

ΠΑΝΕΠΙΣΤΗΜΙΟ ΠΕΙΡΑΙΩΣ
Τμήμα Πληροφορικής



Εργασία Μαθήματος
ΠΡΟΤΥΠΑ ΑΝΑΠΤΥΞΗΣ ΛΟΓΙΣΜΙΚΟΥ

<i>Αριθμός εργασίας – Τίτλος εργασίας</i>	<i>Υλοποίηση Αρχών και Προτύπων Λογισμικού με Σύγχρονες γλώσσες προγραμματισμού</i>
Όνομα φοιτητή	Κωνσταντίνος Καλογερόπουλος
Αρ. Μητρώου	Π19057
Ημερομηνία παράδοσης	12-07-2022



Εκφώνηση εργασίας

Θέμα 1 (8 μονάδες)

Ζητείται να υλοποιηθούν πρωτότυπες υλοποιήσεις με σύγχρονες γλώσσες προγραμματισμού όπως C#, ή rython για τις παρακάτω αρχές αντικειμενοστρεφούς προγραμματισμού και για πρότυπα ανάπτυξης λογισμικού:

1. Αρχή μοναδικού σκοπού (single responsibility principle), Αρχή Κλειστού – Ανοικτού κώδικα (open closed principle) (1 μονάδα)
2. Αρχή αντικατάστασης της Liskov (Liskov substitution), Αρχή διαχωρισμού με διεπαφές (Interface segregation) με πολλαπλή κληρονομικότητα (1 μονάδα)
3. Αρχή αντιστροφής εξαρτήσεων (Dependency Inversion) (2 μονάδες)
4. Πρότυπο Σχεδίασης Adapter (2 μονάδες)
5. Πρότυπο Σχεδίασης Factory & Abstract Factory (2 μονάδες)

Για κάθε ένα από τα παραπάνω θα δημιουργήσετε ένα ενδεικτικό παράδειγμα που θα υλοποιεί το πρότυπο και θα το χρησιμοποιεί και θα παραδώσετε το class diagram, και τον πηγαίο κώδικα σε φάκελο project από IDE της επιλογής σας (π.χ. Visual Studio, κλπ).

Θέμα 2 (2 μονάδες)

Θα μελετήσετε πρότυπα που δίνονται στο βιβλίο [Cloud Design Patterns Book της Microsoft Press](#) που διατίθεται δωρεάν ηλεκτρονικά.

Επιλέξτε ένα από τα παρακάτω ζεύγη προτύπων προς μελέτη:

(Α)

Health Endpoint Monitoring Pattern

Competing Consumers Pattern

(Β)

Event Sourcing Pattern

Command and Query Responsibility Segregation (CQRS) Pattern

(Γ)

Federated Identity Pattern

Retry Pattern

Θα επιλέξετε ένα ζεύγος προτύπων από τα παραπάνω για τα οποία:

1. Θα δημιουργήσετε παρουσίαση Powerpoint ως 15 διαφάνειες
2. Θα γράψετε μία σύντομη αναφορά στα ελληνικά για κάθε πρότυπο στην οποία θα παρουσιάσετε ενδεικτικό παράδειγμα χρήσης του προτύπου (ενδεικτικά απαιτούνται 2-5 σελίδες για κάθε πρότυπο)
3. Η αναφορά σας θα πρέπει να έχει εξώφυλλο με τα στοιχεία (ονοματεπώνυμο, ΑΜ και **το email που χρησιμοποιείτε για login στο teams**) κάθε μέλους της ομάδας

Μπορείτε να χρησιμοποιήσετε τα παραδείγματα που δίνονται στο βιβλίο για να βασίσετε τα παραδείγματα χρήσης τους που θα παρουσιάσετε και θα παραδώσετε το σχετικό class diagram και πηγαίο κώδικα.



ΠΙΝΑΚΑΣ ΠΕΡΙΕΧΟΜΕΝΩΝ

1	Εισαγωγή	4
2	Design Patterns.....	4
2.1	Αρχή Μοναδικού Σκοπού (Single Responsibility Principle)	4
2.2	Αρχή Ανοικτού – Κλειστού κώδικα (open closed principle).....	5
2.3	Αρχή αντικατάστασης της Liskov (Liskov substitution).....	7
2.4	Αρχή διαχωρισμού με διεπαφές (Interface segregation) με πολλαπλή κληρονομικότητα	9
2.5	Αρχή αντιστροφής εξαρτήσεων (Dependency Inversion).....	11
2.6	Πρότυπο Σχεδίασης Adapter	13
2.7	Πρότυπο Σχεδίασης Factory.....	15
2.8	Πρότυπο Σχεδίασης Abstract Factory	17
3	Cloud Design Patterns	19
3.1	Health Endpoint Monitoring Pattern	19
3.2	Competing Consumers Pattern	22
4	Βιβλιογραφικές Πηγές.....	25



1 Εισαγωγή

Στην πρώτη μέρος της εργασίας γίνεται περιγραφή των πιο γνωστών προτύπων ανάπτυξης λογισμικού (SOLID, Factory και Adapter). Στο δεύτερο μέρος, γίνεται αναφορά στα Health Endpoint Monitoring Pattern και Competing Consumers Pattern. Για κάθε πρότυπο, υπάρχει υλοποιημένο ένα παράδειγμα επεξήγησης στην C# με το αντίστοιχο διάγραμμα κλάσεων (class diagram).

2 Design Patterns

2.1 Αρχή Μοναδικού Σκοπού (Single Responsibility Principle)

Η αρχή αυτή υποστηρίζει ότι μια κλάση πρέπει να έχει μόνο ένα λόγο να αλλάξει και δεν πρέπει να έχει περισσότερες από μία αρμοδιότητες.

Πιο συγκεκριμένα, ας δούμε το παρακάτω παράδειγμα:

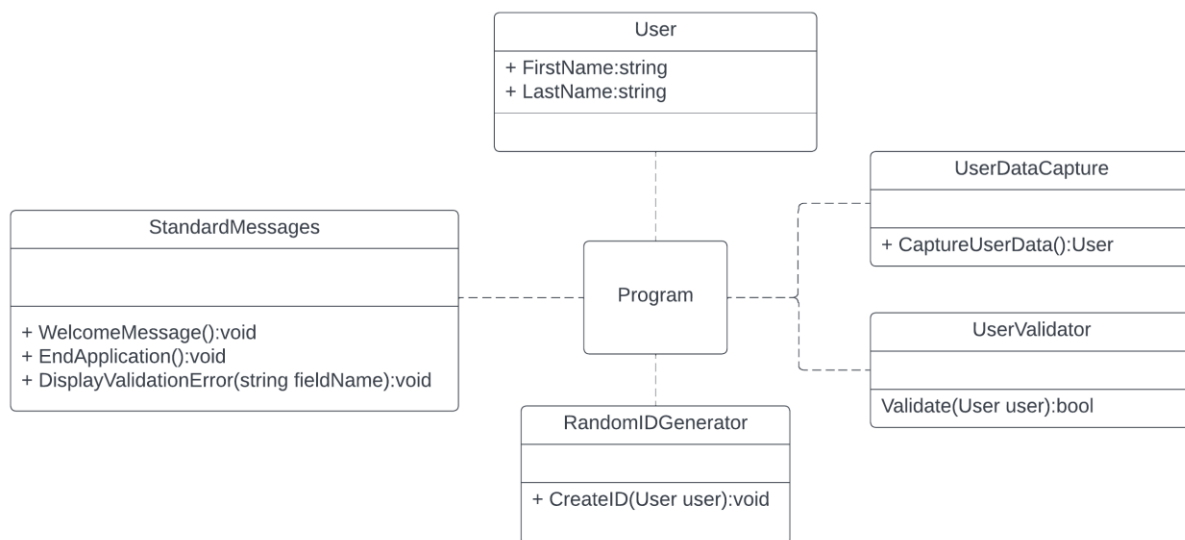
Έστω ένα πρόγραμμα το οποίο ζητάει από τον χρήστη το όνομα και το επώνυμο του, κάνει έλεγχο ορθότητας των δεδομένων που εισάγονται και παράγει ένα τυχαίο αριθμό(id) που αντιστοιχεί για αυτόν.

Οι παραπάνω λειτουργίες γίνονται αρκετά συχνά σε μια μόνο κλάση μέσα στην εκτελέσιμη μέθοδο(main). Όμως, με αυτό τον τρόπο παραβιάζεται η SRP, διότι η κλάση εκτελεί παραπάνω από μια αρμοδιότητες.

Για να λυθεί το προηγούμενο πρόβλημα, θα δημιουργηθούν οι επόμενες κλάσεις (όπως φαίνονται και στο διάγραμμα κλάσεων), οι οποίες θα περιέχουν μεθόδους που υλοποιούν μόνο λειτουργίες που σχετίζονται με το όνομα των κλάσεων:

- User, η κλάση αυτή αναφέρεται στον χρήστη και ουσιαστικά περιλαμβάνει τις πληροφορίες του (Firstname, Lastname).
- StandardMessages, η κλάση αυτή περιλαμβάνει τις μεθόδους για την προβολή μηνυμάτων στην οθόνη του χρήστη, για παράδειγμα «Welcome User, please type your name».
- UserDataCapture, η κλάση αυτή εμπεριέχει τις μεθόδους για την καταγραφή των απαντήσεων (όνομα, επίθετο) του χρήστη στο πρόγραμμα.
- UserValidator, η κλάση αυτή περιλαμβάνει τις μεθόδους για τον έλεγχο εγκυρότητας των δεδομένων που εισάγει ο χρήστης.
- RandomIDGenerator, η κλάση αυτή έχει την μέθοδο, η οποία δημιουργεί ένα τυχαίο αναγνωριστικό ID για κάθε χρήστη.

Έτσι πλέον, η κλάση Program, ελέγχει την ροή του προγράμματος καλώντας τις κατάλληλες μεθόδους, χωρίς να υλοποιεί όλες τις λειτουργίες μέσα στην μέθοδο main όπως είχε συμβεί αρχικά.



