Паттерны программирования

Ткаченко П.И.

Паттерн

Концепцию паттернов впервые описал Кристофер Александер в книге «Язык шаблонов. Города. Здания. Строительство».

Идея показалась заманчивой четвёрке авторов: Эриху Гамме, Ричарду Хелму, Ральфу Джонсону, Джону Влиссидесу. В 1995 году они написали книгу «Design Patterns: Elements of Reusable Object-Oriented Software»

это часто встречаемое решение определённой проблемы при проектировании архитектуры программ.

Описания паттернов обычно очень формальны и чаще всего состоят из таких пунктов:

1. проблемы, которую решает паттерн;
2. мотивации к решению проблемы способом, который предлагает паттерн;
3. структуры классов, составляющих решение;
4. примера на одном из языков программирования;
5. особенностей реализации в различных контекстах;
6. связей с другими паттернами.

Паттерны нужны потому что

Проверенные решения

Стандартизация кода

Общий программистский словарь

Классификация паттернов

Паттерны отличаются по уровню сложности, детализации и охвата проектируемой системы.

Низкоуровневые и простые паттерны — идиомы. Они не очень универсальные, так как применимы только в рамках одного языка программирования.

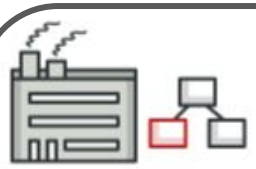
Универсальные — архитектурные паттерны, которые можно реализовать практически на любом языке. Они нужны для проектирования всей программы, а не отдельных её элементов.

Порождающие паттерны беспокоятся о гибком создании объектов без внесения в программу лишних зависимостей (удобное и безопасное создание новых объектов или даже целых семейств объектов)

Структурные паттерны показывают различные способы построения связей между объектами (построение удобных в поддержке иерархий классов)

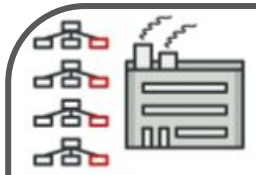
Поведенческие паттерны заботятся об эффективной коммуникации между объектами (решают задачи эффективного и безопасного взаимодействия между объектами программы)

Порождающие паттерны



Фабричный Метод
Factory Method

Определяет общий интерфейс для создания объектов в суперклассе, позволяет изменять тип создаваемых объектов

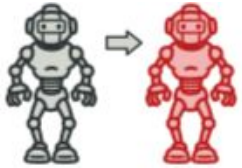


Абстрактная Фабрика
Abstract Factory

Позволяет создавать семейства связанных объектов, не привязываясь к конкретным классам создаваемых объектов

Строитель
Builder

Позволяет создавать сложные объекты пошагово. Один и тот же код используется для получения разных объектов



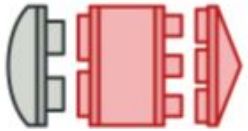
Прототип
Prototype

Позволяет копировать объекты не вдаваясь в подробности их реализации

Одиночка
Singleton

Гарантирует, что у класса есть только один экземпляр, и предоставляет к нему глобальную точку доступа

Структурные паттерны

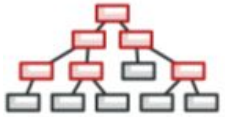


Адаптер
Adapter

Позволяет объектом с несовместимыми интерфейсами работать вместе

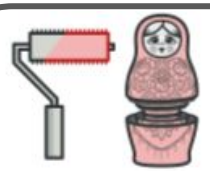
Мост
Bridge

Разделяет один или несколько классов на две отдельные иерархии – абстракцию и реализацию, позволяя изменять их независимо друг от друга



Компоновщик
Composite

Позволяет сгруппировать объекты в древовидную структуру, а затем работать с ними так, если бы это был единичный объект



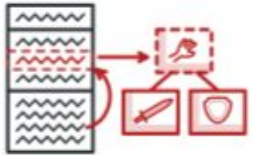
Декоратор
Decorator

Позволяет динамически добавлять объектам новую функциональность, оборачивая их в полезные «обёртки»

Фасад
Facade

Предоставляет простой интерфейс к сложной системе классов, библиотеке.

Поведенческие паттерны



Стратегия

Strategy

Определяет семейство схожих алгоритмов и помещает каждый из них в собственный класс. После чего, алгоритмы можно заменять прямо во время исполнения программы.

Снимок

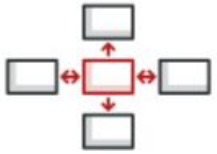
Memento

Позволяет делать снимки состояния объектов, не раскрывая подробностей их реализации. Затем снимок можно использовать.

