ΠΑΝΕΠΙΣΤΗΜΙΟ ΠΕΛΟΠΟΝΝΗΣΟΥ

Επωφελείς Συνδυασμοί και Εφαρμογές των Αντικειμενοστρεφών Μοτίβων Σχεδίασης

του

Απόστολου Γρηγόρη - 2022202202006

Εκπόνηση διπλωματικής ως μέρος του
Π.Μ.Σ. στην Επιστήμη Υπολογιστών

στη
Σχολή Οικονομίας και Τεχνολογίας
Τμήμα Πληροφορικής και Τηλεπικοινωνιών

Επιβλέπων Καθηγητής: Δρ. Γρηγόριος Δημητρουλάκος
Συνεπιβλέπων Καθηγητής: Δρ. Κωνσταντίνος Βασιλάκης

12-03-2024

Δήλωση Συγγραφικής Ιδιότητας

Εγώ, ο Απόστολος Γρηγόρης, δηλώνω ότι αυτή η διπλωματική εργασία με τίτλο, επωφελείς συνδυασμοί και εφαρμογές των αντικειμενοστρεφών μοτίβων σχεδίασης, και η δουλειά που παρουσιάζεται σε αυτή, είναι δικά μου. Επιβεβαιώνω ότι:

- Αυτή η δουλειά πραγματοποιήθηκε ολοκληρωτικά κατά την υποψηφιότητά μου για τίτλο μεταπτυχιακών σπουδών σε αυτό το πανεπιστήμιο.
- Κανένα μέρος αυτής της πτυχιακής εργασίας δεν έχει προηγουμένως κατατεθεί για την απόκτηση πτυχίου ή άλλου τίτλου σε αυτό ή άλλο πανεπιστήμιο.
- Όπου έχω συμβουλευτεί την δημοσιευμένη δουλειά τρίτων, αυτό αποδίδεται ορθώς.
- Όπου έχω παραθέσει από δουλειά τρίτων, η πηγή δίνεται πάντα. Με εξαίρεση αυτές τις παραθέσεις, αυτή η πτυχιακή εργασία είναι εξ ολοκλήρου προσωπική μου δουλειά.
- Έχω παραθέσει όλες τις κύριες πηγές βοήθειας.

Υπογραφή:
Απόστολος Γρηγόρης

Ημερομηνία:
12 - 03 - 2024

ΠΑΝΕΠΙΣΤΗΜΙΟ ΠΕΛΟΠΟΝΝΗΣΟΥ

Σύνοψη

Σχολή Οικονομίας και Τεχνολογίας
Τμήμα Πληροφορικής και Τηλεπικοινωνιών

Μεταπτυχιακός Τίτλος Σπουδών

του Απόστολου Γρηγόρη - 2022202202006

Η παρούσα διπλωματική εργασία πραγματεύεται μια λεπτομερή μελέτη των αντικειμενοστρεφών μοτίβων σχεδίασης. Παρουσιάζει τα παραδοσιακά αντικειμενοστρεφή μοτίβα σχεδίασης του βιβλίου "Design Patterns: Elements of Reusable Object-Oriented Software"[1], και διερευνά τη συνεργιστική εφαρμογή αυτών των μοτίβων σχεδίασης, αποδεικνύοντας πώς ο συνδυασμός τους μπορεί να αντιμετωπίσει αποτελεσματικά πολύπλοκες σχεδιαστικές προκλήσεις, ενισχύοντας την αρθρωτότητα, την επεκτασιμότητα και τη συντηρησιμότητα των συστημάτων λογισμικού. Εξετάζονται διάφοροι συνδυασμοί μοτίβων και παρουσιάζεται η εφαρμογή τους σε πρακτικά σενάρια για την ανάπτυξη ευέλικτων, επεκτάσιμων και συντηρήσιμων αρχιτεκτονικών λογισμικού.

Ευχαριστίες

Η ολοκλήρωση της μεταπτυχιακής αυτής εργασίας θα ήταν αδύνατη χωρίς την πολύτιμη υποστήριξη, επίβλεψη και ποιοτική καθοδήγηση του επιβλέποντα καθηγητή μου, Δρ. Γρηγόριου Δημητρουλάκου. Του εκφράζω ένα βαθύ ευχαριστώ για όλη τη βοήθεια που μου προσέφερε.

Επίσης, ευχαριστώ τον συνεπιβλέποντα καθηγητή μου Δρ. Κωνσταντίνο Βασιλάκη που ήταν αρωγός σε αυτή την προσπάθεια.

Τέλος, ευχαριστώ την οικογένειά μου, για την ανεκτίμητη στήριξή τους.

Περιεχόμενα

Κατάλογος Σχημάτων

Κατάλογος Πινάκων

Εισαγωγή

Στο δυναμικό και εξελισσόμενο πεδίο της μηχανικής λογισμικού, η διαρκής αναζήτηση αποδοτικών στρατηγικών σχεδιασμού που να εξασφαλίζουν την ευελιξία, συντηρησιμότητα και επεκτασιμότητα των συστημάτων, αποτελεί μια συνεχή πρόκληση. Η συνεισφορά των Erich Gamma, Richard Helm, Ralph Johnson και John Vlissides, γνωστών ως ¨συμμορία των τεσσάρων¨, με τη δημοσίευση του βιβλίου τους "Design Patterns: Elements of Reusable Object-Oriented Software" το 1994 [1], έθεσε ένα σημαντικό ορόσημο στον τομέα. Τα μοτίβα σχεδίασης που παρουσιάζονται στο εν λόγω βιβλίο έχουν εξελιχθεί σε σημαντικά εργαλεία για αρχιτέκτονες και προγραμματιστές λογισμικού, και προσφέρουν επαναχρησιμοποιήσιμες λύσεις για κοινά προβλήματα σχεδιασμού. Ωστόσο, παρά την ευρύτητα της εφαρμογής τους, η χρήση τους απαιτεί υψηλό επίπεδο εμπειρίας και δεν είναι πάντα ικανά να αντιμετωπίσουν πιο σύνθετα ζητήματα από μόνα τους.

Η παρούσα διπλωματική εργασία επιχειρεί να εμβαθύνει στην μελέτη των μοτίβων σχεδίασης με δύο κύριους στόχους. Πρώτον, στοχεύει στην παροχή μιας λεπτομερούς εισαγωγής στα κλασσικά μοτίβα σχεδίασης που αναφέρονται στο βιβλίο, περιγράφοντας τις θεωρητικές τους βάσεις, παρουσιάζοντας τις δομές τους μέσω διαγραμμάτων κλάσης, και κατηγοριοποιώντας τα σε κατασκευαστικά, δομικά και συμπεριφορικά, ανάλογα με τη φύση τους. Η ανάλυση αυτή αποσκοπεί στην κατανόηση της βασικής δομής και των πιθανών εφαρμογών τους. Δεύτερον, επιδιώκει να εξερευνήσει τους τρόπους με τους οποίους αυτά τα μοτίβα μπορούν να συνδυαστούν για τη δημιουργία προηγμένων αρχιτεκτονικών λύσεων, ανταποκρινόμενων στην αυξανόμενη πολυπλοκότητα και κλίμακα των σύγχρονων εφαρμογών λογισμικού. Μέσω εμπειρικών μελετών περίπτωσης, η διπλωματική θα αποδείξει την αποτελεσματικότητα των συνδυασμών μοτίβων στην επίλυση σύνθετων αρχιτεκτονικών προκλήσεων, επισημαίνοντας τα οφέλη και τις προκλήσεις της εφαρμογής τους.

Με έναν συνδυασμό θεωρητικής ανάλυσης και πρακτικής εφαρμογής, αυτή η εργασία αποσκοπεί να προσφέρει μια εμπεριστατωμένη βιβλιογραφική ανασκόπηση των μοτίβων σχεδίασης και να την εμπλουτίσει με πραγματικές περιπτώσεις εφαρμογής. Οι μελέτες περίπτωσης προσφέρουν πρακτικά παραδείγματα του πώς μπορούν να εφαρμοστούν επιτυχώς καινοτόμοι συνδυασμοί μοτίβων σχεδίασης, παρέχοντας σημαντικές γνώσεις για τα πλεονεκτήματα και τις προκλήσεις που συνδέονται με αυτούς.

Ο κώδικας που παρουσιάζεται σε αυτή τη διπλωματική εργασία αναπτύχθηκε με τη χρήση της γλώσσας προγραμματισμού C#, είναι πλήρως εκτελέσιμος και έχει δοκιμαστεί για να διασφαλιστεί η λειτουργικότητα του. Το κύριο περιβάλλον ανάπτυξης (IDE) που χρησιμοποιήθηκε, είναι το Visual Studio Code (VS Code).

Κεφάλαιο 1

Περίληψη - Αντικείμενο της Διπλωματικής

Η παρούσα διπλωματική εργασία διακρίνεται σε δύο κύρια μέρη. Στο πρώτο μέρος, το οποίο αποτυπώνεται στο δεύτερο κεφάλαιο, διερευνώνται τα παραδοσιακά αντικειμενοστρεφή μοτίβα σχεδίασης του βιβλίου "Design Patterns: Elements of Reusable Object-Oriented Software"[1], τα οποία κατηγοριοποιούνται σε τρείς κατηγορίες, καθεμία εκ των οποίων εξετάζεται διεξοδικά:

Κατασκευαστικά:

Τα κατασκευαστικά μοτίβα επικεντρώνονται σε ευέλικτες μεθοδολογίες κατασκευής αντικειμένων, και διασφαλίζουν ότι η κύρια αρχιτεκτονική του συστήματος παραμένει αποσυνδεδεμένη από τις λεπτομέρειες της δημιουργίας αντικειμένων. Αυτό προωθεί την αρθρωτότητα και την επεκτασιμότητα. Τα κύρια κατασκευαστικά μοτίβα είναι τα ακόλουθα:

- Factory Method: Ορίζει μια διεπαφή δημιουργίας αντικειμένων, και επιτρέπει παράλληλα στις υποκλάσεις να προσαρμόζουν τον τύπο του επιστρεφόμενου αντικειμένου.
- Abstract Factory: Παρέχει μια διεπαφή δημιουργίας οικογενειών συναφών αντικειμένων χωρίς να προσδιορίζει τις συγκεκριμένες κλάσεις τους.
- Builder: Επιτρέπει τη σταδιακή κατασκευή σύνθετων αντικειμένων και διαχωρίζει τη διαδικασία κατασκευής τους από την αναπαράστασή τους.

- Prototype: Επικεντρώνεται στην κλωνοποίηση αντικειμένων και επιτρέπει σε αντικείμενα να δημιουργούν αντίγραφα του εαυτού τους.
- Singleton: Εξασφαλίζει ότι μια κλάση έχει μόνο ένα στιγμιότυπο και παρέχει καθολική πρόσβαση σε αυτό.

Αυτά τα μοτίβα εξορθολογίζουν τη διαδικασία δημιουργίας αντικειμένων και προάγουν τη συντηρησιμότητα και την επεκτασιμότητα του κώδικα, ωστόσο μπορεί να εισάγουν πολυπλοκότητα και να απαιτούν τον ορισμό πολλαπλών κλάσεων.

Δομικά:

Τα δομικά μοτίβα ασχολούνται με τη σύνθεση κλάσεων ή αντικειμένων για την κατασκευή μεγαλύτερων δομών. Συμβάλλουν στη διασφάλιση ότι πιθανές αλλαγές σε ένα μέρος του συστήματος, δεν απαιτούν εκτεταμένες τροποποιήσεις στο υπόλοιπο σύστημα, καθιστώντας το έτσι, πιο ευέλικτο σε μεταβολές.

- Adapter: Επιτρέπει σε αντικείμενα κλάσεων με ασύμβατες διεπαφές, να συνεργάζονται μεταξύ τους.
- Bridge: Αποσυνδέει μια αφαίρεση από την υλοποίησή της, επιτρέποντάς τους να αναπτύσσονται ανεξάρτητα μεταξύ τους.
- Composite: Επιτρέπει την ομοιόμορφη αντιμετώπιση μεμονωμένων και σύνθετων αντικειμένων, και απλοποιεί τις αλληλεπιδράσεις με αυτά.
- Decorator: Προσθέτει δυναμικά συμπεριφορές σε υφιστάμενα αντικείμενα κλάσεων.
- Facade: Παρέχει μια απλή διεπαφή που απλοποιεί την πρόσβαση σε πολύπλοκα συστήματα.
- Flyweight: Μειώνει την κατανάλωση μνήμης, με το διαμοιρασμό κοινών δεδομένων, μεταξύ παρόμοιων αντικειμένων.
- Proxy: Ελέγχει την πρόσβαση σε αντικείμενα και τους προσθέτει λειτουργικότητες όπως το lazy loading.

Αυτά τα μοτίβα διευκολύνουν την κατασκευή πολύπλοκων, αλλά επεκτάσιμων και συντηρήσιμων αρχιτεκτονικών λογισμικού, ωστόσο μπορεί να εισάγουν πρόσθετα επίπεδα αφαίρεσης, και πολυπλοκότητα.

Συμπεριφορικά:

Τα συμπεριφορικά μοτίβα ασχολούνται με την αποτελεσματική επικοινωνία μεταξύ των αντικειμένων. Βοηθούν στον ορισμό του τρόπου αλληλεπίδρασης των αντικειμένων με τρόπο που να αυξάνει την ευελιξία στη διεξαγωγή της επικοινωνίας.

- Chain of Responsibility: Επιτρέπει τη μεταβίβαση ενός αιτήματος κατά μήκος μίας αλυσίδας χειριστών.
- Command: Ενθυλακώνει αιτήματα σε αυτόνομα αντικείμενα που περιλαμβάνουν όλες τις απαραίτητες πληροφορίες που αφορούν το κάθε αίτημα.
- Iterator: Επιτρέπει την διάσχιση των στοιχείων μιας συλλογής, χωρίς να αποκαλύπτει λεπτομέρειες που αφορούν τη δομή της.
- Mediator: Μειώνει την άμεση επικοινωνία μεταξύ αντικειμένων, μέσω της εισαγωγής μια κεντρικής αρχής που διαχειρίζεται τον τρόπο αλληλεπίδρασης αυτών.
- Memento: Επιτρέπει σε αντικείμενα να αποθηκεύουν και να επαναφέρουν προηγούμενες καταστάσεις τους, χωρίς να εκθέτουν λεπτομέρειες της υλοποίησής τους.
- Observer: Ορίζει μια εξάρτηση ένα-προς-πολλά μεταξύ αντικειμένων, έτσι ώστε όταν ένα αντικείμενο αλλάζει κατάσταση, όλα τα εξαρτώμενα αντικείμενά του να ειδοποιούνται και να ενημερώνονται αυτόματα.
- State: Επιτρέπει σε αντικείμενα να μεταβάλλουν τη συμπεριφορά τους, όταν αλλάζει η εσωτερική τους κατάσταση.
- Strategy: Ορίζει μια οικογένεια αλγορίθμων, ενθυλακώνει κάθε έναν από αυτούς σε ξεχωριστή κλάση, και τους καθιστά εναλλάξιμους.
- Template Method: Ορίζει το σκελετό ενός αλγορίθμου σε κάποια υπερκλάση, και επιτρέπει σε υποκλάσεις της, να υπερκαλύπτουν ορισμένα βήματα χωρίς να αλλάζουν τη δομή του.
- Visitor: Επιτρέπει την προσθήκη νέων λειτουργιών σε υπάρχουσες δομές αντικειμένων, χωρίς να τις τροποποιεί.

Αυτά τα μοτίβα εξορθολογίζουν τον τρόπο αλληλεπίδρασης των αντικειμένων και τις διαδικασίες λήψης αποφάσεων, βελτιστοποιώντας την επικοινωνία τους και καθιστώντας τον κώδικα πιο εύκολα τροποποιήσιμο και συντηρήσιμο.

Το δεύτερο μέρος που είναι κατ΄ ουσίαν το κεφάλαιο υπ΄ αριθμό 3, ερευνά τη συνεργιστική εφαρμογή των αντικειμενοστρεφών μοτίβων σχεδίασης. Σε αντίθεση με την παραδοσιακή προσέγγιση της εφαρμογής αυτών των μοτίβων μεμονωμένα, αυτό το κεφάλαιο παρουσιάζει το πως ο συνδυασμός τους μπορεί να αντιμετωπίσει πιο αποτελεσματικά πολύπλοκες προκλήσεις σχεδίασης, ενισχύοντας έτσι την αρθρωτότητα, την επεκτασιμότητα και τη συντηρησιμότητα των συστημάτων λογισμικού.

Συνδυασμός Builder & Abstract Factory:

Στο υποκεφάλαιο 3.1, εξετάζεται ο συνδυασμός των Builder και Abstract Factory μοτίβων με σκοπό την αποτελεσματική δημιουργία πολύπλοκων αντικειμένων, των οποίων τα επιμέρους στοιχεία μπορεί να έχουν διαφορετικές υλοποιήσεις. Το σενάριο επικεντρώνεται σε ένα σύστημα διαμόρφωσης Η/Υ που στοχεύει στην κατασκευή Η/Υ, χρησιμοποιώντας εξαρτήματα που ποικίλλουν ανάλογα με τον επιδιωκόμενο σκοπό του Η/Υ. Το Abstract Factory χρησιμοποιείται για τη δημιουργία οικογενειών συναφών αντικειμένων (π.χ. εξαρτήματα Η/Υ) αποκρύπτοντας τις συγκεκριμένες κλάσεις τους, ενώ το μοτίβο Builder ορίζει τη διαδικασία κατασκευής ενός υπολογιστή βήμα προς βήμα. Αυτή η προσέγγιση επιτρέπει την εύκολη εναλλαγή οικογενειών εξαρτημάτων που χρησιμοποιούνται για τη σύνθεση ενός Η/Υ, μέσω αλλαγής του χρησιμοποιούμενου εργοστασίου, προσαρμόζοντας έτσι τη διαμόρφωση του. Η υλοποίηση σε C# και το διάγραμμα κλάσης παρουσιάζουν το πως αυτά τα μοτίβα συνεργάζονται για την κατασκευή ενός ευέλικτου και κλιμακώσιμου τέτοιου συστήματος. Ωστόσο, δεν αποφεύγεται η αυξημένη πολυπλοκότητα που προκύπτει από την εισαγωγή πολλών κλάσεων και διεπαφών.

Συνδυασμός Builder & Composite :

Στο υποκεφάλαιο 3.2, ερευνάται ο συνδυασμός των Builder και Composite μοτίβων με σκοπό την παροχή μιας δομημένης προσέγγισης για την κατασκευή πολύπλοκων ιεραρχιών δεδομένων, όπως είναι είναι ένα σύστημα αρχείων. Το Builder μοτίβο ενσωματώνει τη λογική κατασκευής του συστήματος αρχείων, ενώ το Composite μοτίβο παρέχει μια ενιαία διεπαφή για τη διαχείριση τόσο των σύνθετων αντικειμένων (φάκελοι) όσο και των κόμβων φύλλων (αρχεία), διευκολύνοντας λειτουργίες όπως ο υπολογισμός του συνολικού μεγέθους του συστήματος. Αυτή η προσέγγιση παρουσιάζει υψηλή επεκτασιμότητα, καθώς επιτρέπει την προσθήκη νέων τύπων κόμβων και λειτουργιών με ελάχιστο αντίκτυπο στη συνολική πολυπλοκότητα. Ωστόσο, η χρήση του Builder για τη διαχείριση Composite δομών ίσως προκαλεί επιβάρυνση στην απόκριση, ως απόρροια της προσθήκης επιπέδου αφαίρεσης πάνω από αυτές.

Συνδυασμός Iterator & Factory Method:

Στο υποκεφάλαιο 3.3, εξετάζεται ο συνδυασμός των Iterator και Factory Method μοτίβων με σκοπό την αποτελεσματική δημιουργία ενός αξιόπιστου μηχανισμού παροχής αλγορίθμων διάσχισης συλλογών με κοινή διεπαφή, που θα αποκρύπτει τις λεπτομέρειες υλοποίησης αυτών. Η υλοποίηση σε C# και το διάγραμμα κλάσης, παρουσιάζουν τον τρόπο με τον οποίο το Factory Method μοτίβο ορίζει έναν ομοιόμορφο τρόπο δημιουργίας αλγορίθμων διάσχισης μέσω της διεπαφής ICollectionCreator και των κλάσεων ListCreator HeapCreator που λειτουργούν ως εργοστάσια για τους Ιτερατορς λίστας και σωρού, αντίστοιχα. Επίσης, παρουσιάζουν το πως το μοτίβο Iterator τυποποιεί τη συμπεριφορά διάσχισης σε όλες τις συλλογές, μέσω της διεπαφής IIterator.o συνδυασμός επιδεικνύει σημαντική ευελιξία και προσαρμοστικότητα, καθώς επιτρέπει την εισαγωγή νέων τύπων συλλογών και αλγορίθμων διάσχισης με ελάχιστες αλλαγές, και προσφέρει μια κοινή διεπαφή για τη διάσχιση συλλογών, χωρίς να εκτίθενται οι υποκείμενες λεπτομέρειες υλοποίησης στον κώδικα πελάτη. Ωστόσο, το επίπεδο αφαίρεσης που εισάγεται πάνω από τις συλλογές μπορεί να προκαλέσει κάποια επίπτωση στο χρόνο απόκρισης τους.

Συνδυασμός Iterator & Composite:

Στο υποκεφάλαιο 3.4, ερευνάται ο συνδυασμός των Iterator και Composite μοτίβων με σκοπό την αποτελεσματική δημιουργία ενός αξιόπιστου μηχανισμού διάσχισης πολύπλοκων ιεραρχικών δομών. Ο συνδυασμός επιτυγχάνει την κατασκευή αυτού του μηχανισμού και παράλληλα αποκρύπτει τις λεπτομέρειες της εσωτερικής δομής, από τον κώδικα πελάτη. Το Composite μοτίβο διαχειρίζεται τις σχέσεις μεταξύ των στοιχείων και επιτρέπει την πολυμορφική αντιμετώπισή τους, ενώ το Iterator μοτίβο παρέχει έναν τυποποιημένο τρόπο προσπέλασης και διάσχισης τους. Αυτή η προσέγγιση απλοποιεί τον κώδικα πελάτη, καθώς αποκρύπτονται οι λεπτομέρειες υλοποίησης των αλγορίθμων διάσχισης και αναπαραστάσεων των δομών. Η εφαρμογή της συνίσταται για δομές με λίγους τύπους κόμβων και αλγορίθμους διάσχισης, αλλά μπορεί να γίνει πολύπλοκη για μεγαλύτερες δομές με πολλούς αλγορίθμους διάσχισης.

Συνδυασμός Visitor & Composite:

Το υποκεφάλαιο 3.5, εξετάζει το συνδυασμό των Visitor και Composite μοτίβων με σκοπό τη διευκόλυνση προσθήκης και εκτέλεσης νέων λειτουργιών σε πολύπλοκες ιεραρχικές δομές αντικειμένων, χωρίς αυτές να μεταβάλλονται. Αυτή η προσέγγιση επιτρέπει τη δυναμική επέκταση της λειτουργικότητας των στοιχείων μιας ιεραρχίας, όπως είναι μια οργανωτική δομή τμημάτων και υπαλλήλων, καθώς αποσυνδέει τη λογική εκτέλεσης λειτουργιών από τη

δομική σύνθεση. Μέσω της υλοποίησης ενός παραδείγματος σε C#, επιδεικνύεται το πώς επισκέπτες μπορούν να πλοηγούνται στα στοιχεία μιας ιεραρχίας τμημάτων - υπαλλήλων και να εκτελούν λειτουργίες σε αυτά, χωρίς να τροποποιείται η υφιστάμενη δομή. Ο συνδυασμός φαίνεται να προσφέρει σημαντική ευελιξία και προσαρμοστικότητα στην προσθήκη νέων λειτουργιών, ωστόσο αυτή η ενσωμάτωση μπορεί να εγείρει ανησυχίες σχετικά με την ενθυλάκωση και την επεκτασιμότητα, ειδικά όταν διαχειρίζεται πολυάριθμους επισκέπτες και παράλληλα εισάγονται συχνά νέοι τύποι κόμβων. Αντιθέτως, αναδεικνύεται σε σενάρια όπου λειτουργίες προστίθενται και εκτελούνται σε σχετικά σταθερές (ως προς τους τύπους κόμβων) σύνθετες δομές.

Συνδυασμός Chain of Responsibility & Composite:

Το υποκεφάλαιο 3.6, μελετά το συνδυασμό των Chain of Responsibility και Composite μοτίβων με σκοπό τη διαχείριση συμβάντων σε πολύπλοκες ιεραρχικές δομές αντικειμένων, όπως ιεραρχίες στοιχείων UI. Αυτός ο συνδυασμός επιτρέπει την επεξεργασία συμβάντων στο εκάστοτε επίπεδο ιεραρχίας που είναι καταλληλότερο να την εκτελέσει, μέσω υποβολής των συμβάντων σε διάσχιση των επιπέδων της ιεραρχίας μέχρι αυτά να υποστούν επεξεργασία ή να απορριφθούν εν τέλει από τον κόμβο ρίζα. Παρουσιάζεται ένα παράδειγμα συστήματος UI στοιχείων (κουμπιά και πάνελ), και σενάρια επέκτασης πάνω σε αυτό, που αποδεικνύουν το πώς νέα στοιχεία UI και νέοι τύποι συμβάντων μπορούν να εισαχθούν απρόσκοπτα με ελάχιστες τροποποιήσεις. Ενώ αυτή η προσέγγιση ενισχύει την επεκτασιμότητα του συστήματος και την ευελιξία στην προσθήκη νέων στοιχείων και δυνατοτήτων διαχείρισης συμβάντων, προσθέτει επίσης πολυπλοκότητα στην παρακολούθηση της ροής των συμβάντων και μπορεί να επηρεάσει την απόδοση σε βαθιές ένθετες ιεραρχίες.

Συνδυασμός Chain of Responsibility & Command:

Το υποκεφάλαιο 3.7, διερευνά το συνδυασμό των Chain of Responsibility και Command μοτίβων και παρουσιάζει δύο διαφορετικές προσεγγίσεις για το πως μπορεί να επιτευχθεί αυτό. Στην πρώτη προσέγγιση, τα Command αντικείμενα αντιπροσωπεύουν τους χειριστές της αλυσίδας και συνθέτουν μια pipeline. Κάθε αντικείμενο Command ειδικεύεται σε μια συγκεκριμένη λειτουργία, την οποία εκτελεί πάνω στα δεδομένα που δέχεται ως αιτήματα. Έπειτα, μεταβιβάζει τα μεταμορφωμένα πλέον δεδομένα στον επόμενο χειριστή της αλυσίδας. Το εξεταζόμενο σενάριο, υλοποιημένο σε C#, παρουσιάζει την κατασκευή μιας pipeline που επικυρώνει, μεταμορφώνει, εμπλουτίζει και τέλος επεξεργάζεται δεδομένα που λαμβάνει ως είσοδο - αίτημα. Αυτή η αρχιτεκτονική επιτρέπει τον καθαρό διαχωρισμό ανησυχιών, κατά τον οποίο κάθε στάδιο επεξεργασίας επικεντρώνεται σε μια συγκεκριμένη

λειτουργία, ενισχύοντας έτσι την προσαρμοστικότητα και τη συντηρησιμότητα σε σενάρια που απαιτούν δημιουργία δυναμικά μεταβαλλόμενων pipeline. Στη δεύτερη προσέγγιση, τα Command αντικείμενα αντιπροσωπεύουν τα αιτήματα της αλυσίδας και ενθυλακώνουν τις πληροφορίες που είναι απαραίτητες για την εκτέλεση κάποιας λειτουργίας. Αντιθέτως, οι χειριστές της αλυσίδας αντιπροσωπεύουν διαφορετικά πλαίσια επεξεργασίας. Επομένως, με την τροφοδότηση ενός αιτήματος σε αυτή τη δομή, επιτυγχάνεται εκτέλεση της ίδιας εντολής σε διαφορετικά πλαίσια επεξεργασίας. Το εξεταζόμενο σενάριο που παρουσιάζεται σε C#, αφορά τη διαχείριση της ροής υπογραφής εγγράφων σε έναν οργανισμό. Η λειτουργία της υπογραφής ενσωματώνεται σε ένα αντικείμενο Command και τα διαφορετικά τμήματα του οργανισμού αναπαριστούν τους διαφορετικούς χειριστές - πλαίσια επεξεργασίας. Και αυτή η προσέγγιση αποδεικνύεται οτι παρουσιάζει υψηλή προσαρμοστικότητα και συντηρησιμότητα, και οτι επιτυγχάνει καθαρό διαχωρισμό ανησυχιών.

Συνδυασμός Command & Memento:

Το υποκεφάλαιο 3.8 εξετάζει το συνδυασμό των Command και Memento μοτίβων με σκοπό την υλοποίηση ανακλητών λειτουργιών σε εφαρμογές. Οι λεπτομέρειες των λειτουργιών ενθυλακώνονται στα αντικείμενα Command, και η αναστρεψιμότητα εξασφαλίζεται από τη χρήση του Memento. Παρουσιάζεται ένα παράδειγμα υλοποιημένο σε C# που αφορά την κατασκευή ενός μηχανισμού λήψης και επαναφοράς στιγμιοτύπων της κατάστασης εγγράφων. Κατά το παράδειγμα αυτό, τα αντικείμενα Command υλοποιούν λειτουργίες εκτέλεσης και αναίρεσης εντολών, τα Memento αντικείμενα αντιπροσωπεύουν στιγμίοτυπα καταστάσεων, και μια κλάση με όνομα CommandInvoker επιτελεί το ρόλο του Invoker για το Command μοτίβο και του Caretaker για το Memento. Ο εξεταζόμενος συνδυασμός φαίνεται να διατηρεί σαφή διαχωρισμό ανησυχιών και να παρουσιάζει υψηλή προσαρμοστικότητα και συντηρησιμότητα. Ωστόσο, η διαχείριση στιγμιότυπων αντικειμένων, θα μπορούσε να επηρεάσει αρνητικά την απόδοση, όσον αφορά την εισαγωγή καθυστέρησης και τη χρήση μνήμης.

Συνδυασμός Command, State, Strategy, Composite & Singleton:

Το υποκεφάλαιο 3.9 ερευνά μια περίπτωση χρήσης που συνδυάζει τα Command, State, Strategy, Composite και Singleton μοτίβα, με σκοπό την κατασκευή ενός ευέλικτου συστήματος ελέγχου ασφαλείας, το οποίο θα μπορεί να κλειδώνει, να ξεκλειδώνει και να ανταποκρίνεται σε παραβιάσεις ασφαλείας μέσω ενός κεντρικού σημείου ελέγχου, καμερών, αισθητήρων κίνησης και ζωνών ασφαλείας. Η εν λόγω περίπτωση χρήσης υλοποιήθηκε σε C#, και τα εμπλεκόμενα μοτίβα χρησιμοποιούνται ως εξής: το Command μοτίβο υλοποιεί τις ενέργειες

κλειδώματος και ξεκλειδώματος ως εντολές, ενώ το State μοτίβο διαχειρίζεται τις καταστάσεις των στοιχείων (κλειδωμένα ή ξεκλείδωτα), βάση των οποίων καθορίζεται ο τρόπος απόκρισης τους. Έπειτα, το Strategy μοτίβο επιτρέπει τη δυναμική αλλαγή στρατηγικών απόκρισης, όπως ειδοποίηση της αστυνομίας ή ενεργοποίηση της σειρήνας. Το Composite οργανώνει το σύστημα σε μια ιεραρχική δομή ζωνών, καμερών και αισθητήρων και επιτρέπει τη συλλογική αλλαγή της κατάστασης των στοιχείων του συστήματος. Τέλος, το Singleton μοτίβο συγκεντρώνει τη διαχείριση του συστήματος, αποτελώντας το κεντρικό σημείο ελέγχου που παράλληλα διαχειρίζεται και την επιλεγμένη στρατηγικής απόκρισης. Αυτή η αρχιτεκτονική παρουσιάζει και αποδεικνύει την ικανότητα των μοτίβων σχεδίασης να κατασκευάζουν σύνθετα συστήματα που υλοποιούν πληθώρα συμπεριφορών, και επιτυγχάνουν παράλληλα να διατηρούνται εύκολα επεκτάσιμα και συντηρήσιμα. Μειονέκτημα φαίνεται να είναι η επιβάρυνση της αρχιτεκτονικής με πολλές κλάσεις και διεπαφές.

Κεφάλαιο 2

Αντικειμενοστρεφή Μοτίβα Σχεδίασης

Το τρέχον κεφάλαιο μελετά τα παραδοσιακά αντικειμεντοστρεφή μοτίβα σχεδίασης που παρουσιάζονται στο βιβλίο "Design Patterns: Elements of Reusable Object-Oriented Software" [1]. Αυτά τα μοτίβα αποτελούν τυποποιημένες λύσεις σε κοινές προκλήσεις σχεδιασμού της μηχανικής λογισμικού και διακρίνονται σε τρείς κατηγορίες: κατασκευαστικά, δομικά και συμπεριφορικά.

2.1 Creational Patterns

Τα κατασκευαστικά μοτίβα σχεδίασης παίζουν κομβικό ρόλο στον αντικειμενοστρεφή προγραμματισμό, καθώς προσφέρουν ευέλικτες μεθοδολογίες δημιουργίας αντικειμένων. Διασφαλίζουν ότι η αρχιτεκτονική των αναπτυσσόμενων συστημάτων παραμένει αποσυνδεδεμένη από τις λεπτομέρειες που αφορούν στη δημιουργία αντικειμένων, προωθώντας έτσι την αρθρωτότητα και την επεκτασιμότητα του κώδικα. Κατ΄ ουσίαν, πραγματεύονται μηχανισμούς δημιουργίας αντικειμένων και επιδιώκουν να δημιουργούν αντικείμενα με τρόπο κατάλληλο για την εκάστοτε κατάσταση. Επίσης, συγκεντρώνουν τη διαδικασία δημιουργίας αντικειμένων σε ένα σημείο, επιτυγχάνοντας έτσι να μειώνουν την εξάρτηση από συγκεκριμένες κλάσεις και να ενισχύουν την ευελιξία και τη δυνατότητα επαναχρησιμοποίησης του κώδικα.