Εικόνα 1 Class Diagram for SRP demonstration

Output προγράμματος:

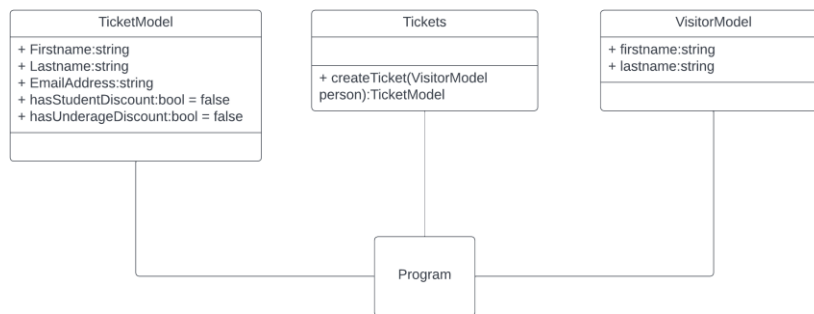
```
C:\Users\kkalo\OneDrive\Documents\GitHub\DesignPatterns\Final\SRP\SRP_Demo\SRP_Demo\bin\Debug\SRP_Dem
Welcome to my application!
What is your first name: Kostas
What is your last name: Kalogeropoulos
Random ID for Kostas Kalogeropoulos is 25806d50-8c17-4787-8d82-4ab77ac9da34
Press enter to close...
```

Εικόνα 2 Output Προγράμματος SRP

2.2 Αρχή Ανοικτού – Κλειστού κώδικα (open closed principle)

Η αρχή ανοικτού κλεισίματος δηλώνει ότι "οι οντότητες λογισμικού πρέπει να είναι ανοιχτές για επέκταση, αλλά κλειστές για τροποποίηση". Δηλαδή, μια τέτοια οντότητα μπορεί να επιτρέψει την επέκτασή της χωρίς τροποποίηση του πηγαίου κώδικα (Open to extension – Closed to modification).

Έστω ένα πρόγραμμα διαχείρισης εισιτηρίων σε ένα ζωολογικό κήπο. Το πρόγραμμα λαμβάνει τα στοιχεία ενός επισκέπτη και του επιστρέφει ένα εισιτήριο ανάλογα με το ποσοστό έκπτωσης που δικαιούται. Το αρχικό διάγραμμα κλάσεων είναι το παρακάτω:



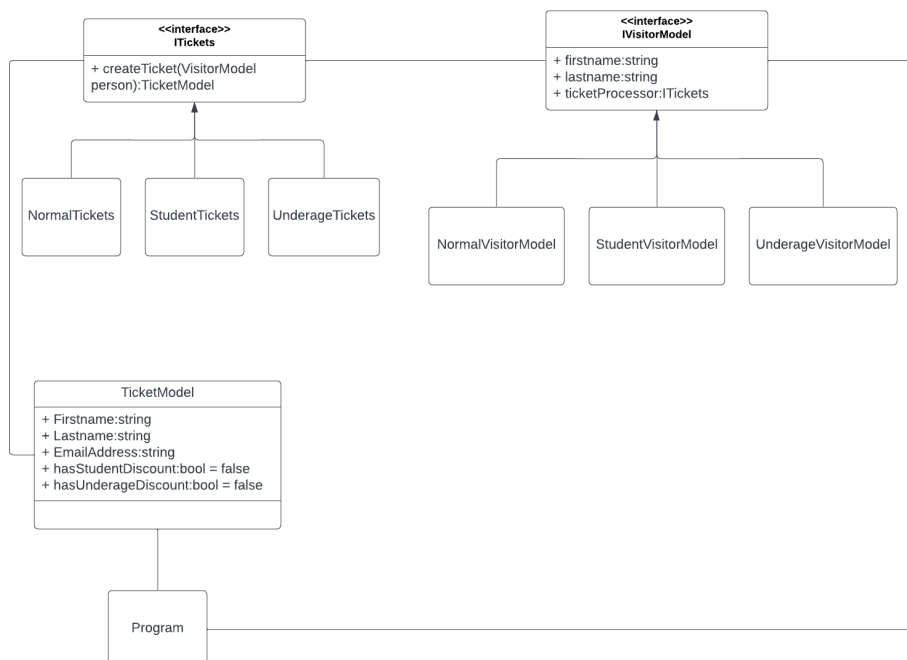
Εικόνα 3 Αρχικό class diagram πριν την εφαρμογή του OCP

Τι θα συμβεί όμως, εάν χρειάζεται να προστεθεί στο ήδη υλοποιημένο σύστημα εισιτηρίων, ένα καινούργιο είδος έκπτωσης;

Σε αυτή την περίπτωση, θα έπρεπε να τροποποιηθεί η μέθοδος createTicket στην κλάση Tickets, με μια δομή επανάληψης ή ένα switch statement, ώστε να πραγματοποιείται έλεγχος στην ιδιότητα του κάθε επισκέπτη (αν είναι φοιτητής, άτομο με ειδικές ανάγκες, κ.λπ). Έτσι, θα εισάγονταν αλλαγές στο υπάρχον σύστημα και παράλληλα η πιθανότητα να υπάρξουν λάθη (bugs) στον κώδικα αυξάνεται.

Μια λύση στο πρόβλημα είναι η χρήση των διεπαφών (interfaces). Οι διεπαφές (interfaces) επιτυγχάνουν την μείωση των εξαρτήσεων μεταξύ των κλάσεων. Άρα, με αυτό το τρόπο, γίνεται να προστεθούν αλλαγές στην υλοποίηση χωρίς να χρειαστεί να αλλάξει ο κορμός του προγράμματος.

Το πρόγραμμα με την εφαρμογή του OCP, περιγράφεται στο παρακάτω διάγραμμα κλάσεων:



Εικόνα 4 Τελικό class diagram μετά την εφαρμογή OCP



Επομένως, πλέον, στην περίπτωση που χρειάζεται να προστεθεί ένα νέο είδος έκπτωσης εισιτηρίου, θα δημιουργείται μια νέα κλάση που θα υλοποιεί την διεπαφή (interface) `ITickets`, η οποία θα καθορίζει το ποσοστό έκπτωσης και άλλη μια κλάση που υλοποιεί την διεπαφή (interface) `IVisitorModel`, η οποία περιέχει τα στοιχεία του νέου επισκέπτη και συσχετίζει πιο εισιτήριο ανήκει σε αυτόν (πχ. Στον `NormalVisitorModel` ανήκει το `NormalTicket`).

Output προγράμματος:

```
C:\Users\kkalo\OneDrive\Documents\GitHub\DesignPatterns\Final\SOLID\OCP\OCP_Demo\OCP_Demo\bin\Debug\OCP_Demo.exe
Kostas Kalogeropoulos: KKalogeropoulos@gmail.com hasUnderageDiscount: False hasStudentDiscount: False
Tom Storm: TStorm@gmail.com hasUnderageDiscount: False hasStudentDiscount: True
Nancy Roman: NRoman@gmail.com hasUnderageDiscount: True hasStudentDiscount: False
```

Εικόνα 5 Output προγράμματος OCP Demo

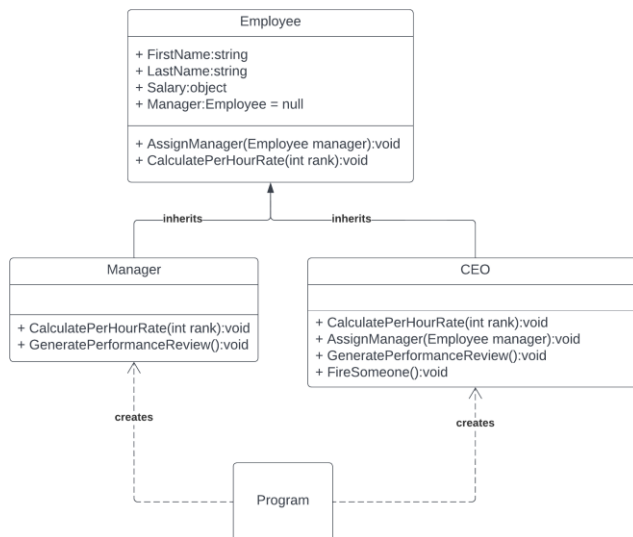
2.3 Αρχή αντικατάστασης της Liskov (Liskov substitution)

Η αρχή ορίζει ότι τα αντικείμενα μιας υπερκλάσης πρέπει να μπορούν να αντικατασταθούν με αντικείμενα των υποκλάσεων της χωρίς να προκύπτει κάποια εξαίρεση(exception) στην εφαρμογή. Αυτό απαιτεί τα αντικείμενα των υποκλάσεων να συμπεριφέρονται με τον ίδιο τρόπο όπως τα αντικείμενα των υπερκλάσεων τους.

Μια παρακαμφθείσα (overwritten) μέθοδος μιας υποκλάσης πρέπει να δέχεται τις ίδιες τιμές παραμέτρων εισόδου με τη μέθοδο της υπερκλάσης. Αυτό σημαίνει ότι γίνεται να εφαρμοστούν λιγότερο περιοριστικοί κανόνες (πχ. λιγότερες παραμέτρους), αλλά δεν επιτρέπεται να επιβάλλονται αυστηρότεροι. Διαφορετικά, οποιοσδήποτε κώδικας που καλεί αυτήν τη μέθοδο σε ένα αντικείμενο της υπερκλάσης μπορεί να προκαλέσει εξαίρεση(exception), εάν κληθεί με ένα αντικείμενο της υποκλάσης.

Παρόμοιοι κανόνες ισχύουν για την τιμή επιστροφής της μεθόδου. Η επιστρεφόμενη τιμή μιας μεθόδου της υποκλάσης πρέπει να συμμορφώνεται με τους ίδιους κανόνες με την τιμή επιστροφής της μεθόδου της υπερκλάσης.

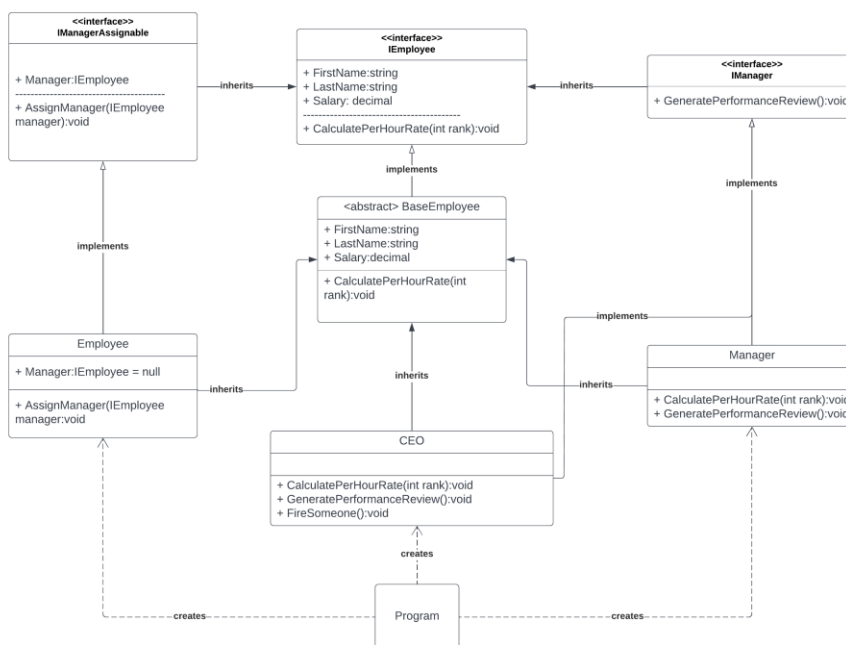
Έστω ένα πρόγραμμα διαχείρισης εργαζομένων μιας επιχείρησης που περιγράφεται στο παρακάτω διάγραμμα κλάσεων:



Εικόνα 6 Διάγραμμα κλάσεων αρχικού προγράμματος

Παρατηρείται ότι στην κλάση CEO, η μέθοδος AssignManager δεν έχει νόημα ύπαρξης καθώς ο CEO δεν είναι υπάλληλος ώστε να έχει προϊστάμενο. Έτσι, εάν δημιουργηθεί ένας νέος CEO μέσω του αντικείμενου της υπερκλάσης Employee, τότε θα προκύψει εξαίρεση (exception) στο πρόγραμμα. Με αυτό τον τρόπο παραβιάζεται η αρχή αντικατάστασης της Liskov.