Итератор

Iterator

Даёт возможность последовательно обходить элементы составных объектов, не раскрывая их внутреннего представления



Посредник

Mediator

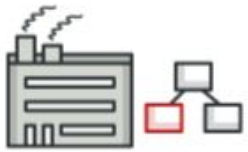
Позволяет уменьшить связанность множества классов между собой, благодаря перемещению этих связей в один класс-посредник

Наблюдатель

Observer

Создаёт механизм подписки, позволяющий одним объектам следить и реагировать на события, происходящие в других объектах

Паттерны - назначение

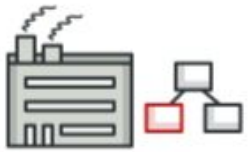


Фабричный Метод

Factory Method

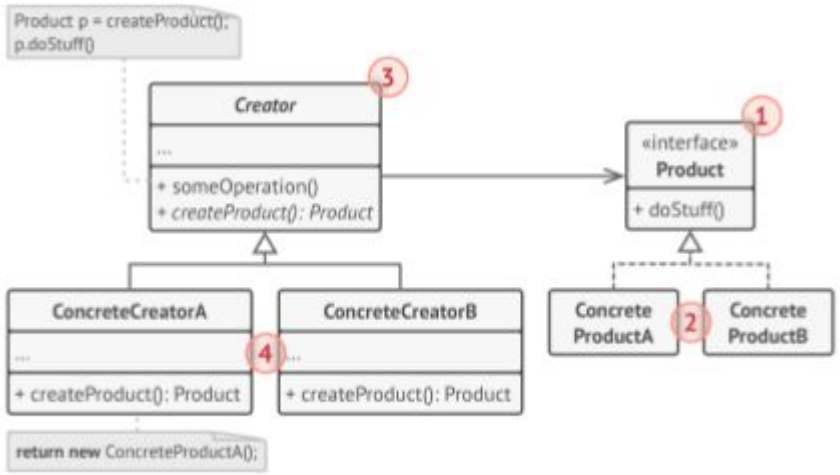
Определяет общий интерфейс для создания объектов в суперклассе, позволяет изменять тип создаваемых объектов

Паттерны – из чего состоит (блоки/части/участники)



Фабричный Метод

Factory Method

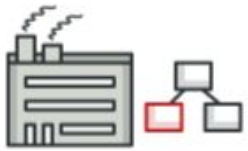


1. **Продукт** определяет общий интерфейс объектов, которые может произвести создатель и его подклассы.
2. **Конкретные продукты** содержат код различных продуктов. Продукты будут отличаться реализацией, но интерфейс у них будет общий.
3. **Создатель** объявляет фабричный метод, создающий объекты через общий интерфейс продуктов.
4. **Конкретные создатели** по-своему реализуют фабричный метод, производя те или иные конкретные продукты

Паттерны – когда нужно и можно использовать

1. Когда необходимо отделить код производства продуктов от остального кода, который эти продукты использует.
2. Когда мы хотим дать возможность пользователям расширять части вашего фреймворка или библиотеки.
3. Когда мы хотим экономить системные ресурсы, повторно используя уже созданные объекты, вместо создания новых.

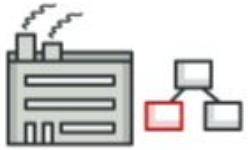
Паттерны – разновидность шаблона



Фабричный Метод
Factory Method

Порождающие паттерны беспокоятся о гибком создании объектов без внесения в программу лишних зависимостей (удобное и безопасное создание новых объектов или даже целых семейств объектов)

Паттерны – последовательность реализации

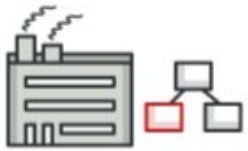


Фабричный Метод

Factory Method

1. Приведите все создаваемые продукты к общему интерфейсу.
2. В классе, который производит продукты, создайте пустой фабричный метод. В качестве возвращаемого типа укажите общий интерфейс продукта.
3. Затем, пройдитесь по коду класса и найдите все участки, создающие продукты. Поочерёдно замените эти участки вызовами фабричного метода, перенося в него код создания различных продуктов.
4. Для каждого типа продуктов заведите подкласс и переопределите в нём фабричный метод. Переместите туда код создания соответствующего продукта из суперкласса.
5. Если создаваемых продуктов слишком много для существующих подклассов создателя, вы можете подумать о введении параметров в фабричный метод, которые позволят возвращать различные продукты в пределах одного подкласса

Паттерны – контрольные точки успешного применения



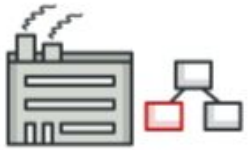
Фабричный Метод

Factory Method

1. Мы можем добавлять новые продукты не меняя код, который эти продукты использует.
2. Пользователи могут добавлять продукты и это не убивает программу.
3. Появляется возможность использовать ранее написанный код повторно.

Паттерны –

плюсы



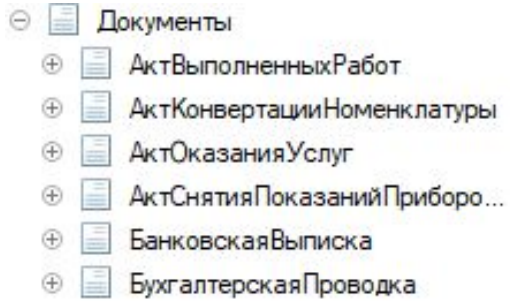
Фабричный Метод
Factory Method

1. Избавляет класс от привязки к конкретным классам продуктов.
2. Выделяет код производства продуктов в одно место, упрощая поддержку кода.
3. Упрощает добавление новых продуктов в программу.
4. Реализует принцип открытости/закрытости.