Τα κύρια κατασκευαστικά μοτίβα είναι τα ακόλουθα και θα αναλυθούν παρακάτω:

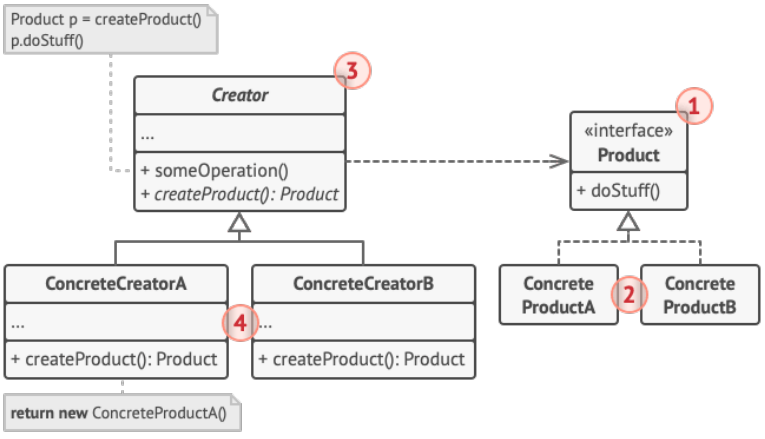
- Factory Method
- Abstract Factory
- Builder
- Prototype
- Singleton

2.1.1 Factory Method

Το Factory Method αποτελεί ένα κατασκευαστικό μοτίβο σχεδίασης το οποίο ενισχύει μία υπερκλάση με την προσθήκη μιας διεπαφής για την παραγωγή αντικειμένων. Παράλληλα, επιτρέπει σε κλάσεις που την επεκτείνουν, να προσαρμόζουν το είδος των παραγόμενων αντικειμένων τους.

Η βασική ιδέα είναι η χρήση μιας μεθόδου εργοστάσιο για την αρχικοποίηση αντικειμένων, αντί της απευθείας χρήσης κάποιου κατασκευαστή. Αυτή η προσέγγιση επιτρέπει μεγαλύτερη ευελιξία και αποσυνδέει τον κώδικα πελάτη από τις συγκεκριμένες κλάσεις που χρησιμοποιούνται στην εφαρμογή.

Σχετικά με τις λεπτομέρειες υλοποίησής του, το Factory Method χρησιμοποιεί μια αφηρημένη κλάση δημιουργό και μια διεπαφή προϊόν. Η κλάση δημιουργός περιλαμβάνει μια μέθοδο εργοστάσιο ή οποία επιστρέφει στιγμιότυπα κλάσεων που υλοποιουν τη διεπαφή προϊόν. Η μέθοδος αυτή είναι συνήθως αφηρημένη και πρέπει να υπερκαλύπτεται από υποκλάσεις της κλάσης δημιουργού.



Σχημα 2.1: Διάγραμμα Κλάσης Factory Method [2]

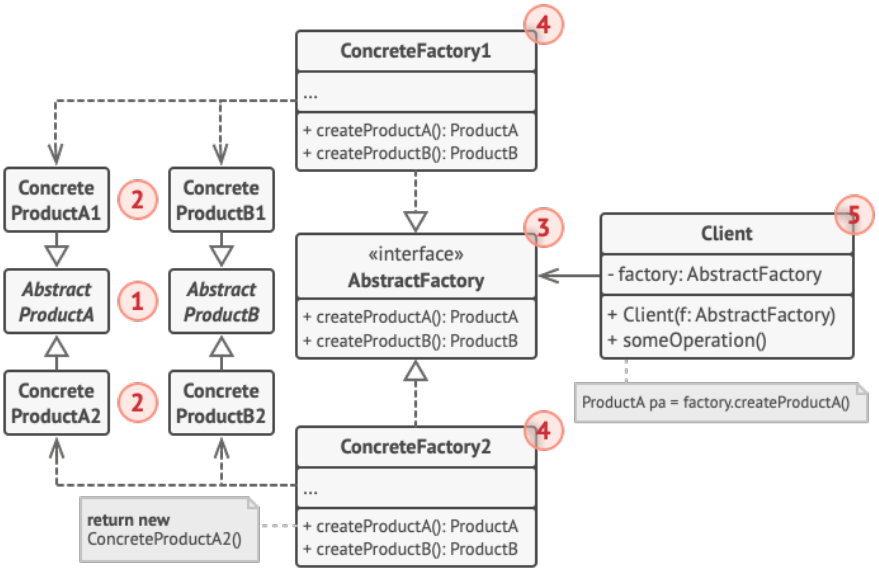
Όσον αφορά τα οφέλη χρήσης του μοτίβου, το Factory Method επιτρέπει την απομόνωση της διαδικασίας δημιουργίας προϊόντων σε ένα μέρος του προγράμματος, καθιστώντας τον κώδικο πιο εύκολο στη συντήρηση. Επίσης, αποτρέπει τη στενή σύζευξη μεταξύ κλάσεων, μέσω της αφαίρεσης που χρησιμοποιεί, και διαθέτει δομή που του επιτρέπει την εύκολη προσθήκη νέων τύπων προϊόντων, χωρίς να ελλοχεύει κίνδυνος πρόκλησης προβλημάτων στον κώδικα πελάτη. Ωστόσο, αξίζει να αναφερθεί ως μειονέκτημα, οτι η χρήση του μπορεί να οδηγήσει σε αυξημένη πολυπλοκότητα και εισαγωγή πολλαπλών υποκλάσεων.

2.1.2 Abstract Factory

Το Abstract Factory αποτελεί ένα σχεδιαστικό μοτίβο που εισάγει μια διεπαφή για τη δημιουργία οικογενειών συναφών αντικειμένων, χωρίς την ανάγκη να καθορίζεται η ακριβής κλάση τους. Η χρήση του μοτίβου, επιτρέπει σε ένα σύστημα να παραμένει ανεξάρτητο από τον τρόπο δημιουργίας, σύνθεσης και αναπαράστασης των προϊόντων του.

Σχετικα με την εφαρμογή του, το Abstract Factory χρησιμοποιείται κυρίως σε περιπτώσεις όπου είναι απαραίτητη η αλληλεπίδραση με πολλαπλές οικογένειες σχετιζόμενων προϊόντων και επιπλέον, δεν είναι επιθυμητή η εξάρτηση από τις συμπαγείς κλάσεις των προϊόντων, είτε επειδή αυτές δεν είναι εκ των προτέρων γνωστές, είτε διότι κρίνεται σκόπιμη η παροχή μόνο των διεπαφών στον κώδικα πελάτη, με σκοπό την απόκρυψη των λεπτομερειών υλοποίησης.

Η υλοποίηση του Abstract Factory περιλαμβάνει μια διεπαφή αφηρημένου εργοστασίου, η οποία ορίζει ένα σύνολο μεθόδων για παραγωγή προϊόντων. Επιπλέον, συμπαγείς κλάσεις εργοστασίων, κάθε μια εκ των οποίων πρέπει να υλοποιεί την προαναφερθείσα διεπαφή, για τη παραγωγή συμπαγών προϊόντων. Ο κώδικας πελάτη θα πρέπει να εργάζεται με τα εργοστάσια και τα προϊόντα, μόνο μέσω των διεπαφών τους.



Σχημα 2.2: Διάγραμμα Κλάσης Abstract Factory [2]

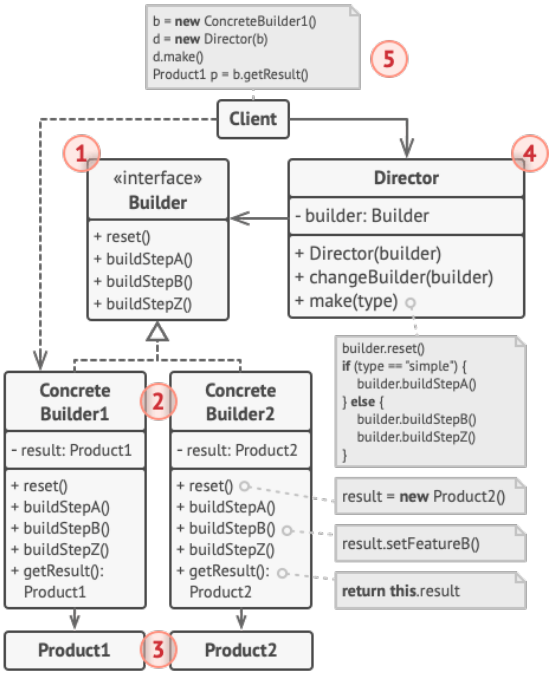
Με τη χρήση του Abstract Factory επιτυγχάνεται υψηλή ευελιξία στην εργασία με οικογένειες σχετιζόμενων μεταξύ τους αντικειμένων. Επίσης, αποτρέπεται η στενή εξάρτηση μεταξύ συμπαγών προϊόντων και του κώδικα πελάτη. Ακόμα, όπως και με το Factory Method, το μοτίβο αυτό επιτρέπει την απομόνωση της διαδικασίας δημιουργίας προϊόντων σε ένα μέρος του προγράμματος, καθιστώντας τον κώδικα πιο εύκολο στη συντήρηση και διαθέτει δομή που του επιτρέπει την εύκολη προσθήκη νέων τύπων προϊόντων, χωρίς να ελλοχεύει κίνδυνος πρόκλησης προβλημάτων στον κώδικα πελάτη. Ωστόσο, η χρήση του μπορεί να οδηγήσει σε αυξημένη πολυπλοκότητα και εισαγωγή πολλαπλών νέων κλάσεων και διεπαφών.

2.1.3 Builder

Το Builder αποτελεί ένα κατασκευαστικό μοτίβο σχεδίασης που επιτρέπει την βήμα προς βήμα κατασκευή σύνθετων αντικειμένων. Διαχωρίζει την κατασκευή ενός σύνθετου αντικειμένου από την αναπαράστασή του, επιτρέποντας έτσι στην ίδια διαδικασία κατασκευής να δημιουργεί διαφορετικές αναπαραστάσεις.

Φαίνεται ιδιαιτέρως χρήσιμο σε περιπτώσεις που ένα αντικείμενο πρέπει να δημιουργηθεί - συνθεθεί από πολλά διαφορετικά μέρη ή αντικείμενα, και είτε η διαμόρφωσή του μπορεί να παρουσιάσει πολλές παραλλαγές, είτε η διαδικασία κατασκευής του είναι περίπλοκή.

Η υλοποίηση του Builder περιλαμβάνει μια διεπαφή Builder η οποία ορίζει μεθόδους για την κατασκευή των διαφορετικών μερών του προϊόντος, και συμπαγείς κλάσεις που υλοποιούν την προαναφερθείσα διεπαφή και παρέχουν συγκεκριμένες υλοποιήσεις για αυτά τα μέρη. Ακόμα, προαιρετικά μπορεί να περιλαμβάνει μια κλάση Director που ελέγχει τη διαδικασία κατασκευής, κάνοντας χρήση ενός στιγμιοτύπου Builder. Όταν δεν υπάρχει κλάση Director, ο κώδικας πελάτη εργάζεται με τη διεπαφή του μοτίβου, ενώ όταν υπάρχει, δημιουργεί στιγμιότυπό της, στο οποίο περνάει ένα αντικείμενο Builder.



Σχημα 2.3: Διάγραμμα Κλάσης Builder [2]

Στα πλεονεκτήματα του Builder συγκαταλέγεται η παροχή σαφούς διαχωρισμού και λεπτομερούς ελέγχου της διαδικασίας κατασκευής. Επίσης, η ευελιξία στη δυνατότητα δημιουργίας διαφορετικών αναπαραστάσεων ενός προϊόντος με επαναχρησιμοποίηση του ίδιου κώδικα κατασκευής. Ως μειονέκτημα, όπως και με τα προηγούμενα μοτίβα, χαρακτηρίζεται η αύξηση της πολυπλοκότητας του κώδικα, καθώς το μοτίβο απαιτεί τη δημιουργία πολλαπλών νέων κλάσεων.

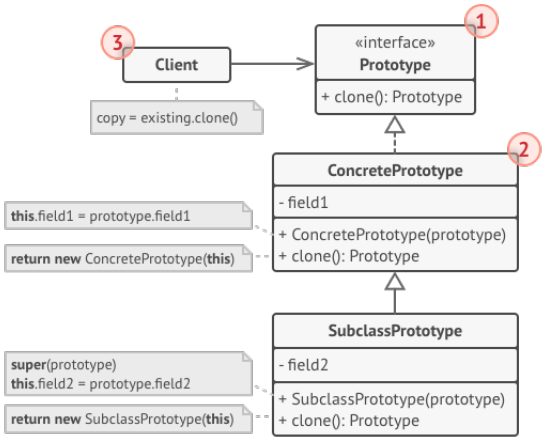
2.1.4 Prototype ή Clone

Το Prototype είναι ένα κατασκευαστικό μοτίβο σχεδίασης που επικεντρώνεται στην κλωνοποίηση αντικειμένων. Επιτρέπει δηλαδή σε ένα αντικείμενο χαρακτηριζόμενο ως πρωτότυπο,

να δημιουργεί αντίγραφα του εαυτού του, διαφυλάττοντας με αυτό τον τρόπο τον υπόλοιπο κώδικα από εξαρτήσεις προς την συμπαγή κλάση του.

Φαίνεται ιδιαιτέρως χρήσιμο σε περιπτώσεις που η δημιουργία ενός νέου στιγμιοτύπου σαν αντίγραφο κάποιου άλλου, εξυπηρετεί καλύτερα από τη δημιουργία ενός από το μηδέν. Επίσης, χρησιμοποιείται όταν οι κλάσεις προς αρχικοποίηση καθορίζονται κατά τον χρόνο εκτέλεσης, και σε περιπτώσεις όπου είναι επιθυμητή η αποφυγή δημιουργίας ιεραρχιών εργοστασίου και η μείωση του αριθμού υποκλάσεων που διαφέρουν μόνο στον τρόπο με τον οποίο αρχικοποιούν τα αντικείμενά τους.

Η υλοποίηση του Prototype περιλαμβάνει μια διεπαφή πρωτοτύπου, η οποία ορίζει μια μέθοδο Clone για την κλωνοποίηση του εκάστοτε αντικειμένου, και συμπαγείς κλάσεις που υλοποιούν την προαναφερθείσα διεπαφή και τη μέθοδο κλωνοποίησης. Η μέθοδος αυτή, πρέπει να επιστρέφει ένα αντίγραφο του αντικειμένου, εξασφαλίζοντας οτι το νέο δημιουργηθέν αντικείμενο έχεις τις ίδιες ιδιότητες και τιμές με το αρχικό.



Σχημα 2.4: Διάγραμμα Κλάσης Prototype [2]

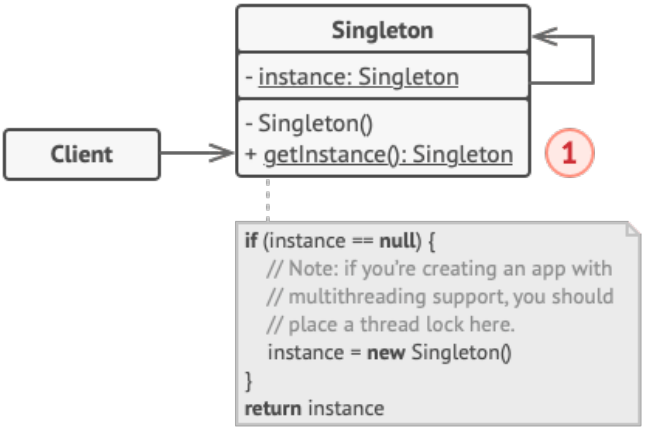
Με τη χρήση του Prototype, επιτυγχάνεται η αποφυγή δαπανηρού κώδικα αρχικοποίησης, μέσω της επαναχρησιμοποίησης υφιστάμενων αντικειμένων. Επίσης, το μοτίβο αυτό προσφέρει μια εναλλακτική λύση, έναντι της χρήσης υποκλάσεων για υλοποίηση προσαρμοσμένων διαφορφώσεων για σύνθετα αντικείμενα και κάνει γενικότερα τη δημιουργία αυτών, πιο εύκολη. Αρνητικό αυτής της προσέγγισης, είναι πως μπορεί να καταστεί πολύπλοκη σε περιπτώσεις που αντικείμενα έχουν περίπλοκες αναφορές σε άλλα αντικείμενα και σε περιπτώσεις ύπαρξης εμφωλευμένων αντικειμένων.

2.1.5 Singleton

Το Singleton είναι ένα κατασκευαστικό μοτίβο σχεδίασης που εξασφαίζει οτι μια κλάση έχει μόνο ένα στιγμιότυπο και για το οποίο παρέχει πρόσβαση σε όλους.

Χρησιμοποιείται σε περιπτώσεις που απαιτείται μια κλάση ενός προγράμματος να έχει ακριβώς ένα στιγμιότυπο διαθέσιμο για όλους τους πελάτες και επιλέγεται για τη διαχείριση πρόσβασης σε πόρους που είναι κοινοί σε πολλαπλά συστήματα, όπως αρχεία ρυθμίσεων.

Η υλοποίηση του Singleton περιλαμβάνει συνήθως μια μοναδική κλάση, η οποία είναι η ίδια υπεύθυνη για τη δημιουργία και τον κύκλο ζωής της. Κύριο μέλημα είναι ο ορισμός του κατασκευστή της κλάσης ως ιδιωτικού, για την αποφυγή χρήσης του τελεστή new επί της κλάσης από κώδικα τρίτου. Η κλάση διασφαλίζει οτι υπάρχει μόνο ένα στιγμιότυπό της, το οποίο μέσω μιας μεθόδου της, είτε επιστρέφει, είτε όταν δεν υπάρχει, πρώτα το δημιουργεί και έπειτα το επιστρέφει.



Σχημα 2.5: Διάγραμμα Κλάσης Singleton [2]

Με τη χρήση του Singleton διασφαλίζεται οτι μια κλάση έχει ένα μοναδικό στιγμιότυπο και οτι παρέχεται καθολική πρόσβαση σε αυτό. Επίσης, υποστηρίζεται lazy-loading, καθώς το singleton αντικείμενο αρχικοποιείται όταν ζητηθεί για πρώτη φορά. Ωστόσο, η χρήση του μπορεί να επιφέρει και προβλήματα. Η υλοποίηση του μοτίβου μπορεί να προκαλέσει μια καθολικότητα στην εφαρμογή, κατά την οποία τα τμήματα του προγράμματος γνωρίζουν υπερβολικά πολλά το ένα για το άλλο. Επίσης χρήζει ειδικής αντιμετώπισης σε πολυνηματικά περιβάλλοντα και μπορεί να δυσκολέψει τη διαδικασία του unit testing, καθώς είναι δύσκολη η αντικατάσταση - ομοίωσή του.

2.2 Structural Patterns

Τα δομικά μοτίβα σχεδίασης ασχολούνται με τον τρόπο με τον οποίο κλάσεις και αντικείμενα συντίθενται για να σχηματίζουν μεγαλύτερες δομές. Αυτά τα μοτίβα επικεντρώνονται στη διαχείριση των σχέσεων μεταξύ οντοτήτων, με κατάλληλο τρόπο που προωθεί την απλότητα, την ευελιξία και την αποτελεσματικότητα. Παράλληλα, διασφαλίζουν οτι αλλαγές που γίνονται σε ένα μέρος του συστήματος, δεν απαιτούν την τροποποίηση όλου του υπόλοιπου συστήματος.

Τα κύρια δομικά μοτίβα είναι τα ακόλουθα και θα αναλυθούν παρακάτω:

- Adapter
- Bridge
- Composite
- Decorator
- Facade
- Flyweight
- Proxy

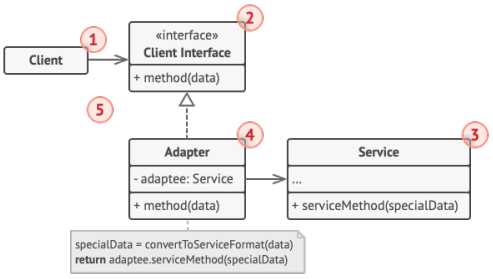
2.2.1 Adapter

Το Adapter αποτελεί ένα δομικό μοτίβο σχεδίασης που επιτρέπει σε αντικείμενα κλάσεων με ασύμβατες διεπαφές, να συνεργάζονται. Λειτουργεί ως ενδιάμεσος μεταξύ των διεπαφών αυτών και τους επιτρέπει να επικοινωνούν.

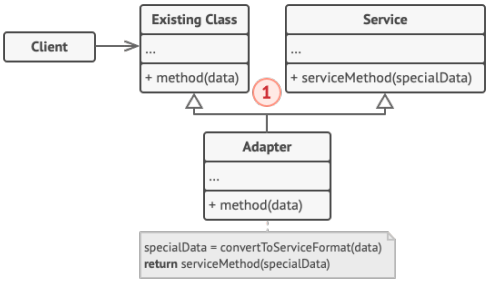
Χρησιμοποιείται σε περιπτώσεις που είναι επιθυμητή η εργασίς με υφιστάμενες κλάσεις, των οποίων οι διεπαφές δεν είναι συμβατές με τον υπόλοιπο κώδικα και η τροποποίηση των οποίων είναι αδύνατη ή δεν καθίσταται πρακτική. Η χρήση του είναι συνήθης σε συστήματα στα οποία πρέπει να ενσωματωθούν νέα στοιχεία, χωρίς να τροποποιηθεί ο υφιστάμενος κώδικας. Τέτοια συστήματα συχνά είναι είτε παλιά συστήματα, είτε βιβλιοθήκες τρίτων.

Η υλοποίηση του Adapter έχει δύο παραλλαγές. Η πρώτη ονομάζεται Object adapter και χρησιμοποιεί σύνθεση. Κατ΄ αυτήν, ο αντάπτορας υλοποιεί τη διεπαφή του ενός αντικειμένου

και ενθυλακώνει μέσα του το άλλο. Η δεύτερη ονομάζεται Class adapter και χρησιμοποιεί κληρονομικότητα. Σε αυτή την περίπτωση, ο αντάπτορας κληρονομεί και από τις δύο διεπαφές. Για την υλοποίηση αυτής, απαιτείται η χρησιμοποιούμενη γλώσσα να υποστηρίζει πολλαπλή κληρονομικότητα.



Σχημα 2.6: Διάγραμμα Κλάσης Object adapter [2]



Σχημα 2.7: Διάγραμμα Κλάσης Class adapter [2]

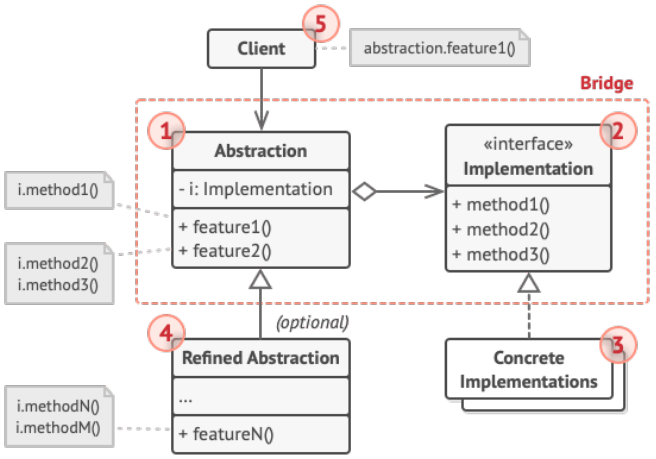
Στα πλεονεκτήματα του Adapter, συγκαταλέγεται η προσθήκη δυνατότητας συνεργασίας κλάσεων με ασύμβατες διεπαφές, χωρίς να απαιτείται η τροποποίησή τους. Ωστόσο, αναδεικνύονται ανησυχίες σχετικά με την αύξηση της πολυπλοκότητας του κώδικα που προκύπτει από την εισαγωγή πρόσθετων επιπέδων και αφαιρέσεων με τη μορφή κλάσεων και διεπαφών.

2.2.2 Bridge

Το Bridge είναι ένα δομικό μοτίβο σχεδίασης που επιτρέπει τον διαχωρισμό μεγάλων κλάσεων και συνόλων κλάσεων σε δύο ιεραρχίες: την αφαίρεση και την υλοποίηση. Αυτές οι ιεραρχίες μπορούν να αναπτύσσονται ανεξάρτητα μεταξύ τους.

Το μοτίβο χρησιμοποιείται σε περιπτώσεις που κρίνεται αναγκαία η αποφυγή μόνιμης σύζευξης μεταξύ μιας αφαίρεσης και της υλοποίησής της. Αποτελεί ιδανική λύση, όταν απαιτείται η υποστήριξη πολλών υλοποιήσεων μιας κοινής αφαίρεσης, για παράδειγμα, στην εργασία με πολλούς εξυπηρετητές βάσεων δεδομένων. Επιλέγεται επίσης, αντί της ανάπτυξης υποκλάσεων.

Η υλοποίηση του Bridge περιλαμβάνει μία κλάση για την αφαίρεση και μία διεπαφή για τις υλοποιήσεις, μαζί με τις συμπαγείς κλάσεις που την υλοποιούν. Η αφαίρεση περιλαμβάνει αναφορά προς τη διεπαφή των υλοποιήσεων. Υπάρχει δηλαδή μεταξύ τους, σχέση σύνθεσης. Ο κώδικας πελάτη αλληλεπιδρά με την αφαίρεση και δε γνωρίζει λεπτομέρειες για τις υλοποιήσεις. Προαιρετικά, μπορούν να υπάρχουν και κλάσεις που επεκτείνουν την αφαίρεση, με σκοπό την παροχή παραλλαγών στη λογική ελέγχου.



Σχημα 2.8: Διάγραμμα Κλάσης Bridge [2]

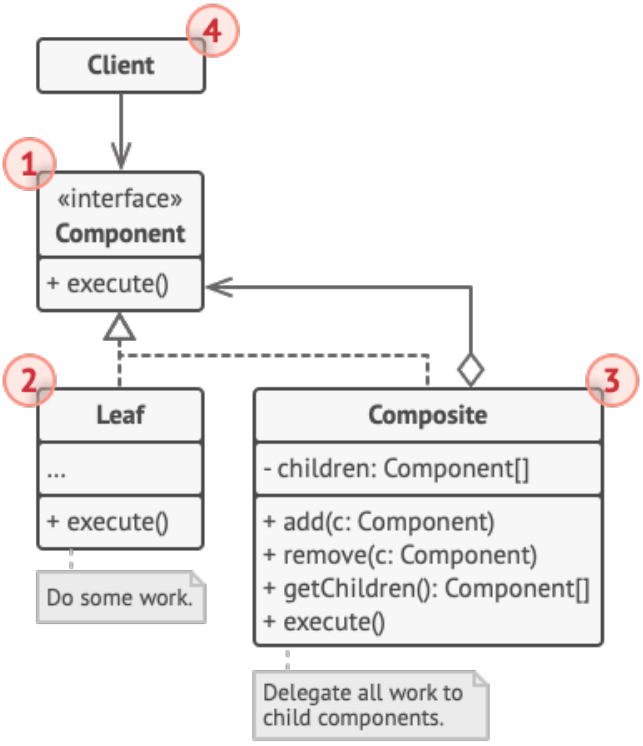
Με τη χρήση του Bridge μοτίβου, επιτυγχάνεται αυξημένη ευελιξία, καθώς επιτρέπονται τροποποιήσεις στην αφαίρεση και στις επιμέρους υλοποιήσεις, με ανεξάρτητο τρόπο. Επίσης, το μοτίβο δεν προάγει την αύξηση των ιεραρχιών κλάσεων και αναγκάζει τον κώδικα πελάτη να αλληλεπιδρά με το επίπεδο αφαίρεσης, χωρίς να του εκθέτει λεπτομέρειες που αφορούν την υλοποίηση. Μειονεκτήματα του μοτίβου είναι η εισαγωγή πολυπλοκότητας, όπως και με τα προηγούμενα μοτίβα, και η πιθανότητα έντονης περίπλεξης του κώδικα, όταν αυτό εφαρμόζεται σε υψηλά συνεκτικές κλάσεις.

2.2.3 Composite

Το Composite αποτελεί ένα δομικό μοτίβο σχεδίασης που επιτρέπει τη σύνθεση αντικειμένων σε δενδρικές δομές, με σκοπό την αναπαράσταση ιεραρχιών μέρους-όλου. Δίνει τη δυνατότητα στον κώδικα πελάτη να αντιμετωπίζει αντικείμενα και συνθέσεις αντικειμένων με κοινό τρόπο.

Χρησιμοποιείται όταν είναι επιθυμητή η ομοιόμορφη μεταχείριση απλών και σύνθετων αντικειμένων από τον κώδικα πελάτ, και καθίσταται ιδανικό για υλοποίηση δενδρικών δομών.

Η υλοποίησή του Composite απαιτεί τον ορισμό μιας διεπαφής ή αφηρημένης κλάσης, η οποία θα ορίζει τις κοινές λειτουργίες τόσο για τα απλά όσο και για τα σύνθετα αντικείμενα. Επίσης, δύο τύπους κλάσεων που θα αναπαριστούν αυτές τις δύο κατηγορίες: φύλλο και σύνθετο. Το φύλλο αντιπροσωπεύει απλά αντικείμενα που επιτελούν εργασίες και δεν έχουν παιδιά, ενώ το σύνθετο περιλαμβάνει φύλλα και άλλα σύνθετα, και αναθέτει τις εργασίες του σε αυτά.



Σχημα 2.9: Διάγραμμα Κλάσης Composite [2]

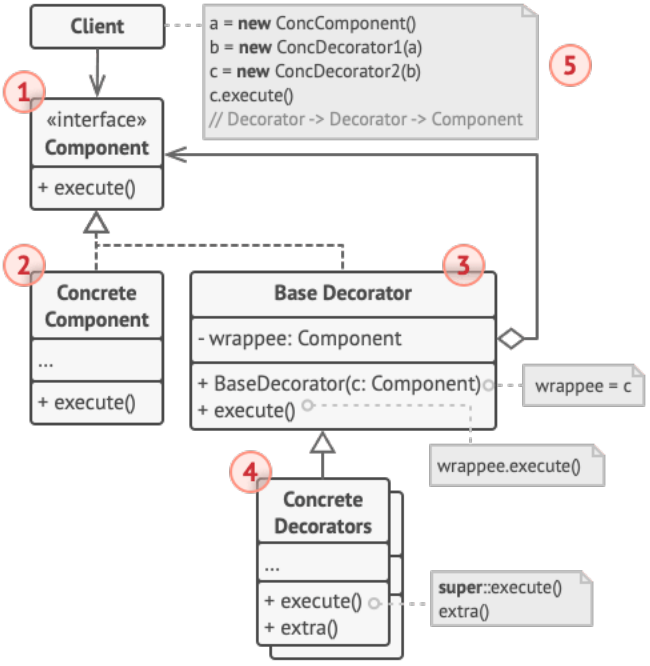
Στα πλεονεκτήματα του Composite συγκαταλέγεται η διευκόλυνση της εργασίας με δενδρικές δομές, με τη χρήση του πολυμορφισμού και της αναδρομής. Επίσης, η ευκολία εισαγωγής νέων στοιχείων που συμμορφώνονται με την κοινή διεπαφή. Ωστόσο, σε ορισμένες περιπτώσεις, είναι δύσκολη η παροχή κοινής διεπαφής για κλάσεις των οποίων η λειτουργικότητα διαφέρει σε μεγάλο βαθμό. Επομένως, ελλοχεύει κίνδυνος υπεργενίκευσης της σχεδίασης. Γενικότερα, η κακή χρήση του μοτίβου μπορεί να οδηγήσει σε περίπλοκες δομές, των οποίων η διαχείριση καθίσταται δυσχερής.

2.2.4 Decorator

Το Decorator είναι ένα δομικό μοτίβο σχεδίασης που επιτρέπει την προσθήκη νέα συμπεριφοράς σε μεμονωμένα αντικείμενα, χωρίς να επηρεάζεται η συμπεριφορά άλλων αντικειμένων της ίδιας κλάσης. Παρέχει επομένως, μια ευέλικτη εναλλακτική λύση, έναντι της χρήσης υποκλάσεων για επέκταση της λειτουργικότητας. Η προσθήκη της συμπεριφοράς επιτυγχάνεται με την περιτύλιξη των αντικειμένων αυτών, από άλλα ειδικά αντικείμενα που ενθυλακώνουν αυτές τις συμπεριφορές.

Το μοτίβο αυτό, χρησιμοποιείται όταν είναι επιθυμητή η δυνατότητα δυναμικής προσθήκης επιπλέον συμπεριφορών σε αντικείμενα, χωρίς να επηρεάζεται ο υπόλοιπος κώδικας που χρησιμοποιεί αυτά τα αντικείμενα, ή τα υπόλοιπα αντικείμενα της ίδιας κλάσης. Είναι ιδανικό σε περιπτώσεις που η επέκταση της λειτουργικότητας κάποιας κλάσης με υποκλάσεις της, καθίσταται ανέφικτη ή ασύμφορη. Επίσης, χρησιμεύει στην κατασκευή αντικειμένων με σύνθετες συμπεριφορές, που προκύπτουν από τη σύνθεση πολλών απλών.

Για την υλοποίησή του Decorator, απαιτείται ορισμός μιας διεπαφής την οποία θα πρέπει να υλοποιούν τόσο τα αντικείμενα στα οποία πρόκειται να προστεθεί λειτουργικότητα, όσο και οι διακοσμητές. Επιπλέον, τουλάχιστον μια κλάση της οποίας τα στιγμιότυπα θα αντιπροσωπεύουν τα εν λόγω αντικείμενα. Τέλος, μια κλάση διακοσμητή που θα περιλαμβάνει ένα αντικείμενο που υλοποιεί την προαναφερθείσα διεπαφή, και προσαρμοσμένους διακοσμητές που θα κληρονομούν από την εν λόγω κλάση.



