МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
НАЦІОНАЛЬНОМУ УНІВЕРСИТЕТІ “ЛЬВІВСЬКА ПОЛІТЕХНІКА”

Лабораторна робота №2-11

з дисципліни
«Математичні методи дослідження операцій»

Виконав:
студент групи КН-207
Тимків Андрій
Викладач:
Бойко Н.І.

Львів – 2019

Патетрн Фасад

Використоуається коли нам потрібно приховати складність певних завдань. Наприклад, бібліотека, вона пропонує розробнику певні методи і в цьому випадку розробнику зовсім не потрібно знати всі внутрішні процеси, які повертають результат, який йому потрібно

У своїй програмі я використовую різні сторонні бібліотеки: це і є один з прикладів патерну “Фасад” у моєму коду. Також файл **main.go** у якому я ініціалізую всі компоненти своєї програми це також приклад патерну “Фасад”

```
func main() {
	cfgPath := flag.String("p", "./config/api.yaml", "Path to config file")
	flag.Parse()
	cfg, err := config.Load(*cfgPath)
	checkErr(err)
	checkErr(Start(cfg))
}
func Start(cfg *config.Configuration) error {
	db, err := gorm.GetDbInstance(&cfg.DB)
	if err != nil {
		return err
	}
	e := server.New()
	shopDB := gormsqlS.NewShop(db)
	shopGr := e.Group("/shop")
	st.NewHTTP(ss.New(shopDB), shopGr)
	server.Start(e, &server.Config{
		Port:		cfg.Server.Port,
		ReadTimeoutSeconds:  cfg.Server.ReadTimeout,
		WriteTimeoutSeconds: cfg.Server.WriteTimeout,
		Debug:		cfg.Server.Debug,
	})
	return nil
}
```

Паттерн Одинак (Singleton)

Одинак — це породжувальний патерн проектування, який гарантує, що клас має лише один екземпляр, та надає глобальну точку доступу до нього. Тобто вмсі інші класи використовують тільки один екземпляр (instance) цього класу. Один з набільш поширених у використанні.

У своїй програмі я використав цей патерн для створення зєднання з базою даних. Це тривіальний приклад використання цього патерну, адже нам не потрібно кожного разу, використовуючи базу даних, створювати до неї нове зєднання

```
	type DB struct {
	gorm.DB
}
var dbInstance *DB
func GetDbInstance(cfg *config.Database) (*DB, error) {
	if dbInstance == nil {
		bdInstance, err := New(cfg)
		return bdInstance, err
	}
	return dbInstance, nil
}
func New(cfg *config.Database) (*DB, error) {
	connectString := fmt.Sprintf("%s:%s@tcp(localhost:3306)/%s?charset=utf8&parseTime=True&loc=Local", cfg.Username,
cfg.Password, cfg.Name)
	b := backoff.NewExponentialBackOff()
	b.MaxElapsedTime = 1 * time.Second
	var db gorm.DB
	operation := func() error {
		dbT, err := gorm.Open("mysql", connectString)
		if err != nil {
			return err
```

```
            }
            db = *dbT
            return nil
        }
        err := backoff.Retry(operation, b)
        if err != nil {
            log.Fatalf("error after retrying: %v", err)
        }
        //defer db.Close()
        db.AutoMigrate(sa.Shop{}, sa.Address{})
        dbIn := DB{db}
        return &dbIn, nil
}
```

Патерн Декоратор

Цей патерн дозволяє декорувати вже існуючий тип більшою кількістю функціональних «фіч» без прямого редагування цього типу. Я використовую патерн декоратор для ініціалізування структури своєї БД:

```
type DB struct {
    gorm.DB
}
```

Тут тип gorm.DB це декоратор, який декорує тип DB. В ньому містяться свої методи, якими тип DB теж може користуватись без явного ініціалізування

Патерн Стратегія

Поведінковий патерн — використоуває різні алгорити для досягнення конкретної функціональності. Тобто, алгоритми різні — результат один. Ці алгоритми приховані за інтерфейсом і, звичайно ж, є взаємозамінними. У своїй програмі я використовує цей патерн

```
type Service interface {
    View(echo.Context, string)(*sa.Shop, error)
    ViewAll(echo.Context) ([]sa.Shop, error)
    GetAddressByShopId(echo.Context, string) (*sa.Address, error)
}
func New(sdb DB) *Shop {
    return &Shop{sdb: sdb}
}
type Shop struct {
    sdb DB
}
type DB interface {
    View(id string)(*sa.Shop, error)
    ViewAll() ([]sa.Shop, error)
    GetAddress(id int)(*sa.Address, error)
}
```

Як ми бачимо Service і DB — інтерфейсси. Ми можемо використовувати будь-які структури які будуть реалізовувати всі методи цього інтерфейсу. В моєму випадку я використовую тільки один спосіб реалізації цих функцій, проте, при потребі, можна створити іншу структуру яка буде реалізовуватви всі функції інтерфейсу, але іншим способом, та використовувати її без жодної різниці у результаті.

Патерн Спостерігач

це поведінковий патерн проектування, який створює механізм підписки, що дає змогу одним об’єктам стежити й реагувати на події, які відбуваються в інших об’єктах.

У своєму коді для демонстрації цього патерну я використав сервіс nats який реалізовує цей патерн. Я підняв nats servrer який слугує для передавання повідомлень між паблішарами і субскрайберами. Створив паблішера, який передає меседж субскрайберам при запиті на сервер /**shops/view/{id}**

```
func (s *Shop) View(c echo.Context, id string) (*sa.Shop, error) {
	shop, err := s.sdb.View(id)
	if err != nil {
		return nil, err
	}
	if err := s.nats.PushMessage(shop, TOPIC); err != nil {
		log.Printf("failed pushing user into nuts; err: %v", err)
		return nil, err
	}
	return shop, nil
}
```

І створив субскрайбера, який підпусується на певний топік, в нашому випадку “shops” і отирмує повідомлення по цьому топіку та записує їх у файл.

```
func Subscriber(connN *nats.Conn, subj string) {
	natsService := messages.Create(connN)
	outPath := "./pubsub/" + subj + ".txt"
	err := natsService.Subscribe(subj, outPath)
	if err != nil {
		log.Println(err)
	}
	log.Printf("New subscriber, listening subject %s", subj)
}
```

```
func (s *Service) Subscribe(subject string, outPath string) error {
	_, err := s.connectionN.Subscribe(subject, func(m *nats.Msg) {
		msg := m.Data
		log.Printf("Recieved message for %s subscriber", subject)
		if err := writeToFile(outPath, msg); err != nil {
			log.Printf("failed writing msg into file from nats; err: %v", err)
		}
	})
	if err != nil {
		return err
	}
	return nil
}
```

https://github.com/atymkiv/sa