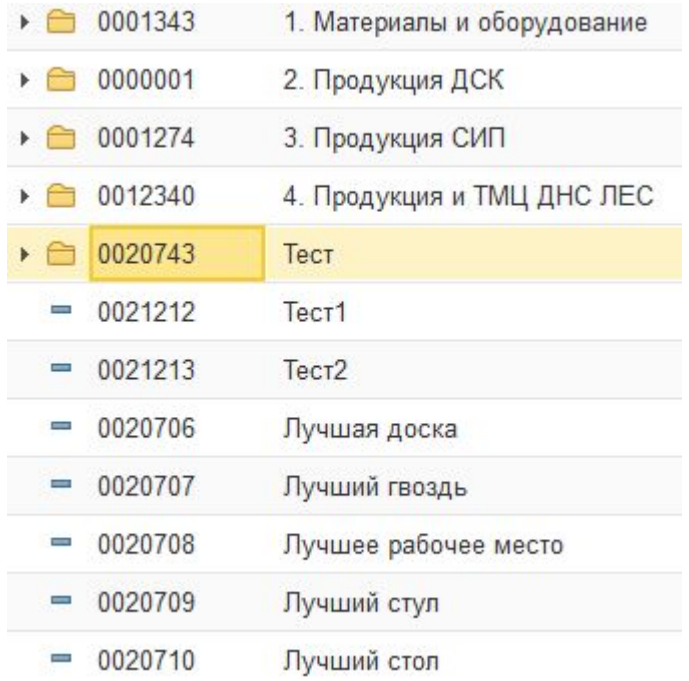
минусы

1. Может привести к созданию больших параллельных иерархий классов, так как для каждого класса продукта надо создать свой подкласс создателя

Паттерны – возможность применения в 1с
(практические примеры)

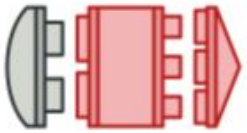


Программисты 1с, могут пользоваться созданием различных объектов, у которых уже установлены основные моменты поведения и при этом есть возможность доработать.



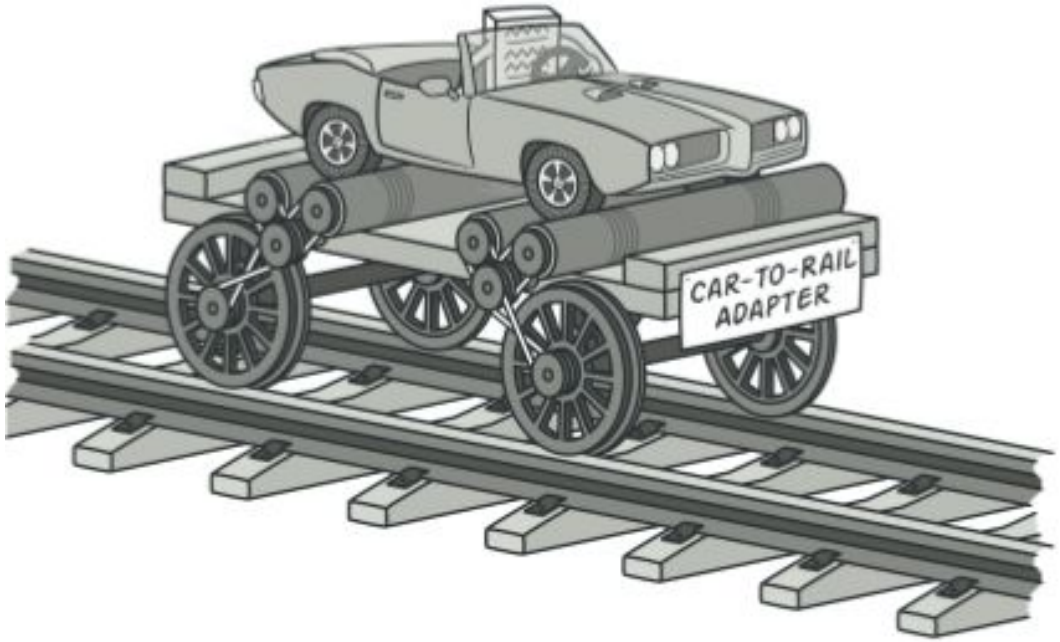
Пользователи могут создавать новые объекты в справочниках, не меняя код и имя возможность работать с созданными объектами.

Паттерны - назначение



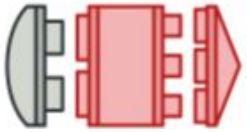
Адаптер

Adapter



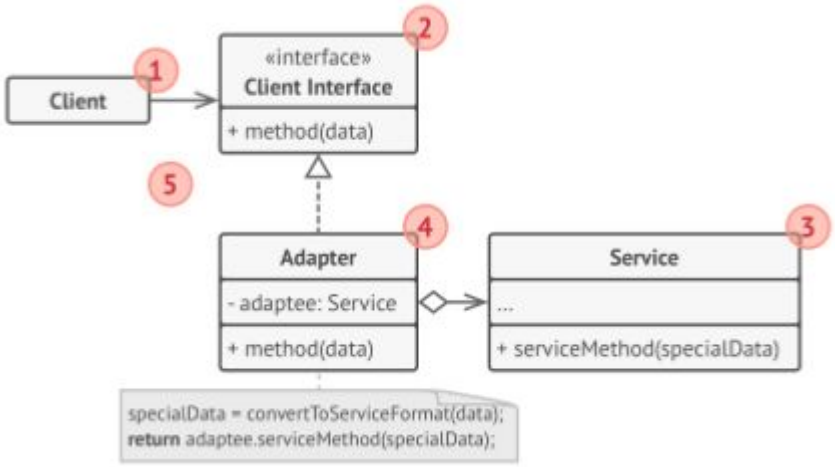
Позволяет объектом с несовместимыми интерфейсами работать вместе