Το πρόβλημα διορθώνεται εφαρμόζοντας διεπαφές (interfaces) και υλοποιώντας την κληρονομικότητα με τέτοιο τρόπο ώστε να μην προκύπτουν εξαιρέσεις (exceptions). Ακολουθεί το τελικό διάγραμμα κλάσεων:



Εικόνα 7 Διάγραμμα κλάσεων τελικού προγράμματος

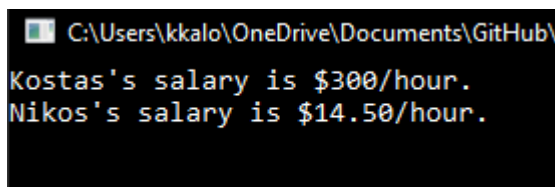
Παρατηρείται ότι οι κλάσεις Employee, CEO, Manager (υποκλάσεις) κληρονομούν από την κλάση BaseEmployee (υπερκλάση).

Η BaseEmployee δηλώθηκε ως abstract κλάση διότι δεν θέλουμε να γίνεται απευθείας χρήση (instantiation) της από το πρόγραμμα, παρόλο που δεν θα προκύπτει πρόβλημα με την ανάθεση αντικειμένου μέσω αυτής καθώς είναι η υπερκλάση.

Ακόμα, δημιουργήθηκαν οι διεπαφές (interfaces) IManagerAssignable, IManager και IEmployee ώστε να μειωθεί η εξάρτηση (coupling) μεταξύ των κλάσεων και να γίνει ξεκάθαρος διαχωρισμός των αρμοδιοτήτων.

Επομένως, με τις αλλαγές αυτές, δεν παραβιάζεται πλέον η αρχή του Liskov καθώς βελτιώθηκε η κληρονομικότητα σε σχέση με το αρχικό πρόγραμμα και τηρήθηκαν πιστά οι κανόνες του συγκεκριμένου προτύπου σχεδίασης.

Output προγράμματος:

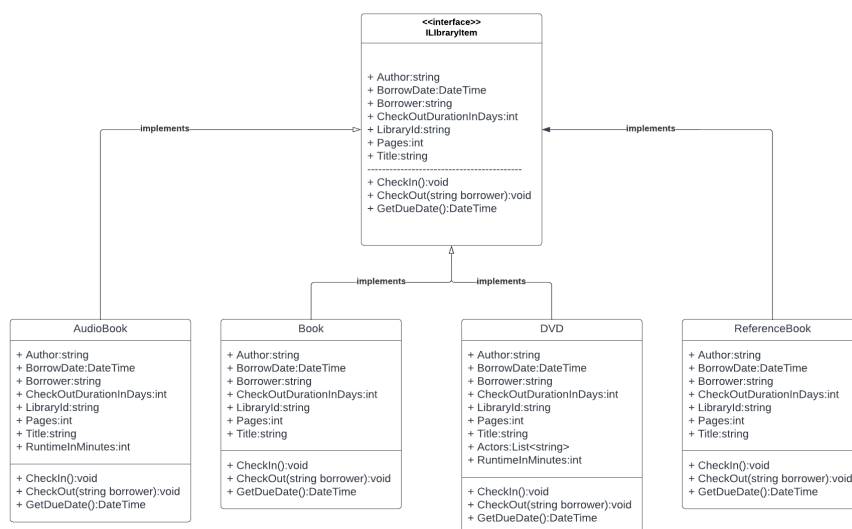


Εικόνα 8 Output προγράμματος LSP Demo

2.4 Αρχή διαχωρισμού με διεπαφές (Interface segregation) με πολλαπλή κληρονομικότητα

Η αρχή του διαχωρισμού διεπαφών δηλώνει ότι καμία κλάση δεν πρέπει να αναγκάζεται να εξαρτάται από διεπαφές (interfaces) που δεν χρησιμοποιεί.

Έστω ένα πρόγραμμα που προσομοιάζει την λειτουργία βιβλιοθήκης. Ακολουθεί το αρχικό διάγραμμα κλάσεων πριν την υλοποίηση του ISP.

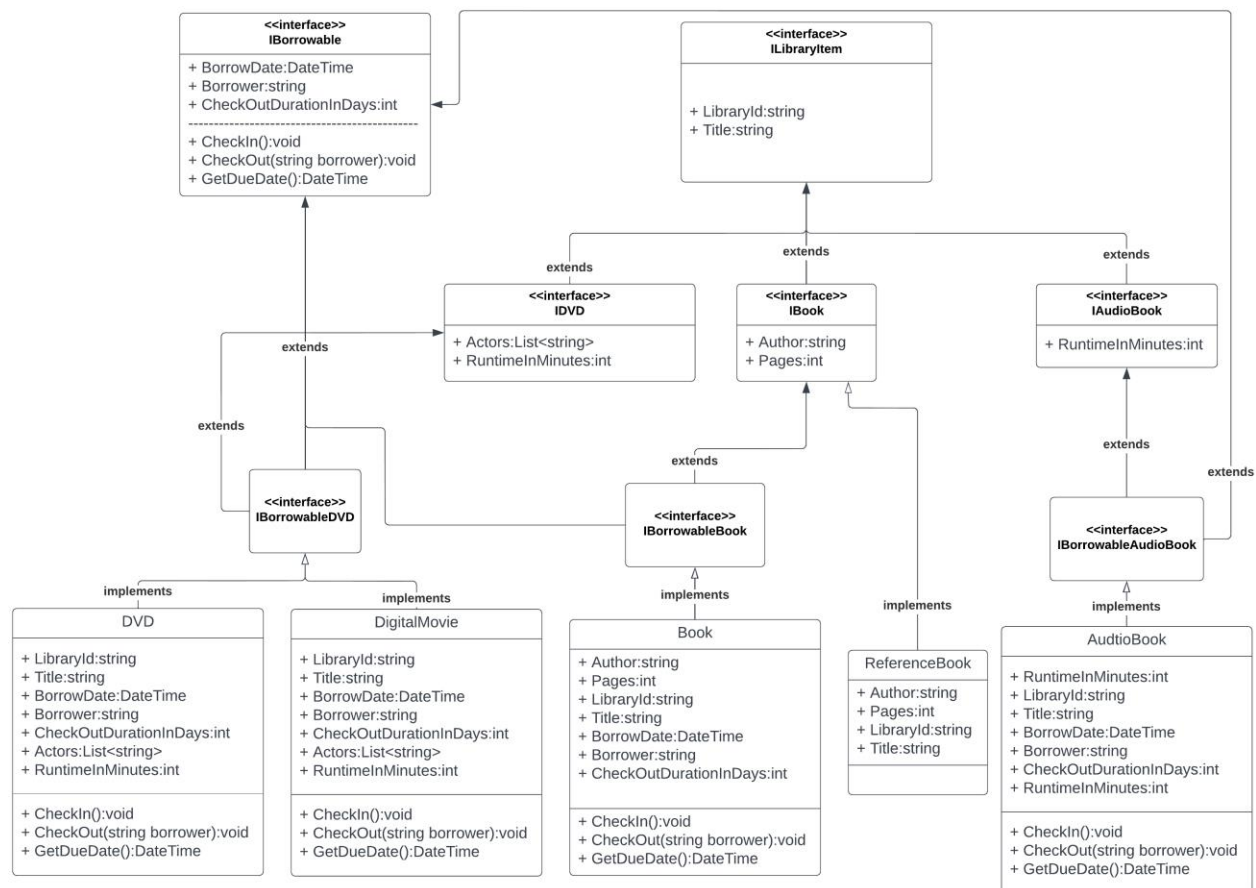


Εικόνα 9 Διάγραμμα Κλάσεων πριν την εφαρμογή του ISP

Παρατηρείται ότι όλες οι κλάσεις (AudioBook, Book, DVD, ReferenceBook) υλοποιούν μία κοινή διεπαφή (interface), την `ILibraryItem`. Όμως, σε αυτό το σημείο προκύπτουν τα προβλήματα εξάρτησης των κλάσεων από μεθόδους που δεν χρησιμοποιούν.

Πιο συγκεκριμένα, η κλάση `ReferenceBook`, αναγκάζεται να υλοποιήσει τις μεθόδους `CheckIn()`, `Checkout()` και να έχει τις ιδιότητες `Borrower` και `BorrowDate` ενώ το συγκεκριμένο βιβλίο δεν ανήκει σε αυτά τα οποία υπάρχει η δυνατότητα δανεισμού. Τα ίδια προβλήματα δημιουργούνται και στις υπόλοιπες κατηγορίες προϊόντων και υπηρεσιών.

Η λύση προέρχεται με τον διαχωρισμό της διεπαφής (interface) `ILibraryItem` σε υποκατηγορίες διεπαφών, οι οποίες αναφέρονται στο αντίστοιχο προϊόν και υπηρεσία της βιβλιοθήκης. Επίσης, δημιουργούνται νέες διεπαφές που ομαδοποιούν τις λειτουργίες πολλών ξεχωριστών διεπαφών, μέσω της κληρονομικότητας. Ακολουθεί το διάγραμμα κλάσεων μετά την εφαρμογή του ISP.



Εικόνα 10 Διάγραμμα κλάσεων μετά την εφαρμογή του ISP

Παρατηρείται ότι πλέον, τα προϊόντα και οι υπηρεσίες της βιβλιοθήκης, υλοποιούν τις διεπαφές που αναφέρονται σε αυτά. Με αυτό τον τρόπο, κανένα δεν εξαρτάται από τις ιδιότητες κάποιου άλλου και επιτυγχάνεται μια ανεξαρτησία μεταξύ τους.

