

Design Patterns

Шаблоны проектирования

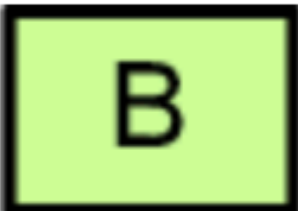
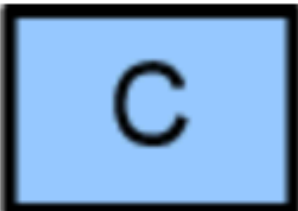
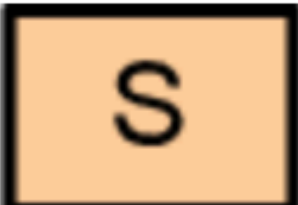
- Приемы объектно-ориентированного программирования. Паттерны проектирования - Книга, написанная этими четырьмя авторами: Эрих Гамма, Ричард Хелм, Ральф Джонсон, Джон Воссидес. (gof)
- Существует 23 классических шаблонов

Зачем они нужны?

- Писать однородный код
- Помогает проще понимать чужой код
- Подчиняется принципам

Какие виды бывают?

Виды паттернов

-  — поведенческие (behavioral);
-  — порождающие (creational);
-  — структурные (structural).

Их правда 23?

Список шаблонов

C Абстрактная фабрика

S Адаптер

S Мост

C Строитель

B Цепочка обязанностей

B Команда

S Компоновщик

S Декоратор

S Фасад

C Фабричный метод

S Приспособленец

B Интерпретатор

B Итератор

B Посредник

B Хранитель

C Прототип

S Прокси

B Наблюдатель

C Одиночка

B Состояние

B Стратегия

B Шаблонный метод

B Посетитель

Задание

Реализуйте builder UNNotificationContent

DRY, KISS, YAGNI, SOLID

Don't repeat yourself

- Случаем нарушения DRY называют WET (Write everything twice)
- «Каждая часть знаний должна иметь единственное, непротиворечивое представление в рамках одной системы»

Keep it simple stupid

- Принцип проектирование, принятый в ВМС США в 1960 году
- Суть в том чтобы избегать ненужной сложности
- Даже Леонардо да Винчи говорил так «Less is more»

You ain't gonna need it

- Принцип проектирования ПО, при котором в качестве основной ценности декларируется отказ от избыточной функциональности, то есть - отказ от функциональности, в которой нет необходимости в текущий момент.
- В связке с SOLID дает плоды чистого кода)

SOLID



SOLID

SOLID

- Придумал все эти принципы Роберт Мартин (Uncle Bob)
- Книга «Clean Architecture» описывает архитектуру, построенную из кирпичиков SOLID
- Применяется и упоминается только в контексте ООП
- Нужно проектировать модули, которые переиспользуемы, легко понимаемы, способствуют изменениям

S - Single Responsibility Principle (SRP)

A module should be responsible to one, and only one, actor.

Старая формулировка: A module should have one, and only one, reason to change.

Часто ее трактовали следующим образом: *Модуль должен иметь только одну обязанность*. И это главное заблуждение при знакомстве с принципами. Все несколько хитрее.

На каждом проекте люди играют разные роли (actor): Аналитик, Проектировщик интерфейсов, Администратор баз данных. Естественно, один человек может играть сразу несколько ролей. В этом принципе речь идет о том, что изменения в модуле может запрашивать одна и только одна роль. Например, есть модуль, реализующий некую бизнес-логику, запросить изменения в этом модуле может только Аналитик, но никак не DBA или UX.

O - The Open Closed Principle (OCP)

A software artifact should be open for extension but closed for modification.

Старая формулировка: You should be able to extend a classes behavior, without modifying it.

Это определенно может ввести в ступор. Как можно расширить поведение класса без его модификации? В текущей формулировке Роберт Мартин оперирует понятием артефакт, т.е. jar, dll, gem, npm package. Чтобы расширить поведение, нужно воспользоваться динамическим полиморфизмом.

Например, наше приложение должно отправлять уведомления. Используя dependency inversion, наш модуль объявляет только интерфейс отправки уведомлений, но не реализацию. Таким образом, логика нашего приложения содержится в одном dll файле, а класс отправки уведомлений, реализующий интерфейс — в другом. Таким образом, мы можем без изменения (перекомпиляции) модуля с логикой использовать различные способы отправки уведомлений.

Этот принцип тесно связан с LSP и DIP, которые мы рассмотрим далее.

L - The Liskov Substitution Principle (LSP)

Имеет сложное математическое определение, которое можно заменить на: Функции, которые используют базовый тип, должны иметь возможность использовать подтипы базового типа, не зная об этом.

Классический пример нарушения. Есть базовый класс `Stack`, реализующий следующий интерфейс: `length`, `push`, `pop`. И есть потомок `DoubleStack`, который дублирует добавляемые элементы. Естественно, класс `DoubleStack` нельзя использовать вместо `Stack`.

У этого принципа есть забавное следствие: *Объекты, моделирующие сущности, не обязаны реализовывать отношения этих сущностей*. Например, у нас есть целые и вещественные числа, причем целые числа — подмножество вещественных. Однако, `double` состоит из двух `int`: мантисы и экспоненты. Если бы `int` наследовал от `double`, то получилась бы забавная картина: родитель содержит 2-х своих детей.

В качестве второго примера можно привести Generics. Допустим, есть базовый класс `Shape` и его потомки `Circle` и `Rectangle`. И есть некая функция `Foo(List<Shape> list)`. Мы считаем, что `List<Circle>` можно привести к `List<Shape>`. Однако, это не так. Допустим, это приведение возможно, но тогда в `list` можно добавить любую фигуру, например `rectangle`. А изначально `list` должен содержать только объекты класса `Circle`.

I - The Interface Segregation Principle (ISP)

Make fine grained interfaces that are client specific.

Под интерфейсом здесь понимается именно Java, C# интерфейс. Разделение интерфейса облегчает использование и тестирование модулей.

Для Objective-C и Swift интерфейсам может являться protocol.

D - The Dependency Inversion Principle (DIP)

Depend on abstractions, not on concretions.

- Модули верхних уровней не должны зависеть от модулей нижних уровней. Оба типа модулей должны зависеть от абстракций.
- Абстракции не должны зависеть от деталей. Детали должны зависеть от абстракций.

Что такое модули верхних уровней? Как определить этот уровень? Как оказалось, все очень просто. Чем ближе модуль к вводу/выводу, тем ниже уровень модуля. Т.е. модули, работающие с BD, интерфейсом пользователя, низкого уровня. А модули, реализующие бизнес-логику — высокого уровня.

К прочтению:

- «Чистый код» Роберт Мартин
- Каталог шаблонов проектирования - <https://refactoring.guru/design-patterns/catalog>

Домашнее задание

Реализовать: Mediator, Memento, Facade, Singleton (Как минимум 4 любых)