Паттерны – из чего состоит (блоки/части/участники)



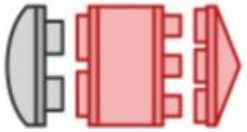
Адаптер

Adapter



1. **Клиент** — это класс, который содержит существующую бизнес-логику программы.
2. **Клиентский интерфейс** описывает протокол, через который клиент может работать с другими классами.
3. **Сервис** – это какой-то полезный класс, обычно сторонний. Клиент не может использовать этот класс напрямую, так как сервис имеет непонятный ему интерфейс.
4. **Адаптер** — это класс, который может одновременно работать и с клиентом, и с сервисом.
5. Работая с адаптером через интерфейс, клиент не привязываться к конкретному классу адаптера. Благодаря этому, вы можете добавлять в программу новые виды адаптеров, независимо от клиентского кода.

Паттерны – когда нужно и можно использовать

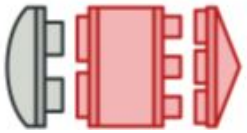


Адаптер

Adapter

1. Когда мы хотим использовать сторонний класс, но его интерфейс не соответствует остальному коду приложения.
2. Когда нам нужно использовать несколько существующих подклассов, но в них не хватает какой-то общей функциональности.

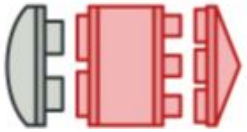
Паттерны – разновидность шаблона



Адаптер
Adapter

Структурные паттерны показывают различные способы построения связей между объектами (построение удобных в поддержке иерархий классов)

Паттерны – последовательность реализации

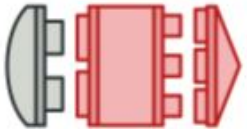


Адаптер

Adapter

1. Убедитесь, что у вас есть два класса с неудобными интерфейсами
2. Опишите клиентский интерфейс, через который классы приложения смогли бы использовать сторонний класс
3. Создайте класс адаптера, реализовав этот интерфейс
4. Поместите в адаптер поле-ссылку на объект-сервис. В большинстве случаев, это поле заполняется объектом, переданным в конструктор адаптера.
5. Адаптер должен делегировать основную работу сервису
6. Приложение должно использовать адаптер только через клиентский интерфейс. Это позволит легко изменять и добавлять адаптеры в будущем.

Паттерны – контрольные точки успешного применения



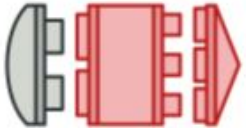
Адаптер

Adapter

1. Два объекта, которые раньше не могли взаимодействовать – теперь могут работать вместе

Паттерны –

плюсы



Адаптер

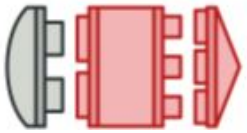
Adapter

1. Отделяет и скрывает от клиента подробности преобразования различных интерфейсов

минусы

1. Усложняет код программы за счёт дополнительных классов.

Паттерны – возможность применения в 1с

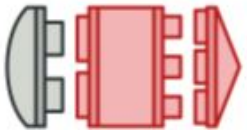


Адаптер

Adapter

Однозначно возможно и переменяется

Паттерны – практические примеры



Адаптер

Adapter

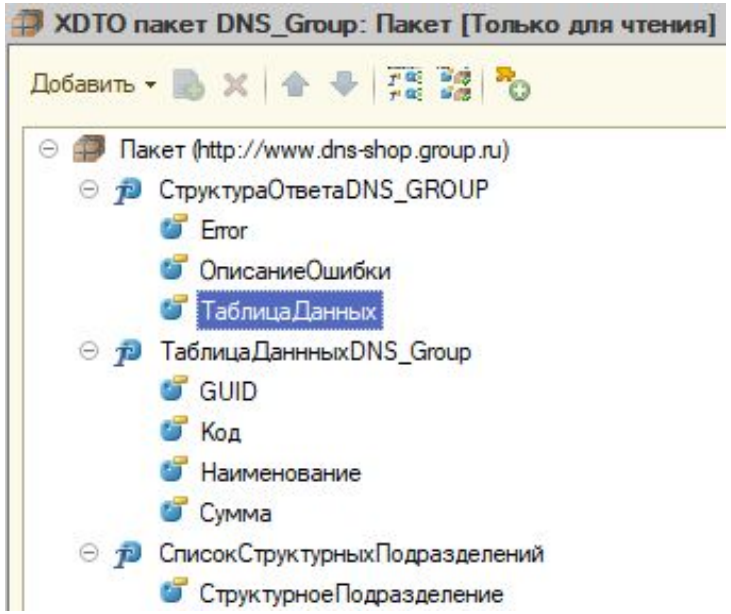
```
Для каждого Отбор Из Отборы Цикл

    Если НЕ ЗначениеЗаполнено(Параметры[Отбор.ИмяРеквизита]) Тогда
        Продолжить;
    КонецЕсли;

    Если ТипЗнч(Параметры[Отбор.ИмяРеквизита]) = Тип("Структура") Тогда
        Отбор.ВидСравнения = Параметры[Отбор.ИмяРеквизита].ВидСравнения;
        Отбор.ЗначениеОтбора = Параметры[Отбор.ИмяРеквизита].ЗначениеОтбора;
    Иначе
        Отбор.ЗначениеОтбора = Параметры[Отбор.ИмяРеквизита];
    КонецЕсли;

КонецЦикла;
```