Σχημα 2.10: Διάγραμμα Κλάσης Decorator [2]

Όσον αφορά τα πλεονεκτήματα χρήσης του, το Decorator προσφέρει μεγαλύτερη ευελιξία στην επέκταση της λειτουργικότητας των αντικειμένων, συγκριτικά με τη στατική κληρονομικότητα. Επιτρέπει τη δυναμική προσθήκη των πρόσθετων συμπεριφορών σε αντικείμενα, καθώς και το συνδυασμό αυτών. Αρνητικό του, είναι πως όταν έχει προκύψει ένας μεγάλος συνδυασμός - στίβα διακοσμητών σε ένα αντικείμενο, καθίσταται δύσκολη τόσο η αφαίρεση

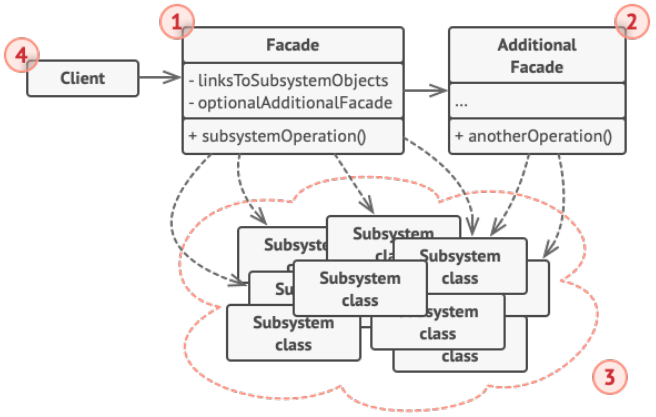
κάποιου ενδιάμεσου διακοσμητή, όσο και η κατανόηση του τελικού περιτυλιγμένου αντικειμένου. Επομένως, η εκτεταμένη χρήση του μπορεί να οδηγήσει σε δυσνόητο και πολύπλοκο κώδικα.

2.2.5 Facade

Το Facade είναι ένα δομικό μοτίβο σχεδίασης που παρέχει μια απλοποιημένη διεπαφή για ένα σύνθετο σύστημα κλάσεων, μια βιβλιοθήκη ή ένα framework. Το μοτίβο αυτό, εισάγει μια κλάση η οποία δρά ως μια απλοποιημένη και ενοποιημένη διεπαφή για πιο πολύπλοκα υποσυστήματα, καθιστώντας έτσι το σύνολο τους πιο εύχρηστο και κατανοητό.

Χρησιμοποιείται όταν κρίνεται επιθυμητή η παροχή μιας απλής και περιορισμένης, αλλά πιο άμεσης προσέγγισης διεπαφής, σε ένα πολύπλοκο υποσύστημα. Επίσης, χρησιμεύει στη δόμηση συστημάτων σε επίπεδα. Μια κλάση Facade μπορεί να λειτουργήσει ως σημείο εισόδου σε ένα επίπεδο του συστήματος, συντελώντας έτσι στη χαλαρή σύζευξη και στη μείωση των εξαρτήσεων.

Για την υλοποίησή του Facade, απαιτείται τουλάχιστον μια κλάση, με αναφορές προς κλάσεις του υποσυστήματος, η οποία θα γνωρίζει πως να κατευθύνει τα αιτήματα του κώδικα πελάτη στα κατάλληλα αντικείμενα του υποσυστήματος και να διαχειριστεί τις λειτουργίες των υποσυστημάτων που την αφορούν. Μπορούν να υπάρχουν πολλαπλές τέτοιες κλάσεις, ώστε να μη μολύνεται μια μοναδική κλάση με άσχετες μεταξύ τους λειτουργίες, που θα την μετέτρεπαν σε άλλη μια περίπλοκη δομη. Σε ορισμένες περιπτώσεις, το Facade μπορεί να γίνει το μοναδικό σημείο επικοινωνίας με ένα υποσύστημα και ο κώδικας πελάτη να αλληλεπιδρά μόνο με αυτό για να αποκτήσει πρόσβαση στο εν λόγω υποσύστημα.



Σχημα 2.11: Διάγραμμα Κλάσης Facade [2]

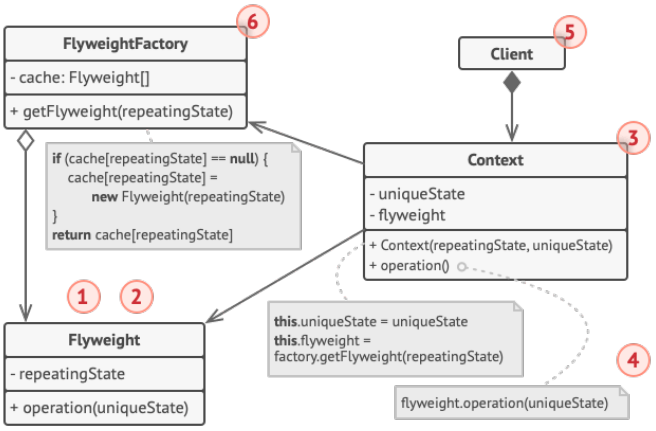
Με τη χρήση του Facade μοτίβου, επιτυγχάνεται η εισαγωγή μιας υψηλού επιπέδου διεπαφής πάνω από πολύπλοκα συστήματα, η οποία τα καθιστά πιο κατανοητά και εύχρηστα. Ωστόσο, μια απλουστευμένη διεπαφή μπορεί να προκαλέσει περιορισμούς σε έμπειρους χρήστες που πρέπει να χρησιμοποιήσουν απευθείας ένα σύνθετο σύστημα. Ακόμα, ελλοχεύει ο κίνδυνος μετατροπής της κλάσης Facade σε μια σύνθετη δομή που είναι συνδεδεμένη με όλες τις κλάσεις της εφαρμογής.

2.2.6 Flyweight ή Cache

Το Flyweight είναι ένα δομικό μοτίβο σχεδίασης που ελαχιστοποιεί τη χρήση της μνήμης, μέσω του διαμοιρασμού όσο το δυνατόν περισσότερων κοινών δεδομένων, μεταξύ παρόμοιων αντικειμένων. Κρατάει δηλαδή την κοινή πληροφορία σε ένα σημείο, αντι να επιτρέπει σε κάθε αντικείμενο να την επαναλαμβάνει.

Χρησιμοποιείται όταν ένα πρόγραμμα πρέπει να υποστηρίξει παράλληλα έναν μεγάλο αριθμό όμοιων αντικειμένων που περιλαμβάνουν κοινή πληροφορία, και αυτό προκαλεί προβλήματα στη μνήμη.

Το μοτίβο περιλαμβάνει τρία κύρια στοιχεία: το Flyweight, το Context και το εργοστάσιο Flyweight. Η πρώτη κλάση περιλαμβάνει το τμήμα της κατάστασης του αρχικού αντικειμένου που μπορεί να διαμοιραστεί και σε άλλα αντικείμενα. Η αποθηκευμένη αυτή κατάσταση χαρακτηρίζεται ως ενδογενής. Η δεύτερη κλάση περιλαμβάνει την εξωγενή κατάσταση, εκείνη δηλαδή που είναι διαφορετική για κάθε αντικείμενο. Αυτή, όταν προστίθεται μέσα της ένα αντικείμενο Flyweight, αναπαριστά την πλήρη κατάσταση του αρχικού αντικειμένου. Τέλος, το εργοστάσιο που διαχειρίζεται μια πληθώρα υφιστάμενων flyweights, είτε τροφοδοτεί το Context με αυτό που καλύπτει τα εκάστοτε κριτήρια, είτε το δημιουργεί και το επιστρέφει αν αυτό δεν υπάρχει.



Σχημα 2.12: Διάγραμμα Κλάσης Flyweight [2]

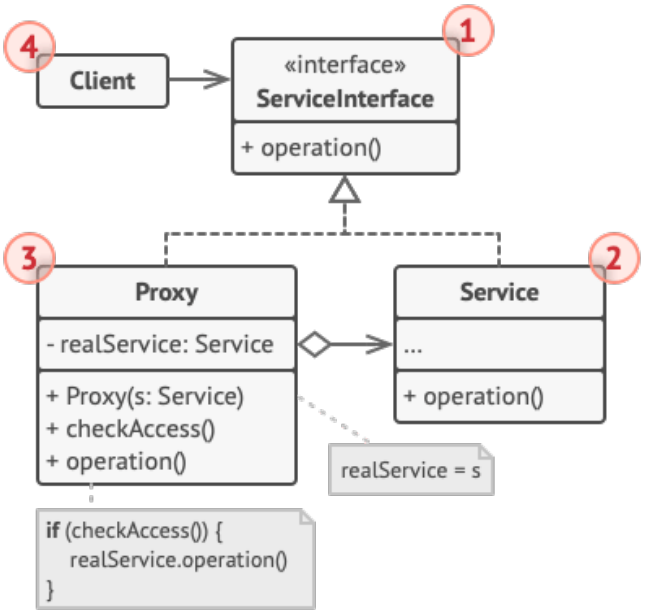
Κύριο πλεονέκτημα της χρήσης του Flyweight μοτίβου είναι η ελαχιστοποίηση της κατανάλωσης μνήμης. Αυτό μπορεί να βελτιστοποιήσει σημαντικά την απόδοση συστημάτων μεγάλης κλίμακας. Το αρνητικό που επιφέρει, είναι πως αυτός ο διαχωρισμός των καταστάσεων κάθε αντικειμένου, αυξάνει την πολυπλοκότητα του κώδικα και πως η σύνθεση της πληροφορίας, όποτε αυτό απαιτείται, κοστίζει σε κύκλους της επεξεργαστικής μονάδας.

2.2.7 Proxy

Το Proxy είναι ένα δομικό μοτίβο σχεδίασης που παρέχει ένα υποκατάστατο στη θέση ενός άλλου αντικειμένου, για την άσκηση ελέγχου πρόσβασης στο τελευταίο.

Το μοτίβο αυτό χρησιμοποιείται όταν είναι επιθυμητή η άσκηση ελέγχου πρόσβασης σε ένα αντικείμενο, είτε για λόγους ασφαλείας, είτε για διαχείριση του κύκλου ζωής του αντικειμένου, είτε για προσθήκη πρόσθετων λειτουργιών, όπως lazy loading και καταγραφή συμβάντων. Ακόμα, χρησιμεύει στην περίπτωση που ένα αντικείμενο βρίσκεται σε απομακρυσμένη τοποθεσία και ο Proxy αναλαμβάνει τις λεπτομέρειες που αφορούν τη δικτυακή επικοινωνία.

Όσον αφορά την υλοποίησή του, απαιτείται η εισαγωγή μιας κλάσης Proxy, η οποία θα πρέπει να υλοποιεί την ίδια διεπαφή με το αντικείμενο που αντιπροσωπεύει, και να έχει αναφορά προς αυτό. Η κλάση αυτή, αφού επιτελέσει τις αντίστοιχες λειτουργίες που πηγάζουν από τον εκάστοτε ρόλο της (από αυτους που αναφέρθηκαν στην ανωτέρω παράγραφο) αναθέτει όλες τις εργασίες στο εν λόγω αντικείμενο.



Σχημα 2.13: Διάγραμμα Κλάσης Proxy [2]

Με τη χρήση του Proxy μοτίβου, επιτυγχάνεται η άσκηση ελεγχόμενης πρόσβασης επί αντικειμένων, η διαχείριση του κύκλού ζωής τους, και η προσθήκη πρόσθετης λειτουργικότητας σε αυτά. Επίσης μπορούν να εισάγονται νέοι τύποι Proxy, χωρίς να τροποποείται το αντικείμενο στόχος ή ο κώδικας πελάτη. Τα μεινεκτήματά του, πηγάζουν από την προσθήκη του πρόσθετου επιπέδου αφαίρεσης. Αυτό μπορεί να οδηγήσει σε αυξημένο χρόνο απόκρισης και σε αύξηση της πολυπλοκότητας, καθώς είσαγονται νέες κλάσεις οι οποίες πρέπει επιπρόσθετα να παραμένουν συνεπείς με τη διεπαφή του αντικειμένου.

2.3 Behavioral Patterns

Τα συμπεριφορικά μοτίβα σχεδίασης ασχολούνται με αλγορίθμους, και με την αποτελεσματική επικοινωνία και ανάθεση αρμοδιοτήτων μεταξύ αντικειμένων. Βοηθούν στον καθορισμό του τρόπου αλληλεπίδρασης και επικοινωνίας των αντικειμένων, και του τρόπου διαχείρισης της ροής ελέγχου.

Τα κύρια συμπεριφορικά μοτίβα είναι τα ακόλουθα και θα αναλυθούν παρακάτω:

- Chain of Responsibility
- Command

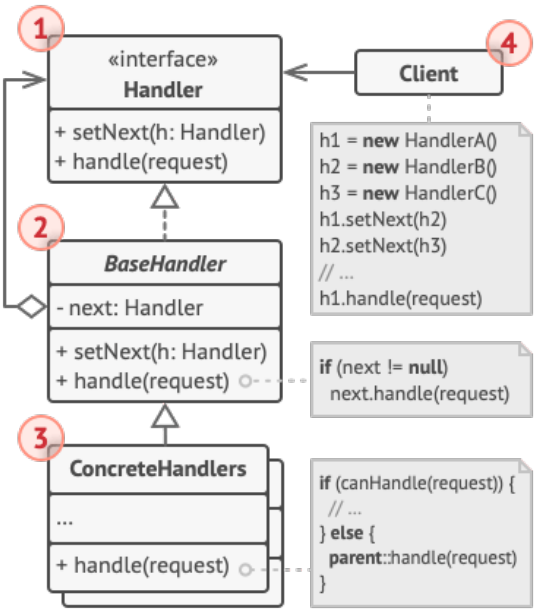
- Iterator
- Mediator
- Memento
- Observer
- State
- Strategy
- Template Method
- Visitor

2.3.1 Chain of Responsibility

Το Chain of Responsibility είναι ένα συμπεριφορικό μοτίβο σχεδίασης που επιτρέπει τη μεταβίβαση ενός αιτήματος, κατά μήκος μίας λίστας χειριστών. Κατά τη λήψη ενός αιτήματος, κάθε χειριστής αποφασίζει είτε να επεξεργαστεί το αίτημα, είτε να το μεταβιβάσει στον επόμενο χειριστή της αλυσίδας.

Χρησιμοποιείται όταν ο αποστολέας ενός αιτήματος και οι παραλήπτες, θα πρέπει να λειτουργούν ανεξάρτητα. Το μοτίβο φαίνεται ιδιαίτερα χρήσιμο σε περιπτώσεις που απαιτείται χειρισμός αιτημάτων από ένα σύνολο χειριστών, αλλά δεν είναι εκ των προτέρων γνωστά ούτε ο ακριβής τύπος του αιτήματος, ούτε ο ακριβής χειριστής. Εφαρμόζεται επίσης, όταν το σύνολο των χειριστών και η σειρά τους πρέπει να μπορούν να καθορίζονται δυναμικά.

Η υλοποίηση του μοτίβου απαιτεί την εισαγωγή μιας διεπαφής Handler που θα ορίζει μια μέθοδο για το χειρισμό αιτημάτων και μια για τον ορισμό του επόμενου χειριστή της αλυσίδας. Επίσης, τις συμπαγείς υλοποιήσεις των χειριστών που θα υλοποιούν αυτή τη διεπαφή. Προαιρετικά, αναμέσα από τη διεπαφή και τις κλάσεις αυτές, μπορεί να προστεθεί μια κλάση που θα περιλαμβάνει την κοινή συμπεριφορά των χειριστών, για αποφυγή επανάληψης κώδικα.



Σχημα 2.14: Διάγραμμα Κλάσης Chain of Responsibility [2]

Το Chain of Responsibility μοτίβο έχει το θετικό ότι διαχωρίζει τις κλάσεις που αποστέλλουν αιτήματα, από αυτές που τα υλοποιούν. Επίσης, παρέχει ευελιξία στην ανάθεση αρμοδιοτήτων, καθώς επιτρέπει την δυναμική προσθαφαίρεση χειριστών, χωρίς να επηρεάζεται ο κώδικας πελάτη. Ωστόσο, ένα άιτημα μπορεί να περάσει από πολλούς χειριστές πριν υποβληθεί σε επεξεργασία, πράγμα που προκαλεί καθυστέρηση, ενώ παράλληλα δεν παρέχεται καμία εγγύηση οτι το αίτημα θα διεκπεραιωθεί αν δεν υπάρχει κατάλληλος χειριστής στην αλυσίδα.

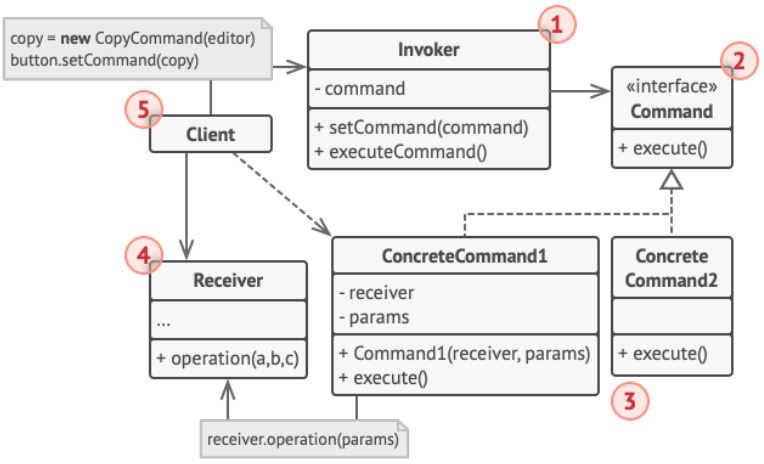
2.3.2 Command

Το Command είναι ένα συμπεριφορικό μοτίβο σχεδίασης που μετατρέπει αιτήματα σε αυτόνομα αντικείμενα που περιλαμβάνουν όλες τις απαραίτητες πληροφορίες που αφορούν το κάθε αίτημα. Αυτός ο μετασχηματισμός καθιστά ικανή την τροφοδότηση μεθόδων με διαφορετικά αιτήματα, και την υποστήριξη λειτουργιών όπως η ηθελημένη καθυστέρηση της εκτέλεσης ενός αιτήματος, η τοποθέτησή του σε ουρά εκτέλεσης ή η ανάκλησή του.

Το μοτίβο επιλέγεται όταν είναι επιθυμητή η παραμετροποίηση αντικειμένων, ώστε να περιλαμβάνουν λεπτομέρειες που αφορούν λειτουργίες. Επίσης, αποτελεί ιδανική λύση σε σενάρια όπου λειτουργίες πρέπει να τοποθετηθούν σε ουρά, να χρονοπρογραμματιστούν για εκτέλεση, ή και να εκτελεστούν από απόσταση. Ακόμα, χρησιμεύει σε περιπτώσεις

που πρέπει να υποστηριχθούν λειτουργίες αναίρεσης, καθώς επιτρέπει σε τρίτες κλάσεις να διατηρούν ιστορικό των εκτελεσμένων λειτουργιών.

Για την κατασκευή του Command μοτίβου, απαιτείται ο ορισμός μιας διεπαφής Command η οποία συνήθως περιλαμβάνει μια μέθοδο εκτέλεσης της εντολής που αντιπροσωπεύει, καθώς και οι συμπαγείς υλοποιήσεις της. Ακόμα, η ύπαρξη μιας κλάσης που πυροδοτεί τις εντολές και που δεν έχει γνώση των προαναφερθέντων συμπαγών υλοποιήσεων, και μιας κλάσης που εκτελεί ουσιαστικά την πραγματική εργασία, όταν καλείται η μέθοδος εκτέλεσης της εντολής. Ο κώδικας πελάτη θα πρέπει να υλοποιεί συμπαγή αντικείμενα εντολών και να τους μεταβιβάζει όλες τις απαραίτητες πληροφορίες που αφορούν το αίτημα.



Σχημα 2.15: Διάγραμμα Κλάσης Command [2]

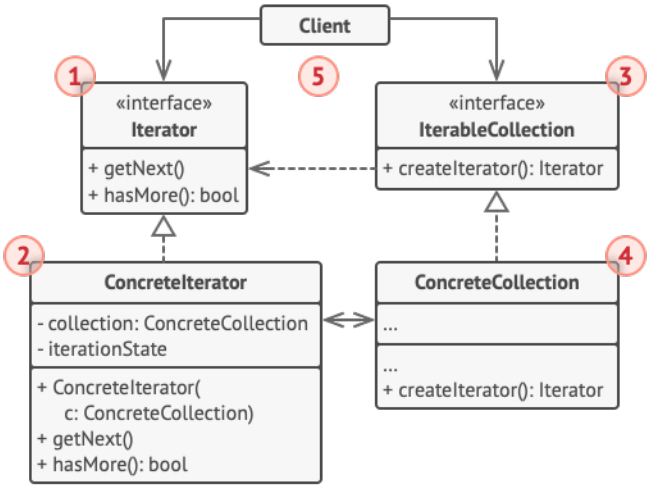
Ένα από τα πλεονεκτήματα του μοτίβου είναι ο διαχωρισμός των ανησυχιών. Επιτυγχάνει τον διαχωρισμό των κλάσεων που αποστέλλουν αιτήματα, από αυτές που τα υλοποιούν. Επίσης, επιτρέπει την εισαγωγή νέων εντολών, χωρίς να επηρεάζεται ο κώδικας πελάτη, και καθιστά δυνατό το συνδυασμό αυτών, για την εκτέλεση σύνθετων λειτουργιών. Εντούτοις, η προσέγγιση αυτή μπορεί να προκαλέσει αυξημένη πολυπλοκότητα, καθώς εισάγει ένα επιπλέον επίπεδο ανάμεσα σε αποστολείς και παραλήπτες αιτημάτων και κατά συνέπεια, νέες κλάσεις.

2.3.3 Iterator

Το Iterator είναι ένα συμπεριφορικό μοτίβο σχεδίασης που επιτρέπει την διάσχιση και προσπέλαση των στοιχείων μιας συλλογής, χωρίς να αποκαλύπτει λεπτομέρειες που αφορούν τη δομή της.

Το Iterator μοτίβο χρησιμοποιείται σε περιπτώσεις που απαιτείται η παροχή ενός ενιαίου τρόπου διάσχισης των στοιχείων κάποιας συλλογής, της οποίας η υποκείμενη δομή δεν είναι επιθυμητό να είναι γνωστή στον κώδικα πελάτη, είτε λόγω πολυπλοκότητας της δομής δεδομένων, είτε για λόγους ασφαλείας. Επίσης, όταν είναι επιθυμητή η υποστήριξη πολλαπλών μεθόδων διάσχισης σε συλλογές που θα υλοποιούν την ίδια διεπαφή. Ακόμα, χρησιμοποιείται για την απόζευξη των συλλογών από τους αλγορίθμους διάσχισης, με στόχο την καλύτερη οργάνωση και επαναχρησιμοποίηση του κώδικα.

Για την κατασκευή του μοτίβου απαιτείται μια διεπαφή Iterator η οποία θα ενσωματώνει λειτουργίες προσπέλασης του επόμενου στοιχείου και ελέγχου του αν η συλλογή στην οποία αναφέρεται περιέχει στοιχεία. Έπειτα, οι συμπαγείς κλάσεις που την υλοποιούν, και που υποστηρίζουν συγκεκριμένους αλγόριθμους διάσχισης και αναφέρονται σε συγκεκριμένες συλλογές. Τέλος, μια διεπαφή για τις συλλογές, η οποία θα ορίζει μια μέθοδο δημιουργίας Iterator συμβατών με την εκάστοτε συλλογή, και οι συμπαγείς συλλογές που θα την υλοποιούν.



Σχημα 2.16: Διάγραμμα Κλάσης Iterator [2]

Στα πλεονεκτήματα της χρήσης του μοτίβου συγκαταλέγονται η απλοποίηση του κώδικα πελάτη, η απόζευξη των αλγορίθμων διάσχισης από τις συλλογές, η δυνατότητα ομοιόμορφης χρήσης διαφορετικών αλγορίθμων διάσχισης σε συλλογές, και η ευκολία εισαγωγής νέων τύπων συλλογών και αλγορίθμων διάσχισής τους, χωρίς να επηρεάζεται ο υφιστάμενος κώδικας. Ακόμα, η δυνατότητα ταυτόχρονης διάσχισης μιας συλλογής από διαφορετικούς Iterator, καθώς ο καθένας θα έχει την δική του ιδιωτική κατάσταση. Μειονέκτημα της χρήσης του μπορεί να θεωρηθεί η αύξηση του χρόνου απόκρισης και της πολυπλοκότητας,

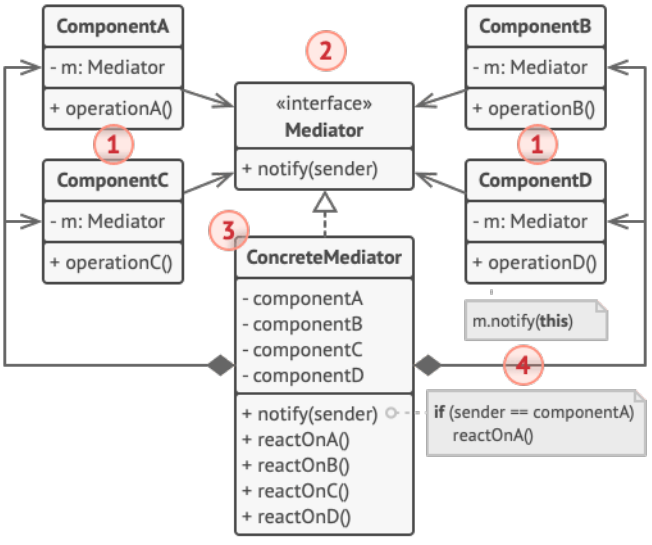
καθώς εισάγεται ένα επιπλέον επίπεδο αφαίρεσης πάνω από τις συλλογές, πράγμα αχρείαστο σε περιπτώσεις χρήσης απλών συλλογών.

2.3.4 Mediator

Το Mediator είναι ένα συμπεριφορικό μοτίβο σχεδίασης που μειώνει την άμεση επικοινωνία μεταξύ αντικειμένων, μέσω της εισαγωγής μια κεντρικής αρχής που διαχειρίζεται τον τρόπο αλληλεπίδρασης αυτών. Αυτή η προσέγγιση μειώνει τις εξαρτήσεις μεταξύ αλληλεπιδρόντων αντικειμένων, διευκολύνοντας έτσι τη διαχείριση και συντήρησή τους.

Το μοτίβο χρησιμοποιείται όταν η δημιουργία αλλαγών σε ένα σύστημα ή η επέκτασή του, καθίστανται δυσχερείς, λόγω πολύπλοκων αλληλεπιδράσεων και υψηλής σύζευξης μεταξύ των υφιστάμενων αντικειμένων. Επίσης, όταν εξαιτίας της υψηλής σύζευξης μεταξύ των τμημάτων του κώδικα, δεν είναι εφικτή η επαναχρησιμοποίηση επιμέρους τμημάτων. Ακόμα, χρησιμοποιείται ως εναλλακτική έναντι της εισαγωγής υποκλάσεων για υλοποίηση προσαρμοσμένων διαφορφώσεων αλληλεπίδρασης μεταξύ αντικειμένων.

Η υλοποίηση του Mediator μοτίβου απαιτεί τον ορισμό μιας διεπαφής Mediator που θα περιλαμβάνει μεθόδους για αποστολή και λήψη μηνυμάτων, με σκοπό την διαχείριση της επικοινωνίας μεταξύ μερών του κώδικα. Επίσης, τις συμπαγείς υλοποιήσεις αυτής που θα διαθέτουν αναφορές προς αυτά τα μέρη. Τέλος, τα τμήματα - κλάσεις του κώδικα που θα περιλαμβάνουν κάποια επιχειρησιακή λογική και θα γνωρίζουν μόνο τη διεπαφή Mediator.



Σχημα 2.17: Διάγραμμα Κλάσης Mediator [2]

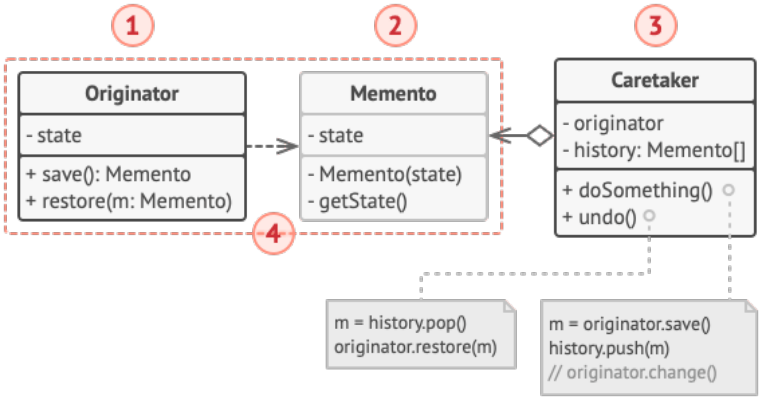
Με τη χρήση του μοτίβου μειώνεται η πολυπλοκότητα αλληλεπίδρασης των αντικειμένων και επιτυγχάνεται η καλύτερη κατανόηση και συντήρηση του κώδικα. Επίσης, ενισχύεται η ευελιξία επικοινωνίας των αντικειμένων, καθώς μπορούν να εφαρμοστούν αλλαγές απευθείας στον Mediator, χωρίς να απαιτούνται τροποποιήσεις στα επιμέρους τμήματα. Τέλος, επιτυγχάνεται χαμηλή σύζευξη μεταξύ των αντικειμένων, πράγμα που προωθεί την επαναχρησιμοποίηση κώδικα και διευκολύνει τη διαδικασία εφαρμογής δοκιμών και ελέγχων στον κώδικα. Ωστόσο, όσο περισσότερη λογική προστίθεται στον Mediator, τόσο αυξάνεται ο κίνδυνος να αυξηθεί η πολυπλοκότητά του και αυτός να μετατραπεί σε μονόλιθο. Επίσης, το επιπλέον επίπεδο αφαίρεσης μπορεί να έχει επιπτώσεις στην απόδοση του συστήματος.

2.3.5 Memento

Το Memento είναι ένα συμπεριφορικό μοτίβο σχεδίασης που επιτρέπει σε αντικείμενα να αποθηκεύουν και να επαναφέρουν προηγούμενες καταστάσεις τους, χωρίς να εκθέτουν λεπτομέρειες της υλοποίησής τους.

Το μοτίβο αυτό χρησιμοποιείται κυρίως για την υλοποίηση μηχανισμών αναίρεσης ή για αποθήκευση στιγμιοτύπων της κατάστασης ενός αντικειμένου και επαναφοράς τους, κατά το δοκούν. Ακόμα, όταν είναι επιθυμητή η διατήρηση της κατάστασης ενός αντικειμένου, χωρίς να παρέχεται πρόσβαση σε εσωτερικές λεπτομέρειές του και χωρίς να παραβιάζεται η ενθυλάκωση.

Η υλοποίηση του μοτίβου περιλαμβάνει την εισαγωγή μιας κλάσης Originator που αντιπροσωπεύει το αντικείμενο του οποίου η κατάσταση πρέπει να αποθηκευτεί, και της κλάσης Memento που αντιπροσωπεύει την αποθηκευμένη κατάσταση. Τέλος, μιας κλάσης φροντιστή που είναι υπεύθυνη για τη λήψη, φύλαξη και επαναφορά των αποθηκευμένων καταστάσεων του Originator.



Σχημα 2.18: Διάγραμμα Κλάσης Memento [2]

Η χρήση του Memento μοτίβου προσφέρει το πλεονέκτημα της εύκολης διαχείρισης του ιστορικού των καταστάσεων ενός αντικειμένου. Επιτυγχάνει να μην επιφορτίζεται επιπλέον το αρχικό αντικείμενο, καθώς οι αποθηκευμένες καταστάσεις υπόκεινται σε διαχείριση από την κλάση φροντιστή, και παράλληλα κρατάει την εσωτερική κατάσταση του αντικειμένου προστατευμένη από εξωτερικές αλλαγές. Ωστόσο, η αλόγιστη χρήση του μοτίβου, ειδικά για αποθήκευση καταστάσεων με μεγάλο μέγεθος, μπορεί να οδηγήσει στην αυξημένη κατανάλωση μνήμης. Επίσης, η διαδικασία επαναφοράς καταστάσεων μπορεί να είναι περίπλοκη, ειδικά αν υπάρχουν πολλές εξαρτήσεις προς το αντικείμενο. Επιπλέον φόρτο στην πολυπλοκότητα μπορεί να έχει επίσης η υποχρέωση της κλάσης φροντιστή να διαχειρίζεται τον κύκλο ζωής των αρχικών αντικειμένων.

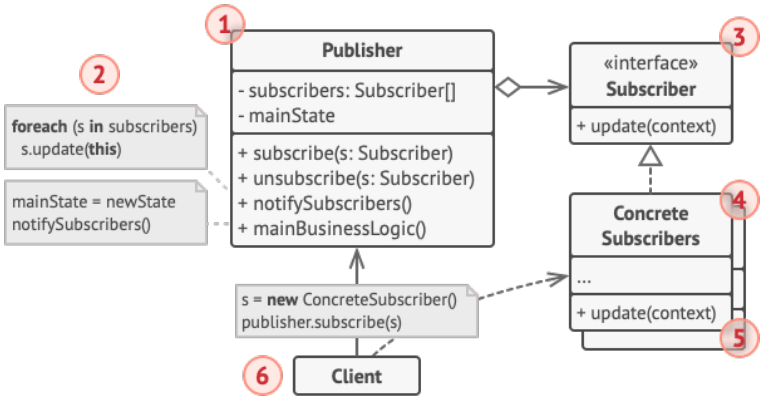
2.3.6 Observer

Το Observer είναι ένα συμπεριφορικό μοτίβο σχεδίασης που ορίζει μια εξάρτηση ¨ένα προς πολλά' μεταξύ αντικειμένων, έτσι ώστε όταν η κατάσταση του ενός αλλάζει, τα εξαρτώμενα από αυτό αντικείμενα να μπορούν να ειδοποιηθούν άμεσα.

