**Екзаменаційна робота  
з навчальної дисципліни «Конструювання програмного забезпечення»**

**Виконав студент групи П-41  
Туров Кіріл Віталійович**

**Завдання:**

1. Основи успішного кодування програмного забезпечення.
2. Пишіть поняття малих та великих програмних систем. Надайте приклади.
3. Практичне завдання. Розробити застосування, яке реалізує патерн Абстрактна фабрика та демонструє гру футбол.

**Порядок виконання роботи**

**Теоретична частина**

1. **Опишіть особливості водоспадного стилю виробництва програмного забезпечення.**

Водоспадний (Waterfall) стиль виробництва програмного забезпечення є одним з найстаріших та найбільш традиційних підходів до розробки програмного забезпечення. Цей метод передбачає послідовну послідовність етапів, які виконуються в строго визначеному порядку. Основні особливості водоспадного стилю включають:

1. Фазовість: Розробка водоспадним методом складається з послідовного виконання фаз, таких як аналіз, проектування, реалізація, тестування та впровадження. Кожна фаза має чітко визначені вхідні та вихідні дані, і вони виконуються відповідно до заданих критеріїв завершення.

2. Строга послідовність: Кожна фаза починається тільки після завершення попередньої фази. Наприклад, розробка не може початися до завершення фази аналізу, а тестування не може розпочатися до завершення фази реалізації.

3. Чітке документування: Водоспадний підхід передбачає детальне документування кожної фази, включаючи вимоги, специфікації, плани, звіти тощо. Це допомагає забезпечити стабільність і відстеження процесу розробки.

4. Великі обсяги передробки: Оскільки водоспадний підхід передбачає строгу послідовність етапів, великі зміни вимог або дизайну, які виявляються на пізніших етапах, можуть вимагати значних зусиль з передробки.

5. Менша гнучкість: Водоспадний підхід менш гнучкий у порівнянні з іншими методами розробки, такими як Agile, оскільки він передбачає строгу послідовність етапів і може бути важко адаптуватися до змін вимог чи умов.

Хоча водоспадний підхід має свої переваги, зокрема простоту та чіткість процесу розробки, він також може бути недостатньо гнучким для деяких проектів, особливо тих, де вимоги чи умови розвитку швидко змінюються.

1. **Дайте визначення малих та великих програмних систем.**

Малі програмні системи представляють собою програмні продукти, які мають обмежену складність та обсяг функціональності. Ці системи зазвичай складаються з невеликої кількості компонентів або модулів і здатні вирішувати обмежений набір задач або конкретну проблему. Наприклад, програма для конвертації одного формату файлу в інший, або простий веб-додаток для запису нотаток - це приклади малих програмних систем.

Основна риса малих програмних систем полягає в їхній простоті та легкості у використанні. Розробка таких систем зазвичай вимагає менших зусиль на аналіз, проектування та впровадження. Такі системи часто швидше створюються, оскільки вони мають меншу кількість функцій та компонентів, що потребують розробки.

Великі програмні системи характеризуються великим обсягом функціональності та складністю. Ці системи зазвичай складаються з великої кількості компонентів або модулів і мають здатність вирішувати широкий спектр задач або набір пов'язаних проблем. Операційні системи, великі веб-портали, системи управління базами даних - це лише кілька прикладів великих програмних систем.

Розробка великих програмних систем зазвичай потребує більше часу, ресурсів та зусиль у порівнянні з малими системами. Це через складність управління великим обсягом коду, велику кількість функціональності та потребу відповідності високим стандартам якості. Також, важливо враховувати необхідність ретельного тестування та підтримки протягом усього життєвого циклу великої програмної системи.

**Практична частина**

1. **Розробити застосування, яке реалізує патерн Прототип на прикладі імітації рибок у акваріумі.**

Код програми з середи розробки XCode та використанням мови програмування Swift з фраємворком SwiftUI:

import SwiftUI

// Клас, що представляє рибку

class Fish: ObservableObject, NSCopying {

var name: String

init(name: String) {

self.name = name

}

// Метод копіювання

func copy(with zone: NSZone? = nil) -> Any {

let copy = Fish(name: self.name)

return copy

}

}

// Головний клас для керування акваріумом

class AquariumManager: ObservableObject {

@Published var fishes: [Fish] = []