Отчет который может принимать и обрабатывать разные типы принимаемых параметров.

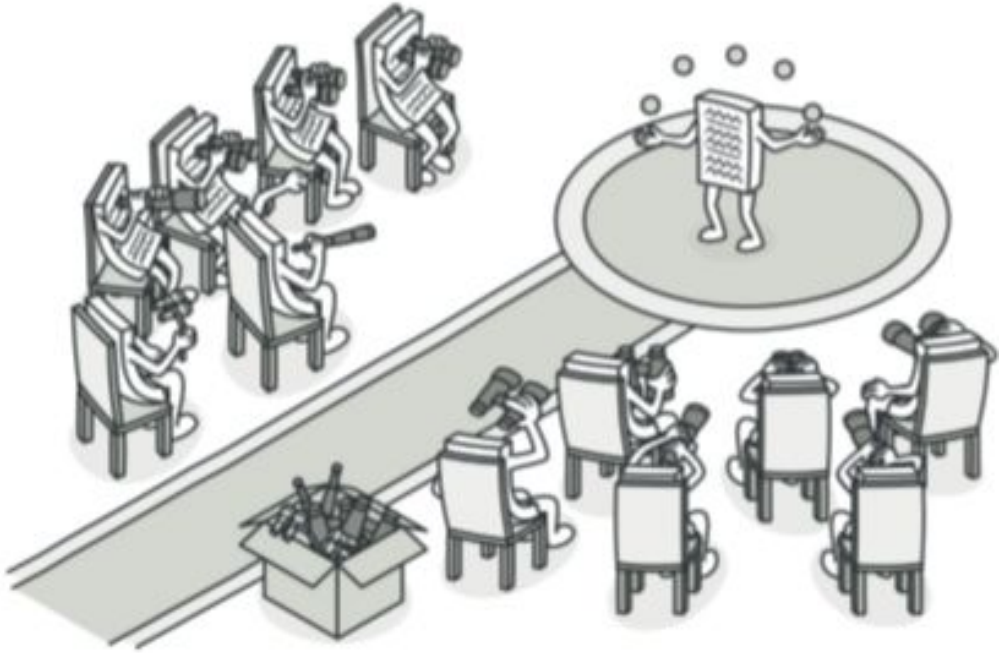


Пакет XDTO, который фактически служит адаптером между двумя и более базами.

Паттерны - назначение

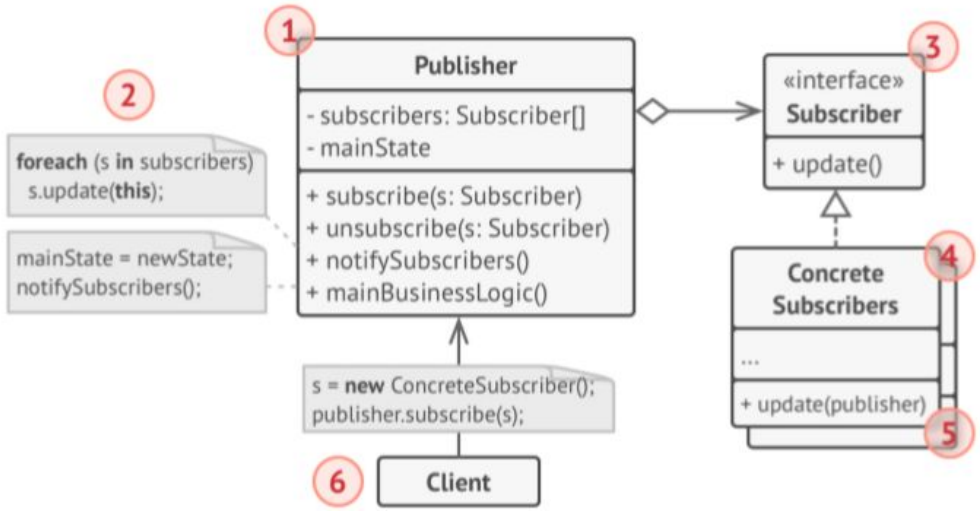
Наблюдатель

Observer



Создаёт механизм подписки, позволяющий одним объектам следить и реагировать на события, происходящие в других объектах

Паттерны – из чего состоит (блоки/части/участники)