Το μοτίβο χρησιμοποιείται σε περιπτώσεις που υπάρχει η απαίτηση, όταν αλλάζει η κατάσταση ενός αντικειμένου - εκδότη, να πρέπει να ειδοποιηθούν και ίσως να ενημερώσουν την κατάστασή τους άλλα αντικείμενα - συνδρομητές, οι ταυτότητες των οποίων είτε δεν είναι γνωστές εκ των προτέρων, είτε αυτά πρέπει να αλλάζουν δυναμικά, χωρίς να τροποποιείται το αντικείμενο - εκδότης.

Η υλοποίηση του μοτίβου απαιτεί την εισαγωγή μιας κλάσης - εκδότη που θα ορίζει μεθόδους για προσθήκη, αφαίρεση και ειδοποίηση συνδρομητών, και θα διατηρεί μια λίστα των

συνδρομητών του. Επίσης, τον ορισμό μιας διεπαφής για τους συνδρομητές, η οποία θα διαθέτει μια μέθοδο ειδοποίησης τους, που θα καλείται από τον εκδότη. Τέλος, οι συμπαγείς υλοποιήσεις της τελευταίας διεπαφής.



Σχημα 2.19: Διάγραμμα Κλάσης Observer [2]

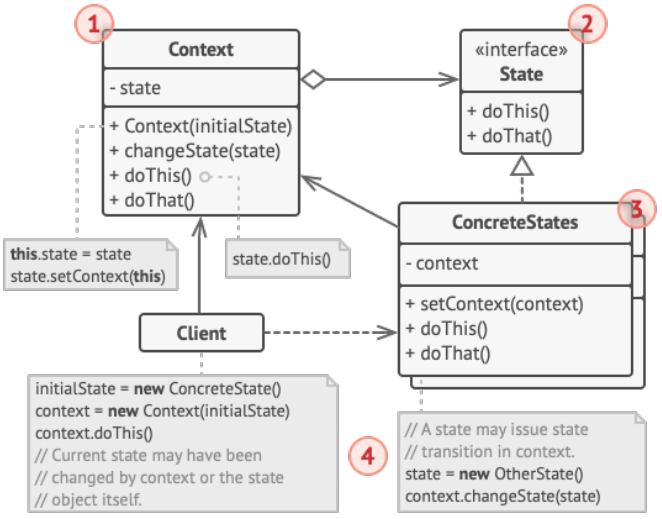
Τα πλεονεκτήματα του Observer μοτίβου είναι πως υποστηρίζει δυναμική προσθαφαίρεση συνδρομητών σε έναν εκδότη, χωρίς να επηρεάζεται ο κώδικάς του, και πως ταυτόχρονα προσφέρει κοινή διεπαφή συνδρομητών για οποιονδήποτε εκδότη. Αυτό ενισχύει την απόζευξη μεταξύ των συμπαγών κλάσεων εκδοτών και συνδρομητών. Τα μειονεκτήματά του, είναι η αδυναμία άσκησης ελέγχου στην σειρά με την οποία ενημερώνονται οι συνδρομητές, και η επιβάρυνση της απόδοσης εκείνων των συστημάτων που συντηρούν μεγάλο αριθμό παρατηρητών ή διαθέτουν πολυπλοκη διαδικασία ενημέρωσης.

2.3.7 State

Το State είναι ένα συμπεριφορικό μοτίβο σχεδίασης που επιτρέπει σε αντικείμενα να μεταβάλλουν τη συμπεριφορά τους, όταν αλλάζει η εσωτερική τους κατάσταση. Αντιμετωπίζει την κατάσταση των αντικειμένων σαν ανεξάρτητο αντικείμενο που μπορεί να τροποποιηθεί κατά τη διάρκεια της εκτέλεσης, από κάποια λειτουργία. Με αυτό τον τρόπο, δημιουργείται η ψευδαίσθηση οτι τα αντικείμενα αλλάζουν κλάση.

Το μοτίβο χρησιμοποιείται όταν η συμπεριφορά κάποιου αντικειμένου εξαρτάται από την κατάστασή του, και η κατάσταση αυτή πρέπει να αλλάζει δυναμικά. Επίσης, εξυπηρετεί στην εξάλειψη μεγάλων διακλαδώσεων δομών ελέγχου ροής που χρησιμοποιούνται για να καθορίζουν τον τρόπο συμπεριφοράς αντικειμένων ανάλογα με την κατάστασή τους, καθώς οργανώνει τις καταστάσεις και τις αντίστοιχες συμπεριφορές τους, σε διαφορετικές κλάσεις.

Η υλοποίηση του State μοτίβου απαιτεί τον ορισμό μιας διεπαφής State που θα ορίζει τις κοινές μεθόδους που θα έχουν όλες οι συμπαγείς υλοποιήσεις της. Κάθε συμπαγής υλοποίηση θα αναπαριστά μια κατάσταση την οποία μπορούν να λάβουν τα αντικείμενα. Ακόμα, χρειάζεται η υλοποίηση μιας κλάσης Context που θα αφορά το γενικό πλαίσιο και θα διατηρεί αναφορά προς ένα στιγμιότυπο της πρώτης διεπαφής. Η κλάση αυτή θα μεταβιβάζει στο στιγμιότυπο όποια εργασία είναι σχετική με κατάσταση.



Σχημα 2.20: Διάγραμμα Κλάσης State [2]

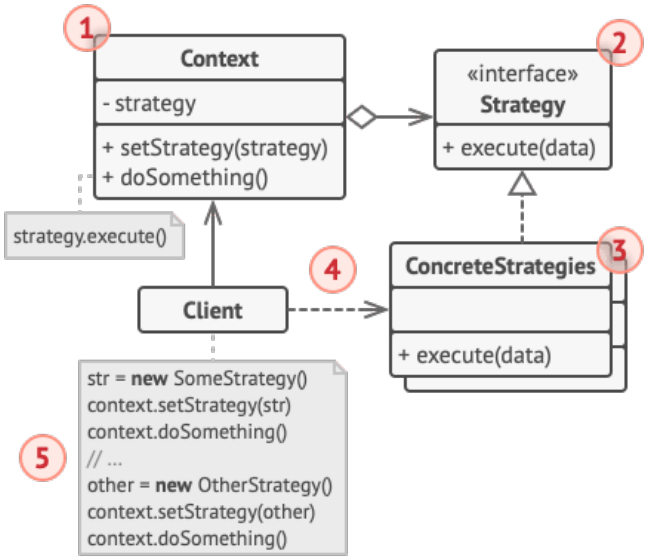
Με τη χρήση του State μοτίβου, επιτυγχάνονται σημαντικά οφέλη. Αρχικά, διαχωρίζονται οι συμπεριφορές που αφορούν συγκεκριμένες καταστάσεις, σε ξεχωριστές κλάσεις, με αποτέλεσμα να μειώνονται σύνθετες διακλαδώσεις δομών ελέγχου. Επίσης, η εισαγωγή νέων καταστάσεων και συμπεριφορών γίνεται χωρίς αλλαγές στον υπόλοιπο κώδικα. Ωστόσο, αυξάνεται και πάλι ο αριθμός των κλάσεων, και στις περιπτώσεις που απαιτείται αναφορά από κάποια αντικείμενα State προς την κλάση Context, προκύπτουν κυκλικές εξαρτήσεις η οποίες είναι δύσκολες στη διαχείριση.

2.3.8 Strategy

Το Strategy είναι ένα συμπεριφορικό μοτίβο σχεδίασης που επιτρέπει τον ορισμό οικογενειών από αλγορίθμους, τους οποίους ενθυλακώνει και καθιστά εναλλάξιμους μεταξύ τους. Επιτρέπει δηλαδή ανάλογα με το εκάστοτε πλαίσιο, τη δυναμική αλλαγή αλγορίθμων που χρησιμοποιούνται από την εφαρμογή.

Το μοτίβο χρησιμοποιείται όταν είναι επιθυμητή η εφαρμογή διαφορετικών παραλλαγών κάποιου αλγορίθμου σε ένα αντικείμενο, και η δυναμική επιλογή ή εναλλαγή αυτών. Επίσης, όταν επιδιώκεται ο διαχωρισμός των λεπτομερειών υλοποίησης ενός αλγορίθμου από τον κώδικα που τον χρησιμοποιεί, και για την αντικατάσταση σύνθετων διακλαδώσεων δομών ελέγχου που έχουν ρόλο να αποφαίνονται για τη συμπεριφορά ενός αλγορίθμου.

Η υλοποίηση του Strategy μοτίβου απαιτεί την εισαγωγή μιας διεπαφής Strategy που θα ορίζει τη μέθοδο που πρέπει να υλοποιούν όλες οι συμπαγείς στρατηγικές. Ακόμα, μιας κλάσης Context που θα διατηρεί αναφορά προς μια συμπαγή στρατηγική, με την οποία θα επικοινωνεί μέσω της προαναφερθείσας διεπαφής.



Σχημα 2.21: Διάγραμμα Κλάσης Strategy [2]

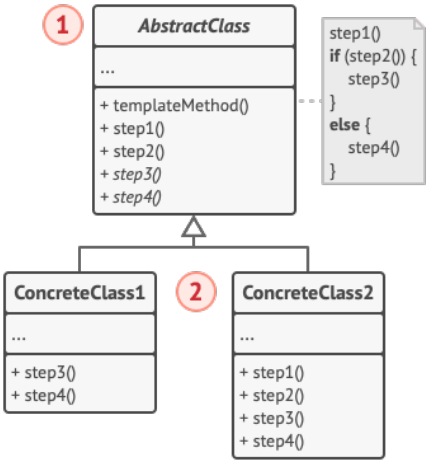
Στα πλεονεκτήματα χρήσης του μοτίβου συγκαταλέγονται η ευελιξία στην εναλλαγή αλγορίθμων με δυναμικό τρόπο, η χαλαρή σύζευξη μεταξύ αλγορίθμων και υπόλοιπου κώδικα, και η ευκολία εισαγωγής νέων στρατηγικών χωρίς τροποποίηση του υπάρχοντος κώδικα. Μειονέκτήματα αποτελούν η αύξηση της πολυπλοκότητας που προκύπτει από την εισαγωγή νέων κλάσεων, και η απαίτηση να γνωρίζει ο κώδικας πελάτη τις διαφορές μεταξύ αλγορίθμων, για να μπορεί να κάνει την αντίστοιχη επιλογή. Επίσης, η επιλογή του μοτίβου κρίνεται περιττή σε περιπτώσεις χρήσης γλωσσών προγραμματισμού που μπορούν να υλοποιούν διαφορετικές παραλλαγές αλγορίθμων μέσα σε ανώνυμες μεθόδους.

2.3.9 Template Method

Το Template Method αποτελεί ένα συμπεριφορικό μοτίβο σχεδίασης το οποίο ορίζει το σκελετό ενός αλγορίθμου σε κάποια υπερκλάση, και επιτρέπει σε υποκλάσεις της, να υπερκαλύπτουν ορισμένα βήματα χωρίς να αλλάζουν τη δομή του.

Το μοτίβο χρησιμοποιείται σε περιπτώσεις ύπαρξης ενός κοινού αλγορίθμου που μπορεί να παρουσιάζει μικρές παραλλαγές στην εκτέλεση ορισμένων βημάτων του, τα οποία ο κώδικας πελάτη θα πρέπει να μπορεί να επεκτείνει, αλλά όχι να τροποποιεί τον υπόλοιπο αλγόριθμο.

Η υλοποίησή του Template Method μοτίβου περιλαμβάνει τον ορισμό μια αφηρημένης υπερκλάσης, η οποία δηλώνει αφενός αφηρημένες μεθόδους που αντιπροσωπεύουν τα βήματα του αλγορίθμου, και αφετέρου μια μέθοδο που καλεί τις μεθόδους αυτές με συγκεκριμένη σειρά. Ακόμα, τις υποκλάσεις της που υπερκαλύπτουν τις προαναφερθέντες αφηρημένες μεθόδους.



Σχημα 2.22: Διάγραμμα Κλάσης Template Method [2]

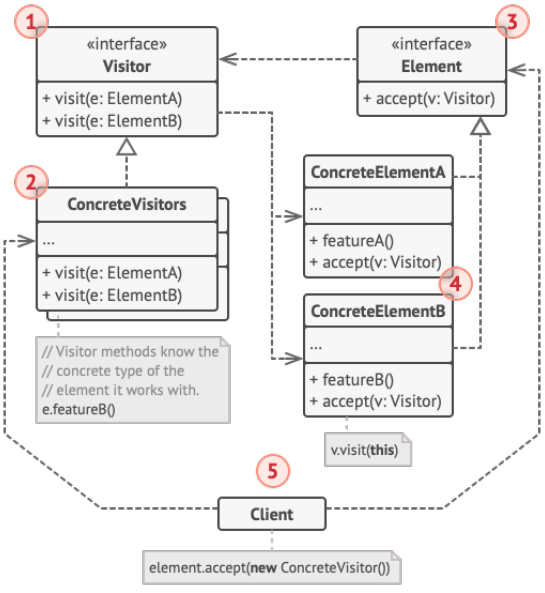
Το μοτίβο προάγει την επαναχρησιμοποίηση κώδικα και τη συγκέντρωση κοινών τμημάτων κώδικα αλγορίθμων σε ένα σημείο, μειώνοντας έτσι τον πλεονασμό. Επίσης, προσφέρει την ευελιξία στις υποκλάσεις να προσαρμόζουν βήματα των αλγορίθμων, διατηρώντας αμετάβλητη τη δομή αυτών. Ωστόσο, αυτή η στατικότητα στη δομή των αλγορίθμων μπορεί να εισάγει περιορισμούς στον κώδικα πελάτη που ίσως επιθυμεί να μπορεί να την τροποποιεί. Ακόμα, η διαιρεμένη υλοποίηση αλγορίθμων σε υπερκλάσεις και υποκλάσεις μπορεί να κάνει τον κώδικα λιγότερο ευανάγνωστο και περισσότερο δυσνόητο.

2.3.10 Visitor

Το Visitor είναι ένα συμπεριφορικό μοτίβο σχεδίασης που επιτρέπει το διαχωρισμό των αλγορίθμων, από τα αντικείμενα στα οποία λειτουργούν. Επιτρέπει την προσθήκη νέων λειτουργιών σε υπάρχουσες δομές αντικειμένων, χωρίς να τις τροποποιεί.

Χρησιμοποιείται για την εκτέλεση λειτουργιών σε αντικείμενα που είναι οργανωμένα σε πολύπλοκες δομές. Αυτές οι λειτουργίες σχετίζονται με τις κλάσεις των αντικειμένων, και τα αντικείμενα μπορεί να είναι στιγμιότυπα διαφορετικών κλάσεων και να συμπεριφέρονται διαφορετικά με βάση αυτές. Επίσης, το μοτίβο εφαρμόζεται για τη διατήρηση της επιχειρησιακής λογικής των κλάσεων, καθαρής από λειτουργίες που δεν σχετίζονται άμεσα με τις βασικές τους δραστηριότητες.

Η υλοποίηση του Visitor μοτίβου απαιτεί την εισαγωγή μιας διεπαφής Visitor που θα δηλώνει ένα σύνολο μεθόδων για την επίσκεψη συμπαγών στοιχείων που μπορούν να δεχθούν επίσκεψη. Επίσης, συμπαγών υλοποιήσεων αυτής. Ακόμα, περιλαμβάνει μια διεπαφή Element η οποία δηλώνει μια μέθοδο αποδοχής επισκεπτών, και τις συμπαγείς υλοποιήσεις αυτής, που καλούν τις προαναφερθέντες μεθόδους επίσκεψης των συμπαγών επισκεπτών, με παράμετρο τον εαυτό τους.



Σχημα 2.23: Διάγραμμα Κλάσης Visitor [2]

Η χρήση του μοτίβου προσφέρει το θετικό ότι μπορεί να εισάγει καινούργιες συμπεριφορές σε κλάσεις, χωρίς να απαιτούνται τροποποιήσεις αυτών. Επίσης, διαχωρισμό ανησυχιών,

καθώς διαχωρίζει τα αντικείμενα από τις λειτουργίες που εκτελούνται σε αυτά. Ωστόσο, για να μπορούν οι επισκέπτες να εκτελούν τις λειτουργίες τους, συνήθως χρειάζονται πρόσβαση στην εσωτερική κατάσταση των στοιχείων, πράγμα που παραβιάζει την αρχή της ενθυλάκωσης. Επιπλέον, για κάθε νέα κλάση στοιχείων που προστίθεται, πρέπει να ενημερώνονται κατάλληλα όλοι οι επισκέπτες, γεγονός που ανεβάζει το κόστος συντήρησης του μοτίβου.

Συνοπτικά, συμπεραίνεται πως τα αντικειμενοστρεφή μοτίβα σχεδίασης είναι παραπάνω από απλές λύσεις σε επαναλαμβανόμενα προβλήματα. Ενσωματώνουν αρχές της καλής σχεδίασης, όπως η αφαίρεση, η ενθυλάκωση και η αρθρωτότητα του κώδικα, και παρέχουν στέρεες βάσεις για τη δημιουργία αξιόπιστου, επεκτάσιμου και συντηρήσιμου λογισμικού.

Κεφάλαιο 3

Εφαρμοσμένοι Συνδυασμοί Αντικειμενοστρεφών Μοτίβων Σχεδίασης

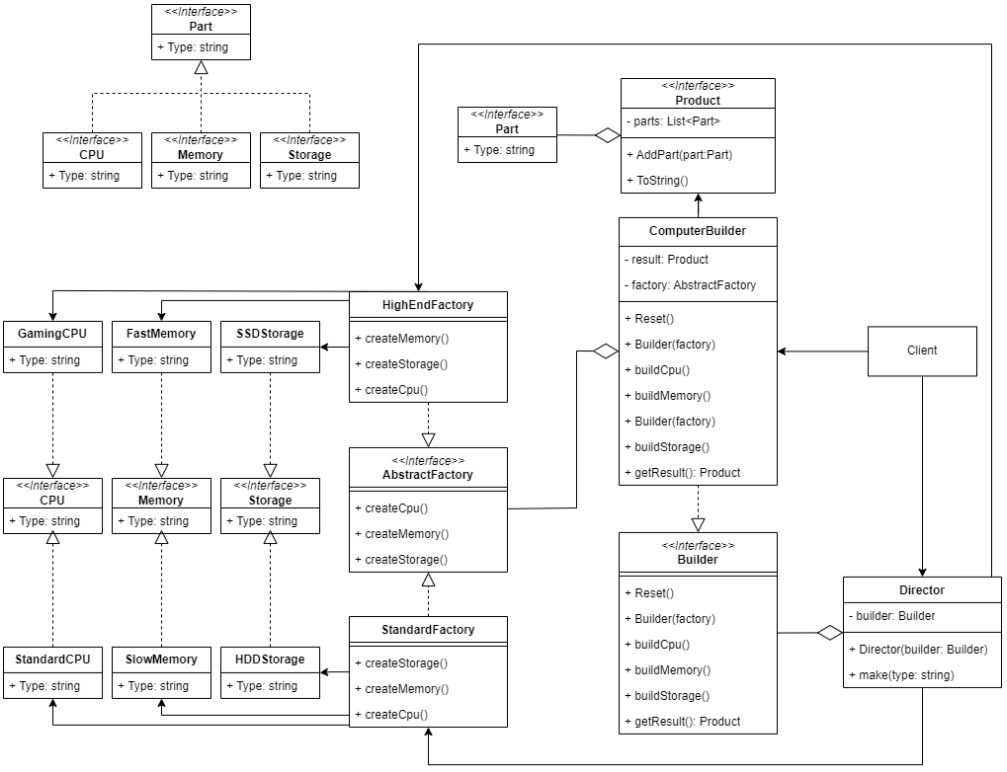
Στο κεφάλαιο αυτό, θα διερευνηθούν κάποιοι τρόποι με τους οποίους τα παραδοσιακά αντικειμεντοστρεφή μοτίβα σχεδίασης του προηγούμενου κεφαλαίου, μπορούν να συνδυαστούν για τη δημιουργία προηγμένων αρχιτεκτονικών λύσεων.

3.1 Builder & Abstract Factory

Στο πρώτο υποκεφάλαιο θα διερευνηθεί ο συνδυασμός της εφαρμογής του Builder και του Abstract Factory μοτίβου σε μια ενιαία υλοποίηση. Αυτός ο συνδυασμός αντιμετωπίζει αποδοτικά τις δυσκολίες που σχετίζονται με την υλοποίηση περίπλοκων αντικειμένων, όπως Η/Υ που περιλαμβάνουν πολλά και διαφορετικά εξαρτήματα που μπορεί να έχουν διαφορετικές υλοποιήσεις.

Το σενάριο που εξετάζεται, αφορά ένα σύστημα διαμόρφωσης Η/Υ. Αυτό το σύστημα θα στοχεύει στην κατασκευή Η/Υ χρησιμοποιώντας εξαρτήματα που ποικίλλουν ανάλογα με τον επιδιωκόμενο σκοπό του Η/Υ. Για παράδειγμα, ένας Η/Υ που προορίζεται για παιχνίδια απαιτεί εξαρτήματα υψηλής απόδοσης, όπως γρήγορο επεξεργαστή, μνήμη υψηλής ταχύτητας και εξωτερική κάρτα γραφικών, ενώ ένας συμβατικός Η/Υ γραφείου μπορεί να χρησιμοποιεί πιο συμβατικά εξαρτήματα.

Ακολουθεί διάγραμμα κλάσης και υλοποίηση σε κώδικα C# για το εν λόγω σενάριο:



Σχημα 3.1: Διάγραμμα Κλάσης Builder & Abstract Factory

Το Abstract Factory μοτίβο χρησιμοποιείται για τη δημιουργίων οικογενειών από σχετιζόμενα μεταξύ τους αντικείμενα, όπως εξαρτήματα Η/Υ, χωρίς να προσδιορίζει τις συγκεκριμένες κλάσεις τους. Η διεπαφή Abstract Factory δηλώνει μεθόδους για τη δημιουργία διαφορετικών εξαρτημάτων, όπως μνήμη, επεξεργαστή και αποθηκευτικό μέσο, και τα συμπαγή εργοστάσια HighEndFactory & StandardFactory υλοποιούν αυτές τις μεθόδους και παράγουν προϊόντα - εξαρτήματα των διαφορετικών τύπων που αναφέρθηκαν.

```
1          public interface AbstractFactory
2          {
3              public Memory CreateMemory();
4              public CPU CreateCPU();
5              public Storage CreateStorage();
6          }
7
8          public class HIghEndFactory : AbstractFactory
9          {
```

```
10              public Memory CreateMemory()
11              {
12                  return new FastMemory();
13              }
14              public CPU CreateCPU()
15              {
16                  return new GamingCPU();
17              }
18              public Storage CreateStorage()
19              {
20                  return new SSDStorage();
21              }
22
23          }
24
25          public class StandardFactory : AbstractFactory
26          {
27              public Memory CreateMemory()
28              {
29                  return new SlowMemory();
30              }
31              public CPU CreateCPU()
32              {
33                  return new StandardCPU();
34              }
35              public Storage CreateStorage()
36              {
37                  return new HDDStorage();
38              }
39          }
```

Στο Builder μοτίβο, η διεπαφή Builder ορίζει τη σύμβαση για την κατασκευή ενός Η/Υ βήμα προς βήμα. Οι μέθοδοι BuildCPU, BuildStorage και BuildMemory αντιπροσωπεύουν τα βήματα που απαιτούνται για την κατασκευή των διαφορετικών εξαρτημάτων του Η/Υ. Η κλάση ComputerBuilder υλοποιεί την προαναφερθείσα διεπαφή και χρησιμοποιεί ένα Abstract Factory για να αποκτήσει τα εξαρτήματα. Η προσέγγιση αυτή επιτρέπει την αλλαγή της σύνθεσης των Η/Υ με την αλλαγή του εκάστοτε χρησιμοποιούμενου εργοστασίου, διαφοροποιώντας έτσι τις διαμορφώσεις των Η/Υ ανάλογα με τη χρήση για την οποία προορίζονται. Η κλάση Director ενορχηστρώνει τη διαδικασία κατασκευής Η/Υ, κάνοντας

χρήση ενός Builder. Αποτελεί ένα στρώμα αφαίρεσης που περιλαμβάνει τα βήματα που απαιτούνται για την κατασκευή. Τέλος, η κλάση Product αντιπροσωπεύει τον Η/Υ - προϊόν, ως σύνθεση διαφορετικών εξαρτημάτων.

```
1       public interface Builder
2       {
3           public void Reset();
4           public void SetFactory(AbstractFactory factory);
5           public void BuildCPU();
6           public void BuildMemory();
7           public void BuildStorage();
8           public Product GetResult();
9
10      }
11
12      public class ComputerBuilder : Builder
13      {
14          private Product result;
15          private AbstractFactory factory;
16          public void Reset()
17          {
18              this.result = new Product();
19          }
20          public void SetFactory(AbstractFactory factory)
21          {
22              this.factory = factory;
23          }
24          public void BuildCPU() => result.AddPart(factory.CreateCPU());
25          public void BuildMemory() => result.AddPart(factory.CreateMemory());
26          public void BuildStorage() => result.AddPart(factory.CreateStorage());
27          public Product GetResult() => result;
28      }
29
30      public class Director
31      {
32          Builder builder;
33          public Director(Builder builder)
34          {
35              this.builder = builder;
36          }
```

```
37
38          public void Make(string type)
39          {
40              builder.Reset();
41              if (type == "gaming") builder.SetFactory(new HIghEndFactory());
42              else if (type == "standard") builder.SetFactory(new StandardFactory());
43
44              builder.BuildCPU();
45              builder.BuildMemory();
46              builder.BuildStorage();
47          }
48      }
49
50      public class Product
51      {
52          private List<Part> parts = new List<Part>();
53
54          public void AddPart(Part part)
55          {
56              parts.Add(part);
57          }
58
59          public void ToString()
60          {
61              Console.WriteLine("\nComputer contains: ");
62              foreach (var part in parts)
63              {
64                  Console.WriteLine(part.Type);
65              }
66          }
67      }
```

Ακολουθούν οι διεπαφές Part, Memory, CPU και Storage και οι αντίστοιχες κλάσεις που αντιπροσωπεύουν τα διαφορετικά εξαρτήματα ενός Η/Υ.

```
1       public interface Part
2       {
3           string Type { get; }
4       }
5       public interface Memory : Part
```

```
6       {
7           string Type { get; }
8       }
9
10      public class FastMemory : Memory
11      {
12          public string Type => "3200Mhz Memory";
13      }
14
15      public class SlowMemory : Memory
16      {
17          public string Type => "1333Mhz Memory";
18      }
19
20      public interface CPU : Part
21      {
22          string Type { get; }
23      }
24
25      public class GamingCPU : CPU
26      {
27          public string Type => "Gaming Processor";
28      }
29
30      public class StandardCPU : CPU
31      {
32          public string Type => "Standard Processor";
33      }
34
35      public interface Storage : Part
36      {
37          string Type { get; }
38      }
39
40      public class HDDStorage : Storage
41      {
42          public string Type => "HDD Storage";
43      }
44
45      public class SSDStorage : Storage
46      {
```

```
47          public string Type => "SSD Storage";
48      }
```

Ο κώδικας πελάτη χρειάζεται να γνωρίζει μόνο τους συμπαγείς κατασκευαστές και την κλάση Director.

```
1       class Program
2       {
3           static void Main()
4           {
5               ComputerBuilder builder = new ComputerBuilder();
6               Director director = new Director(builder);
7               director.Make("gaming");
8               builder.GetResult().ToString();
9           }
10      }
```

Στο σημείο αυτό θα διερευνηθεί ο αντίκτυπος που θα έχει η εισαγωγή νέων τύπων εξαρτημάτων και κατηγοριών Η/Υ στο σύστημα. Στην τρέχουσα κατάσταση υφίστανται δύο κατηγορίες επεξεργαστών, μνήμης και μέσων αποθήκευσης, και δύο τύποι Η/Υ. Έστω ότι θεωρούμε n τους τύπους των Η/Υ και m τον αριθμό των κατηγοριών εξαρτημάτων που περιλαμβάνει κάθε Η/Υ. Σε περίπτωση που επιθυμούμε να προσθέσουμε μία επιπλέον κατηγορία εξαρτημάτων Η/Υ (διεπαφή και κλάσεις που την υλοποιούν), θα πρέπει να γίνουν οι κάτωθι τροποποιήσεις:

- Προσθήκη μιας διεπαφής PowerSupply και δύο κλάσεων StandardPowerSupply και HighEndPowerSupply.
- Προσθήκη μεθόδου για δημιουργία του νέου εξαρτήματος Η/Υ στο Abstract Factory και σε κάθε συμπαγή υλοποίησή του.
- Προσθήκη μεθόδου BuilderPowerSupply στη διεπαφή του Builder και αντίστοιχη υλοποίησή της στην κλάση ComputerBuilder
- Προσθήκη κλήσης της μεθόδου BuilderPowerSupply στον Director, και της CreatePowerSupply στον Builder.

Άρα, προστίθενται 1 διεπαφή, 2 κλάσεις, n+3 μέθοδοι και 2 κλήσεις μεθόδων και επομένως η πολυπλοκότητα εκτιμάται ως O(n).

Σε περίπτωση που επιθυμούμε να προσθέσουμε ένα επιπλέον τύπο Η/Υ (π.χ. Κβαντικό Η/Υ), θα πρέπει να γίνουν οι κάτωθι τροποποιήσεις:

- Προσθήκη νέας κλάσης QuantumFactory που κληρονομεί από το Abstract Factory.
- Προσθήκη m κλάσεων (μια για κάθε τύπο εξαρτήματος Η/Υ).
- Προσθήκη νέου ελέγχου στη μέθοδο Make του Director, για κάλυψη της περίπτωσης Quantum.
- Προσθήκη κλήσης της Make μεθόδου στον κώδικα πελάτη, με παράμετρο το λεκτικό "quantum".

Άρα, προστίθενται m+1 κλάσεις, 1 έλεγχος και 1 κλήση μεθόδου και επομένως η πολυπλοκότητα εκτιμάται ως O(m).

Ο συνδυασμός αυτός φαίνεται να παρέχει έναν καλά δομημένο τρόπο διαχείρισης της δημιουργίας νέων τύπων Η/Υ. Αρχικά, καθιστά αρκετά ευέλικτη την κατασκευή σύνθετων αντικειμένων, των οποίων τα μέρη πρέπει να επιλέγονται από οικογένειες αντικειμένων. Τα μέρη αυτά παρέχονται από το Abstract Factory και συναρμολογούνται από τον Builder. Επίσης, ο Director ενθυλακώνει τη λογική και τις πληροφορίες κατασκευής, με αποτέλεσμα ο κώδικας πελάτη να μη μολύνεται με αχρείαστη πληροφορία. Συμπεραίνεται ακόμα από τα σενάρια που εξετάστηκαν, πως το δημιουργηθέν σύστημα κατασκευής Η/Υ είναι εύκολα επεκτάσιμο. Ωστόσο, δεν μπορεί να παραβλεφθεί πως ο συνδυασμός των δύο μοτίβων φαίνεται να εισάγει πολυπλοκότητα στο σύστημα, καθώς απαιτεί τον ορισμό πολλών κλάσεων και διεπαφών. Επίσης, προκαλεί ανησυχίες η κατανομή της λογικής της κατασκευής αντικειμένων σε πολλά αντικείμενα.

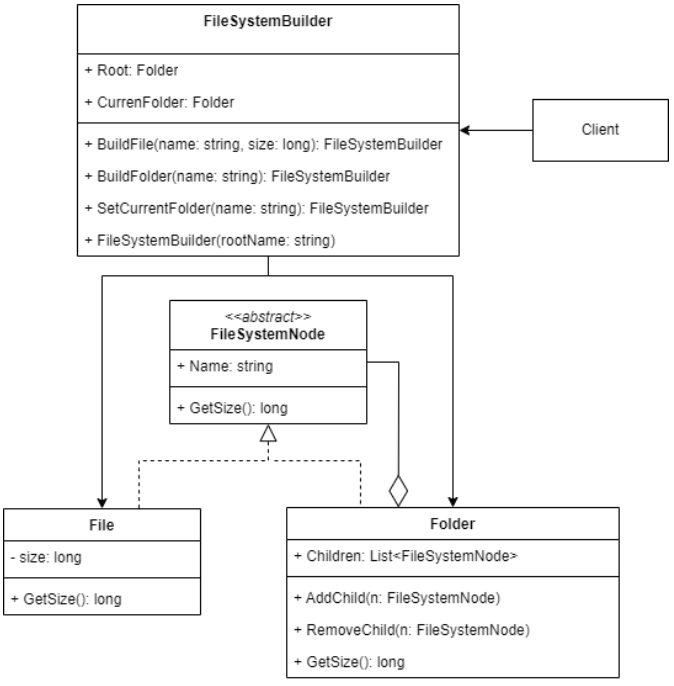
Πριν την επιλογή αυτού του είδους υλοποίησης, αντί κάποιας άλλης πιο απλής και άμεσης στην προσέγγισή της, απαιτείται να διερευνηθούν οι ανάγκες του εκάστοτε project. Λαμβάνονται υπόψη παράμετροι όπως η συχνότητα εισαγωγής νέων τύπων εξαρτημάτων Η/Υ και τύπων Η/Υ, και η επιθυμία για πιο αρθρωτό και επεκτάσιμο ή πιο απλό κώδικα, λαμβάνοντας υπόψη το επιπλέον κόστος διατήρησης της πιο οργανωμένης δομής της πρώτης περίπτωσης.