Output προγράμματος:

```
Select C:\Users\kkalo\OneDrive\Documents\GitHub\DesignPatterns\Final\SOLID\ISP\ISP_Demo\ISP_Demo\bin\Debug\ISP
DVD borrowing details loading...
Borrower: Kostas, BorrowDate: 7/8/2022 10:21:48 PM, Title: Learning Design Patterns
Book borrowing details loading...
Borrower: Kostas, BorrowDate: 7/8/2022 10:21:48 PM, Title: Harry Potter
Movie borrowing details loading...
Borrower: Kostas, BorrowDate: 7/8/2022 10:21:48 PM, Title: James Bond part1
Audio borrowing details loading...
Borrower: Kostas, BorrowDate: 7/8/2022 10:21:48 PM, Title: Sleeping sounds 24/7
Reference Book created. Title: SOLID Principles, Library Id: 213
```

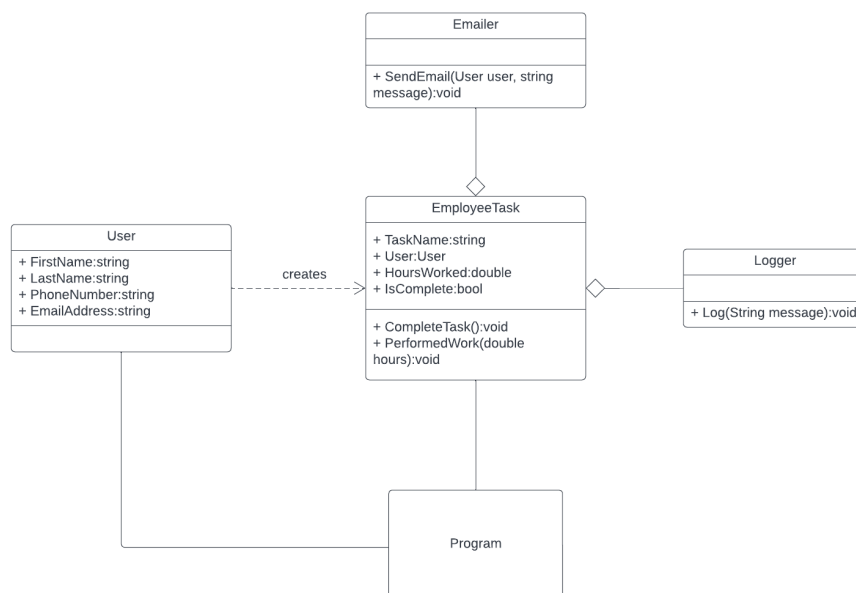
Εικόνα 11 Output προγράμματος ISP_Demo

2.5 Αρχή αντιστροφής εξαρτήσεων (Dependency Inversion)

Στον αντικειμενοστραφή σχεδιασμό, η αρχή της αντιστροφής εξάρτησης είναι μια συγκεκριμένη μεθοδολογία για χαλαρή σύζευξη μονάδων λογισμικού (low coupling between classes).

Η Αρχή Αντιστροφής Εξαρτήσεων (DIP) δηλώνει ότι οι μονάδες (modules) υψηλού επιπέδου δεν πρέπει να εξαρτώνται από μονάδες χαμηλού επιπέδου. Και οι δύο πρέπει να εξαρτώνται από αφαιρέσεις (abstractions). Οι αφαιρέσεις (abstractions) δεν πρέπει να εξαρτώνται από λεπτομέρειες. Οι λεπτομέρειες πρέπει να εξαρτώνται από αφαιρέσεις (abstractions).

Έστω ένα πρόγραμμα το οποίο προσομοιάζει την αποστολή email από έναν χρήστη. Το αρχικό διάγραμμα κλάσεων πριν την εφαρμογή του DIP είναι το παρακάτω:



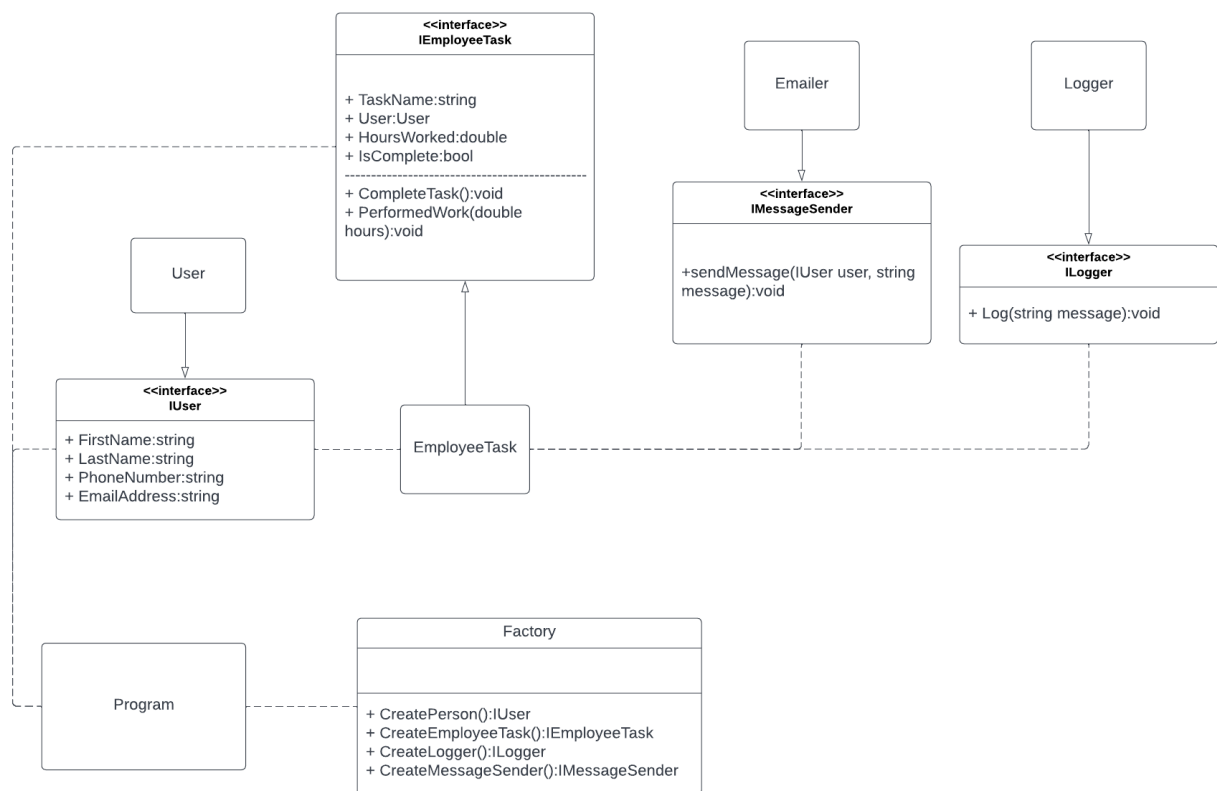
Εικόνα 12 Διάγραμμα κλάσεων πριν την εφαρμογή του DIP



Παρατηρείται ότι η κλάση EmployeeTask, μία υψηλού επιπέδου κλάση, που εκτελεί την διαδικασία αποστολής ενός email, εξαρτάται από την κλάση EMailer και την κλάση Logger, οι οποίες είναι χαμηλού επιπέδου κλάσεις.

Σύμφωνα με την αρχή DIP, και οι υψηλού και οι χαμηλού επιπέδου κλάσεις πρέπει να βασίζονται σε αφαιρέσεις (abstractions) και οι αφαιρέσεις (abstractions) δεν πρέπει να εξαρτώνται από λεπτομέρειες, εννοώντας ότι δεν μας αφορά το πώς θα υλοποιηθούν οι λειτουργίες αλλά το ότι *πρόκειται να γίνουν*. Ένας τρόπος για να υλοποιηθούν οι αφαιρέσεις (abstractions) είναι με την χρήση διεπαφών (interfaces).

Με βάση τα προηγούμενα, το νέο διάγραμμα κλάσεων που προκύπτει μετά την εφαρμογή του DIP είναι το παρακάτω:



Εικόνα 13 Διάγραμμα κλάσεων μετά την εφαρμογή του DIP

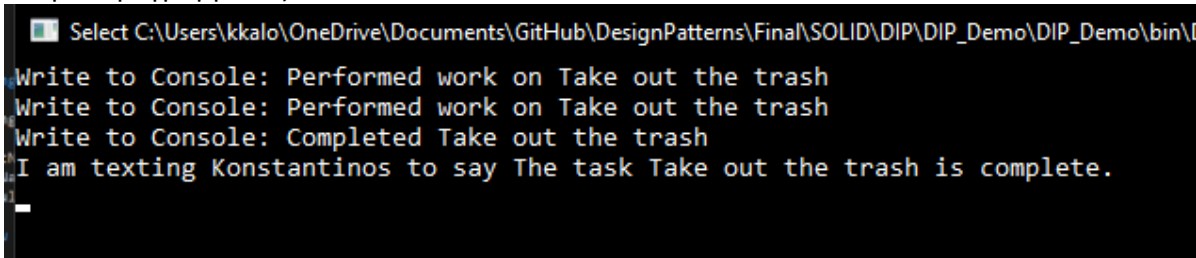
Παρατηρείται, ότι πλέον καμία κλάση δεν εξαρτάται από καμία άλλη με άμεσο τρόπο. Αντίθετα, όλες οι κλάσεις πλέον αναφέρονται σε αφαιρέσεις (abstractions) που επιτυγχάνονται μέσω των διεπαφών (interfaces). Κάποια από τα πολύ σημαντικά πλεονεκτήματα που δίνονται χάρη στην χρήση των διεπαφών (interfaces), είναι η καλύτερη συντηρησιμότητα (maintainability) και οι δυνατότητες εξέλιξης του προγράμματος.



Για παράδειγμα, εάν χρειάζονταν να γίνει αλλαγή στο υλοποιημένο σύστημα ανταλλαγής μηνυμάτων από Email σε Texter(άμεσο chatting), θα υπήρχε η δυνατότητα να δημιουργηθεί μία κλάση Texter που να υλοποιεί το interface IMessageSender και η μόνη αλλαγή που θα χρειάζονταν να γίνει, θα συνέβαινε στην κλάση Factory. (Πιο συγκεκριμένα, θα άλλαζε ο τύπος αντικείμενου που επιστρέφεται στην μέθοδο CreateMessageSender() από EMailer σε Texter).

Επομένως, γίνεται φανερό ότι η χαμηλή σύζευξη μεταξύ των κλάσεων που επιτυγχάνεται μέσω την διεπαφών (interfaces) είναι πολύ σημαντική στην επεκτασιμότητα του προγράμματος.

Output προγράμματος:



Εικόνα 14 Output προγράμματος DIP Demo

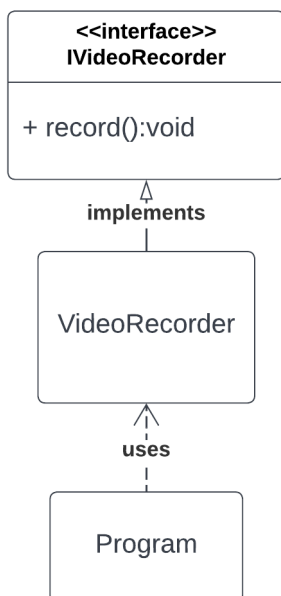
2.6 Πρότυπο Σχεδίασης Adapter

Το πρότυπο σχεδίασης Adapter λειτουργεί ως γέφυρα μεταξύ δυο ανεξάρτητων ή μη συμβατών μεταξύ τους διεπαφών (interfaces).