1. **Издатель** владеет внутренним состоянием, изменение которого интересно для подписчиков. Он содержит механизм подписки — список подписчиков, а также методы подписки/отписки.
2. Когда внутреннее состояние издателя меняется, он оповещает своих подписчиков. Для этого издатель проходит по списку подписчиков и вызывает их метод оповещения, заданный в интерфейсе подписчика.
3. **Подписчик** определяет интерфейс, которым пользуется издатель для отправки оповещения. В большинстве случаев, для этого достаточно единственного метода.
4. **Конкретные подписчики** выполняют что-то в ответ на оповещение, пришедшее от издателя. Эти классы должны следовать общему интерфейсу подписчиков, чтобы издатель не зависел от конкретных классов подписчиков.
5. По приходу оповещения, подписчику нужно получить обновлённое состояние издателя. Издатель может передать это состояние через параметры метода оповещения.
6. **Клиент** создаёт объекты издателей и подписчиков, а затем регистрирует подписчиков на обновления в издателях.

Паттерны – когда нужно и можно использовать

1. Когда нам что-то нужно делать при наступлении определенного состояния.
2. Когда одни объекты должны следить за состоянием других объектов.

Паттерны – разновидность шаблона

Поведенческие паттерны заботятся об эффективной коммуникации между объектами (решают задачи эффективного и безопасного взаимодействия между объектами программы)

Паттерны – последовательность реализации

1. Разбейте вашу функциональность на две части: независимое ядро и опциональные зависимые части. Независимое ядро станет издателем. Зависимые части станут подписчиками
2. Создайте интерфейс подписчиков. Обычно, в нём достаточно определить единственный метод оповещения.
3. Создайте интерфейс издателей и опишите в нём операции управления подпиской. Помните, что издатель должен работать только с общим интерфейсом подписчиков.
4. Создайте классы конкретных издателей. Реализуйте их так, чтобы при каждом изменении состояния, они слали оповещения всем своим подписчикам
5. Реализуйте метод оповещения в конкретных подписчиках. Издатель может отправлять какие-то данные вместе с оповещением (например, в параметрах). Возможен и другой вариант, когда подписчик, получив оповещение, сам берёт из объекта издателя нужные данные. Но при этом подписчик привяжет себя к конкретному классу издателя
6. Клиент должен создавать необходимое количество объектов подписчиков и подписывать их у издателей

Паттерны – контрольные точки успешного применения

1. Происходит нужное действие при наступлении определенного события.
2. Одни объекты способны отслеживать состояние других объектов

Паттерны –

плюсы

1. Издатель не зависит от конкретных классов подписчиков
2. Вы можете подписывать и отписывать получателей на лету
3. Реализует принцип открытости/закрытости