3.2 Builder & Composite

Σε αυτό το υποκεφάλαιο θα διερευνηθεί ο συνδυασμός της εφαρμογής του Builder και του Composite μοτίβου σε μια ενιαία υλοποίηση. Αυτός ο συνδυασμός προσφέρει μια δομημένη προσέγγιση για την κατασκευή σύνθετων ιεραρχικών δομών που προσομοιάζουν δέντρα, όπως είναι ένα σύστημα αρχείων.

Το σενάριο που εξετάζεται, αφορά τη δημιουργία ενός μηχανισμού κατασκευής συστημάτων αρχείων. Τα εν λόγω συστήματα θα περιλαμβάνουν ως κόμβους, αρχεία που θα λειτουργούν ως φύλλα, και φακέλους που θα λειτουργούν ως σύνθετα αντικείμενα και θα μπορούν να περιλαμβάνουν άλλα αρχεία και φακέλους μέσα τους.

Ακολουθεί διάγραμμα κλάσης και υλοποίηση σε κώδικα C# για το εν λόγω σενάριο:



Σχημα 3.2: Διάγραμμα Κλάσης Builder & Composite

Το Builder μοτίβο ενθυλακώνει την κατασκευαστική λογική του συστήματος αρχείων. Η κλάση FileSystemBuilder διατηρεί αναφορές αφενός προς τον γονικό φάκελο Root που αντιπροσωπεύει όλο το σύστημα αρχείων και αφετέρου προς τον τρέχων - επιλεγμένο φάκελο. Επίσης, παρέχει μεθόδους για την κατασκευή φακέλων και αρχείων, και για τον ορισμό τρέχοντος φακέλου εργασίας με σκοπό τη διευκόλυνση της πλοήγησης στους φάκελους.

```
public class FileSystemBuilder
{
    public Folder Root {get;}
    public Folder CurrentFolder {get; set;}

    public FileSystemBuilder(string rootName)
    {
        Root = new Folder(rootName);
        CurrentFolder = Root;
    }

    public FileSystemBuilder BuildFolder(string name)
    {
        var folder = new Folder(name);
        CurrentFolder.AddChild(folder);
        return this;
    }

    public FileSystemBuilder BuildFile(string name, long size)
    {
        var file = new File(name, size);
        CurrentFolder.AddChild(file);
        return this;
    }

    public FileSystemBuilder SetCurrentFolder(string currentFolderName)
    {
        var folderStack = new Stack<Folder>();
        folderStack.Push(Root);
        while (folderStack.Any())
        {
            var currentFolder = folderStack.Pop();

            if(currentFolderName == currentFolder.Name)
            {
                this.CurrentFolder = currentFolder;
                return this;
            }

            foreach (var item in currentFolder.Children.OfType<Folder>())
            {
```

```
                    folderStack.Push(item);
                }
            }
            throw new Exception($"Folder name: '{ currentFolderName }' not found!"
                );
        }

        public Folder Build() {
            return Root;
        }
    }
```

Στο Composite μοτίβο, η αφηρημένη κλάση FileSystemNode ορίζει τη σύμβαση για το ποια στοιχεία και συμπεριφορά πρέπει να έχει κάθε κόμβος. Η κλάση Folder ενεργεί ως σύνθετη και μπορεί να περιέχει άλλους φακέλους και αρχεία, ενώ η κλάση File αντιπροσωπεύει έναν κόμβο φύλλο που δεν επιτρέπεται να έχει παιδιά. Και οι δύο κλάσεις επεκτείνουν την προαναφερθείσα αφηρημένη κλάση, επιτρέποντας έτσι την πολυμορφική αντιμετώπισή των αντικειμένων τους από τον κώδικα πελάτη. Η μέθοδος GetSize, η οποία υπερκαλύπτεται και στις δύο κλάσεις, υπολογίζει στην κλάση File το μέγεθος των αρχείων και στην κλάση Folder το συνολικό μέγεθος όλων των στοιχείων ενός φακέλου, παρουσιάζοντας έτσι τον τρόπο κατά τον οποίο λειτουργίες μπορούν να εκτελεστούν ομοιόμορφα τόσο σε σύνθετα αντικείμενα όσο και σε φύλλα.

```
public abstract class FileSystemNode
{
    public string Name { get; set; }

    protected FileSystemNode(string name)
    {
        Name = name;
    }

    public abstract long GetSize();
}

public class File : FileSystemNode
{
    public long Size { get; }
```

```
16              public File(string name, long Size) : base(name)
17              {
18                  this.Size = Size;
19              }
20
21              public override long GetSize()
22              {
23                  return Size;
24              }
25          }
26
27          public class Folder : FileSystemNode
28          {
29              public List<FileSystemNode> Children {get;} = new List<FileSystemNode>();
30
31              public Folder(string name) : base(name) { }
32
33              public void AddChild(FileSystemNode child)
34              {
35                  Children.Add(child);
36              }
37
38              public void RemoveChild(FileSystemNode child)
39              {
40                  Children.Remove(child);
41              }
42
43              public override long GetSize()
44              {
45                  return Children.Sum(x => x.GetSize());
46              }
47          }
```

Ο κώδικας πελάτη κάνει χρήση του Builder για την κατασκευή μιας ιεραρχικής δομής φακέλων και αρχείων, και υπολογίζει το συνολικό μέγεθος του γονικού φακέλου Root.

```
1           class Program
2           {
3               static void Main(string[] args)
4               {
```

```
5                   var builder = new FileSystemBuilder("root")
6                   .BuildFolder("Folder0")
7                   .SetCurrentFolder("Folder0")
8                   .BuildFile("File_0_0.txt", 12000)
9                   .BuildFile("File_0_1.mkv", 1000000)
10                  .BuildFolder("SubFolder0_0")
11                  .SetCurrentFolder("SubFolder0_0")
12                  .BuildFile("File_0_0_0.mp3", 20000)
13                  .BuildFile("File_0_0_1.pdf", 18000)
14                  .SetCurrentFolder("root")
15                  .BuildFolder("Folder1")
16                  .BuildFile("File_1_0.apk", 250000)
17                  .BuildFile("File_1_1.exe", 87000000)
18                  .Build();
19
20                  Console.WriteLine($"Total size of (root): { builder.GetSize() } Bytes"
                        );
21              }
22          }
```

Στο σημείο αυτό θα γίνει αξιολόγηση του συνδυασμού στα σενάρια εισαγωγής νέων τύπων κόμβων και νέων μεθόδων που θα εκτελούν λειτουργίες στους κόμβους. Στην τρέχουσα κατάσταση υφίστανται δύο κατηγορίες κόμβων και μια μέθοδος που εκτελεί λειτουργία σε αυτούς. Έστω οτι θεωρούμε n κατηγορίες κόμβων. Σε περίπτωση που επιθυμούμε να προσθέσουμε μία επιπλέον κατηγορία κόμβου, θα πρέπει να γίνουν οι κάτωθι τροποποιήσεις:

- Προσθήκη μιας κλάσης Shortcut που θα κληρονομεί από την FileSystemNode και θα υπερκαλύπτει τη μέθοδο GetSize .
- Προσθήκη μεθόδου BuildShortcut(name, destinationName) για δημιουργία του νέου είδους κόμβου στον Builder.

Άρα, προστίθεται 1 κλάση και 1 μέθοδος (O(1) πολυπλοκότητα). Αυτό το σενάριο καλύπτει και τις δύο πιθανές ευρύτερες κατηγορίες κόμβων (φύλλο και σύνθετο).

Σε περίπτωση που επιθυμούμε να προσθέσουμε μία επιπλέον μέθοδο που εκτελεί κάποια λειτουργία στους κόμβους, θα πρέπει να γίνουν οι κάτωθι τροποποιήσεις:

- Προσθήκη της μεθόδου PrintDetails στην αφηρημένη κλάση FileSystemNode και υπερφορτωμένων υλοποιήσεών της, σε κάθε κλάση που κληρονομεί από την τελευταία.

Άρα, προστίθενται n+1 μέθοδοι (O(n) πολυπλοκότητα).

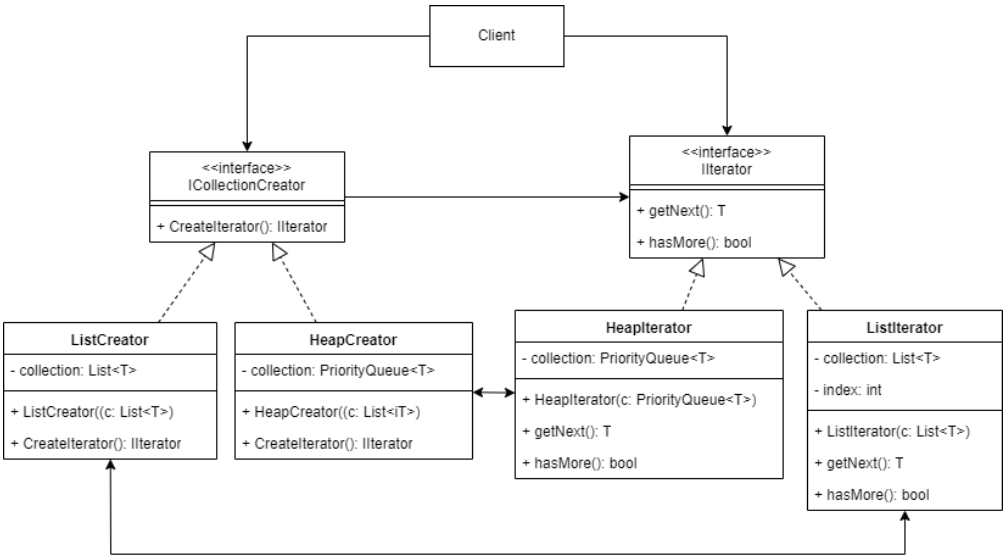
Όπως διαπιστώνεται, ο συνδυασμός των Composite και Builder μοτίβων επιτρέπει τη σαφή και δομημένη κατασκευή σύνθετων ιεραρχικών συστημάτων. Το Builder μοτίβο προσφέρει μια βήμα προς βήμα προσέγγιση και το Composite μοτίβο παρέχει μια ενοποιημένη διεπαφή διαχείρισης σύνθετων αντικειμένων και φύλλων. Επίσης, φαίνεται να επιδεικνύει υψηλό βαθμό επεκτασιμότητας, καθώς μπορούν να προστεθούν νέα στοιχεία χωρίς να μεταβάλλεται σημαντικά το υπόλοιπο σύστημα. Ωστόσο, οφείλει να σημειωθεί πως η εισαγωγή ενός Builder για τη διαχείριση μιας Composite δομής, προσθέτει ένα επίπεδο αφαίρεσης πάνω από το σύστημα, το οποίο ενδεχομένως να εισάγει επιβάρυνσή στην απόκρισή του.

3.3 Iterator & Factory Method

Σε αυτό το υποκεφάλαιο θα διερευνηθεί ο συνδυασμός του Iterator και του Factory Method μοτίβου σε μια ενιαία υλοποίηση. Αυτός ο συνδυασμός επιλέγεται με σκοπό τη δημιουργία ενός αξιόπιστου μηχανισμού παροχής αλγορίθμων διάσχισης συλλογών, ο οποίος θα αποκρύπτει παράλληλα τις λεπτομέρειες υλοποίησης των τρόπων διάσχισης από τον κώδικα πελάτη.

Το σενάριο που εξετάζεται, αφορά την κατασκευή ενός τέτοιου συστήματος που θα επιτρέπει σε συλλογές να δημιουργούν και να επιστρέφουν κατάλληλους Iterators για τη διάσχισή τους, και θα αποτελείται από τις συλλογές αυτές και τους αντίστοιχους αλγόριθμους διάσχισης.

Ακολουθεί διάγραμμα κλάσης και υλοποίηση σε κώδικα C# για το εν λόγω σενάριο:



Σχημα 3.3: Διάγραμμα Κλάσης Iterator & Factory Method

Στο Factory Method μοτίβο, η διεπαφή ICollectionCreator<T> είναι εκείνη που ορίζει τη μέθοδο εργοστασίου και που παρέχει έναν ομοιόμορφο τρόπο δημιουργίας αλγορίθμων διάσχισης για τις συλλογές. Οι συμπαγείς κλάσεις ListCreator<T> και HeapCreator<T> που την υλοποιούν, λειτουργούν ώς προσαρμοσμένα εργοστάσια δημιουργίας Iterator για λίστες και σωρούς αντίστοιχα. Κατά κανόνα, κάθε τέτοια κλάση θα πρέπει να διατηρεί μέσα της μια συλλογή και να επιστρέφει τον αντίστοιχο IIterator<T> που είναι κατάλληλος για να προσπελάσει το συγκεκριμένο τύπο της.

```
1           interface ICollectionCreator<T> where T : IComparable<T>
2           {
3               IIterator<T> CreateIterator();
4           }
5
6           class ListCreator<T> : ICollectionCreator<T> where T : IComparable<T>
7           {
8               private List<T> collection;
9
10              public ListCreator(List<T> collection)
11              {
12                  this.collection = collection;
13              }
14
15              public IIterator<T> CreateIterator()
16              {
```

```
17                  return new ListIterator<T>(collection);
18              }
19          }
20
21          class HeapCreator<T> : ICollectionCreator<T> where T : IComparable<T>
22          {
23              private PriorityQueue<T> collection;
24
25              public HeapCreator(PriorityQueue<T> collection)
26              {
27                  this.collection = collection;
28              }
29
30              public IIterator<T> CreateIterator()
31              {
32                  return new HeapIterator<T>(collection);
33              }
34          }
```

Όσον αφορά το Iterator μοτίβο, η IIterator<T> διεπαφή ορίζει την τυποποιημένη συμπεριφορά διάσχισης συλλογών, με τον ορισμό δύο μεθόδων για τον έλεγχο του αν η συλλογή είναι κενή και για την επιστροφή του επόμενου στοιχείου της συλλογής. Αυτό διασφαλίζει τη συνέπεια στον τρόπο με τον οποίο διατρέχονται οι διαφορετικές συλλογές. Οι κλάσεις ListIterator<T> και HeapIterator<T> αποτελούν προσαρμοσμένες υλοποιήσεις της προαναφερθείσας διεπαφής για τη διάσχιση συλλογών τύπου λίστας και σωρού αντίστοιχα και ενθυλακώνουν τις λεπτομέρειες της λογικής διάσχισης αυτών των τύπων συλλογών.

```
1           interface IIterator<T> where T : IComparable<T>
2           {
3               bool HasMore();
4               T GetNext();
5           }
6
7           class ListIterator<T> : IIterator<T> where T : IComparable<T>
8           {
9               private List<T> collection;
10              private int index = 0;
11
12              public ListIterator(List<T> collection)
```

```
13              {
14                  this.collection = collection;
15              }
16
17              public bool HasMore()
18              {
19                  return index < collection.Count;
20              }
21
22              public T GetNext()
23              {
24                  if (HasMore())
25                  {
26                      return collection[index++];
27                  }
28                  throw new InvalidOperationException("No more elements in the
                        collection.");
29              }
30          }
31
32          class HeapIterator<T> : IIterator<T> where T : IComparable<T>
33          {
34              private PriorityQueue<T> collection;
35
36              public HeapIterator(PriorityQueue<T> collection)
37              {
38                  this.collection = collection;
39              }
40
41              public bool HasMore()
42              {
43                  return collection.Count > 0;
44              }
45
46              public T GetNext()
47              {
48                  if (HasMore())
49                  {
50                      return collection.Dequeue();
51                  }
```

```
52                 throw new InvalidOperationException("No more elements in the
                       collection.");
53             }
54         }
```

Παρακάτω παρέχεται η συγκεκριμένη υλοποίηση της ουράς προτεραιότητας που χρησιμοποιείται.

```
1          public class PriorityQueue<T> where T : IComparable<T>
2          {
3              private List<T> heap = new List<T>();
4              public int Count => heap.Count;
5
6              public PriorityQueue() { }
7
8              public PriorityQueue(IEnumerable<T> collection)
9              {
10                 foreach (var item in collection)
11                 {
12                     Enqueue(item);
13                 }
14             }
15
16             public void Enqueue(T x)
17             {
18                 heap.Add(x);
19                 int i = Count - 1;
20
21                 while (i > 0)
22                 {
23                     int p = (i - 1) / 2;
24                     if ( heap[p].CompareTo(x) <= 0) break;
25
26                     heap[i] = heap[p];
27                     i = p;
28                 }
29
30                 if (Count > 0) heap[i] = x;
31             }
32
```

```
33              public T Dequeue()
34              {
35                  T target = Peek();
36                  T root = heap[Count - 1];
37                  heap.RemoveAt(Count - 1);
38
39                  int i = 0;
40                  while (i * 2 + 1 < Count)
41                  {
42                      int a = i * 2 + 1;
43                      int b = i * 2 + 2;
44                      int c = b < Count && heap[b].CompareTo(heap[a]) < 0 ? b : a;
45
46                      if (heap[c].CompareTo(root) >= 0) break;
47                      heap[i] = heap[c];
48                      i = c;
49                  }
50
51                  if (Count > 0) heap[i] = root;
52                  return target;
53              }
54              public T Peek()
55              {
56                  if (Count == 0) throw new InvalidOperationException("Queue is empty.")
                        ;
57                  return heap[0];
58              }
59          }
```

Τέλος, ο κώδικας πελάτη λαμβάνει αντικείμενα ICollectionCreator<T> και χρησιμοποιεί τους Iterator που παράγουν, για να διασχίσει τις συλλογές τους.

```
1           class Program
2           {
3               static void Main(string[] args)
4               {
5                   // Initialization code
6                   List<int> listCollection = new List<int> { 1, 2, 3, 4, 5, 6 };
7                   ICollectionCreator<int> listFactory = new ListCreator<int>(
                        listCollection);
```

```
8
9               PriorityQueue<int> heapCollection = new PriorityQueue<int>(new[] { 7,
                    8, 9, 10, 11, 12 });
10              ICollectionCreator<int> heapFactory = new HeapCreator<int>(
                    heapCollection);
11
12              //Client code
13              IIterator<int> listIterator = listFactory.CreateIterator();
14              Console.WriteLine("ListCollection elements:");
15              while (listIterator.HasMore())
16              {
17                  Console.Write(listIterator.GetNext() + " ");
18              }
19              Console.WriteLine();
20
21              IIterator<int> heapIterator = heapFactory.CreateIterator();
22              Console.WriteLine("\nHeapCollection elements:");
23              while (heapIterator.HasMore())
24              {
25                  Console.Write(heapIterator.GetNext() + " ");
26              }
27          }
28      }
```

Στο σημείο αυτό θα γίνει ανάλυση του συνδυασμού στο σενάριο εισαγωγής νέου τύπου συλλογής που ικανοποιείται από υπάρχων αλγόριθμο διάσχισης, εισαγωγής νέου αλγορίθμου διάσχισης για υφιστάμενο τύπο συλλογής, και εισαγωγής νέου ζεύγους συλλογής - αλγορίθμου διάσχισης. Στην τρέχουσα κατάσταση υφίστανται δύο κατηγορίες συλλογών και αλγορίθμων διάσχισης. Σε περίπτωση που επιθυμούμε να προσθέσουμε μία επιπλέον κατηγορία συλλογής, θα πρέπει να γίνουν οι κάτωθι τροποποιήσεις:

- Προσθήκη μιας κλάσης BinaryTree<T>.
- Προσθήκη μιας κλάσης BinaryTreeCreator<T> που θα κληρονομεί από την ICollectionCreator<T> και θα επιστρέφει ένα στιγμιότυπο της HeapIterator<T>.

Άρα, προστίθενται 2 κλάσεις (O(1) πολυπλοκότητα).

Σε περίπτωση που επιθυμούμε να εισάγουμε έναν επιπλέον αλγόριθμο διάσχισης για μια υπάρχουσα συλλογή, θα πρέπει να γίνουν οι κάτωθι τροποποιήσεις:

- Προσθήκη μιας κλάσης ReverseListIterator<T> που θα κληρονομεί από την IIterator<T> και θα διασχίζει μια List<T> αντίστροφα.
- Προσθήκη μιας μεθόδου εργοστάσιο στην κλάση ListCreator<T> ή τροποποίηση της υπαρχουσας για να παράγει ένα στιγμιότυπο της ReverseListIterator<T> με κριτήριο κάποια συνθήκη.

Άρα, προστίθενται 1 κλάση και 1 μέθοδος (O(1) πολυπλοκότητα).

Στο τελευταίο σενάριο εξετάζεται η εισαγωγή ενός νέου ζεύγους συλλογής - αλγορίθμου διάσχισης, όπως Graph<T> και DepthFirstIterator<T>. Για την υλοποίηση αυτή θα απαιτούνταν οι κάτωθι τροποποιήσεις:

- Προσθήκη μιας κλάσης Graph<T>.
- Προσθήκη μιας κλάσης DepthFirstIterator<T> που θα κληρονομεί από την IIterator<T> και θα διασχίζει έναν γράφο.
- Προσθήκη μιας κλάσης GraphCreator<T> που θα κληρονομεί από την ICollectionCreator<T> και θα επιστρέφει ένα στιγμιότυπο της DepthFirstIterator<T>.

Άρα, προστίθενται 3 κλάσεις (O(1) πολυπλοκότητα).

Όπως διαπιστώνεται, ο συνδυασμός των Iterator και Factory Method μοτίβων επιδεικνύει σημαντική ευελιξία και προσαρμοστικότητα σε διάφορα σενάρια. Η εισαγωγή νέων τύπων συλλογής και αλγορίθμων διάσχισης μπορεί να επιτευχθεί με ελάχιστες αλλαγές. Παράλληλα, επιτρέπει στον κώδικα πελάτη τη δυνατότητα διάσχισης συλλογών μέσω μιας κοινής διεπαφής, χωρίς να του κάνει γνωστές τις λεπτομέρειες υλοποίησης αυτών και των αλγορίθμων διάσχισης τους. Αρνητικό του συνδυασμού αυτού μπορεί να θεωρηθεί η μικρή επιβάρυνση που μπορεί να προκαλέσει το επίπεδο αφαίρεσης που εισάγεται πάνω από τις συλλογές.

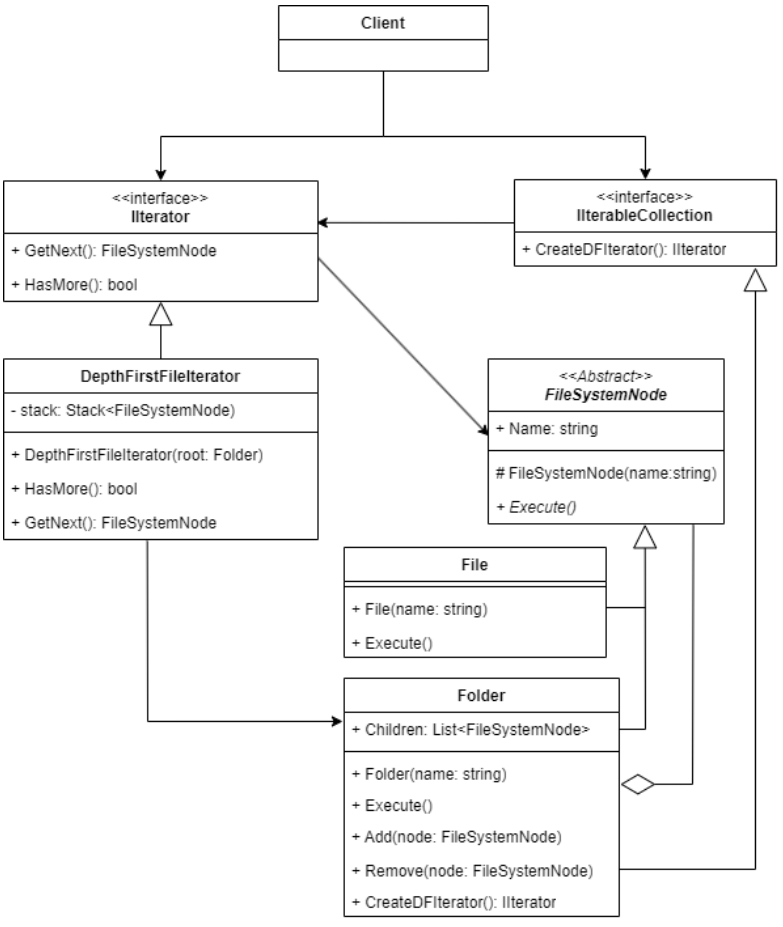
3.4 Iterator & Composite

Σε αυτό το υποκεφάλαιο θα διερευνηθεί ο συνδυασμός του Iterator και του Composite μοτίβου σε μια ενιαία υλοποίηση. Αυτός ο συνδυασμός επιλέγεται με σκοπό τη δημιουργία

ενός αξιόπιστου μηχανισμού διάσχισης πολύπλοκων ιεραρχικών δομών, με απλό και ομοιόμορφο τρόπο, ο οποίος δε θα εκθέτει τις εσωτερικές αναπαραστάσεις των δομών αυτών. Το Composite μοτίβο θα διαχειρίζεται τις σχέσεις μεταξύ των στοιχείων και το Iterator μοτίβο θα παρέχει πρόσβαση σε αυτά.

Το σενάριο που εξετάζεται, αφορά τη κατασκευή ενός μηχανισμού διάσχισης συστημάτων αρχείων. Κάθε τέτοιο σύστημα θα περιλαμβάνει ως κόμβους, αρχεία που θα λειτουργούν ως φύλλα, και φακέλους που θα λειτουργούν ως σύνθετα αντικείμενα και θα μπορούν να περιλαμβάνουν άλλα αρχεία και φακέλους μέσα τους.

Ακολουθεί διάγραμμα κλάσης και υλοποίηση σε κώδικα C# για το εν λόγω σενάριο:



Σχημα 3.4: Διάγραμμα Κλάσης Iterator & Composite

Στο Composite μοτίβο, η αφηρημένη κλάση FileSystemNode ορίζει τη σύμβαση για το ποια στοιχεία και συμπεριφορά πρέπει να έχει κάθε κόμβος. Η κλάση Folder ενεργεί ως σύνθετη και μπορεί να περιέχει άλλους φακέλους και αρχεία, ενώ η κλάση File αντιπροσωπεύει

έναν κόμβο φύλλο που δεν επιτρέπεται να έχει παιδιά. Και οι δύο κλάσεις υλοποιούν την προαναφερθείσα αφηρημένη κλάση, επιτρέποντας έτσι την πολυμορφική αντιμετώπισή των αντικειμένων τους από τον κώδικα πελάτη. Η μέθοδος Execute, η οποία υπερκαλύπτεται και στις δύο κλάσεις, τυπώνει στην κλάση File το όνομα του εκάστοτε αρχείου και στην κλάση Folder το όνομα του φακέλου και όλων των κόμβων που είναι κάτω από αυτόν. Η κλάση Folder υλοποιεί επιπλέον τη διεπαφή IIterableCollection για την προσθήκη της συμπεριφοράς κατασκευής Iterator αντικειμένων.

```
1   public abstract class FileSystemNode
2   {
3       public string Name { set; get; }
4
5       protected FileSystemNode(string name)
6       {
7           Name = name;
8       }
9       public abstract void Execute();
10  }
11
12  public interface IIterableCollection
13  {
14      public IIterator createDFIterator();
15  }
16
17  public class File : FileSystemNode
18  {
19      public File(string name) : base(name) { }
20      public override void Execute()
21      {
22          Console.WriteLine($"Opened File: {Name}");
23      }
24  }
25
26  public class Folder : FileSystemNode, IIterableCollection
27  {
28      public List<FileSystemNode> Children { get; } = new List<FileSystemNode
            >();
29
30      public Folder(string name) : base(name) { }
```

```
31              public override void Execute()
32              {
33                  Console.WriteLine($"Opened Folder: {Name}");
34                  foreach (var node in Children)
35                  {
36                      node.Execute();
37                  }
38              }
39
40              public void Add(FileSystemNode node)
41              {
42                  Children.Add(node);
43              }
44
45              public void Remove(FileSystemNode node)
46              {
47                  Children.Remove(node);
48              }
49
50              public IIterator createDFIterator()
51              {
52                  return new DepthFirstFileIterator(this);
53              }
54          }
```

Όσον αφορά το Iterator μοτίβο, η IIterator<T> διεπαφή ορίζει την τυποποιημένη συμπεριφορά διάσχισης συλλογών, με τον ορισμό δύο μεθόδων για τον έλεγχο του αν η συλλογή είναι κενή και για την επιστροφή του επόμενου στοιχείου της συλλογής. Αυτό διασφαλίζει τη συνέπεια στον τρόπο με τον οποίο διατρέχονται οι διαφορετικές συλλογές. Η κλάση DepthFirstFileIterator<T> αποτελεί προσαρμοσμένη υλοποίηση της προαναφερθείσας διεπαφής για τη διάσχιση του Composite συστήματος αρχείων και ενθυλακώνει της λεπτομέρειες της δομής του και της λογικής διάσχισής του.

```
1           public interface IIterator
2           {
3               bool HasMore();
4               FileSystemNode GetNext();
5           }
6
```

```
public class DepthFirstFileIterator : IIterator
{
    private Stack<FileSystemNode> stack = new Stack<FileSystemNode>();

    public DepthFirstFileIterator(Folder root)
    {
        stack.Push(root);
    }

    public bool HasMore()
    {
        return stack.Count > 0;
    }

    public FileSystemNode GetNext()
    {
        if (!HasMore()) return null;

        var current = stack.Pop();

        if (current is Folder Folder)
        {
            foreach (var element in Folder.Children.AsEnumerable().Reverse())
            {
                stack.Push(element);
            }
        }
        return current;
    }
}
```

Ο κώδικας πελάτη λαμβάνει ένα αντικείμενο IIterableCollection και χρησιμοποιεί τον Iterator που παράγει, για να διασχίσει τη δομή του.

```
class Program
{
    static void Main(string[] args)
    {
        // Initialization code
        var root = new Folder("root");
```

```
7                   var subDir1 = new Folder("subDir1");
8                   var subDir2 = new Folder("subDir2");
9                   var file1 = new File("file1");
10                  var file2 = new File("file2");
11                  var file3 = new File("file3");
12
13                  subDir1.Add(file1);
14                  subDir2.Add(file2);
15                  root.Add(subDir1);
16                  root.Add(subDir2);
17                  root.Add(file3);
18
19                  //Client code
20                  var depthFirstIterator = root.createDFIterator();
21
22                  Console.WriteLine("Depth-First Traversal of File System:");
23                  while (depthFirstIterator.HasMore())
24                  {
25                      var item = depthFirstIterator.GetNext();
26                      item.Execute();
27                  }
28              }
29          }
```

Στο σημείο αυτό θα γίνει ανάλυση του συνδυασμού στο σενάριο εισαγωγής νέου τύπου κόμβου και εισαγωγής νέου αλγορίθμου διάσχισης. Στην τρέχουσα κατάσταση υφίστανται δύο τύποι κόμβων και ένας αλγόριθμος διάσχισης. Έστω n ο αριθμός των διαφορετικών τύπων IIterator και m ο αριθμός των διαφορετικών τύπων κόμβων που υλοποιούν την IIterableCollection. Σε περίπτωση που επιθυμούμε να προσθέσουμε έναν επιπλέον τύπο FileSystemNode, θα πρέπει να γίνουν οι κάτωθι τροποποιήσεις:

Περίπτωση εισαγωγής φύλλου:

- Προσθήκη μιας κλάσης FileShortcut που θα κληρονομεί από την FileSystemNode.

Περίπτωση εισαγωγής σύνθετου αντικειμένου:

- Προσθήκη μιας κλάσης FolderShortcut που θα κληρονομεί από την FileSystemNode και θα υλοποιεί την IIterableCollection.

- Κατάλληλη τροποποίηση της συνθήκης της δομής ελέγχου που ελέγχει τον τύπο του τρέχοντος κόμβου στη μέθοδο GetNext κάθε συμπαγούς IIterator.

Άρα, στην πρώτη περίπτωση προστίθεται 1 κλάση και στην δεύτερη, 1 κλάση και τροποποίηση μιας συνθήκης ελέγχου σε n κλάσεις (O(n) πολυπλοκότητα).

Σε περίπτωση που επιθυμούμε να εισάγουμε έναν επιπλέον αλγόριθμο διάσχισης, θα πρέπει να γίνουν οι κάτωθι τροποποιήσεις:

- Προσθήκη μιας κλάσης BreadthFirstFileIterator που θα υλοποιεί την IIterator<T> και θα διασχίζει την ιεραρχική δομή κατά πλάτος.
- κατάλληλη τροποποίηση της createDFIterator μεθόδου ή προσθήκη μεθόδου createBFIterator σε κάθε τύπο κόμβου που υλοποιεί τη διεπαφή IIterableCollection.

Άρα, προστίθεται 1 κλάση και m μέθοδοι (O(m) πολυπλοκότητα).