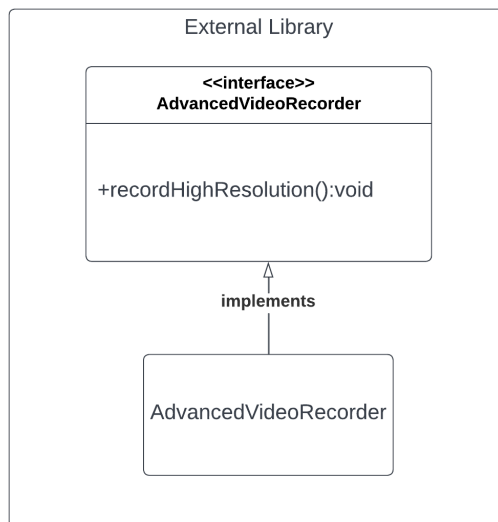
Το πρότυπο αυτό, εμπεριέχει μια κλάση η οποία είναι υπεύθυνη για την ομαδοποίηση των λειτουργιών μεταξύ των ανεξάρτητων ή μη συμβατών διεπαφών (interfaces). Για παράδειγμα, ένα θέλουμε να χρησιμοποιήσουμε μια εξωτερική βιβλιοθήκη στο πρόγραμμα μας, μπορούμε να κάνουμε χρήση του προτύπου adapter.

Ένα παράδειγμα μέσα από την αληθινή ζωή, θα μπορούσε να είναι η χρήση ενός φορτιστή για το κινητό μας σε χώρες με διαφορετικό πρότυπο ηλεκτρικής πρίζας(Αγγλία/Γαλλία). Έτσι, για να τοποθετήσουμε τον φορτιστή σε μια ηλεκτρική πρίζα, θα χρειαζόμασταν έναν αντάπτορα, δηλαδή μια συσκευή που μετέτρεπε τον φορτιστή μας συμβατό με το εκάστοτε πρότυπο πρίζας κάθε χώρας. Την ίδια λογική ακολουθεί το πρότυπο adapter και στο λογισμικό μας.

Έστω ένα πρόγραμμα που καταγράφει βίντεο από μία οθόνη ενός υπολογιστή. Το πρόγραμμα έχει την δυνατότητα να βιντεοσκοπεί σε ανάλυση 1080p (Full HD) αλλά ο πελάτης χρειάζεται και την δυνατότητα καταγραφής σε ανάλυση 2160p (4K). Έτσι, θα χρειαστεί να προστεθεί αυτή η δυνατότητα από μία εξωτερική βιβλιοθήκη που την έχει υλοποιήσει.

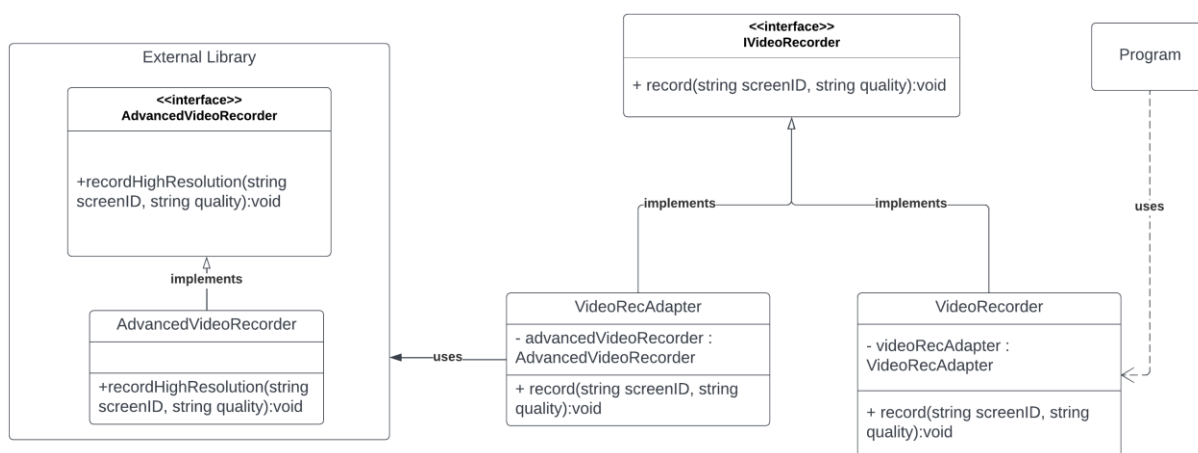


Εικόνα 15 Αρχικό πρόγραμμα



Εικόνα 16 External Library

Πώς θα «ενωθεί» το πρόγραμμα καταγραφής βίντεο με την εξωτερική βιβλιοθήκη;
Για να γίνει αυτό, θα χρειαστεί να υλοποιηθεί το πρότυπο σχεδίασης Adapter.

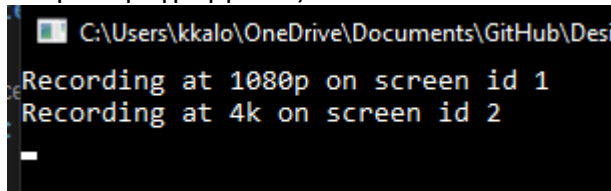


Εικόνα 17 Το διάγραμμα κλάσεων μετά την εφαρμογή του Adapter Pattern

Παρατηρείται ότι δημιουργήθηκε μια κλάση VideoRecAdapter, η οποία προσαρμόζει την εξωτερική βιβλιοθήκη στην υπάρχουσα υλοποίηση του προγράμματος. Η κλάση αυτή ομαδοποιεί τις λειτουργίες της βιβλιοθήκης AdvancedVideoRecorder και τις προσαρμόζει στις

μεθόδους του προγράμματος. Αυτό συμβαίνει διότι υλοποιεί την ίδια διεπαφή (interface) με την κλάση VideoRecorder, οπότε με αυτό τον τρόπο υποχρεώνεται να διατηρεί την ίδια λογική του προγράμματος. Επομένως, η χρήση του Adapter είναι πολύ σημαντική για να επεκταθούν οι δυνατότητες ενός λογισμικού, χωρίς να τροποποιηθεί η δομή του σε μεγάλο βαθμό.

Output προγράμματος:



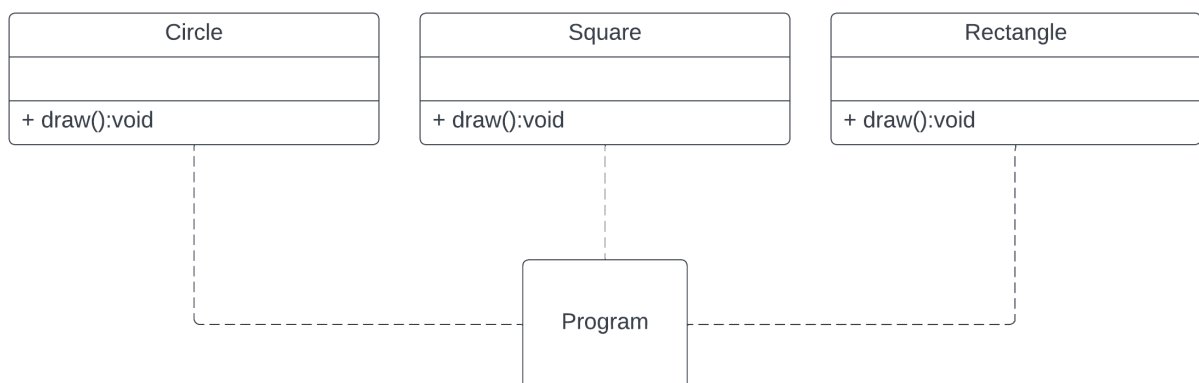
Εικόνα 18 Output προγράμματος AdapterDemo

2.7 Πρότυπο Σχεδίασης Factory

Το πρότυπο σχεδίασης Factory είναι από τα πιο γνωστά και παρέχει τους καλύτερους τρόπους δημιουργίας ενός αντικειμένου.

Πιο αναλυτικά, το Factory pattern, επιτρέπει σε μια κλάση Factory να παράγει αντικείμενα άλλων κλάσεων. Με αυτό τον τρόπο αποφεύγεται η άμεση αρχικοποίηση ενός αντικειμένου μίας κλάσης B μέσα σε μία κλάση A καθώς η κλάση Factory αναλαμβάνει την αρχικοποίηση όλων των αντικειμένων. Έτσι, μειώνεται η εξάρτηση μεταξύ των κλάσεων, καθιστώντας με αυτόν τον τρόπο, το πρόγραμμα πιο ευέλικτο και επεκτάσιμο.

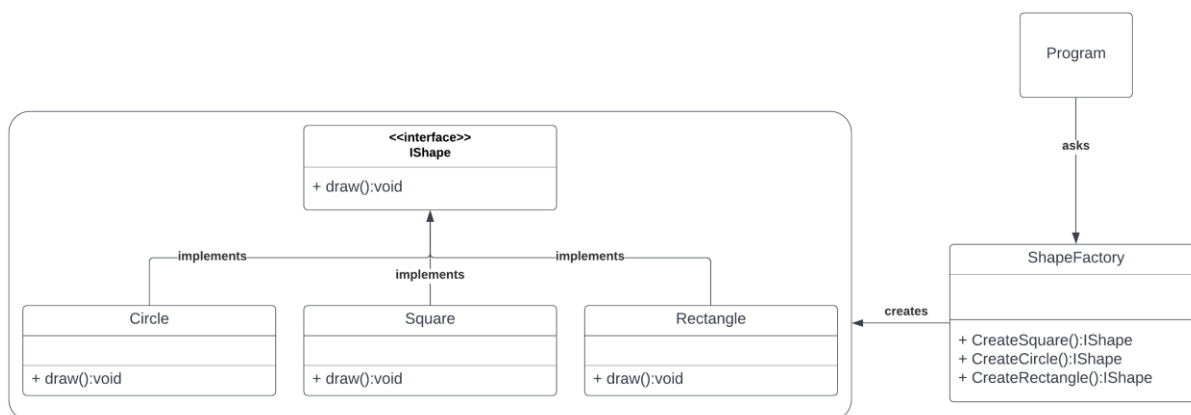
Έστω ένα πρόγραμμα που προσομοιάζει την δημιουργία γεωμετρικών σχημάτων (κύκλος, τετράγωνο, ορθογώνιο). Το αρχικό διάγραμμα κλάσεων έχει ως εξής:



Εικόνα 19 Διάγραμμα κλάσεων πριν την εφαρμογή του Factory Pattern

Παρατηρείται ότι το πρόγραμμα για να δημιουργήσει ένα νέο γεωμετρικό σχήμα, πρέπει κάθε φορά να δημιουργεί ένα νέο αντικείμενο της εκάστοτε κλάσης, εκθέτοντας με αυτόν τον τρόπο όλες τις πληροφορίες στον πελάτη (client). Αυτό διορθώνεται εύκολα με το πρότυπο σχεδίασης Factory.