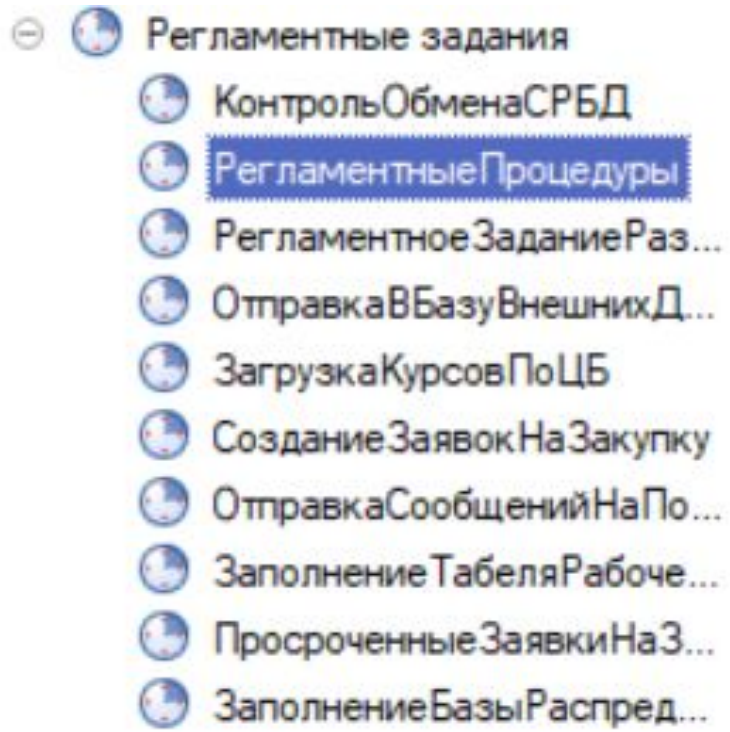
минусы

1. Наблюдатели оповещаются в случайном порядке

Паттерны – возможность применения в 1с

Однозначно возможно и переменяется

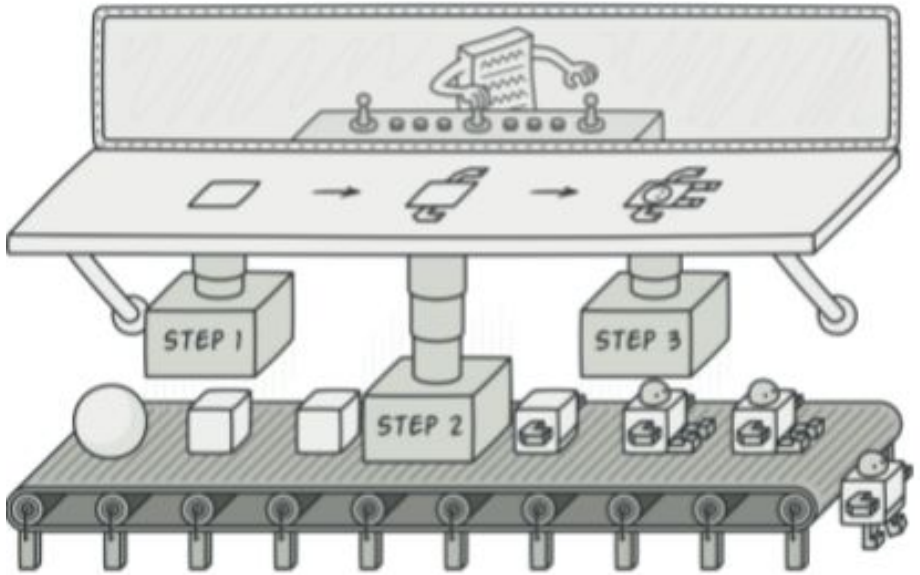
Паттерны – практические примеры



Паттерны - назначение

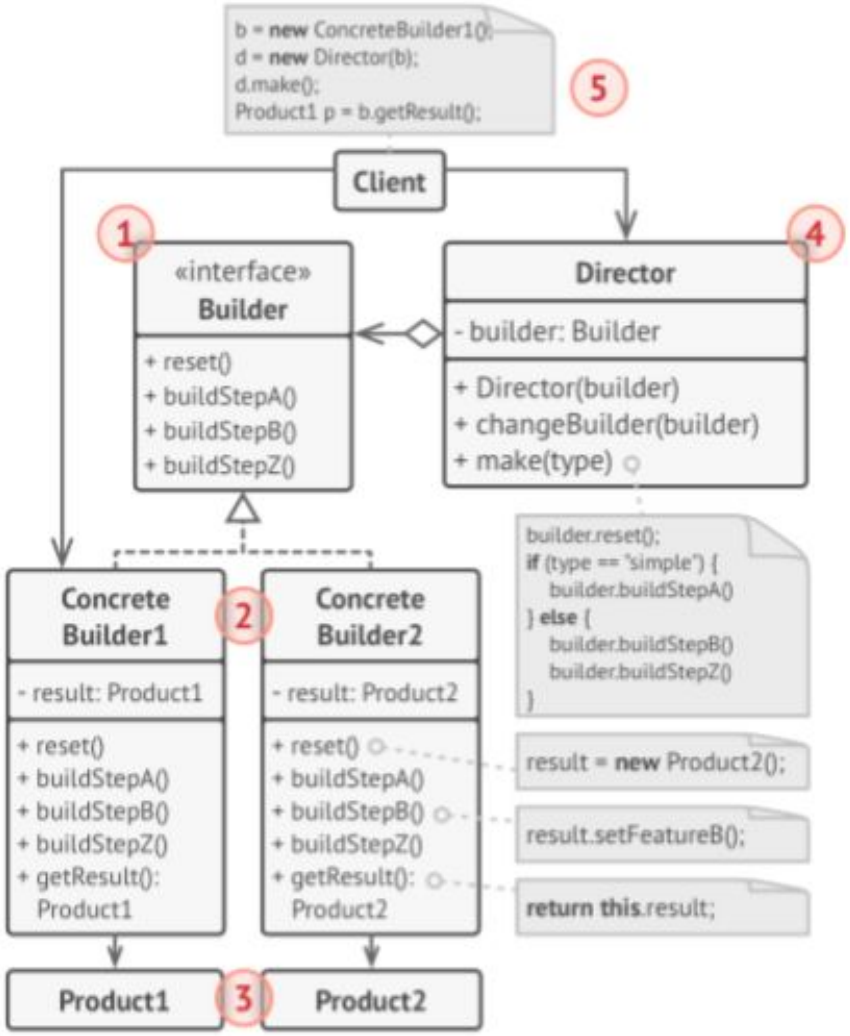
Строитель

Builder



Позволяет создавать сложные объекты пошагово. Один и тот же код используется для получения разных объектов

Паттерны – из чего состоит (блоки/части/участники)



1. **Интерфейс строителя** объявляет шаги конструирования продуктов, общие для всех видов строителей.
2. **Конкретные строители** реализуют строительные шаги, каждый по-своему. Конкретные строители могут производить разнородные объекты, не имеющие общего интерфейса.
3. **Продукт** — создаваемый объект. Продукты, сделанные разными строителями, не обязаны иметь общий интерфейс.
4. **Директор** определяет порядок вызова строительных шагов для производства той или иной конфигурации объектов.
5. Обычно, **Клиент** подаёт в конструктор директора уже готовый объект-строитель, и в дальнейшем данный директор использует только его. Но возможен и другой вариант, когда клиент передаёт строителя через параметр строительного метода директора. В этом случае можно каждый раз применять разных строителей для производства различных представлений объектов.