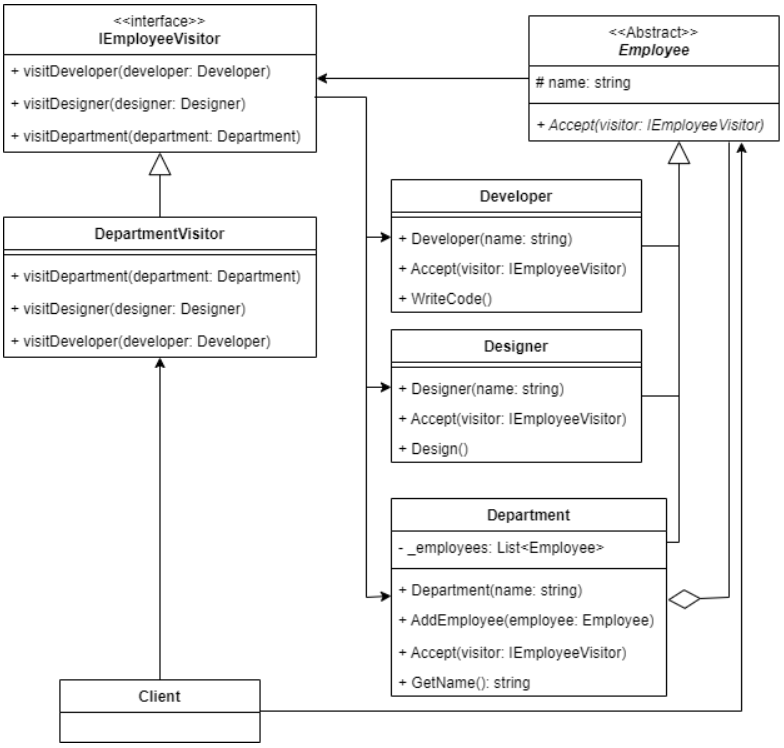
Όπως διαπιστώνεται, ο συνδυασμός των Iterator και Composite μοτίβων επιτυγχάνει να επιτρέπει στον κώδικα πελάτη τη διάσχιση ιεραρχικών δομών που περιλαμβάνουν διαφορετικούς τύπους αντικειμένων, μέσω μιας κοινής διεπαφής, χωρίς να του κάνει γνωστές τις λεπτομέρειες υλοποίησης αυτών και των αλγορίθμων διάσχισης τους, και κατ΄ επέκταση διατηρώντας τον απλό. Σε απλά σενάρια με λίγους τύπους σύνθετων κόμβων και αλγορίθμων διάσχισης, που είναι το σύνηθες φαινόμενο, συνίσταται η χρήση του συνδυασμού. Ωστόσο, η υλοποίηση του μπορεί να γίνει πολύπλοκη για μεγάλες δομές που περιλαμβάνουν πολλούς τύπους σύνθετους αντικειμένων και πολλούς αλγορίθμους διάσχισης.

3.5 Visitor & Composite

Σε αυτό το υποκεφάλαιο θα διερευνηθεί ο συνδυασμός του Visitor και του Composite μοτίβου σε μια ενιαία υλοποίηση. Αυτός ο συνδυασμός επιλέγεται με σκοπό τη διευκόλυνση προσθήκης και εκτέλεσης νέων λειτουργιών σε πολύπλοκες ιεραρχικές δομές αντικειμένων, χωρίς να τροποποιούνται οι δομές τους. Το Composite μοτίβο θα διαχειρίζεται τις σχέσεις μεταξύ των στοιχείων και το Visitor μοτίβο θα αποσυνδέει τη λογική των λειτουργιών από τη δομή των αντικειμένων.

Το σενάριο που εξετάζεται, αφορά τη διαχείριση και την εκτέλεση λειτουργιών σε μια δομή τμημάτων και εργαζομένων ενός οργανισμού. Το σύστημα αυτό θα περιλαμβάνει ως κόμβους, εργαζόμενους που θα λειτουργούν ως φύλλα και τμήματα που θα λειτουργούν ως σύνθετα αντικείμενα και θα μπορούν να περιλαμβάνουν άλλα τμήματα και εργαζομένους μέσα τους.

Ακολουθεί διάγραμμα κλάσης και υλοποίηση σε κώδικα C# για το εν λόγω σενάριο:



Σχημα 3.5: Διάγραμμα Κλάσης Visitor & Composite

Στο Composite μοτίβο, η αφηρημένη κλάση Employee ορίζει τη μέθοδο Accept που πρέπει να υλοποιεί κάθε στοιχείο της ιεραρχίας, με σκοπό να επιτρέπει στο μοτίβο Visitor να αλληλεπιδρά απρόσκοπτα με κάθε τέτοιο στοιχείο. Οι συμπαγείς υλοποιήσεις της, Developer και Designer αντιπροσωπεύουν κόμβους φύλλων, καθένας με τη δική του συμπεριφορά WriteCode και Design αντίστοιχα. Η υπερκάλυψη της μεθόδου Accept από αυτές, διασφαλίζει ότι καλείται η εκάστοτε κατάλληλη μέθοδος επισκέπτη, επιτρέποντας έτσι την εκτέλεση λειτουργιών που αφορούν τα συγκεκριμένα στοιχεία. Σχετικά με την κλάση Department που υλοποιεί την ίδια διεπαφή, αυτή ενεργεί ως σύνθετη και μπορεί να περιέχει άλλους Employee, επιτρέποντας με αυτό τον τρόπο τη δημιουργία δενδρικής δομής. Επίσης, υπερκαλύπτει τη μέθοδο Accept για να δέχεται έναν επισκέπτη, αλλά και για να τον προωθεί στους κόμβους

παιδιά της.

```
public abstract class Employee
{
    protected string name;
    public abstract void Accept(IEmployeeVisitor visitor);
}

public class Developer : Employee
{
    public Developer(string name)
    {
        this.name = name;
    }

    public override void Accept(IEmployeeVisitor visitor)
    {
        visitor.VisitDeveloper(this);
    }

    public void WriteCode()
    {
        Console.WriteLine($"{name} is writing code.");
    }
}

public class Designer : Employee
{
    public Designer(string name)
    {
        this.name = name;
    }

    public override void Accept(IEmployeeVisitor visitor)
    {
        visitor.VisitDesigner(this);
    }

    public void Design()
    {
        Console.WriteLine($"{name} is designing.");
```

```
40              }
41          }
42
43          public class Department : Employee
44          {
45              private List<Employee> _employees = new List<Employee>();
46
47              public Department(string name)
48              {
49                  this.name = name;
50              }
51
52              public string GetName()
53              {
54                  return name;
55              }
56
57              public override void Accept(IEmployeeVisitor visitor)
58              {
59                  visitor.VisitDepartment(this);
60                  foreach (var employee in _employees)
61                  {
62                      employee.Accept(visitor);
63                  }
64              }
65
66              public void AddEmployee(Employee employee)
67              {
68                  _employees.Add(employee);
69              }
70          }
```

Το Visitor μοτίβο υλοποιείται μέσω της διεπαφής IEmployeeVisitor και της συμπαγούς της κλάσης DepartmentVisitor. Η διεπαφή ορίζει μια μέθοδο επισκέπτη για κάθε συμπαγή κλάση της οποίας αντικείμενο μπορεί να είναι κάποιο στοιχείο της δομής. Ο σχεδιασμός αυτός επιτρέπει την προσθήκη νέων λειτουργιών χωρίς να τροποποιούνται τα στοιχεία της δομής. Η DepartmentVisitor παρέχει προσαρμοσμένη λογική για κάποιο είδος επίσκεψης και ενσωματώνει τις λειτουργίες που πρέπει να εκτελούνται σε κάθε τύπο στοιχείου, διαχωρίζοντας έτσι τη λογική των λειτουργιών από αυτή της δομής.

```
public interface IEmployeeVisitor
{
    void VisitDeveloper(Developer developer);
    void VisitDesigner(Designer designer);
    void VisitDepartment(Department department);
}

public class DepartmentVisitor : IEmployeeVisitor
{
    public void VisitDeveloper(Developer developer)
    {
        developer.WriteCode();
    }

    public void VisitDesigner(Designer designer) {
        designer.Design();
    }

    public void VisitDepartment(Department department) {
        Console.WriteLine($"Evaluating department: {department.GetName()}");
    }
}
```

Ο κώδικας πελάτη λαμβάνει ένα αντικείμενο IEmployee και χρησιμοποιεί τον DepartmentVisitor που παράγει, για να εκτελέσει λειτουργίες στη δομή του.

```
class Program
{
    static void Main()
    {
        // Initialization code
        Department softwareDepartment = new Department("Software Department");
        softwareDepartment.AddEmployee(new Developer("Raj"));
        softwareDepartment.AddEmployee(new Designer("Bob"));

        Department designDepartment = new Department("Design Department");
        designDepartment.AddEmployee(new Designer("Alice"));

        Department company = new Department("Tech Company");
```

```
14                  company.AddEmployee(softwareDepartment);
15                  company.AddEmployee(designDepartment);
16
17                  // Client code
18                  DepartmentVisitor visitor = new DepartmentVisitor();
19                  company.Accept(visitor);
20              }
21          }
```

Στο σημείο αυτό θα γίνει ανάλυση του συνδυασμού στο σενάριο εισαγωγής νέου τύπου κόμβου και εισαγωγής νέου αλγορίθμου επίσκεψης. Στην τρέχουσα κατάσταση υφίστανται τρείς τύποι κόμβων και ένας τύπος επισκέπτη. Έστω n ο αριθμός των διαφορετικών τύπων επισκέπτη. Σε περίπτωση που επιθυμούμε να προσθέσουμε έναν επιπλέον τύπο Employee, θα πρέπει να γίνουν οι κάτωθι τροποποιήσεις:

- Προσθήκη μιας κλάσης Manager που θα κληρονομεί από την Employee και θα υπερκαλύπτει τη μέθοδο Accept.
- Προσθήκη μιας μεθόδου VisitManager(Manager manager) στη διεπαφή IEmployeeVisitor, την οποία θα πρέπει να υλοποιούν όλες οι συμπαγείς της κλάσεις για εκτέλεση λειτουργιών στους κόμβους τύπου Manager.

Άρα, προστίθενται 1 κλάση και n+1 μέθοδοι (O(n) πολυπλοκότητα).

Στην περίπτωση που είναι επιθυμητή η εισαγωγή ενός νέου τύπου επισκέπτη, απαιτούνται οι κάτωθι τροποποιήσεις:

- Εισαγωγή μιας κλάσης EvaluateVisitor που θα υλοποιεί τη διεπαφή IEmployeeVisitor και θα περιλαμβάνει μεθόδους επίσκεψης για κάθε τύπο Employee.

Άρα, προστίθεται 1 κλάση (O(1) πολυπλοκότητα).

Όπως διαπιστώνεται, ο συνδυασμός των Visitor και Composite μοτίβων επιδεικνύει και αυτός ευελιξία και προσαρμοστικότητα. Η εισαγωγή νέων τύπων κόμβων επιφέρει αλλαγές μόνο στους επισκέπτες, χωρίς να επηρεάζεται η υφιστάμενη δομή, και η εισαγωγή νέων τύπων επισκεπτών είναι εξαιρετικά άμεση. Με αυτό τον τρόπο επιτυγχάνεται καθαρός διαχωρισμός μεταξύ της δομής των δεδομένων και των λειτουργιών που εκτελούνται στα στοιχεία

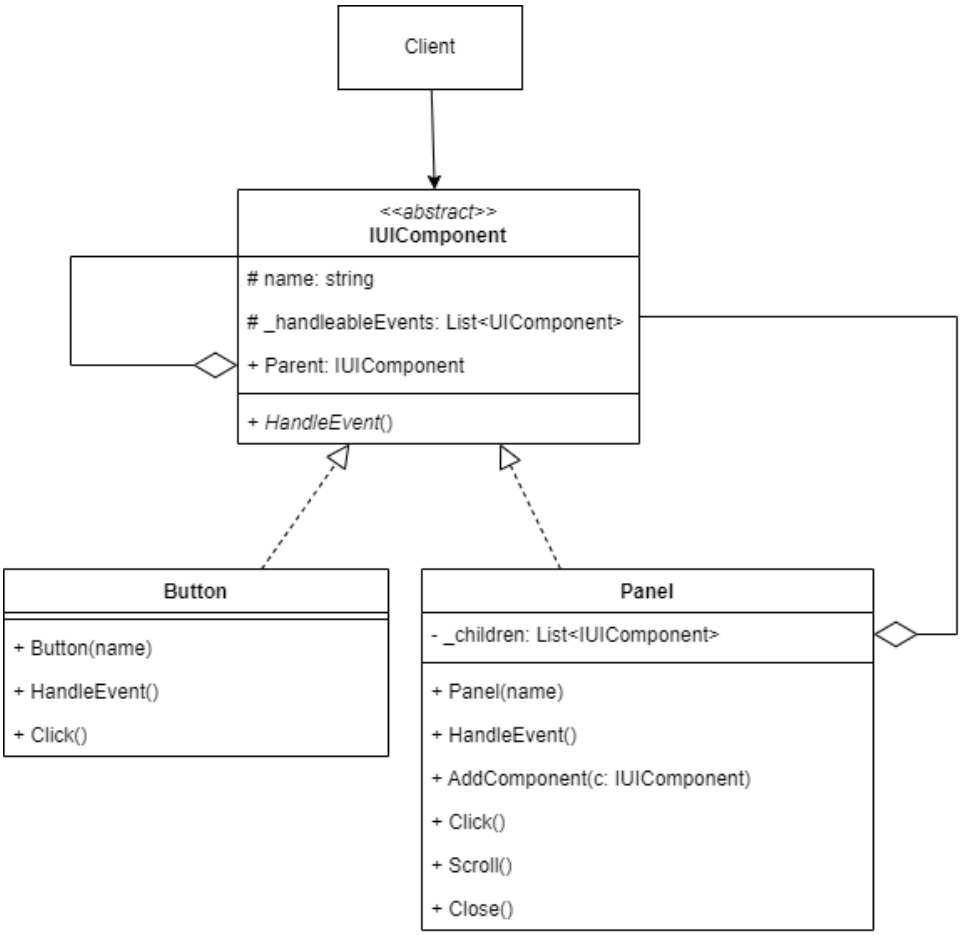
της. Ωστόσο, ελλοχεύει ο κίνδυνος το Visitor μοτίβο να εκθέτει μέρος της εσωτερικής κατάστασης των στοιχείων στα οποία εκτελεί λειτουργίες, οδηγώντας έτσι σε παραβίαση της ενθυλάκωσης. Ακόμα, παρ΄ όλο που η εισαγωγή νέων τύπων κόμβων επιφέρει αλλαγές μόνο στους επισκέπτες, μη επηρεάζοντας την υφιστάμενη δομή του Composite μοτίβου, προκύπτουν ανησυχίες που αφορούν στην κλιμάκωση, σε περίπτωση που συντηρείται μεγάλος αριθμός επισκεπτών και απαιτούνται συχνά εισαγωγές νέων τύπων κόμβων. Επομένως, ο εν λόγω συνδυασμός φαίνεται να αναδεικνύεται σε σενάρια όπου προστίθενται ή αλλάζουν συχνά λειτουργιές που εκτελούνται σε στοιχεία οργανωμένα σε Composite δομές που δεν τροποποιούνται συχνά.

3.6 Chain of Responsibility & Composite

Σε αυτό το υποκεφάλαιο θα διερευνηθεί ο συνδυασμός του Chain of Responsibility και του Composite μοτίβου σε μια ενιαία υλοποίηση. Αυτός ο συνδυασμός επιλέγεται προς υλοποίηση, με σκοπό το χειρισμό συμβάντων σε πολύπλοκες ιεραρχικές δομές αντικειμένων. Εξετάζεται η εισαγωγή ενός μηχανισμού που θα επιτρέπει την επεξεργασία συμβάντων στο εκάστοτε επίπεδο ιεραρχίας που είναι καταλληλότερο να την εκτελέσει. Το Composite μοτίβο θα διαχειρίζεται τη δομή των στοιχείων και το Chain of Responsibility μοτίβο θα επιτρέπει τη μεταβίβαση - προώθηση των συμβάντων κατά μήκος μιας αλυσίδας πιθανών χειριστών έως ότου αυτα υποστούν επεξεργασία, ή στις περιπτώσεις που αυτό δεν δύναται, να απορριφθούν.

Το σενάριο που εξετάζεται, αφορά το χειρισμό συμβάντων σε πολύπλοκες ιεραρχίας στοιχείων UI. Το σύστημα αυτό θα περιλαμβάνει ως κόμβους, κουμπιά και πάνελ που μπορεί να περιλαμβάνουν άλλα πάνελ και κουμπιά μέσα τους.

Ακολουθεί διάγραμμα κλάσης και υλοποίηση σε κώδικα C# για το εν λόγω σενάριο:



Σχημα 3.6: Διάγραμμα Κλάσης Chain of Responsibility & Composite

Η αρχιτεκτονική του μοτίβου έχει σχεδιαστεί γύρω από την αφηρημένη κλάση UIComponent η οποία λειτουργεί ως η βάση για όλα τα στοιχεία του UI. Η αφηρημένη αυτή κλάση καθιερώνει μια ενιαία διεπαφή για τα εν λόγω στοιχεία και ενσωματώνει κοινές λειτουργίες και σχέσεις, όπως χειρισμό συμβάντων και αναφορά προς το εκάστοτε γονικό αντικείμενο. Αυτή η θεμελιώδης δόμηση της, επιτρέπει στο Chain of Responsibility μοτίβο να μεταβιβάζει συμβάντα που δεν δύνανται να υποστούν επεξεργασία από τα στοιχεία που τα λαμβάνουν, στα επόμενα στοιχεία της ιεραρχίας (προς τα πάνω κατεύθυνση). Οι συμπαγείς κλάσεις Button και Panel επεκτείνουν την UIComponent και παρέχουν προσαρμοσμένες υλοποιήσεις για τη μέθοδο HandleEvent. Η κλάση Button επικεντρώνεται στο χειρισμό συμβάντων κλίκ, ενώ η Panel μπορεί να χειριστεί και συμβάντα κύλισης και επιλογής κλεισίματός του, σε συνδυασμό με τη διαχείριση θυγατρικών για αυτή στοιχείων. Η μέθοδος HandleEvent υλοποιείται έτσι ώστε να ελέγχει αν ένα συμβάν είναι μέσα στα επιτρεπτά συμβάντα που μπορεί να χειριστεί η εκάστοτε συμπαγής κλάση. Άν ανήκει σε αυτά, το χειρίζεται. Αλλιώς, το μεταβιβάσει στον γονέα του, εφόσον αυτός υπάρχει.

```
public abstract class UIComponent
{
    protected string name;
    protected List<string> _handleableEvents = new List<string> { "Click" };
    public UIComponent Parent { get; set; }

    public UIComponent(string name)
    {
        this.name = name;
    }

    public virtual void HandleEvent(string uiEvent)
    {
        Parent.HandleEvent(uiEvent);
    }
}

public class Button : UIComponent
{
    public Button(string name) : base(name) { }

    public override void HandleEvent(string uiEvent)
    {
        if (uiEvent == "Click")
        {
            Console.WriteLine($"Event handled by {name}");
            Click();
        }
        else if (Parent != null)
            base.HandleEvent(uiEvent);

        else
            Console.WriteLine($"{name}: Event can't be handled.");
    }

    private void Click()
    {
        Console.WriteLine($"{name}: Clicked.");
    }
}
```

```
41
42          public class Panel : UIComponent
43          {
44              private List<UIComponent> _children = new List<UIComponent>();
45
46              public Panel(string name) : base(name)
47              {
48                  _handleableEvents.AddRange(new string[] { "Scroll", "Close" });
49              }
50
51              public void AddComponent(UIComponent component)
52              {
53                  _children.Add(component);
54                  component.Parent = this;
55              }
56
57              public override void HandleEvent(string uiEvent)
58              {
59                  if (_handleableEvents.Contains(uiEvent))
60                  {
61                      Console.WriteLine($"Event handled by {name}");
62                      switch (uiEvent)
63                      {
64                          case "Click":
65                              Click();
66                              break;
67                          case "Scroll":
68                              Scroll();
69                              break;
70                          case "Close":
71                              Close();
72                              break;
73                      }
74                  }
75                  else if (Parent != null)
76                      base.HandleEvent(uiEvent);
77
78                  else
79                      Console.WriteLine($"{name}: Event can't be handled.");
80              }
81
```

```
82              private void Click()
83              {
84                  Console.WriteLine($"{name}: Clicked.");
85              }
86
87              private void Scroll()
88              {
89                  Console.WriteLine($"{name}: Scrolling.");
90              }
91
92              private void Close()
93              {
94                  Console.WriteLine($"{name}: Closed.");
95              }
96          }
```

Ο κώδικας πελάτη λαμβάνει αντικείμενα UIComponent και καλεί τη μέθοδο HandleEvent τους, με όρισμα κάποιο συμβάν.

```
1           public class Program
2           {
3               public static void Main()
4               {
5                   // Initialization code
6                   Panel mainPanel = new Panel("Main Panel");
7                   Panel subPanel = new Panel("Sub Panel");
8                   Button button1 = new Button("Button 1");
9                   Button button2 = new Button("Button 2");
10
11                  mainPanel.AddComponent(subPanel);
12                  subPanel.AddComponent(button1);
13                  mainPanel.AddComponent(button2);
14
15                  // Client code
16                  button1.HandleEvent("Scroll");
17                  button2.HandleEvent("Click");
18                  button1.HandleEvent("Resize");
19              }
20          }
```

Στο σημείο αυτό θα γίνει αξιολόγηση του συνδυασμού με μια απλή μετρική που θα εξετάσει την ευκολία επέκτασης του συτήματος. Η μετρική θα επικεντρωθεί σε δύο σενάρια: στην εισαγωγή νέου τύπου UIComponent και στην εισαγωγή νέου τύπου συμβάντος. Έστω n ο αριθμός των τύπων UI στοιχείων. Σε περίπτωση που επιθυμούμε να προσθέσουμε έναν επιπλέον τύπο UIComponent, θα πρέπει να γίνουν οι κάτωθι τροποποιήσεις:

- Προσθήκη μιας κλάσης TextField που θα επεκτείνει την UIComponent, θα υπερκαλύπτει τη μέθοδο HandleEvent και θα διαθέτει επιπλέον είδη συμβάντων που θα μπορεί να διαχειριστεί, όπως "Typing".

Άρα, προστίθεται 1 κλάση (O(1) πολυπλοκότητα).

Στην περίπτωση που είναι επιθυμητή η εισαγωγή ενός νέου τύπου συμβάντος, όπως το "Hover" απαιτούνται οι κάτωθι τροποποιήσεις:

- Τροποποίηση των συμπαγών UI στοιχείων που θα πρέπει να μπορούν να το χειριστούν, μέσω προσθήκης του τύπου αυτού στα _handleableEvents τους (αν είναι καθολικό το συμβάν για όλα τα στοιχεία, μπορεί να προστεθεί στην UIComponent).
- Προσθήκη Hover μεθόδου σε όλα τα στοιχεία που θα πρέπει να μπορούν να το χειριστούν το εν λόγω συμβάν.
- Τροποποίησης των συνθηκών ελέγχου της εκάστοτε μεθόδου HandleEvent και κλήσης της εκάστοτε προσαρμοσμένης υλοποίησης της μεθόδου Hover στην πυροδότηση του "Hover" συμβάντος.

Έστω m: {1,n} ο αριθμός των στοιχείων που θα πρέπει να μπορούν να το χειριστούν το εν λόγω συμβάν. Τότε, εισάγονται m τροποποιήσεις κλάσεων, μ κλάσεις (O(m) πολυπλοκότητα).

Όπως διαπιστώνεται, ο συνδυασμός των Chain of Responsibility και Composite μοτίβων παρουσιάζει έναν αποδοτικό μηχανισμό σχεδιασμού ιεραρχιών από UI στοιχεία, ικανών να χειρίζονται διαφορετικά συμβάντα. Το παραχθέν σύστημα φαίνεται να παρέχει υψηλή επεκτασιμότητα, καθώς επιτρέπει την εισαγωγή νέων τύπων στοιχείων και δυνατοτήτων χειρισμού συμβάντων, χωρίς να απαιτεί την εφαρμογή σημαντικών τροποποιήσεων στον κώδικα. Τα νέα στοιχεία απαιτούν μόνο την υλοποίηση των αντίστοιχων κλάσεων, και η εισαγωγή νέων τύπων συμβάντων επηρεάζει μόνο τα άμεσα επηρεαζόμενα στοιχεία. Ωστόσο, παρ΄ ότι

ο εξεταζόμεμος συνδυασμός προσφέρει πλεονεκτήματα, σε περιπτώσεις ύπαρξης βαθιών ιεραρχιών στοιχείων ίσως δυχεραίνεται η παρακολούθηση των διαδρομών διάδοσης συμβάντων και ενδεχομένως να υπάρχουν επιπτώσεις στην επίδοση.

3.7 Chain of Responsibility & Command

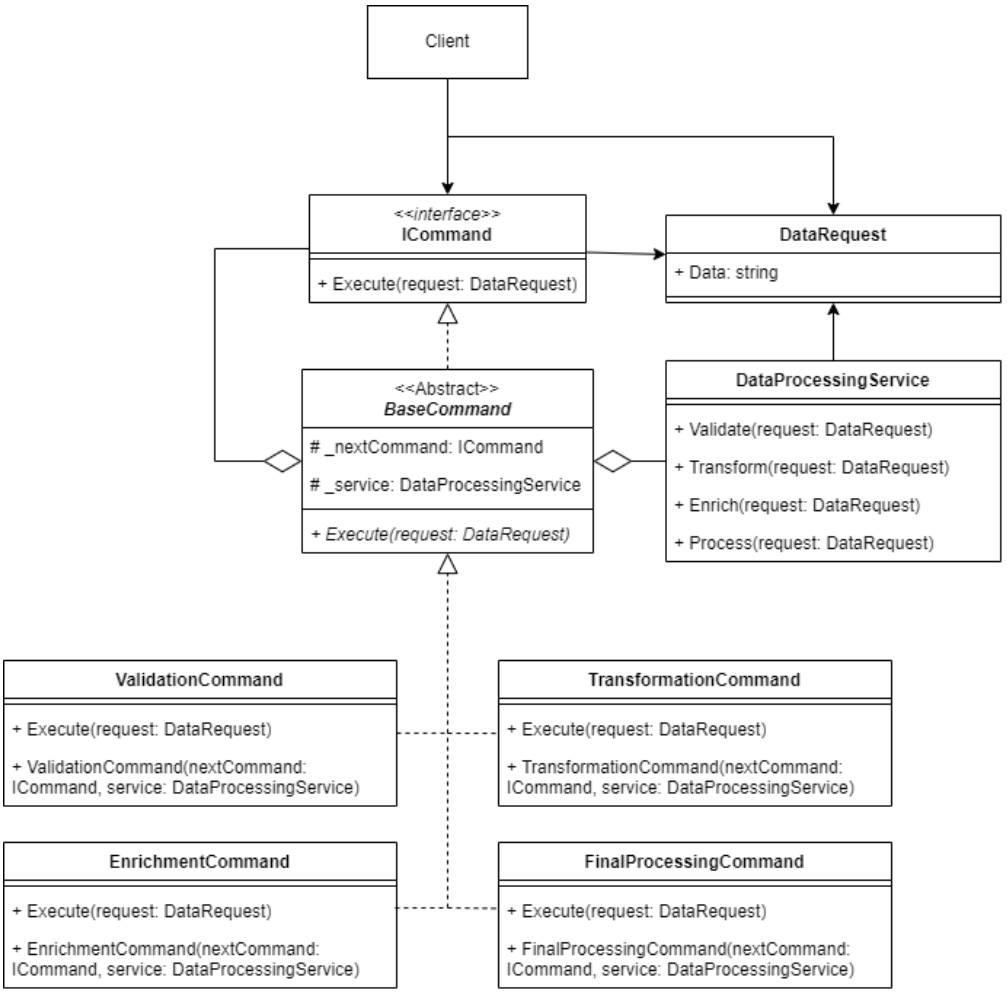
Σε αυτό το υποκεφάλαιο θα διερευνηθεί ο συνδυασμός του Chain of Responsibility και του Command μοτίβου σε μια ενιαία υλοποίηση. Πιο συγκεκριμένα, θα εξεταστούν δύο διαφορετικοί τρόποι συνδυασμού αυτών των μοτίβων. Στον πρώτο, τα Command αντικείμενα θα αντιπροσωπεύουν τους handlers και θα συνθέτουν μια αλυσίδα, κατά την οποία θα εκτελούνται λειτουργίες σε δεδομένα, με κάποια σειρά. Στον δεύτερο, τα Command αντικείμενα θα αντιπροσωπεύουν τα αιτήματα που θα πηγαίνουν στους χειριστές της αλυσίδας προς εκτέλεση και θα περιλαμβάνουν όλες τις απαραίτητες πληροφορίες εκτέλεσης.

3.7.1 Commands ως χειριστές αλυσίδας

Αυτός ο συνδυασμός επιλέγεται προς υλοποίηση, με σκοπό τη δημιουργία ενός δομημένου μηχανισμού επεξεργασίας δεδομένων, ο οποίος θα αποτελείται από μια ακολουθία διαφορετικών λειτουργιών εκτέλεσης. Η δομή αυτής της υλοποίησης θα περιλαμβάνει αντικείμενα τύπου Command, οργανωμένα σε μια ακολουθία μονής σύνδεσης, τα οποία θα λαμβάνουν αιτήματα, θα τα επεξεργάζονται και θα τα προωθούν - διαβιβάζουν στον επόμενο Command - χειριστή της αλυσίδας.

Το σενάριο που εξετάζεται, αφορά τη δημιουργία μιας pipeline, κατά την οποία κάθε χειριστής θα αντιπροσωπεύει ένα διακριτό στάδιο επεξεργασίας δεδομένων, όπως επικύρωση δεδομένων, μετατροπής τους σε κατάλληλη μορφή, εμπλουτισμός με επιπλέον πληροφορία και τελική επεξεργασία. Όπως και στο Chain of Responsibility μοτίβο, η ύπαρξη κάποιου χειριστή στην ακολουθία είναι προαιρετική, και η σειρά οργάνωσής τους μεταβλητή.

Ακολουθεί διάγραμμα κλάσης και υλοποίηση σε κώδικα C# για το εν λόγω σενάριο:



Σχημα 3.7: Διάγραμμα Κλάσης Chain of Responsibility & Command (χειριστές ως αντικείμενα Command

Η αρχιτεκτονική του μοτίβου έχει σχεδιαστεί γύρω από τη διεπαφή ICommand και την αφηρημένη κλάση BaseCommand που την επεκτείνει και λειτουργεί ως η βάση για όλα τα στοιχεία Command. Η αφηρημένη αυτή κλάση, αποτελεί μια ενιαία διεπαφή για τα εν λόγω στοιχεία και ενσωματώνει κοινές λειτουργίες και σχέσεις, όπως εκτέλεση λειτουργιών, και αναφορά προς τον επόμενο χειριστή της αλυσίδας και προς μια υπηρεσία επεξεργασίας δεδομένων. Αυτή η θεμελιώδης δόμησή της, επιτρέπει την υλοποίηση του Chain of Responsibility μοτίβου και κατ΄ επέκταση του μηχανισμού μεταβίβασης αιτημάτων κατά μήκος της αλυσίδας.

```
1          public interface ICommand
2          {
3              void Execute(DataRequest request);
4          }
```

```
5
6           public abstract class BaseCommand : ICommand
7           {
8               protected ICommand _nextCommand;
9               protected DataProcessingService _service;
10
11              public virtual void Execute(DataRequest request)
12              {
13                  _nextCommand?.Execute(request);
14              }
15          }
```

Οι συμπαγείς κλάσεις ValidationCommand, TransformationCommand, EnrichmentCommand και FinalProcessingCommand, επεκτείνουν την BaseCommand και ειδικεύονται σε μια συγκεκριμένη λειτουργία, μέσω της υπερκάλυψης της μεθόδου Execute, η οποία εκτελεί κάποια λειτουργία και έπειτα μεταβιβάζει το αίτημα στον επόμενο χειριστή - Command της αλυσίδας.

```
1           public class ValidationCommand : BaseCommand
2           {
3               public ValidationCommand(ICommand nextCommand, DataProcessingService
                    service)
4               {
5                   _nextCommand = nextCommand;
6                   _service = service;
7               }
8
9               public override void Execute(DataRequest request)
10              {
11                  _service.Validate(request);
12                  base.Execute(request);
13              }
14          }
15
16          public class TransformationCommand : BaseCommand
17          {
18              public TransformationCommand(ICommand nextCommand, DataProcessingService
                    service)
19              {
20                  _service = service;
```

```
21                  _nextCommand = nextCommand;
22              }
23
24              public override void Execute(DataRequest request)
25              {
26                  _service.Transform(request);
27                  base.Execute(request);
28              }
29          }
30
31          public class EnrichmentCommand : BaseCommand
32          {
33              public EnrichmentCommand(ICommand nextCommand, DataProcessingService
                    service)
34              {
35                  _nextCommand = nextCommand;
36                  _service = service;
37              }
38
39              public override void Execute(DataRequest request)
40              {
41                  _service.Enrich(request);
42                  base.Execute(request);
43              }
44          }
45
46          public class FinalProcessingCommand : BaseCommand
47          {
48              public FinalProcessingCommand(ICommand nextCommand, DataProcessingService
                    service)
49              {
50                  _service = service;
51              }
52
53              public override void Execute(DataRequest request)
54              {
55                  _service.Process(request);
56                  base.Execute(request);
57              }
58          }
```

Η κλάση DataRequest αντιπροσωπεύει τα δεδομένα που υπόκεινται σε επεξεργασία από την αλυσίδα, και λειτουργεί ως το αντικείμενο των μετασχηματισμών και των λειτουργιών που εφαρμόζονται. Επιπλέον, η κλάση DataProcessingService παρέχει τις πραγματικές υλοποιήσεις των λειτουργιών που εκτελούνται στα δεδομένα. Κάθε ICommand αντικείμενο καλεί μια μέθοδο αυτής της υπηρεσίας για να εκτελέσει την αντίστοιχη λειτουργία.

```
public class DataRequest
{
    public string Data { get; set; }
}
public class DataProcessingService
{
    public void Validate(DataRequest request)
    {
        Console.WriteLine($"Service: Validating {request.Data}.");
        request.Data = $"validated {request.Data}";
    }

    public void Transform(DataRequest request)
    {
        Console.WriteLine($"Service: Transforming {request.Data}.");
        request.Data = $"transformed {request.Data}";
    }

    public void Enrich(DataRequest request)
    {
        Console.WriteLine($"Service: Enriching {request.Data}.");
        request.Data = $"enriched {request.Data}";
    }

    public void Process(DataRequest request)
    {
        Console.WriteLine($"Service: Final processing of {request.Data}.");
    }
}
```

Ο κώδικας πελάτη λαμβάνει κάποιο αντικείμενο ICommand και καλεί τη μέθοδο Execute του, με όρισμα κάποιο αίτημα που ο ίδιος αρχικοποιεί.

```
public class Program
{
    public static void Main()
    {
        // Initialization code
        var service = new DataProcessingService();

        ICommand finalProcessing = new FinalProcessingCommand(null, service);
        ICommand enrichment = new EnrichmentCommand(finalProcessing, service);
        ICommand transformation = new TransformationCommand(enrichment,
            service);
        ICommand validation = new ValidationCommand(transformation, service);

        // Client code
        var request = new DataRequest { Data = "Sample Data" };
        validation.Execute(request);
    }
}
```

Στο σημείο αυτό θα γίνει αξιολόγηση του συνδυασμού με μια απλή μετρική που θα εξετάσει την προσαρμοστικότητα και τη διατηρησιμότητα αυτής της αρχιτεκτονικής. Η μετρική θα επικεντρωθεί σε δύο σενάρια: στην εισαγωγή νέου τύπου ICommand και στην επεξεργασία της συμπεριφοράς ενός υφιστάμενου Command. Σε περίπτωση που επιθυμούμε να προσθέσουμε έναν επιπλέον τύπο ICommand, θα πρέπει να γίνουν οι κάτωθι τροποποιήσεις:

- Προσθήκη μιας κλάσης DataCleansing που θα επεκτείνει την ICommand και θα υπερκαλύπτει τη μέθοδο Execute.
- Κατάλληλη τροποποίηση του κώδικα αρχικοποίησης, ώστε η κλάση αυτή να λάβει την κατάλληλη θέση στην αλυσίδα.