Το τελικό διάγραμμα κλάσεων έχει ως εξής:



Εικόνα 20 Διάγραμμα κλάσεων μετά την εφαρμογή του Factory Pattern

Παρατηρείται ότι δημιουργήθηκε μία διεπαφή (interface) IShape, την οποία υλοποιούν όλα τα σχήματα. Με αυτόν τον τρόπο, γίνεται εύκολο για την κλάση ShapeFactory να περικλείσει (encapsulate) όλα τα σχήματα ώστε κάθε φορά να δημιουργεί το σωστό σχήμα που ζητάει ο πελάτης. Αυτό έχει ως αποτέλεσμα, επειδή όλα τα σχήματα είναι τύπου IShape, να γενικεύεται η δημιουργία του κάθε σχήματος, εκθέτοντας έτσι λιγότερες πληροφορίες στον πελάτη (client). Επομένως, ο αρχικός στόχος της υλοποίησης έχει επιτευχθεί.

Output προγράμματος:

```
C:\Users\kkalo\OneDrive\Documents\GitHub\DesignP
Inside Circle::draw() method.
Inside Rectangle::draw() method.
Inside Square::draw() method.
```

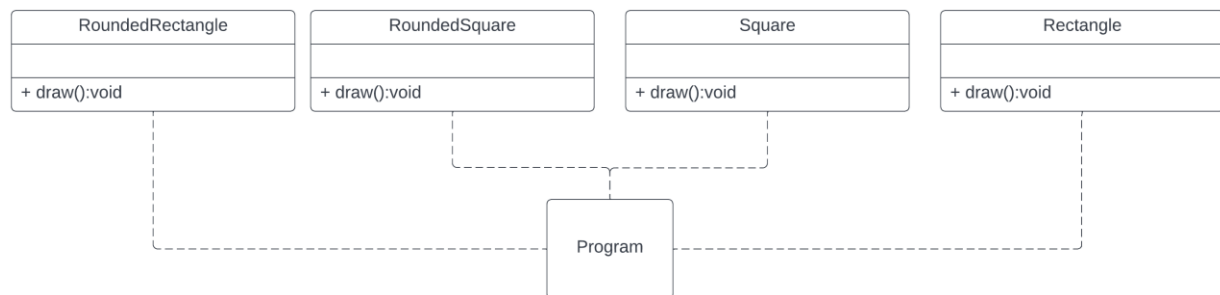
Εικόνα 21 Output Προγράμματος Factory Pattern

2.8 Πρότυπο Σχεδίασης Abstract Factory

Το πρότυπο Abstract Factory αναφέρεται στην δημιουργία αντικείμενων των κλάσεων.

Σε αντίθεση με το απλό Factory pattern, το Abstract Factory αποτελείται από μια υπερκλάση FactoryProducer, η οποία δημιουργεί άλλες υποκλάσεις Factory που παράγουν αντικείμενα ενός συνόλου κλάσεων.

Έστω ένα πρόγραμμα που προσομοιάζει την δημιουργία γεωμετρικών σχημάτων (κύκλος, τετράγωνο, ορθογώνιο), όμως αυτή την φορά, δημιουργεί και σχήματα με ορισμένες ιδιότητες (rounded corners). Το αρχικό διάγραμμα κλάσεων έχει ως εξής:



Εικόνα 22 Διάγραμμα κλάσεων πριν την εφαρμογή του Abstract Factory Pattern

Παρατηρείται ότι το πρόγραμμα για να δημιουργήσει ένα νέο γεωμετρικό σχήμα, πρέπει κάθε φορά να δημιουργεί ένα νέο αντικείμενο της εκάστοτε κλάσης, εκθέτοντας με αυτόν τον τρόπο όλες τις πληροφορίες στον πελάτη (client).

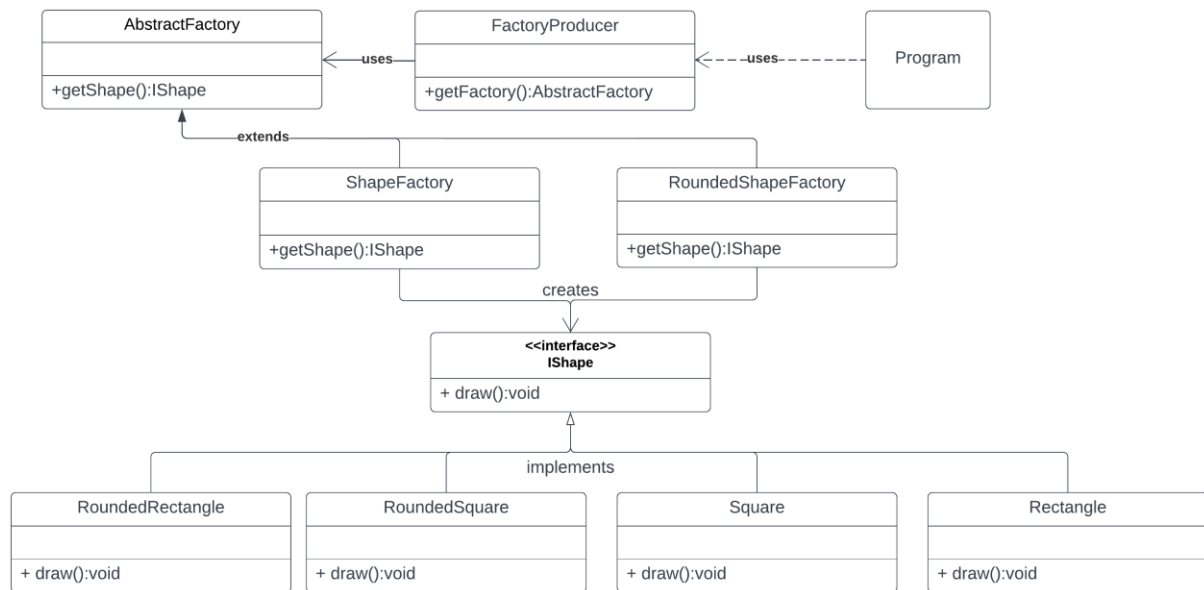
Για να εφαρμοστεί το Abstract Factory Pattern, αρχικά θα δημιουργηθεί μια διεπαφή (interface) IShape, την οποία θα υλοποιούν όλα τα σχήματα (Square, Rectangle, RoundedRectangle, RoundedSquare).

Στην συνέχεια, θα δημιουργηθεί η αφηρημένη (abstract) κλάση AbstractFactory, η οποία θα κληρονομείται από τις κλάσεις ShapeFactory και RoundedShapeFactory.

Έπειτα, δημιουργείται η κλάση FactoryProducer, υπεύθυνη για την δημιουργία των factories των σχημάτων (ShapeFactory, RoundedShapeFactory). Η κλάση αυτή (FactoryProducer), χρησιμοποιεί το αντικείμενο της AbstractFactory, ως μέσο μετάδοσης πληροφορίας για την δημιουργία του κάθε factory.

Με άλλα λόγια, η κλάση FactoryProducer είναι υπεύθυνη για την αρχικοποίηση όλων των factories και για να το καταφέρει αυτό, χρησιμοποιεί την κλάση AbstractFactory (την οποία κληρονομούν τα factories) ώστε να επικοινωνήσει μαζί τους.

Επομένως, ο χρήστης για να υλοποιήσει τα διάφορα σχήματα (Square, Rectangle, RoundedRectangle, RoundedSquare), πρώτα αρχικοποιεί το factory που τα παράγει (ShapeFactory ή RoundedShapeFactory) και τέλος μέσω του αντίστοιχου factory για κάθε σχήμα, τα επιλέγει προς σχεδίαση.



Εικόνα 23 Διάγραμμα κλάσεων μετά την εφαρμογή του Abstract Factory Pattern

Output προγράμματος:

```
C:\Users\kkalo\OneDrive\Documents\GitHub\DesignPa
Inside Rectangle::draw() method.
Inside Square::draw() method.
Inside RoundedRectangle::draw() method.
Inside RoundedSquare::draw() method.
```

Εικόνα 24 Output Προγράμματος Abstract Factory



3 Cloud Design Patterns

3.1 Health Endpoint Monitoring Pattern

Εισαγωγή

Είναι καλή πρακτική, και συνήθως απαιτούμενο στην παραγωγή, η δυνατότητα επίβλεψης web εφαρμογών και των back-end υπηρεσιών, ώστε να διασφαλιστεί η σωστή και ορθή λειτουργία τους.

Το Health Endpoint Monitoring Pattern αναφέρεται στην δημιουργία και υλοποίηση συγκεκριμένων σημείων (endpoints), μέσω των οποίων δίνεται η δυνατότητα στον προγραμματιστή να ελέγχει την «υγεία» του προγράμματος. Πιο συγκεκριμένα, υπάρχει μία ορισμένη χρονική περίοδος για την οποία παράγονται δεδομένα που ανιχνεύουν την συμπεριφορά ενός συνόλου υπηρεσιών ως προς την αξιοπιστία τους(πχ. σύνολο σφαλμάτων), τον χρόνο λειτουργίας τους (uptime availability) και άλλες λειτουργίες που κρίνονται κρίσιμες για συχνή παρακολούθηση.

Περιγραφή προβλήματος

Υπάρχουν πολλοί παράγοντες που επηρεάζουν την λειτουργία των εφαρμογών που φιλοξενούνται στο cloud. Κάποιοι από αυτούς είναι η καθυστέρηση του δικτύου (network latency), η απόδοση και η διαθεσιμότητα των hardware συστημάτων που βρίσκονται κάτω από αυτό, καθώς και η σύνδεση μεταξύ των δύο. Η εφαρμογή μπορεί να αποτύχει μερικώς ή εντελώς για κάποιον από τους προηγούμενους λόγους.

