**Домашняя работа**

**Сапицкий Алексей К4110с**

**Цель работы**

* Изучить паттерны Gang of Four и их применение на практике
* Провести моделирование созданной ранее архитектуры классов на языке Swift

**Реализованные паттерны**

**Адаптер**

В домашнем задании данный паттерн реализован в случае когда, приложению необходимо взаимодействовать с градусником, однако его интерфейс не соответствует ожидаемому. Поэтому необходимо использовать данный паттерн для взаимодействия между компонентами.

**protocol** Device {

**var** id: UUID { **get** }

**func** getAnalytics(

dateProvidingStrategy: DateProvidingStrategy

) -> DeviceAnalytics

**func** sendCommand(command: Action)

**func** undoLast()

}

**protocol** TemperatureIndicator {

**var** id: UUID { **get** }

**func** getTemperature() -> String

}

**final** **class** TemperatureAdapter: Device {

**enum** State {

**case** idle

}

**var** id: UUID {

subject.id

}

**private** **let** subject: TemperatureIndicator

**private** **let** dateProvider: DateProvider

**private** **let** state: State = .idle

**init**(

subject: TemperatureIndicator,

dateProvider: DateProvider

) {

**self**.subject = subject

**self**.dateProvider = dateProvider

}

**func** getAnalytics(

dateProvidingStrategy: DateProvidingStrategy

) -> DeviceAnalytics {

DeviceAnalytics(

date: dateProvider.getCurrentDate(

with: dateProvidingStrategy

),

name: String(describing: subject),

description: "Temperature is \(subject.getTemperature())"

)

}

**func** sendCommand(command: Action) {}

**func** undoLast() {}

}

**Стратегия**

В домашнем задании данный паттерн реализован для использования в провайдере даты. Там стратегия определяет формат в котором передаётся дата запрашивающему компоненту.

**protocol** DateProvider {

**func** getCurrentDate(

with strategy: DateProvidingStrategy

) -> String

}

**final** **class** DateProviderImpl: DateProvider {

**private** **let** logger: Logger

**init**(logger: Logger) {

**self**.logger = logger

}

**func** getCurrentDate(with strategy: DateProvidingStrategy) -> String {

logger.log("Date requested")

**return** strategy.date

}

}

**enum** DateStrategy {

**static** **let** years = YearsStrategy()

**static** **let** full = FullDateStrategy()

}

**protocol** DateProvidingStrategy {

**var** date: String { **get** }

}

**struct** YearsStrategy: DateProvidingStrategy {

**var** date: String {

**let** dateFormatter = DateFormatter()

dateFormatter.dateFormat = "YYYY"

**return** dateFormatter.string(from: Date())

}

}

**struct** FullDateStrategy: DateProvidingStrategy {

**var** date: String {

**let** dateFormatter = DateFormatter()

dateFormatter.dateFormat = "DD-MM-YYYY"

**return** dateFormatter.string(from: Date())

}

}

**Синглтон**

В домашнем задании данный паттерн реализован для создания логгера и предоставления к нему доступа из любой точки программы.

**protocol** Logger {

**func** log(**\_** logMessage: String) -> Void

}

**final** **class** DefaultLogger: Logger {

**static** **let** shared: Logger = DefaultLogger()

**private** **init**() {}

**func** log(**\_** logMessage: String) {

print("\t\(logMessage)")

}

}

**Декоратор**

В домашнем задании данный паттерн реализован для того, чтобы в местах где к логам нужна дата и время была возможность их добавить.

**final** **class** DateLoggerDecorator: Logger {

**private** **let** subject: Logger

**private** **let** dateProvider: DateProvider

**init**(

subject: Logger,

dateProvider: DateProvider

) {

**self**.subject = subject

**self**.dateProvider = dateProvider

}

**func** log(**\_** logMessage: String) {

**let** date = dateProvider.getCurrentDate(with: DateStrategy.full)

print("[\(date)]:")

print("[")

subject.log(logMessage)

print("]")

}

}

**Команда**

В домашнем задании данный паттерн реализован для того, чтобы у приложения имелась возможность передавать команду устройству и при необходимости отменять или повторять её.

**protocol** CommandExecutor {

**func** executeCommand(command: Command)

**func** undo()

**func** redo()

}

**enum** Action {

**case** turnOn

**case** turnOff

}

**struct** Command {

**var** execute: () -> Void

**var** unExecute: () -> Void

}

**final** **class** CommandExecutorImpl: CommandExecutor {

**static** **let** shared: CommandExecutor = CommandExecutorImpl()

**private** **init**() {}

**private** **var** commands: [Command] = []

**func** executeCommand(command: Command) {

commands.append(command)

command.execute()

}

**func** undo() {

commands.last?.unExecute()

}

**func** redo() {

commands.last?.execute()

}

}

**final** **class** AirConditioner: Device {

**enum** State {

**case** off

**case** on

}

**var** id = UUID()

**private** **let** location: DeviceInfo.Location

**private** **let** dateProvider: DateProvider

**private** **var** description: String {

**return** "\(String(describing: **self**))"

}

**private(set)** **var** state: State = .off

**init**(

location: DeviceInfo.Location,

dateProvider: DateProvider

) {

**self**.location = location

**self**.dateProvider = dateProvider

}

**func** getAnalytics(

dateProvidingStrategy strategy: DateProvidingStrategy

) -> DeviceAnalytics {

DeviceAnalytics(

date: dateProvider.getCurrentDate(with: strategy),

name: "Air Conditioner",

description: "\(description) located in \(location.rawValue)"

)

}

**func** sendCommand(command: Action) {

**switch** command {

**case** .turnOn:

**let** command = Command(

execute: { **self**.state = .on },

unExecute: { **self**.state = .off }

)

CommandExecutorImpl.shared.executeCommand(command: command)

**case** .turnOff:

**let** command = Command(

execute: { **self**.state = .off },

unExecute: { **self**.state = .on }

)

CommandExecutorImpl.shared.executeCommand(command: command)

}

}

**func** undoLast() {

CommandExecutorImpl.shared.undo()

}

}

**Фасад**

В домашнем задании данный паттерн реализован для начала распознавания команды голосовым ассистентом, когда за вызовом одного метода из его интерфейса скрыты вызовы к нескольким другим его внешним зависимостям.

**protocol** Assistant {

**func** startRecognition(

handler: **@escaping** (Result<RecognitionResponse, RecognitionError>) -> Void

)

}

**final** **class** AssistantController: Assistant {

**private** **let** voiceRecognizer: VoiceRecognizer

**private** **let** intentRecognizer: IntentRecognizer

**private** **let** intentHandler: IntentHandler

**init**(

voiceRecognizer: VoiceRecognizer,

intentRecognizer: IntentRecognizer,

intentHandler: IntentHandler

) {

**self**.voiceRecognizer = voiceRecognizer

**self**.intentRecognizer = intentRecognizer

**self**.intentHandler = intentHandler

}

**func** startRecognition(

handler: **@escaping** (Result<RecognitionResponse, RecognitionError>) -> Void

) {

voiceRecognizer.recognize(handler: { result **in**

**switch** result {

**case** **let** .success(response):

**let** scenario = intentRecognizer.createScenario(response)

intentHandler.handleIntent(scenario)

**case**.failure:

print("Recognition Error")

}

handler(result)

})

}

}

**Вывод:**

Паттерны проектирования позволяют добиться высокой связности и низкой сцепленности кода. Благодаря этому появляется возможность писать переиспользуемый и масштабируемый код, что крайне важно в крупных и сложных системах, где необходимо обеспечивать взаимодействие и поддержку большого числа компонентов.