Άρα, προστίθεται 1 κλάση (O(1) πολυπλοκότητα).

Στην περίπτωση που είναι επιθυμητή η τροποποίηση της μεθόδου Execute κάποιας κλάσης που επεκτείνει την ICommand, η αλλαγή αυτή γίνεται τοπικά και δεν επηρεάζει καθόλου ούτε την δομή της αλυσίδας, αλλά ούτε κάποια άλλη Command κλάση.

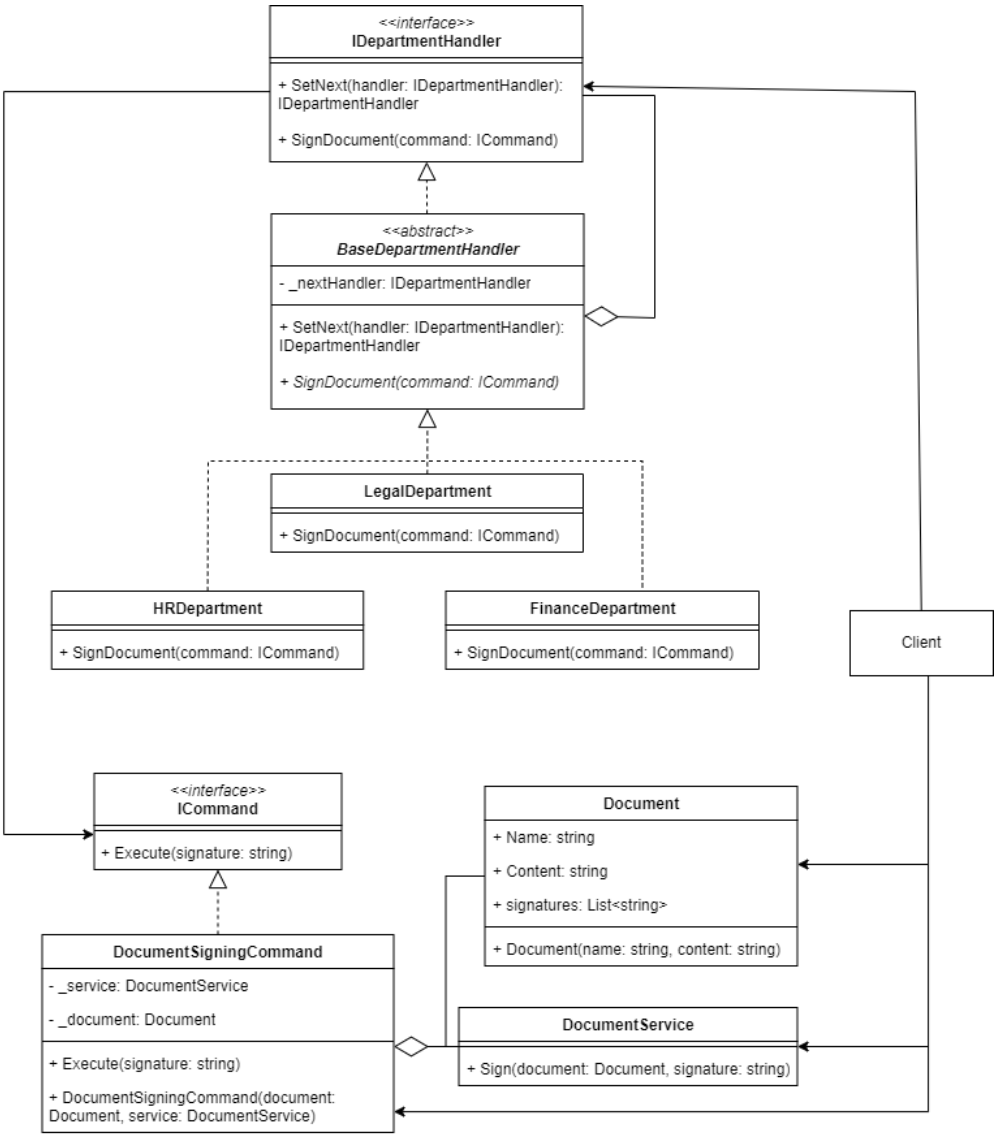
Όπως διαπιστώνεται, ο συνδυασμός των Chain of Responsibility και Command μοτίβων παρουσιάζει υψηλή προσαρμοστικότητα και συντηρησιμότητα σε σενάρια δημιουργίας pipeline. Αυτή η προσέγγιση επιτρέπει τον καθαρό διαχωρισμό ανησυχιών, κατά τον οποίο κάθε στάδιο επεξεργασίας επικεντρώνεται σε μια συγκεκριμένη λειτουργία, και η αλυσίδα διαχειρίζεται τη γενικότερη ροή της επεξεργασίας. Η εισαγωγή νέων σταδίων επεξεργασίας είναι εξαιρετικά απλή και η τροποποίηση των υφιστάμενων εντολών δεν έχει άμεσο αντίκτυπο σε άλλα στοιχεία του συστήματος. Επίσης, καθίσταται δυνατή η δυναμική αλλαγή της σειράς των σταδίων επεξεργασίας και η προσθαφαίρεση αυτών σε μια αλυσίδα. Η διαφαινόμενη ανεξαρτησία των στάδιων επεξεργασίας διευκολύνει και την testing διαδικασία. Αυτό που μπορεί να εγείρει κάποιους προβληματισμούς, είναι η επιβάρυνση στις επιδόσεις, από τη μετάβαση αντικειμένων - αιτημάτων από πολλαπλά αντικείμενα ICommand.

3.7.2 Commands ως αιτήματα αλυσίδας

Αυτός ο συνδυασμός επιλέγεται προς υλοποίηση, με σκοπό τη δημιουργία ενός μηχανισμού διαχείρισης λειτουργιών - αιτημάτων. Η τρέχουσα προσέγγιση ενθυλακώνει μια λειτουργία - αίτημα σε ένα αντικείμενο Command, το οποίο έπειτα μεταβιβάζει σε μια αλυσίδα χειριστών, καθένας εκ των οποίων αντιπροσωπεύει διαφορετικό πλαισίο επεξεργασίας. Κάθε χειριστής, θα λαμβάνει το αίτημα, θα το επεξεργάζεται και θα το προωθεί - διαβιβάζει στον επόμενο χειριστή της αλυσίδας.

Το σενάριο που εξετάζεται αφορά τη δημιουργία ενός μηχανισμού διαχείρισης της ροής υπογραφής εγγράφων σε έναν οργανισμό. Τα επιμέρους τμήματα (υπογράφοντες) θα αναπαριστούν τους χειριστές της αλυσίδας ευθύνης, και η λειτουργία της υπογραφής ενός εγγράφου θα ενθυλακώνεται μέσα σε μια εντολή, η οποία θα μεταβιβάζεται στο πρώτο τμήμα και μέσω της αλυσίδας και στα επόμενα. Σύμφωνα με το Chain of Responsibility μοτίβο, επιτρέπεται η δυναμική τροποποίηση της ακολουθίας επεξεργασίας και η ενσωμάτωση νέων βημάτων επεξεργασίας ή αφαίρεση τους.

Ακολουθεί διάγραμμα κλάσης και υλοποίηση σε κώδικα C# για το εν λόγω σενάριο:



Σχημα 3.8: Διάγραμμα Κλάσης Chain of Responsibility & Command (Command αντικείμενα ως αιτήματα αλυσίδας

Η αρχιτεκτονική του μοτίβου έχει σχεδιαστεί γύρω από τη λογική της αντιμετώπισης κάθε αιτήματος, όπως είναι η ανάγκη υπογραφής ενός εγγράφου, ως ξεχωριστό αντικείμενο ICommand. Κάθε τέτοιο αντικείμενο περικλείει όλες τις πληροφοριές και τις λειτουργίες που απαιτούνται για την διεκπεραίωση του αιτήματος. Η διεπαφή ICommand ορίζει τη μέθοδο Execute που δηλώνει τη μέθοδο εκτέλεσης του αιτήματος. Η κλάση DocumentSigningCommand που επεκτείνει την εν λόγω διεπαφή, ενσωματώνει τη λογική υπογραφής εγγράφων, και διατηρεί αναφορά προς ένα αντικείμενο τύπου Document και προς ένα αντικείμενο τύπου DocumentService.

```
1         public interface ICommand
2         {
```

```
3               void Execute(string signature);
4           }
5
6           public class DocumentSigningCommand : ICommand
7           {
8               private DocumentService _service;
9               private Document _document;
10
11              public DocumentSigningCommand(Document document, DocumentService service)
12              {
13                  _service = service;
14                  _document = document;
15              }
16
17              public void Execute(string signature)
18              {
19                  _service.Sign(_document, signature);
20              }
21          }
```

Η κλάση Document αντιπροσωπεύει τα έγγραφα που διακινούνται προς υπογραφή στην αλυσίδα χειριστών - τμημάτων, και διατηρεί πληροφορίες όπως το όνομα το εγγράφου, το περιεχόμενό του και μια λίστα με τις υπογραφές που αυτό έχει λάβει.

```
1           public class Document
2           {
3               public string Name { get; set; }
4               public string Content { get; set; }
5               public List<string> signatures = new List<string>();
6
7               public Document(string name, string content)
8               {
9                   Name = name;
10                  Content = content;
11              }
12          }
```

Η κλάση DocumentService αποτελεί μια υπηρεσία που παρέχει την πραγματική υλοποίηση της λειτουργίας υπογραφής των εγγράφων. Κάθε αντικείμενο DocumentSigningCommand

καλεί τη Sign μέθοδό της, όποτε χρειάζεται να εκτελέσει αυτή την ενέργεια.

```
public class DocumentService
{
    public void Sign(Document document, string signature)
    {
        document.signatures.Add(signature);
        Console.WriteLine($"Service: Added {signature} to document {document.
            Name}.");
    }
}
```

Στο Chain of Responsibility μοτίβο, εισάγονται η διεπαφή IDepartmentHandler και η αφηρημένη κλάση BaseDepartmentHandler που την επεκτείνει και λειτουργεί ως η βάση για όλες τις κλάσεις που θα επιτελούν το ρόλο του χειριστή (όπως είναι οι HRDepartment, FinanceDepartment και LegalDepartment), καθεμία με τη δική της προσαρμοσμένη υλοποίηση. Η αφηρημένη αυτή κλάση, ορίζει μια ενιαία διεπαφή για τα εν λόγω στοιχεία και ενσωματώνει κοινές σχέσεις και λειτουργίες, όπως η αναφορά προς τον επόμενο χειριστή της αλυσίδας, η μέθοδος ορισμού εκείνου, και η εκτέλεση της λειτουργίας της υπογραφής.

```
public interface IDepartmentHandler
{
    public IDepartmentHandler SetNext(IDepartmentHandler handler);
    public void SignDocument(ICommand command);
}

public abstract class BaseDepartmentHandler : IDepartmentHandler
{
    private IDepartmentHandler _nextHandler;

    public IDepartmentHandler SetNext(IDepartmentHandler handler)
    {
        _nextHandler = handler;
        return handler;
    }

    public virtual void SignDocument(ICommand command)
    {
        _nextHandler?.SignDocument(command);
```

```
        }
    }

    public class HRDepartment : BaseDepartmentHandler
    {
        public override void SignDocument(ICommand command)
        {
            command.Execute("HR Department Signature");
            base.SignDocument(command);
        }
    }

    public class FinanceDepartment : BaseDepartmentHandler
    {
        public override void SignDocument(ICommand command)
        {
            command.Execute("Finance Department Signature");
            base.SignDocument(command);
        }
    }

    public class LegalDepartment : BaseDepartmentHandler
    {
        public override void SignDocument(ICommand command)
        {
            command.Execute("Legal Department Signature");
            base.SignDocument(command);
        }
    }
```

Ο κώδικας πελάτη λαμβάνει ένα αντικείμενο της documentService και τον πρώτο IDepartmentHandler της αλυσίδας, και δημιουργεί ένα αντικείμενο Document και ένα DocumentSigningCommand με ορίσματα την υπηρεσία και το έγγραφο. Έπειτα καλεί τη μέθοδο SignDocument του χειριστή, με όρισμα κάποιο αίτημα που αρχικοποιεί.

```
    public class Program
    {
        public static void Main()
        {
            // Initialization code
```

```
6                   var documentService = new DocumentService();
7                   var hr = new HRDepartment();
8                   var finance = new FinanceDepartment();
9                   var legal = new LegalDepartment();
10
11                  hr.SetNext(finance).SetNext(legal);
12
13                  // Client code
14                  var document = new Document("Report", "Report content");
15                  var signingCommand = new DocumentSigningCommand(document,
                        documentService);
16
17                  hr.SignDocument(signingCommand);
18              }
19          }
```

Στο σημείο αυτό θα γίνει αξιολόγηση του συνδυασμού με μια απλή μετρική που θα εξετάσει την ευελιξία αυτής της αρχιτεκτονικής. Η μετρική θα επικεντρωθεί σε δύο σενάρια: στην εισαγωγή νέου τύπου ICommand και στην εισαγωγή νέου τύπου IDepartmentHandler. Έστω n ο αριθμός των χειριστών της αλυσίδας. Σε περίπτωση που επιθυμούμε να προσθέσουμε έναν επιπλέον τύπο ICommand, θα πρέπει να γίνουν οι κάτωθι τροποποιήσεις:

- Προσθήκη μιας κλάσης DocumentReviewCommand που θα επεκτείνει την ICommand και θα υπερκαλύπτει τη μέθοδο Execute.
- Προσθήκη μεθόδου ReviewDocument στη διεπαφή IDepartmentHandler, στην αφηρημένη κλάση BaseDepartmentHandler και σε κάθε προσαρμοσμένη υλοποίησή της.
- Προσθήκη μεθόδου Review στην κλάση DocumentService

Άρα, προστίθενται 1 κλάση και n+3 μέθοδοι (O(n) πολυπλοκότητα).

Στην περίπτωση που είναι επιθυμητή η προσθήκη ενός νέου τύπου IDepartmentHandler, θα πρέπει να γίνουν οι κάτωθι τροποποιήσεις:

- Προσθήκη μιας κλάσης QualityAssuranceDepartment που θα επεκτείνει την BaseDepartmentHandler και θα υπερκαλύπτει τη μέθοδο SignDocument.
- Κατάλληλη τροποποίηση του κώδικα αρχικοποίησης, ώστε η κλάση αυτή να λάβει την κατάλληλη θέση στην αλυσίδα.

Άρα, προστίθεται 1 κλάση (O(1) πολυπλοκότητα).

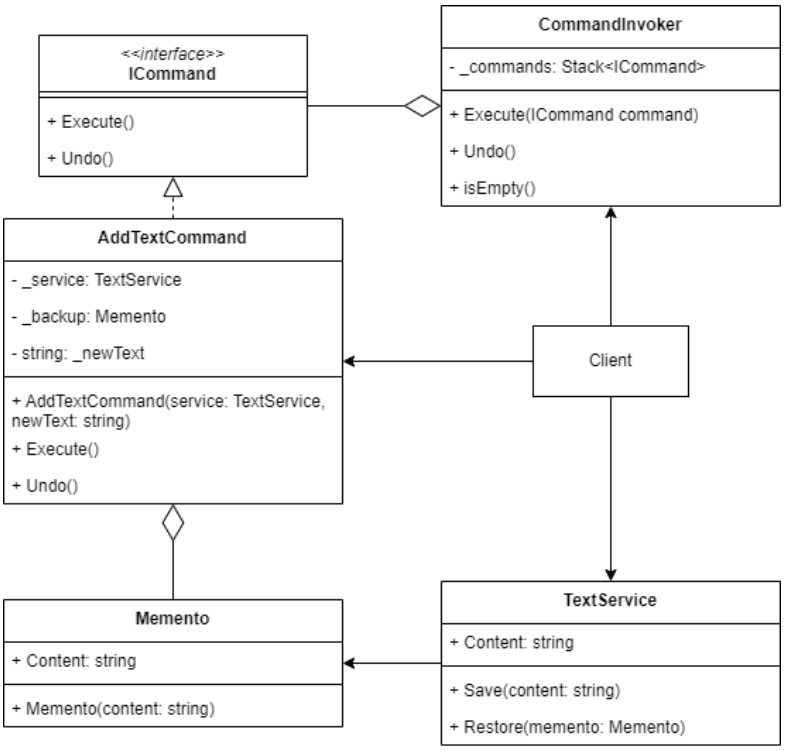
Όπως διαπιστώνεται, ο συνδυασμός των Chain of Responsibility και Command μοτίβων παρουσιάζει υψηλή προσαρμοστικότητα και συντηρησιμότητα σε σενάρια εκτέλεσης της ίδιας λειτουργίας σε μια σειρά διαφορετικών περιβάλλοντων - πλαισίων επεξεργασίας, τα οποία συνδέονται σε μια αλυσίδα. Αυτή η προσέγγιση επιτρέπει τον καθαρό διαχωρισμό ανησυχιών, κατά τον οποίο η επεξεργαστική λογική ενθυλακώνεται στα Command αντικείμενα και αποσυνδέεται από τους χειριστές της αλυσίδας. Η εισαγωγή νέων χειριστών - πλαισίων επεξεργασίας είναι εξαιρετικά απλή, και η εισαγωγή νέων τύπων εντολών επηρεάζει τοπικά μόνο τα άμεσα επηρεαζόμενα μέρη, όπως είναι οι χειριστές και η υπηρεσία που εκτελεί πρακτικά την εκάστοτε λειτουργία. Επίσης, καθίσταται δυνατή η δυναμική αλλαγή της σειράς των σταδίων επεξεργασίας και η προσθαφαίρεση αυτών σε μια αλυσίδα.

3.8 Command & Memento

Σε αυτό το υποκεφάλαιο θα εξεταστεί ο συνδυασμός του Command και του Memento μοτίβου σε μια ενιαία υλοποίηση. Αυτός ο συνδυασμός επιλέγεται με σκοπό την υλοποίηση ανακλητών λειτουργιών σε εφαρμογές. Το Command μοτίβο θα ενθυλακώνει όλες τις απαραίτητες λεπτομέρειες που αφορούν κάθε λειτουργία, και το Memento θα εξασφαλίζει την αναστρεψιμότητα των λειτουργιών αυτών.

Το σενάριο που εξετάζεται, αφορά τη κατασκευή ενός μηχανισμού λήψης στιγμιοτύπων της κατάστασης κάποιου εγγράφου, πριν από κάθε προσθήκη κειμένου σε αυτό, και επαναφοράς τους, σε περίπτωση που απαιτηθεί από τον κώδικα πελάτη.

Ακολουθεί διάγραμμα κλάσης και υλοποίηση σε κώδικα C# για το εν λόγω σενάριο:



Σχημα 3.9: Διάγραμμα Κλάσης Command & Memento

Στο Command μοτίβο, η διεπαφή ICommand ορίζει τη συμπεριφορά που πρέπει να έχει κάθε εντολή, δηλαδή εκτέλεση κάποιας λειτουργίας και αναίρεση αυτής. Η κλάση AddTextCommand επεκτείνει την ανωτέρω διεπαφή και αποτελεί μια προσαρμοσμένη κλάση, της οποίας λειτουργία είναι η προσθήκη κειμένου σε ένα έγγραφο. Επίσης, φροντίζει να κρατά αντίγραφο της κατάστασης του κειμένου, πριν από κάθε τροποποίηση, διασφαλίζοντας την αναστρεψιμότητα της ενέργειας.

```
1           public interface ICommand
2           {
3               public void Execute();
4               public void Undo();
5           }
6
7           public class AddTextCommand : ICommand
8           {
9               private TextService _service;
10              private string _newText;
11              private Memento _backup;
12
13              public AddTextCommand(TextService service, string newText)
```

```
14              {
15                  _service = service;
16                  _newText = newText;
17              }
18
19              public void Execute()
20              {
21                  _backup = _service.Save();
22                  _service.Content += _newText;
23              }
24
25              public void Undo()
26              {
27                  _service.Restore(_backup);
28              }
29          }
```

Η κλάση CommandInvoker έχει διπλό ρόλο. Πέρα από την κλήση των Commands, τα διατηρεί επίσης με τη σειρά εκτέλεσης τους σε μια στοίβα, και επαναφέρει το τελευταίο σε κάθε επιλογή της λειτουργίας αναίρεσης. Αυτός ο διπλός ρόλος Invoker για το Command μοτίβο και Caretaker για το Memento μοτίβο, είναι ζωτικής σημασίας για την παρακολούθηση του ιστορικού των εντολών και τη διευκόλυνση των λειτουργιών αναίρεσης.

```
1           public class CommandInvoker
2           {
3               private Stack<ICommand> _commands = new Stack<ICommand>();
4
5               public void Execute(ICommand command)
6               {
7                   command.Execute();
8                   _commands.Push(command);
9               }
10
11              public void Undo()
12              {
13                  if (_commands.Count > 0)
14                  {
15                      var command = _commands.Pop();
16                      command.Undo();
17                  }
```

```
        }

        public bool IsEmpty()
        {
            return _commands.Count == 0;
        }
    }
```

Στο Memento μοτίβο, η ομώνυμη κλάση δημιουργεί στιγμιότυπα της κατάστασης των αντικειμένων της κλάσης TextService, διατηρώντας το περιεχόμενό τους, πριν από την εκτέλεση κάθε εντολής. Η κλάση TextService έχει το ρόλο του Originator και μπορεί να παράγει στιγμιότυπα της δικής της κατάστασης, καθώς και να επαναφέρει την κατάστασή της από αυτά τα στιγμιότυπα όταν χρειάζεται.

```
    public class Memento
    {
        public string Content { get; private set; }

        public Memento(string content)
        {
            Content = content;
        }
    }

    public class TextService
    {
        public string Content { get; set; }

        public Memento Save()
        {
            return new Memento(Content);
        }

        public void Restore(Memento memento)
        {
            Content = memento.Content;
        }
    }
```

Ο κώδικας πελάτη λαμβάνει ένα αντικείμενο της TextService και της CommandInvoker. Έπειτα μπορεί να δημιουργεί αντικείμενα της AddTextCommand και να τα εκτελεί μέσω του CommandInvoker, καθώς επίσης να τα ανακαλεί.

```
public class Program
{
    public static void Main()
    {
        // Initialization code
        TextService service = new TextService();
        CommandInvoker invoker = new CommandInvoker();

        // Client code
        invoker.Execute(new AddTextCommand(service, "Lorem Ipsum is simply
            dummy text of the printing and typesetting industry."));
        invoker.Execute(new AddTextCommand(service, "Lorem Ipsum has been the
            industry's standard dummy text ever since the 1500s."));

        Console.WriteLine("Content: " + service.Content);

        while (!invoker.IsEmpty())
        {
            invoker.Undo();
            Console.WriteLine("After Undo: " + service.Content);
        };
    }
}
```

Με σκοπό την αξιολόγηση του συνδυασμού ως προς την αποδοτικότητα και την ευελιξία στην υλοποίηση ανακλητών λειτουργιών σε εφαρμογές, θα ακολουθήσει μια απλή μετρική που θα εξετάσει την επεκτασιμότητα και συντηρησιμότητα της αρχιτεκτονικής. Η μετρική θα επικεντρωθεί σε δύο σενάρια: στην εισαγωγή νέου τύπου ICommand και στην επέκταση της λειτουργικότητας της λειτουργίας αναίρεσης. Σε περίπτωση που επιθυμούμε να προσθέσουμε έναν επιπλέον τύπο ICommand, θα πρέπει να γίνουν οι κάτωθι τροποποιήσεις:

- Προσθήκη μιας κλάσης RemoveTextCommand που θα επεκτείνει την ICommand και θα υπερκαλύπτει τις μεθόδους Execute και Undo.
- Κατάλληλη προσθήκη εξάρτησης προς την κλάση αυτή, στον κώδικα πελάτη.

Άρα, προστίθεται 1 κλάση (O(1) πολυπλοκότητα).

Στην περίπτωση που είναι επιθυμητή η επέκταση της λειτουργικότητας τη αναίρεσης, με σκοπό να καθίσταται δυνατή η επανάληψη εντολών που αναιρέθηκαν, θα πρέπει να γίνουν οι κάτωθι τροποποιήσεις:

- Τροποποίηση της CommandInvoker, ώστε να διατηρεί μια ακόμα στοίβα για τις εντολές που ανακλήθηκαν.
- Τροποποίηση της μεθόδου Undo της CommandInvoker, ώστε να προσθέτει την προς αφαίρεση εντολή στη νέα στοίβα, πριν την αφαίρεσή της.
- Προσθήκη μεθόδου Redo στον CommandInvoker που θα λειτουργεί με τρόπο όμοιο με αυτό της Undo

Άρα, εισάγονται αλλαγές μόνο σε 1 κλάση (O(1) πολυπλοκότητα).

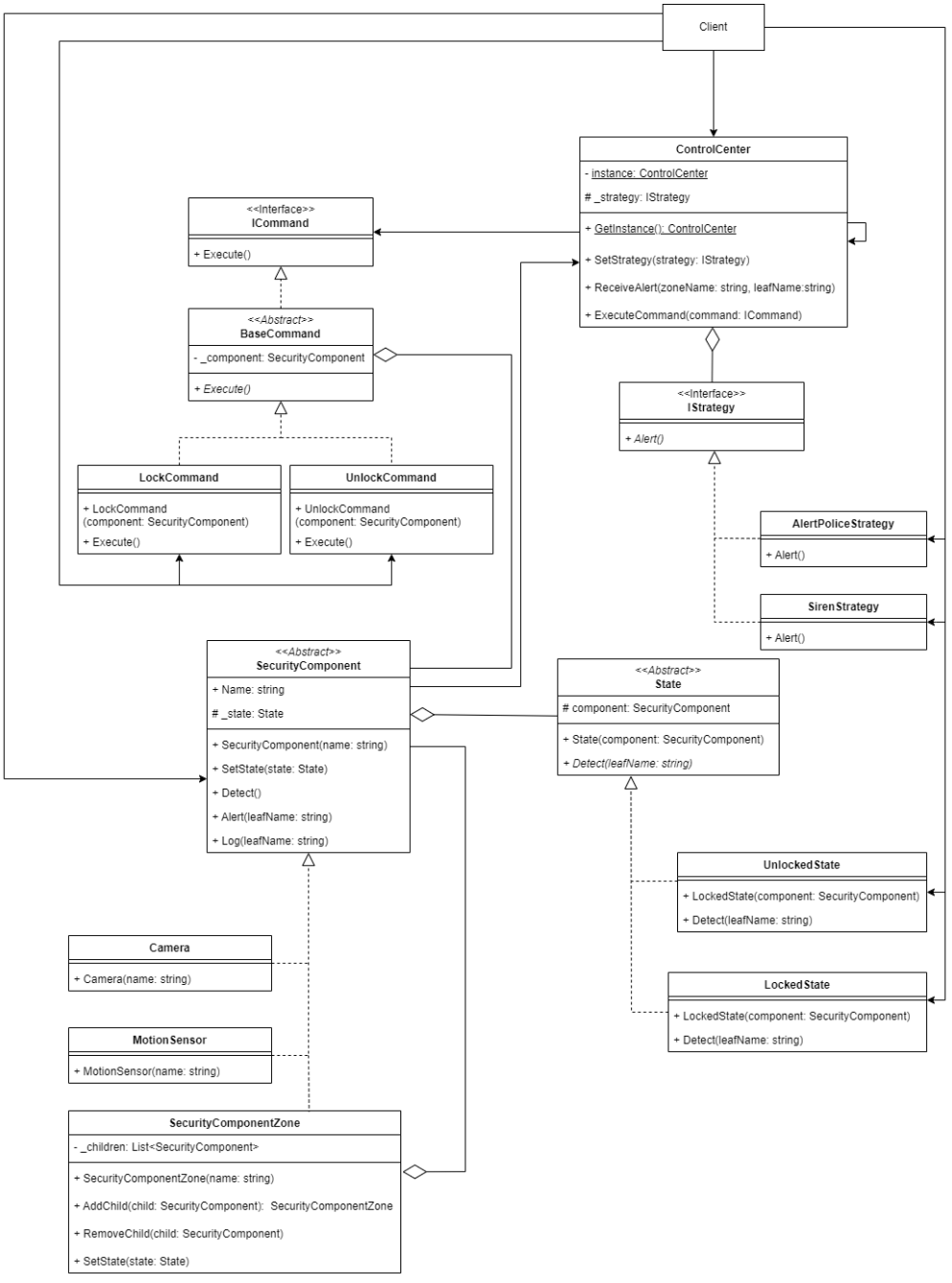
Όπως διαπιστώνεται, ο συνδυασμός των Memento και Command μοτίβων παρουσιάζει υψηλή προσαρμοστικότητα και συντηρησιμότητα σε σενάρια δημιουργίας ανακλητών λειτουργιών. Αυτή η προσέγγιση διατηρεί σαφή διαχωρισμό ανησυχιών. Η εισαγωγή νέων αντικειμένων Command είναι εξαιρετικά απλή, και η προσθήκη νέων δυνατοτήτων αναίρεσης επηρεάζει μόνο τοπικά την κλάση CommandInvoker, χωρίς να απαιτούνται εκτεταμένες αλλαγές στο υπόλοιπο σύστημα. Επομένως, επιτυγχάνονται όλα τα οφέλη που προσφέρει η δυνατότητα αναίρεσης εντολών, και παράλληλα διασφαλίζεται ο διαχωρισμός ανησυχιών και η κλιμακωσιμότητα του συστήματος. Το στοιχείο που χρήζει προσοχής κατα την ανάπτυξη αυτής της αρχιτεκτονικής, είναι η επιβάρυνση της επίδοσης από τον επιπλέον φόρτο διαχείρισης των στιγμιοτύπων, τόσο από άποψη καθυστέρησης, όσο και από κατανάλωσης μνήμης.

3.9 Command, Composite, State, Strategy & Singleton

Αυτό το υποκεφάλαιο θα ασχοληθεί με τη δημιουργία μιας μελέτης περίπτωσης που θα συνδυάζει πέντε από τα μοτίβα σχεδίασης που περιεγράφηκαν στο προηγούμενο κεφάλαιο, με σκοπό την κατασκευή ενός συστήματος ασφαλείας. Τα μοτίβα αυτά, είναι τα Command, Composite, State, Strategy και Singleton. Το εν λόγω σύστημα ασφαλείας θα έχει τη δυνατότητα να οργανώνει τα μέρη του σε ιεραρχικές δενδρικές δομές, καθώς επίσης να

αλλάζει δυναμικά την κατάσταση των αντικειμένων αυτών και τη στρατηγική απόκρισης, όταν ανιχνεύονται απειλές. Θα αποτελείται από ένα κεντρικό σημείο ελέγχου, κάμερες, αισθητήρες κίνησης και ζώνες. Τα στοιχεία αυτά θα διαθέτουν δυνατότητα κλειδώματος, ξεκλειδώματος και αντίδρασης σε παραβιάσεις ασφαλείας.