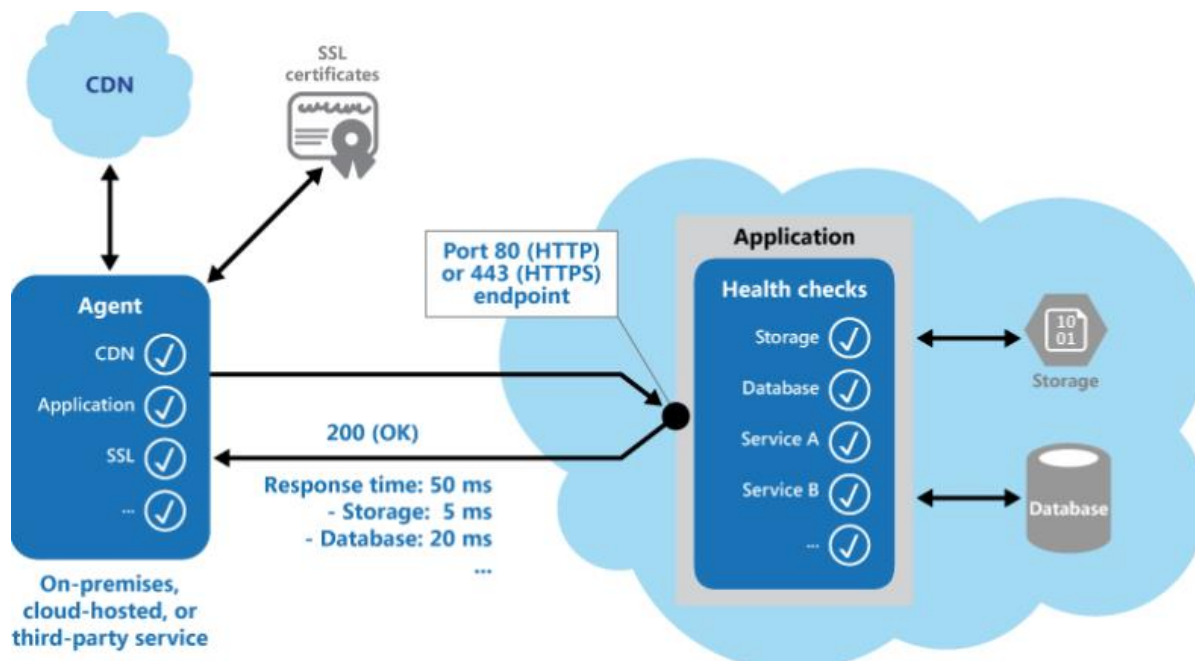
Επομένως, πρέπει αν τακτά χρονικά διαστήματα να πραγματοποιείται έλεγχος σε διάφορα σημεία της εφαρμογής, ώστε να διασφαλιστεί η ορθή λειτουργία της, πράγμα το οποίο απαιτείται συχνά στην συμφωνία που αφορά τις υπηρεσίες που παρέχονται (Service Level Agreement - SLA).

Περιγραφή Λύσης

Σε αυτό το σημείο, λαμβάνει χώρα η υλοποίηση του Health Endpoint Monitoring Pattern. Η εφαρμογή στέλνει ένα αίτημα σε ορισμένα σημεία, τα οποία της επιτρέπουν να πραγματοποιεί τους απαραίτητους ελέγχους και να επιστρέφουν ανά πάσα στιγμή την γενική κατάσταση που επικρατεί μέσω αυτών.

Συχνά, ένας έλεγχος αποτελείται από δύο φάσεις. Η πρώτη φάση είναι οι έλεγχοι που πραγματοποιούνται σε σχέση με το αντίστοιχο αίτημα (requests). Η δεύτερη φάση, είναι η ανάλυση των αποτελεσμάτων που παράγονται από τα εργαλεία που πραγματοποιούν τον έλεγχο, ώστε να εξαχθούν να αντίστοιχα συμπεράσματα.

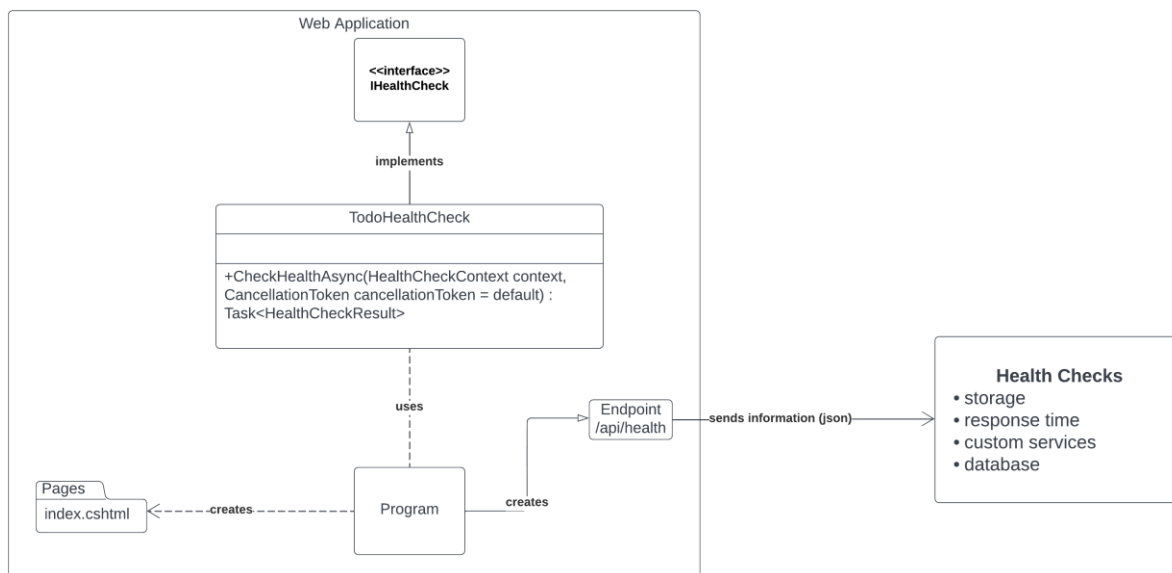
Το παρακάτω διάγραμμα απεικονίζει μία σύνοψη του Health Endpoint Monitoring Pattern.



Εικόνα 25 Σύνοψη του Health Endpoint Monitoring Pattern

Επιπλέον, κάποιοι από τους τυπικούς ελέγχους που πραγματοποιούνται από εργαλεία επίβλεψης περιλαμβάνουν:

1. Ανάλυση των κωδικών που επιστρέφονται. Για παράδειγμα, μία HTTP απάντηση με κωδικό 200 (OK) υποδεικνύει ότι η εφαρμογή αποκρίθηκε χωρίς κάποιο πρόβλημα.
2. Ενδελεχής έλεγχος των ειδοποιήσεων (warnings) που επιστρέφεται ώστε να εντοπιστούν πιθανά λάθη, ακόμα και αν δεν επηρεάζουν την εκτέλεση την εφαρμογή μας (HTTP 200 - OK). Για παράδειγμα, λάθη που αφορούν μόνο ένα μέρος την εφαρμογής όπως ο τίτλος μιας συγκεκριμένης σελίδας να μην είναι σωστός σε σχέση με τις υπόλοιπες.
3. Έλεγχος αν έχει λήξει κάποιο πιστοποιητικό SSL (Πιστοποιητικό ασφαλείας που υποδεικνύεται στον επισκέπτη της σελίδας).
4. Έλεγχος του χρόνου απόκρισης της σελίδας ώστε να βρεθούν διάφορες καθυστερήσεις του δικτύου.
5. Πιστοποίηση ότι η σελίδα που επιστρέφεται μετά από ένα DNS lookup είναι η πράγματι η σωστή. Έτσι, αυτό θα συμβάλει στην αποφυγή ενός κακόβουλο αιτήματος μετά από μία επιτυχημένη επίθεση στον DNS server.



Εικόνα 26 Διάγραμμα Κλάσεων για το Health Endpoint Monitoring Pattern σε εφαρμογή του ASP.NET Core

Πότε πρέπει να γίνεται χρήση του προτύπου;

Το πρότυπο αυτό είναι χρήσιμο για:

- Επίβλεψη ιστοσελίδων και web εφαρμογών για να ελέγχεται η διαθεσιμότητα τους συνεχώς.
- Γενική επίβλεψη web εφαρμογών για την ορθή λειτουργία τους.
- Επίβλεψη υπηρεσιών που σχετίζονται με την web εφαρμογή (middle-tier or shared services), ώστε περίπτωση αποτυχίας κάποιας από αυτές, να απομονώνεται επιτυχώς από το υπόλοιπο σύστημα ώστε να μην διαταράζεται η λειτουργία του.
- Συνεργασία με υπάρχοντα logging και error handling συστήματα. Το πρότυπο αυτό δεν αντικαθιστά απαραίτητα τα παραπάνω συστήματα αλλά λειτουργεί ως επιπλέον μέσο ελέγχου μιας web εφαρμογής.

3.2 Competing Consumers Pattern

Εισαγωγή

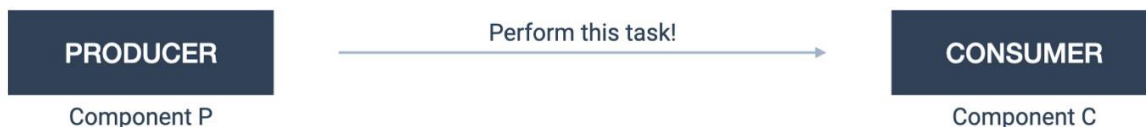
Το Competing Consumers pattern αναφέρεται στο πώς πολλά υποσυστήματα ανταγωνίζονται στο ίδιο κανάλι επικοινωνίας για να αποστείλουν και επεξεργαστούν πολλά μηνύματα ταυτόχρονα.

Με άλλα λόγια, το πρότυπο δίνει τη δυνατότητα σε ένα σύστημα να επεξεργάζεται πολλαπλά μηνύματα ταυτόχρονα για να βελτιστοποιήσει την απόδοση, την επεκτασιμότητα, την διαθεσιμότητα και εξισορρόπηση του φόρτου εργασίας.

Τα συστήματα που στέλνουν τις εργασίες προς υλοποίηση ονομάζονται παραγωγοί και τα συστήματα που αναλαμβάνουν την υλοποίηση, ονομάζονται καταναλωτές.

Περιγραφή προβλήματος

Για παράδειγμα, έστω ένα σύστημα P, το οποίο ζητάει από ένα σύστημα C να εκτελέσει μια εργασία, η οποία συνήθως παίρνει 5 λεπτά να ολοκληρωθεί κατά μέσο όρο.

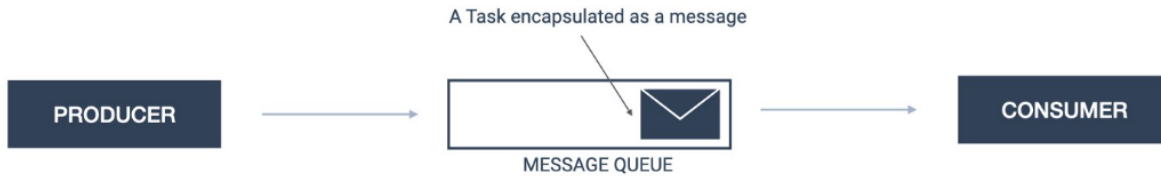


Εικόνα 27 Ένα σύστημα P καλεί ένα σύστημα C για να εκτελέσει ένα task

Η ύπαρξη μίας σύγχρονης επικοινωνίας (synchronous communication) μεταξύ του P και του C, δεν αποτελεί μια καλή επιλογή για διάφορους λόγους.

Πιο συγκεκριμένα, το σύστημα P θα ήταν μπλοκαρισμένο για να εκτελέσει άλλες εργασίες μέχρι το σύστημα C να ολοκληρώσει αυτό που του ανατέθηκε. Επιπλέον, μια εργασία που χρειάζεται 5 λεπτά για να ολοκληρωθεί είναι ένα μεγάλο χρονικό διάστημα για ένα σύντομο HTTP request.