Паттерны – когда нужно и можно использовать

1. Когда нам нужно собирать объекты пошагово
2. Когда нам нужные разные представления одного объекта (Железные и деревянные дома)
3. Когда нужно строить древовидные структуры

Паттерны – разновидность шаблона

Порождающие паттерны беспокоятся о гибком создании объектов без внесения в программу лишних зависимостей (удобное и безопасное создание новых объектов или даже целых семейств объектов)

Паттерны – последовательность реализации

1. Убедитесь в том, что создание разных представлений объекта можно свести к общим шагам.
2. Опишите эти шаги в общем интерфейсе строителей.
3. Для каждого из представлений объекта-продукта создайте по одному классу-строителю и реализуйте их методы строительства
4. Подумайте о создании класса директора. Его методы будут создавать различные конфигурации продуктов, вызывая разные шаги одного и того же строителя.
5. Клиентский код должен будет создавать и объекты строителей, и объект директора. Перед началом строительства, клиент должен связать определённого строителя с директором

Паттерны – контрольные точки успешного применения

1. Мы можем создавать объекты пошагово
2. У нас могут быть разные представления одного объекта

Паттерны –

плюсы

1. Позволяет создавать продукты пошагово.
2. Позволяет использовать один и тот же код для создания различных продуктов.
3. Изолирует сложный код сборки продукта от его основной бизнес-логики.

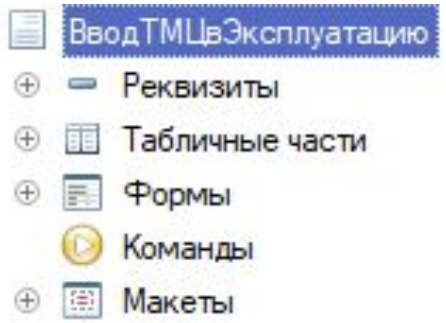
минусы

1. Усложняет код программы за счёт дополнительных классов.
2. Клиент будет привязан к конкретным классам строителей, так как в интерфейсе строителя может не быть метода получения результата.

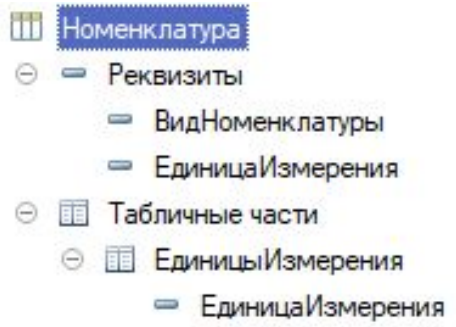
Паттерны – возможность применения в 1с

Относительно возможное применение

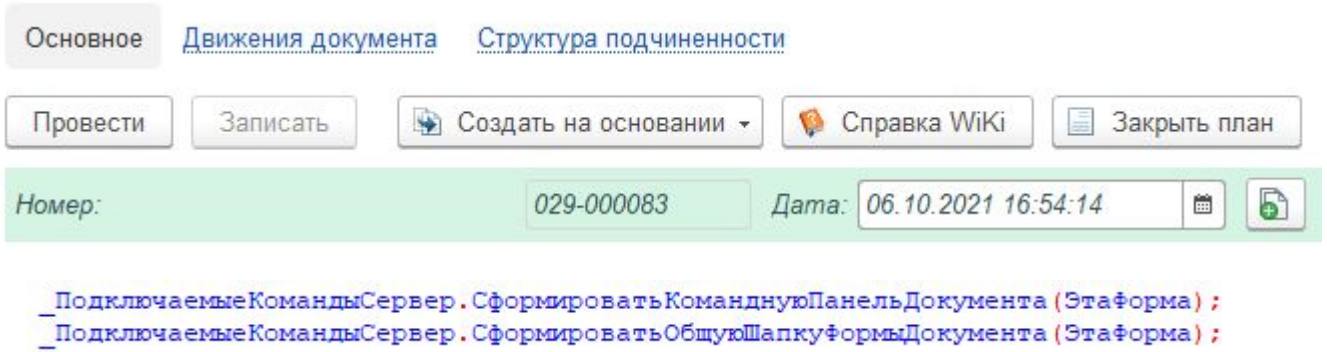
Паттерны – практические примеры



Части документа как части строителя



Элемент справочника как объект, для которого задаются свои строители (Реквизиты и ТЧ)



Форма которая создается в зависимости от условий

Источники:

1. ООП в картинках - https://habr.com/ru/post/463125/
2. Принципы SOLID в картинках - https://habr.com/ru/company/productivity_inside/blog/505430/
3. Принципы для разработки: KISS, DRY, YAGNI, BDUF, SOLID, APO и бритва Оккама - https://habr.com/ru/company/itelma/blog/546372/
4. Три ключевых принципа ПО, которые вы должны понимать - https://habr.com/ru/post/144611/
   Принцип инверсии зависимости (DIP, из блока принципов SOLID) https://confluence.dns-shop.ru/pages/viewpage.action?pageId=39726097&preview=/39726097/39726098/DIP.pptx
5. Принципы SOLID – продолжаем разбор интересных аббревиатур (Радио 1С Энтерпрайз);
6. Швец А. Погружение в паттерны проектирования