J2EE Patterns (Core Design Series): Best Practices and Design Strategies*. Sun Microsystems, Inc., 2003.
13. Morton, J., and J. J. Odell. *Object oriented analysis and design*. Englewood Cliffs (New Jersey): Prentice-Hall, 1992.
14. Grand, Mark. *Patterns in Java: a catalog of reusable design patterns illustrated with UML*. John Wiley & Sons, 2003.
15. Shi, Nija, and Ronald A. Olsson. "Reverse engineering of design patterns from java source code." *Automated Software Engineering, 2006. ASE'06. 21st IEEE/ACM International Conference on*. IEEE, 2006.
16. Cooper, James William. *Java design patterns: a tutorial*. Addison-Wesley Professional, 2000.
17. Fowler, Martin. *Patterns of enterprise application architecture*. Addison-Wesley Longman Publishing Co., Inc., 2002.
18. Shi, Nija, and Ronald A. Olsson. "Reverse engineering of design patterns from java source code." *Automated Software Engineering, 2006. ASE'06. 21st IEEE/ACM International Conference on*. IEEE, 2006.
19. Distefano, Dino, and Matthew J. Parkinson J. "jStar: Towards practical verification for

Java." *ACM Sigplan Notices*. Vol. 43. No. 10. ACM, 2008.
20. Grand, Mark. *Patterns in Java: a catalog of reusable design patterns illustrated with UML*. John Wiley & Sons, 2003.
21. Cox, B.(1986), Object – Oriented Programming :An Evolutionary Approach, Reading , MA: Addison -Wesley
22. Pooley, Robert J. An introduction to programming in SIMULA. Blackwell Scientific Publications, Ltd., 1987.
23. Dahl, Ole-Johan. "The roots of object orientation: the Simula language." *Software pioneers*. Springer Berlin Heidelberg, 2002. 78-90.
24. Goldberg, Adele, and David Robson. *Smalltalk-80: the language and its implementation*. Addison-Wesley Longman Publishing Co., Inc., 1983.
25. Parnas, David Lorge. "Information distribution aspects of design methodology." (1971).
26. Booch, Grady. "UML in action." *Communications of the ACM* 42.10 (1999): 26-28.
27. Gosling, James. *The Java language specification*. Addison-Wesley Professional, 2000.
28. Wolfgang, Pree. *Design patterns for object-oriented software development*. Reading, Mass.: Addison-Wesley, 1994.
29. Schmidt, Douglas C. "Reactor: An object behavioral pattern for concurrent event demultiplexing and dispatching." (1995).
30. Van Welie, Martijn, and Gerrit C. Van der Veer. "Pattern languages in interaction design: Structure and organization." *Proceedings of interact*. Vol. 3. 2003.
31. Ellis, Brian, Jeffrey Stylos, and Brad Myers. "The factory pattern in API design: A usability evaluation." *Proceedings of the 29th international conference on Software Engineering*. IEEE Computer Society, 2007.
32. America, Pierre. "POOL-T: A parallel object-oriented language." *Object-oriented concurrent programming*. MIT press, 1987.
33. Hamilton, Graham, Rick Cattell, and Maydene Fisher. *JDBC Database Access with Java*. Vol. 7. Addison Wesley, 1997.
34. Gilb, Tom, and Susannah Finzi. *Principles of software engineering management*. Vol. 11. Reading, MA: Addison-Wesley, 1988.
35. Bucanek, James. "Model-view-controller pattern." *Learn Objective-C for Java Developers* (2009): 353-402.
36. Brogden, William B., and Chris Minnick. *Java Developer's Guide to E-commerce with XML and JSP*. Sybex, 2001.
37. Berkhemer, Olvert A., et al. "A randomized trial of intraarterial treatment for acute ischemic stroke." *N Engl J Med* 2015.372 (2015): 11-20.
38. Graham, Ian S. *The HTML sourcebook*. John Wiley & Sons, Inc., 1995.