Ακολουθεί διάγραμμα κλάσης και υλοποίηση σε κώδικα C# για το εν λόγω σενάριο:



Σχημα 3.10: Διάγραμμα Κλάσης Μελέτης Περίπτωσης

Στο Command μοτίβο, η διεπαφή ICommand ορίζει τη συμπεριφορά που θα πρέπει να έχει κάθε εντολή, και η αφηρημένη κλάση BaseCommand που την επεκτείνει, διατηρεί την κατάλληλη αναφορά προς ένα συστατικό του συστήματος, που θα είναι και το αντικείμενο της εντολής. Οι κλάσεις LockCommand και UnlockCommand επεκτείνουν την προαναφερθείσα αφηρημένη κλάση και παρέχουν προσαρμοσμένες υλοποιήσεις για κλείδωμα και ξεκλείδωμα συστατικών του συστήματος.

```
1   public interface ICommand
2   {
3       public void Execute();
4   }
5
6   public abstract class BaseCommand : ICommand
7   {
8       protected SecurityComponent component;
9       public virtual void Execute() { }
10  }
11
12  public class LockCommand : BaseCommand
13  {
14      public LockCommand(SecurityComponent component)
15      {
16          this.component = component;
17      }
18
19      public override void Execute()
20      {
21          component.SetState(new LockedState(component));
22      }
23  }
24
25  public class UnlockCommand : BaseCommand
26  {
27      public UnlockCommand(SecurityComponent component)
28      {
29          this.component = component;
30      }
31
32      public override void Execute()
33      {
```

```
34                  component.SetState(new UnlockedState(component));
35              }
36          }
```

Στο State μοτίβο, η αφηρημένη κλάση State και οι προσαρμοσμένες υλοποιήσεις της UnlockedState και LockedState, διαχειρίζονται την κατάσταση των συστατικών του συστήματος. Διατηρούν αντίστροφη αναφορά προς τα στοιχεία που αφορούν, και υπαγορεύοουν τη συμπεριφορά που θα έχουν αυτά σε συμβάντα ανίχνευσης, με κριτήριο το αν εκείνα βρίσκονται σε κλειδωμένη ή ξεκλείδωτη κατάσταση.

```
1           public abstract class State
2           {
3               protected SecurityComponent component;
4               public State(SecurityComponent component) => this.component = component;
5               public abstract void Detect(string leafName);
6           }
7
8           public class LockedState : State
9           {
10              public LockedState(SecurityComponent component) : base(component) { }
11              public override void Detect(string leafName) => component.Alert(leafName)
                    ;
12          }
13
14          public class UnlockedState : State
15          {
16              public UnlockedState(SecurityComponent component) : base(component) { }
17              public override void Detect(string leafName) => component.Log(leafName);
18          }
```

Το Strategy μοτίβο, επεκτείνει το σύστημα με τη δυνατότητα δυναμικής αλλαγής στρατηγικής απόκρισης, σε συμβάντα παραβίασης ασφαλείας. Αποτελείται από την διεπαφή IStrategy και τις προσαρμοσμένες υλοποιήσεις της AlertPoliceStrategy και SirenStrategy για ειδοποίηση της αστυνομίας και ήχηση της σειρήνας αντίστοιχα.

```
1           public interface IStrategy
2           {
3               public void Alert();
```

```
    }

    public class AlertPoliceStrategy : IStrategy
    {
        public void Alert()
        {
            Console.WriteLine("Calling Police");
        }
    }

    public class SirenStrategy : IStrategy
    {
        public void Alert()
        {
            Console.WriteLine("**Siren Noise**");
        }
    }
```

Στο Composite μοτίβο, η αφηρημένη κλάση SecurityComponent ορίζει τα κοινά χαρακτηριστικά και την κοινή συμπεριφορά που πρέπει να έχουν όλα τα συστατικά του συστήματος. Περιλαμβάνει όνομα και ένα αντικείμενο State που αντιπροσωπεύει το αν είναι κλειδωμένο ή όχι. Επίσης, μεθόδους για ορισμό της κατάστασης αυτής, ανίχνευσης συμβάντων και απόκρισης σε αυτά. Οι προσαρμοσμένες κλάσεις Camera και MotionSensor αποτελούν τα φύλλα της δομής. Η SecurityComponentZone λειτουργεί ως σύνθετη και μπορεί να περιλαμβάνει επιπλέον ζώνες, κάμερες και αισθητήρες κίνησης, που θα λαμβάνουν την ίδια κατάσταση με τη δική της, εκτός αν οριστεί πιο ειδικά κάτι διαφορετικό, απευθείας στα θυγατρικά αντικείμενά της.

```
    public abstract class SecurityComponent
    {
        public string Name { get; private set; }
        protected State _state;
        public SecurityComponent(string name){Name = name;}
        public virtual void SetState(State state) => _state = state;
        public virtual void Detect() => _state.Detect(Name);
        public virtual void Alert(string leafName) => ControlCenter.GetInstance()
            .ReceiveAlert(Name, leafName);
        public virtual void Log(string leafName) => Console.WriteLine($"{Name}:
            Detection from {leafName}");
```

```
10          }
11
12          public class Camera : SecurityComponent
13          {
14              public Camera(string name) : base(name) { }
15          }
16
17          public class MotionSensor : SecurityComponent
18          {
19              public MotionSensor(string name) : base(name) { }
20          }
21
22          public class SecurityComponentZone : SecurityComponent
23          {
24              private List<SecurityComponent> _children = new List<SecurityComponent>()
                    ;
25
26              public SecurityComponentZone(string name) : base(name) { }
27
28              public SecurityComponentZone AddChild(SecurityComponent child)
29              {
30                  _children.Add(child);
31                  return this;
32              }
33
34              public void RemoveChild(SecurityComponent child)
35              {
36                  _children.Remove(child);
37              }
38
39              public override void SetState(State state)
40              {
41                  _state = state;
42                  foreach (var component in _children)
43                  {
44                      component.SetState(state);
45                  }
46              }
47          }
```

Το Singleton μοτίβο υλοποιείται στην κλάση ControlCenter και συγκεντρώνει τη διαχείριση

του συστήματος ασφαλείας σε ένα ενιαίο σημείου ελέγχου, στο οποίο παράλληλα ορίζεται και εκτελείται η στρατηγική απόκρισης σε συμβάντα ασφαλείας. Η κλάση διασφαλίζει οτι υπάρχει μόνο ένα στιγμιότυπό της, το οποίο μέσω της στατικής της μεθόδου GetInstance, είτε επιστρέφει, είτε όταν δεν υπάρχει, πρώτα το δημιουργεί και έπειτα το επιστρέφει.

```
1   public class ControlCenter
2   {
3       private static ControlCenter instance;
4       private IStrategy _strategy;
5
6       private ControlCenter() { }
7       public static ControlCenter GetInstance()
8       {
9           if (instance == null)
10          {
11              instance = new ControlCenter();
12          }
13          return instance;
14      }
15      public void SetStrategy(IStrategy strategy) => _strategy = strategy;
16      public void ReceiveAlert(string zoneName, string leafName)
17      {
18          Console.WriteLine($"DANGER! {leafName} detected something in {zoneName
                }");
19          _strategy?.Alert();
20      }
21      public void ExecuteCommand(ICommand command) => command.Execute();
22  }
```

Ο κώδικας πελάτη απεικονίζει τη λειτουργική χρήση του συστήματος, εκτελώντας εντολές για το κλείδωμα ή το ξεκλείδωμα ζωνών, αλλάζοντας στρατηγική απόκρισης και ενεργοποιώντας ανιχνεύσεις για την προσομοίωση συμβάντων ασφαλείας.

```
1   public class Program
2   {
3       public static void Main()
4       {
5           // Initialization code
6           ControlCenter.GetInstance().SetStrategy(new AlertPoliceStrategy());
```

```
7
8                   var camera1 = new Camera("camera1");
9                   var camera2 = new Camera("camera2");
10                  var camera3 = new Camera("camera3");
11                  var sensor1 = new MotionSensor("sensor1");
12                  var mainZone = new SecurityComponentZone("Main Zone");
13                  var zone1 = new SecurityComponentZone("Zone1");
14
15                  zone1.AddChild(camera1).AddChild(sensor1);
16                  mainZone.AddChild(camera2).AddChild(zone1);
17
18                  // Client code
19                  ControlCenter.GetInstance().ExecuteCommand(new UnlockCommand(mainZone)
                        );
20                  ControlCenter.GetInstance().ExecuteCommand(new LockCommand(zone1));
21
22                  camera1.Detect();
23                  camera2.Detect();
24
25                  ControlCenter.GetInstance().SetStrategy(new SirenStrategy());
26                  sensor1.Detect();
27              }
28          }
```

Για να αξιολογηθεί η περίπτωση χρήσης, θα εισαχθεί μια απλή μετρική που θα εστιάζει στη δυνατότητα επέκτασης και στην ευκολία τροποποίησης του μοτίβου. Η μετρική θα επικεντρωθεί σε τρία σενάρια: στην εισαγωγή ενός νέου τύπου SecurityComponent, στην εισαγωγή ενός νέου τύπου IStrategy και στην εισαγωγή ενός νέου τύπου State που θα ενεργοποιείται με την εισαγωγή ενός νέου τύπου ICommand. Σε περίπτωση που επιθυμούμε να προσθέσουμε έναν επιπλέον τύπο SecurityComponent, θα πρέπει να γίνουν οι κάτωθι τροποποιήσεις:

- Προσθήκη μιας κλάσης ThermalCamera ή AdvancedSecurityComponentZone που θα επεκτείνουν την SecurityComponent καθ' ομοίωση των Camera και SecurityComponentZone αντίστοιχα.

Άρα, προστίθεται 1 κλάση (O(1) πολυπλοκότητα).

Στην περίπτωση που είναι επιθυμητή η προσθήκη ενός νέου τύπου IStrategy, για να ενισχυθεί η δυνατότητα απόκρισης του συστήματος σε απειλές, θα πρέπει να γίνουν οι κάτωθι τροποποιήσεις:

- Προσθήκη μιας κλάσης LockdownStrategy που θα επεκτείνει την IStrategy και θα υπερκαλύπτει τη μέθοδο Alert.

Άρα, προστίθεται 1 κλάση (O(1) πολυπλοκότητα).

Στην περίπτωση που είναι επιθυμητή η προσθήκη ενός νέου τύπου State για τα στοιχεία του συστήματος ασφαλείας, που θα υποδεικνύει οτι αυτά βρίσκονται υπό συντήρηση, θα πρέπει να γίνουν οι κάτωθι τροποποιήσεις:

- Προσθήκη μιας κλάσης MaintenanceState που θα επεκτείνει την State και θα υπερκαλύπτει τη μέθοδο Detect.
- Προσθήκη μιας κλάσης MaintenanceCommand που θα επεκτείνει την αφηρημένη κλάση BaseCommand και θα υπερκαλύπτει τη μέθοδο Execute.

Άρα, προστίθεται 2 κλάσεις (O(1) πολυπλοκότητα).

Όπως διαπιστώνεται, ο συνδυασμός των Command, Composite, State, Strategy & Singleton μοτίβων για την δημιουργία ενός συστήματος ελέγχου ασφαλείας, αποτελεί παράδειγμα της ικανότητας των μοτίβων σχεδίασης να κατασκευάζουν πολύπλοκες, επεκτάσιμες και συντηρήσιμες αρχιτεκτονικές λογισμικού. Η εισαγωγή νέων συστατικών συστήματος ασφαλείας και στρατηγικών είναι εξαιρετικά απλή, και η εισαγωγή νέων τύπων καταστάσεων για τα εν λόγω στοιχεία, απαιτεί επίσης λιγοστές αλλαγές. Ακόμα, η αλληλεπίδραση των μοτίβων στην αρχιτεκτονική του συστήματος αναδεικνύει πληθώρα οφελών. Εξασφαλίζεται η ύπαρξη κεντρικού σημείου ελέγχου και συντονισμού, η οργάνωση και ενιαία αντιμετώπιση όλων των συστατικών του συστήματος ασφαλείας, η δυναμική αλλαγή της κατάστασης των εν λόγω συστατικών κατόπιν λήψης της αντίστοιχης εντολής, και η δυναμική αλλαγή στρατηγικών απόκρισης σε συμβάντα ανίχνευσης απειλών. Ως προς τα μειονεκτήματα, ισχύουν όσα έχουν αναφερθεί και στα προηγούμενα υποκεφάλαια για επιβάρυνση στις επιδόσεις και αύξηση της πολυπλοκότητας που προκύπτει από την εισαγωγή πολλαπλών κλάσεων και διεπαφών.

Κεφάλαιο 4

Συμπεράσματα

Στο τελευταίο κεφάλαιο, αφού έχουν παρουσιαστεί τα παραδοσιακά αντικειμενοστρεφή μοτίβα σχεδίασης και έχουν διερευνηθεί κάποιοι από τους πιθανούς επωφελείς συνδυασμούς τους με σκοπό τη δημιουργία προηγμένων αρχιτεκτονικών λύσεων, θα ακολουθήσουν συμπερασματικοί πίνακες. Αυτοί περιλαμβάνουν για καθέναν από αυτούς τους συνδυασμούς, τον στόχο ανάπτυξής του, τα εμπλεκόμενα μοτίβα, τα κύρια στοιχεία και τα σενάρια επέκτασης του.

Συνδυασμός μοτίβων	Abstract Factory & Builder
Στόχος	Κατασκευή σύνθετων αντικειμένων (π.χ. Η/Υ) που έχουν διάφορες προδιαγραφές, με ευέλικτο και κλιμακούμενο τρόπο
Abstract Factory	Διεπαφή για τη δημιουργία οικογενειών από σχετιζόμενα ή εξαρτόμενα μεταξύ τους αντικείμενα, των οποίων οι συμπαγείς κλάσεις δεν εκτίθενται.
Builder	Διαχωρίζει τη διαδικασία κατασκευής ενός σύνθετου αντικειμένου από την αναπαράστασή του, επιτρέποντας στην ίδια διαδικασία κατασκευής να δημιουργεί διαφορετικές αναπαραστάσεις.
Κύρια στοιχεία του Συνδυασμού	- Διεπαφή Abstract Factory: Ορίζει τις μεθόδους δημιουργίας εξαρτημάτων. - Κλάση ComputerBuilder: Υλοποιεί τη διεπαφή Builder για τη σύνθεση των εξαρτημάτων σε ένα τελικό προϊόν. - Κλάση Director: Διαχειρίζεται τη διαδικασία κατασκευής.
Σενάρια επέκτασης	% Ινισιβλε ρυλε το αδδ σπαςε ατ τηε τοπ - Η προσθήκη μιας νέας κατηγορίας εξαρτημάτων (π.χ. PowerSupply) απαιτεί την προσθήκη διεπαφής και συμπαγών κλάσεων για το συστατικό, την ενημέρωση του Abstract Factory και του Builder με νέες μεθόδους και την προσαρμογή του Director ώστε να ενσωματώσει το νέο βήμα κατασκευής. Η πολυπλοκότητα εκτιμάται ως O(n), όπου n ο αριθμός οικογενειών εξαρτημάτων. - Η εισαγωγή ενός νέου τύπου υπολογιστή (π.χ. Κβαντικός υπολογιστής) απαιτεί την προσθήκη μιας νέας κλάσης εργοστασίου που κληρονομεί από το Abstract Factory, την προσθήκη κλάσεων για κάθε τύπο εξαρτήματος και την ενημέρωση του Director για να χειρίζεται τον νέο τύπο. Η πολυπλοκότητα εκτιμάται ως O(m), όπου m ο αριθμός των κατηγοριών εξαρτημάτων που περιλαμβάνει κάθε Η/Υ.

Πινακας 4.1: Abstract Factory & Builder

Συνδυασμός μοτίβων	Composite & Builder
Στόχος	Κατασκευή σύνθετων ιεραρχικών δομών αντικειμένων (π.χ. σύστημα αρχείων) με σαφή και δομημένο τρόπο.
Composite	Επιτρέπει τη σύνθεση αντικειμένων σε δενδρικές δομές, και την ομοιόμορφη διαχείριση και επεξεργασία μεμονωμένων αντικειμένων και συνθέσεων αντικειμένων.
Builder	Ενθυλακώνει την κατασκευαστική λογική της ιεραρχικής δομής.
Κύρια στοιχεία του Συνδυασμού	- Αφηρημένη κλάση FileSystemNode: Ορίζει τη συμπεριφορά των φύλλων και των σύνθετων αντικειμένων. - Κλάσεις File και Folder: Υλοποιήσεις των φύλλων και των σύνθετων αντικειμένων αντίστοιχα. - Κλάση FileSystemBuilder: Κατασκευή και σύνθεση των κόμβων, και κατασκευή ενός συστήματος αρχείων.
Σενάρια επέκτασης	- Η προσθήκη μιας νέας κατηγορίας κόμβων (π.χ. Shortcut) απαιτεί την προσθήκη μίας κλάσης που κληρονομεί από την FileSystemBuilder και την ενημέρωση του Builder με μία νέα μέθοδο. Η πολυπλοκότητα εκτιμάται ως O(1). - Η εισαγωγή μια νέας μεθόδου που θα εκτελεί κάποια λειτουργία στους κόμβους (π.χ. PrintDetails) απαιτεί την προσθήκη μιας νέας μεθόδου σε κάθε τύπο κόμβου. Η πολυπλοκότητα εκτιμάται ως O(n), όπου n ο αριθμός των διαφορετικών τύπων κόμβων.

Πινακας 4.2: Composite & Builder

Συνδυασμός μοτίβων	Iterator & Factory Method
Στόχος	Επίτευξη ευέλικτης και προσαρμόσιμης διάσχισης μιας ποικιλίας τύπων συλλογών με χρήση μιας κοινής διεπαφής.
Iterator	Παρέχει έναν τρόπο διάσχισης των στοιχείων κάποιας σύνθετης δομής διαδοχικά, χωρίς να αποκαλύπτονται η λεπτομέρειες οργάνωσης της εν λόγω δομής.
Factory Method	Δημιουργεί και επιστρέφει αλγορίθμους διάσχισης συλλογών, με κριτήριο τον τύπο της συλλογής και τον επιθυμητό τρόπο διάσχισής της.
Κύρια στοιχεία του Συνδυασμού	- Διεπαφή ICollectionCreator, και κλάσεις ListCreator και HeapCreator: Δημιουργούν και επιστρέφουν αλγορίθμους διάσχισης συλλογών (Iterators). - Διεπαφή IIterator: Διεπαφή διάσχισης των στοιχείων συλλογών. - Προσαρμοσμένες υλοποιήσεις της IIterator.
Σενάρια επέκτασης	- Η προσθήκη ενός νέου τύπου συλλογής απαιτεί την προσθήκη μίας νέας κλάσης συλλογής και μιας αντίστοιχης κλάσης εργοστασίου για την επιστροφή ενός κατάλληλου αλγορίθμου διάσχισης. Η πολυπλοκότητα εκτιμάται ως O(1). - Η εισαγωγή ενός νέου αλγορίθμου διάσχισης για μια υφιστάμενη συλλογή απαιτεί την προσθήκη μιας νέας κλάσης IIterator και την προσθήκη νέας μεθόδου εργοστασίου ή τροποποίηση της υφιστάμενης για επιστροφή του επιθυμητού αλγορίθμου διάσχισης με κριτήριο κάποια συνθήκη. Η πολυπλοκότητα εκτιμάται ως O(1).

Πινακας 4.3: Iterator & Factory Method

Συνδυασμός μοτίβων	Iterator & Composite
Στόχος	Αποτελεσματική διαχείριση ιεραρχικών δομών και δυνατότητα διάσχισής τους μέσω μιας ενοποιημένης διεπαφής.
Iterator	Παρέχει έναν τρόπο διάσχισης των στοιχείων κάποιας σύνθετης δομής διαδοχικά, χωρίς να αποκαλύπτεται η υποκείμενη δομή του.
Composite	Επιτρέπει τη σύνθεση αντικειμένων σε δενδρικές δομές, και την ομοιόμορφη διαχείριση και επεξεργασία μεμονωμένων αντικειμένων και συνθέσεων αντικειμένων.
Κύρια στοιχεία του Συνδυασμού	- FileSystemNode: Αφηρημένη κλάση που αντιπροσωπεύει όλους τους τύπους κόμβων. - Κλάσεις File και Folder: Υλοποιήσεις των φύλλων και των σύνθετων αντικειμένων αντίστοιχα. - IIterableCollection και IIterator: Διεπαφές για τη δημιουργία αλγορίθμων διάσχισης και εκτέλεσης της λειτουργίας αυτής αντίστοιχα.
Σενάρια επέκτασης	- Η προσθήκη ενός νέου τύπου κόμβου απαιτεί την προσθήκη μίας νέας κλάσης κόμβου και επιπλέον την τροποποίηση της μεθόδου GetNext κάθε συμπαγούς IIterator στην περίπτωση εισαγωγής νέου τύπου σύνθετου αντικειμένου. Η πολυπλοκότητα εκτιμάται ως O(n), όπου n ο αριθμός των διαφορετικών τύπων IIterator. - Η εισαγωγή ενός νέου αλγορίθμου διάσχισης απαιτεί την προσθήκη μιας νέας κλάσης IIterator και για κάθε σύνθετο κόμβο, την προσθήκη νέας μεθόδου δημιουργίας του αλγορίθμου διάσχισης ή τροποποίηση της υφιστάμενης για επιστροφή του επιθυμητού αλγορίθμου με κριτήριο κάποια συνθήκη. Η πολυπλοκότητα εκτιμάται ως O(m), όπου m ο αριθμός των διαφορετικών τύπων κόμβων που υλοποιούν την IIterableCollection.

Πινακας 4.4: Iterator & Composite

Συνδυασμός μοτίβων	Visitor & Composite
Στόχος	Διαχωρισμός της λογικής εκτέλεσης των λειτουργιών, από τη δομή των αντικειμένων που υφίστανται την επεξεργασία, ώστε να επιτυγχάνεται η προσθήκη νέων λειτουργιών σε υπάρχουσες δομές αντικειμένων, χωρίς αυτές να τροποποιούνται.
Visitor	Επισκέπτεται τα στοιχεία μιας δομής αντικειμένων και εκτελεί λειτουργίες σε αυτά.
Composite	Επιτρέπει τη σύνθεση αντικειμένων σε δενδρικές δομές, και την ομοιόμορφη διαχείριση και επεξεργασία μεμονωμένων αντικειμένων και συνθέσεων αντικειμένων.
Κύρια στοιχεία του Συνδυασμού	- Employee: Αφηρημένη κλάση που αντιπροσωπεύει όλους τους τύπους κόμβων. - Κλάσεις Developer, Designer και Manager: Τύποι κόμβων που αντιπροσωπεύουν διαφορετικούς ρόλους εργαζομένων. - Κλάση Department: Τύπος σύνθετου αντικειμένου που διατηρεί αντικείμενα τύπου Employee. - IEmployeeVisitor: Διεπαφή για υλοποιήσεις επισκεπτών.
Σενάρια επέκτασης	- Η προσθήκη ενός νέου τύπου κόμβου απαιτεί την προσθήκη μίας νέας κλάσης που θα επεκτείνει την Employee, και επιπλέον την προσθήκη της αντίστοιχης μεθόδου επισκέπτη στη διεπαφή IEmployeeVisitor και σε κάθε υλοποίησή της. Η πολυπλοκότητα εκτιμάται ως O(n), όπου n ο αριθμός των διαφορετικών τύπων επισκέπτη που επεκτείνουν την IEmployeeVisitor. - Η εισαγωγή ενός νέου τύπου επισκέπτη απαιτεί την προσθήκη μιας νέας κλάσης επισκέπτη που θα υλοποιεί την IEmployeeVisitor και θα περιλαμβάνει μεθόδους επίσκεψης για κάθε τύπο Employee. Η πολυπλοκότητα εκτιμάται ως O(1).

Πινακας 4.5: Visitor & Composite

Συνδυασμός μοτίβων	Chain of Responsibility & Composite
Στόχος	Χειρισμός συμβάντων σε πολύπλοκες ιεραρχικές δομές αντικειμένων, και επεξεργασία τους στο εκάστοτε καταλληλότερο επίπεδο της ιεραρχίας.
Chain of Responsibility	Επιτρέπει τη μεταβίβαση συμβάντων από ένα στοιχείο της ιεραρχίας στο γονικό του (και ούτω καθεξής), μέχρι να βρεθεί ο κατάλληλος χειριστής.
Composite	Επιτρέπει την ιεραρχική οργάνωση των αντικειμένων και την ομοιόμορφη διαχείριση και επεξεργασία μεμονωμένων αντικειμένων και συνθέσεων αντικειμένων, ώστε να εξυπηρετείτε ο μηχανισμός μεταβίβασης συμβάντων.
Κύρια στοιχεία του Συνδυασμού	- UIComponent: Αφηρημένη κλάση που αντιπροσωπεύει όλους τους τύπους στοιχείων UI και περιλαμβάνει μια λίστα _handleableEvents που διατηρεί για κάθε στοιχείο, ποια συμβάντα αυτό μπορεί να χειριστεί από μόνο του. - Κλάσεις Button και Panel: Τύποι κόμβων που αντιπροσωπεύουν φύλλα και σύνθετα αντικείμενα αντίστοιχα.
Σενάρια επέκτασης	- Η προσθήκη ενός νέου τύπου κόμβου απαιτεί την προσθήκη μίας νέας κλάσης που θα επεκτείνει την UIComponent και θα υπερκαλύπτει τη μέθοδο HandleEvent. Η πολυπλοκότητα εκτιμάται ως O(1). - Η εισαγωγή ενός νέου τύπου συμβάντος απαιτεί κατάλληλη τροποποίηση κάθε υπάρχουσας κλάσης που επεκτείνει την UIComponent και θα πρέπει να μπορεί να χειριστεί αυτό τον τύπο συμβάντων. Η πολυπλοκότητα εκτιμάται ως O(m), όπου m ο αριθμός των κλάσεων που χρειάζονται τροποποίηση.

Πινακας 4.6: Chain of Responsibility & Composite

Συνδυασμός μοτίβων	Chain of Responsibility & Command - Commands ως χειριστές αλυσίδας
Στόχος	Κατασκευή ενός ευέλικτου και συντηρήσιμου μηχανισμού επεξεργασίας δεδομένων, σε διαδοχικά βήματα που αντιπροσωπεύονται από εντολές.
Chain of Responsibility	Επιτρέπει τη μεταβίβαση αιτημάτων κατά μήκος μιας αλυσίδας πιθανών χειριστών, μέχρι αυτά να διεκπεραιωθούν.
Command	Ενθυλακώνει σε ένα αντικείμενο, όλες τις λεπτομέρειες που αφορούν μια λειτουργία που είναι επιθυμητό να εκτελεστεί σε κάποια δεδομένα.
Κύρια στοιχεία του Συνδυασμού	- BaseCommand: Αφηρημένη κλάση που ορίζει την προκαθορισμένη συμπεριφορά των χειριστών της αλυσίδας. - Κλάσεις ValidationCommand, TransformationCommand, EnrichmentCommand και FinalProcessingCommand: Τύποι εντολών που αντιπροσωπεύουν διαφορετικά στάδια επεξεργασίας. - Κλάση DataRequest: Αντιπροσωπεύει τα υπό επεξεργασία δεδομένα.
Σενάρια επέκτασης	- Η προσθήκη ενός νέου τύπου εντολής απαιτεί την προσθήκη μίας νέας κλάσης που θα επεκτείνει την BaseCommand και θα υπερκαλύπτει τη μέθοδο Execute. Η πολυπλοκότητα εκτιμάται ως O(1). - Η τροποποίηση της μεθόδου Execute κάποιου τύπου εντολής γίνεται τοπικά και χωρίς να επηρεάζεται η δομή της αλυσίδας. Η πολυπλοκότητα εκτιμάται ως O(1).

Πινακας 4.7: Chain of Responsibility & Command - Commands ως χειριστές αλυσίδας

Συνδυασμός μοτίβων	Chain of Responsibility & Command - Commands ως αιτήματα
Στόχος	Κατασκευή ενός ευέλικτου και συντηρήσιμου μηχανισμού επεξεργασίας δεδομένων, όπου τα αιτήματα θα ενθυλακώνουν τη λογική εκτέλεσης λειτουργιών.
Chain of Responsibility	Επιτρέπει τη μεταβίβαση αιτημάτων κατά μήκος μιας αλυσίδας πιθανών χειριστών, μέχρι αυτά να διεκπεραιωθούν.
Command	Ενθυλακώνει σε ένα αντικείμενο, όλες τις λεπτομέρειες που αφορούν μια λειτουργία που είναι επιθυμητό να εκτελεστεί σε κάποια δεδομένα.
Κύρια στοιχεία του Συνδυασμού	- BaseDepartmentHandler: Αφηρημένη κλάση που ορίζει την προκαθορισμένη συμπεριφορά των χειριστών της αλυσίδας. - Κλάσεις HRDepartment, FinanceDepartment και LegalDepartment: Τύποι χειριστών της αλυσίδας. - Κλάση DocumentSigningCommand: Τύπος εντολής υπογραφής εγγράφων. - Κλάση DocumentService: Εκτελεί τις πραγματικές λειτουργίες στα έγγραφα.
Σενάρια επέκτασης	- Η προσθήκη ενός νέου τύπου εντολής απαιτεί την προσθήκη μίας νέας κλάσης που θα επεκτείνει την ICommand και θα υπερκαλύπτει τη μέθοδο Execute. Η πολυπλοκότητα εκτιμάται ως O(1). - Η προσθήκη ενός νέου τύπου χειριστή απαιτεί την προσθήκη μίας νέας κλάσης που θα επεκτείνει την BaseDepartmentHandler και θα υπερκαλύπτει τη μέθοδο SignDocument . Η πολυπλοκότητα εκτιμάται ως O(1).

Πινακας 4.8: Chain of Responsibility & Command - Commands ως αιτήματα

Συνδυασμός μοτίβων	Command & Memento
Στόχος	Υλοποίηση ανακλητών λειτουργιών σε εφαρμογές.
Command	Ενθυλακώνει σε ένα αντικείμενο, όλες τις λεπτομέρειες που αφορούν μια λειτουργία που είναι επιθυμητό να εκτελεστεί σε κάποια δεδομένα.
Memento	Καταγράφει και εξωτερικεύει την εσωτερική κατάσταση ενός αντικειμένου, ώστε το αντικείμενο να μπορεί να επανέλθει σε αυτή την κατάσταση αργότερα.
Κύρια στοιχεία του Συνδυασμού	- Κλάση AddTextCommand: Τύπος εντολής που προσθέτει κείμενο σε ένα άλλο κείμενο. - Κλάση CommandInvoker: Διαχειρίζεται την εκτέλεση και αναίρεση των εντολών, ενεργώντας ως Invoker για το Command και Caretaker για το Memento. - Κλάση Memento: Διατηρεί στιγμιότυπο της κατάστασης κάποιου αντικειμένου πριν από την εκτέλεση μιας εντολής σε αυτό. - Κλάση TextService: Ενεργεί ως Originator και διατηρεί το περιεχόμενο όλου του κειμένου.
Σενάρια επέκτασης	- Η προσθήκη ενός νέου τύπου εντολής απαιτεί την προσθήκη μίας νέας κλάσης που θα επεκτείνει την ICommand και θα υπερκαλύπτει τις μεθόδους Execute και Undo. Η πολυπλοκότητα εκτιμάται ως O(1). - Η προσθήκη δυνατότητας επανάληψης εντολών που αναιρέθηκαν, απαιτεί την τροποποίηση της κλάσης CommandInvoker, ώστε να διαχειρίζεται μια στοίβας επανάληψης εντολών παράλληλα με τη στοίβα αναίρεσης. Η πολυπλοκότητα εκτιμάται ως O(1).

Πινακας 4.9: Command & Memento

Συνδυασμός μοτίβων	Command, Composite, State, Strategy & Singleton
Στόχος	Δημιουργία ενός ευέλικτου, επεκτάσιμου και συντηρήσιμου συστήματος (π.χ. ασφαλειάς) που θα μπορεί να ανταποκρίνεται δυναμικά σε διάφορα συμβάντα, να διαχειρίζεται διάφορα στοιχεία διαφορετικών τύπων, και να αλλάζει τις στρατηγικές απόκρισης ανάλογα με τις ανάγκες.
Command	Ενθυλακώνει σε ένα αντικείμενο, όλες τις λεπτομέρειες που αφορούν μια λειτουργία που είναι επιθυμητό να εκτελεστεί σε κάποια δεδομένα (π.χ. λειτουργίες κλειδώματος και ξεκλειδώματος).
Composite	Επιτρέπει τη σύνθεση των διαφορετικών στοιχείων - αντικειμένων σε μια ιεραρχική δομή, επιτρέποντας την ομοιόμορφη αντιμετώπιση μεμονωμένων και σύνθετων αντικειμένων.
State	Επιτρέπει στα στοιχεία να αλλάζουν τη συμπεριφορά τους με βάση την κατάστασή τους (κλειδωμένα ή ξεκλείδωτα).
Strategy	Επιτρέπει τη δυναμική αλλαγή της απόκρισης του συστήματος σε συμβάντα, επιτρέποντας την εφαρμογή διαφορετικών στρατηγικών απόκρισης (π.χ. ειδοποίηση της αστυνομίας ή ήχηση συναγερμού)
Singleton	Εξασφαλίζει την ύπαρξη ενός μοναδικού σημείου ελέγχου και συντονισμού του συστήματος.
Κύρια στοιχεία του Συνδυασμού	- Αφηρημένη κλάση SecurityComponent και επεκτάσεις της Camera, MotionSensor και SecurityComponentZone: Αναπαριστούν τα στοιχεία - συσκευές ασφαλείας. - Κλάση ControlCenter: Εκτελεί εντολές και διαχειρίζεται τις διαφορετικές στρατηγικές απόκρισης. - Αφηρημένη κλάση State και επεκτάσεις της LockedState και UnlockedState: Αναπαριστούν τις δυνατές καταστάσεις των στοιχείων ασφαλείας. - Αφηρημένη κλάση BaseCommand και επεκτάσεις της LockCommand και UnlockCommand: Αναπαριστούν τις λειτουργίες που εκτελούνται επί των στοιχείων ασφαλείας.
Σενάρια επέκτασης	- Η προσθήκη ενός νέου τύπου κόμβου απαιτεί την προσθήκη μίας νέας κλάσης που θα επεκτείνει την SecurityComponent. Η πολυπλοκότητα εκτιμάται ως O(1). - Η προσθήκη ενός νέου τύπου στρατηγικής απαιτεί την προσθήκη μίας νέας κλάσης που θα επεκτείνει την IStrategy. Η πολυπλοκότητα εκτιμάται ως O(1). - Η προσθήκη ενός νέου τύπου κατάστασης και εντολής που την ενεργοποιεί, απαιτεί την προσθήκη μίας νέας κλάσης που θα επεκτείνει την State και μιας νέας κλάσης που θα επεκτείνει την BaseCommand. Η πολυπλοκότητα εκτιμάται ως O(1).

ΠΙΝΑΚΑΣ 4.10: Command, Composite, State, Strategy & Singleton

Βιβλιογραφία

[1] Ralph Johnson John Vlissides Erich Gamma, Richard Helm. 1994. *Design Patterns: Elements of Reusable Object-Oriented Software* (1st ed.). Addison-Wesley Professional.

[2] Alexander Shvets. 2019. *Dive Into Design Patterns* (1st ed.). Alexander Shvets.