// Метод для додавання нової рибки

func addFish(\_ fish: Fish) {

// Клонування прототипу рибки

let newFish = fish.copy() as! Fish

fishes.append(newFish)

}

}

// Приклад використання

struct ContentView: View {

@ObservedObject var aquariumManager = AquariumManager()

var body: some View {

VStack {

Button("Додати рибку") {

let prototypeFish = Fish(name: "Золота рибка")

aquariumManager.addFish(prototypeFish)

}

List(aquariumManager.fishes, id: \.name) { fish in

Text(fish.name)

}

}

}

}

@main

struct AquariumApp: App {

var body: some Scene {

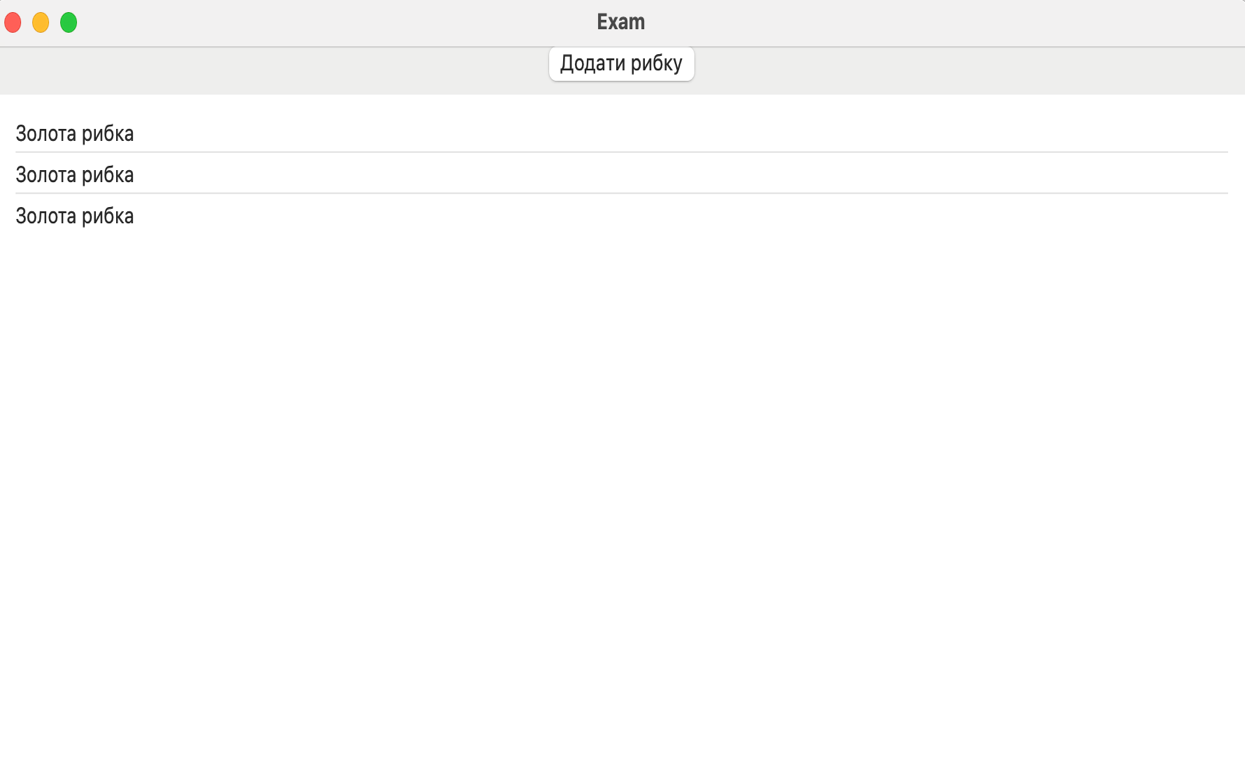
WindowGroup {

ContentView()

}

}

}



Світлина роботи програми

У цьому прикладі я створив клас Fish, що представляє рибку, та клас AquariumManager, відповідуючий за керування акваріумом. Клас Fish реалізує протокол NSCopying, який дозволяє клонувати об'єкти. Коли користувач натискає кнопку "Додати рибку", програма створює новий об'єкт рибки за допомогою клонування прототипу, який міститься в змінній prototypeFish, та додає його до списку рибок акваріуму.