Μια προτεινόμενη λύση θα ήταν να κάνουμε την επικοινωνία ασύγχρονη υλοποιώντας μια ουρά μηνυμάτων/εντολών ανάμεσα στο P και στο C. Το P περικλείει μια εργασία ως ένα μήνυμα και το στέλνει στην ουρά μηνυμάτων. Το C λαμβάνει τις εργασίες από την ουρά μηνυμάτων και τις εκτελεί με ασύγχρονο τρόπο. Έτσι, το P δεν θα είναι μπλοκαρισμένο όσο το C υλοποιεί μια εργασία.



Εικόνα 28 Το P περικλείει την εργασία ως μήνυμα και την αποστέλλει στην ουρά μηνυμάτων. Έπειτα το C λαμβάνει την ουρά και την εκτελεί τις εργασίες ασύγχρονα.

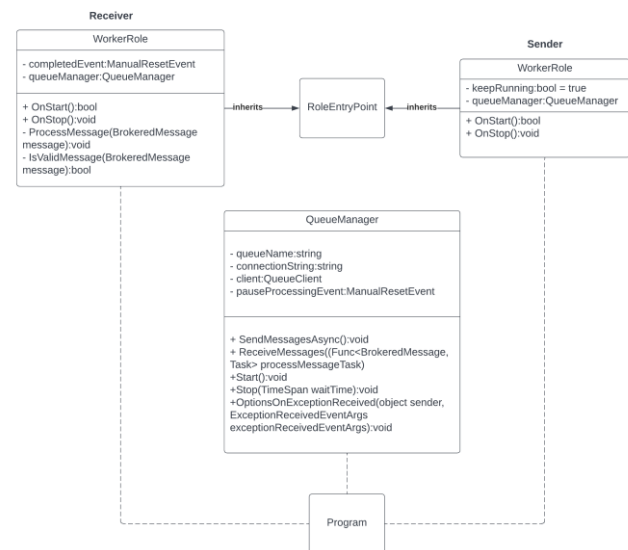
Ωστόσο, έχοντας μόνο ένα σύστημα C δεν θεωρείται ως μία επεκτάσιμη λύση. Εάν το C καταρρεύσει, δεν θα υπάρχει άλλο να το αντικαταστήσει και να αναλάβει αυτά που του έχουν ανατεθεί. Επίσης, το C χρειάζεται να φτάσει το ρυθμό στον οποίο το P τοποθετεί μηνύματα στην ουρά προς υλοποίηση. Αν μία εργασία χρειάζεται 5 λεπτά για να ολοκληρωθεί, τι θα συμβεί εάν 100.000 εργασίες είναι στην αναμονή προς υλοποίηση; Θα χρειάζονταν μέρες.

Πως γίνεται να επεκτείνουμε σε μεγαλύτερο βαθμό την παραπάνω λύση βελτιώνοντας την απόδοση και την διαθεσιμότητα του συστήματος;

Περιγραφή Λύσης

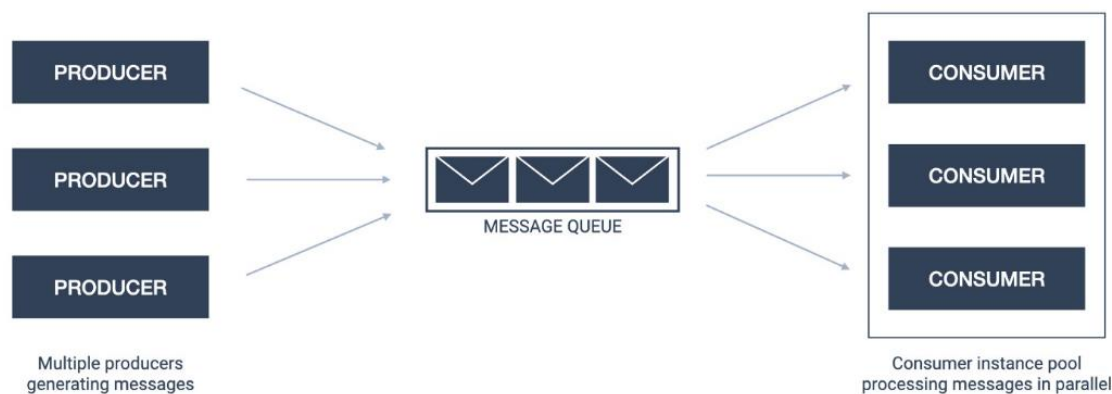
Το πρότυπο σχεδίασης Competing Consumers παρέχει την δυνατότητα σε πολλαπλά υποσυστήματα να υλοποιούν ταυτόχρονα τις εργασίες που λαμβάνουν, στο ίδιο όμως κανάλι επικοινωνίας.

Στο προηγούμενο παράδειγμα, γίνεται να έχουμε πολλά υποσυστήματα σαν το C, τα οποία να ανταγωνίζονται για μηνύματα (τα μηνύματα περικλείουν εργασίες) στην ίδια ουρά μηνυμάτων. Έτσι, θα γίνεται επεξεργασία πολλών μηνυμάτων ταυτόχρονα, ώστε να αδειάσει η ουρά πιο γρήγορα.



Εικόνα 29 Διάγραμμα κλάσεων που περιγράφει το Competing Consumers demo application

Όταν ένα μήνυμα γίνεται διαθέσιμο προς υλοποίηση στην ουρά μηνυμάτων, τότε οποιοδήποτε υποσύστημα θα μπορούσε να το αναλάβει. Το σύστημα μετάδοσης των μηνυμάτων καθορίζει ποιο υποσύστημα θα αναλάβει την εργασία, όμως παράλληλα, τα υποσυστήματα ανταγωνίζονται μεταξύ τους για το ποιο θα γίνει ο ανάδοχος του μηνύματος.



Εικόνα 30 Πρότυπο Σχεδίασης Competing Consumers

Output προγράμματος:

```
PS C:\Users\kkalo\OneDrive\Documents\GitHub\DesignPatterns\Fin
mo> cd .\Producer
PS C:\Users\kkalo\OneDrive\Documents\GitHub\DesignPatterns\Fin
mo\Producer> dotnet run
Send message: Sending Message Id: 1
Send message: Sending Message Id: 2
Send message: Sending Message Id: 3
Send message: Sending Message Id: 4
Send message: Sending Message Id: 5
Send message: Sending Message Id: 6
Send message: Sending Message Id: 7
Send message: Sending Message Id: 8
Send message: Sending Message Id: 9
Send message: Sending Message Id: 10
```

Εικόνα 32 Δημιουργία μηνυμάτων και αποστολή στην ουρά (queue)

```
PS C:\Users\kkalo\OneDrive\Documents\GitHub\DesignPatterns\Fin
mo> cd .\Consumer
PS C:\Users\kkalo\OneDrive\Documents\GitHub\DesignPatterns\Fin
mo\Consumer> dotnet run
Consuming
Recieved: 'Sending Message Id: 119', will take 3 to process
Recieved: 'Sending Message Id: 120', will take 5 to process
Recieved: 'Sending Message Id: 121', will take 2 to process
Recieved: 'Sending Message Id: 122', will take 5 to process
Recieved: 'Sending Message Id: 123', will take 2 to process
Recieved: 'Sending Message Id: 124', will take 2 to process
Recieved: 'Sending Message Id: 125', will take 4 to process
Recieved: 'Sending Message Id: 126', will take 2 to process
Recieved: 'Sending Message Id: 127', will take 3 to process
Recieved: 'Sending Message Id: 128', will take 3 to process
Recieved: 'Sending Message Id: 129', will take 5 to process
Recieved: 'Sending Message Id: 130', will take 2 to process
Recieved: 'Sending Message Id: 131', will take 1 to process
```

Εικόνα 31 Δημιουργία καταναλωτή (consumer) για παραλαβή μηνυμάτων

Πλεονεκτήματα χρήσης του προτύπου

Η διαδικασία του διαμοιρασμού των εργασιών με ασύγχρονο τρόπο σε ένα σύνολο συστημάτων έχει ποικίλα πλεονεκτήματα όρους αξιοπιστίας, ευελιξίας και επεκτασιμότητας.

1. Επεκτασιμότητα

Η πληθώρα των συστημάτων που υλοποιούν τα διάφορα tasks μπορεί να μεγαλώσει ή να μικρύνει σε μέγεθος, ανάλογα με το μήκος της ουράς μηνυμάτων που υπάρχει εκείνη την στιγμή. Με αυτό τον τρόπο, μειώνεται το λειτουργικό κόστος και υπάρχει μια ομαλή επεκτασιμότητα.

2. Αξιοπιστία



Εάν το σύνολο των συστημάτων έχει καλυφθεί πλήρως από διάφορα tasks(ή δεν αποκρίνεται), υπάρχει ακόμα η δυνατότητα τοποθέτησης μηνυμάτων στην ουρά. Έτσι, το σύστημα είναι εν μέρη λειτουργικό.

Η ουρά μηνυμάτων λειτουργεί ως χώρος αναμονής, μέχρι το σύστημα να επανέλθει σε φυσιολογικά επίπεδα. Με αυτό τον τρόπο, αποφεύγεται να χαθεί κάποιο μήνυμα και υπάρχει εγγύηση να παραδοθεί σε κάποιο υποσύστημα τουλάχιστον μία φορά.

3. Ευελιξία

Εάν ένα υποσύστημα αποτύχει να εκτελέσει μια εντολή που έλαβε, η εντολή αυτή θα επιστρέψει στην ουρά μηνυμάτων, ώστε να την αναλάβει κάποιο άλλο υποσύστημα.

Πότε πρέπει να γίνεται χρήση του προτύπου;

1. Οι λειτουργίες ενός προγράμματος είναι χωρισμένες με τέτοιο τρόπο ώστε να υλοποιούνται ασύγχρονα.
2. Οι εργασίες είναι ανεξάρτητες μεταξύ τους και μπορούν να τρέξουν παράλληλα
3. Το μέγεθος των εργασιών αλλάζει συνεχώς, ζητώντας μία επεκτάσιμη λύση.
4. Η λύση πρέπει να παρέχει υψηλή αξιοπιστία και ανθεκτικότητα, εάν μια εργασία αποτύχει να εκτελεστεί.

4 Βιβλιογραφικές Πηγές

1. **Αλέξανδρος Ν. Χατζηγεωργίου**. Αντικειμενοστρεφής Σχεδίαση : Εκδόσεις Κλειδάριθμος (Σύγγραμμα μαθήματος)
2. **Tim Corey**. Design Patterns in C#. <https://youtu.be/dhnsegiPXoo>
3. Διαφάνειες μαθήματος στο e-class(5-Αρχές)
4. Tutorial Point, https://www.tutorialspoint.com/design_pattern
5. LucidChart Το πρόγραμμα στο οποίο υλοποιήθηκαν τα διαγράμματα κλάσεων.
6. Competing Consumers Pattern, [Cloud Design Patterns Book της Microsoft Press](#), σελ. 32
7. Health Endpoint Monitoring Pattern, [Cloud Design Patterns Book της Microsoft Press](#), σελ 79
8. Health Endpoint Monitoring Pattern in asp.net core, example with docker <https://www.c-sharpcorner.com/article/health-monitoring-in-asp-net-core/>