C# Professional

Поведенческие шаблоны проектирования

otus.ru

Проверить, идет ли запись

Меня хорошо видно && слышно?

Ставим “+”, если все хорошо
“-”, если есть проблемы

Тема вебинара

Поведенческие шаблоны проектирования

Михаил Дмитриев

Ведущий программист НИПК Электрон

Разрабатываю и поддерживаю приложения для работы с радиологическими комплексами

https://t.me/sf321

Правила вебинара

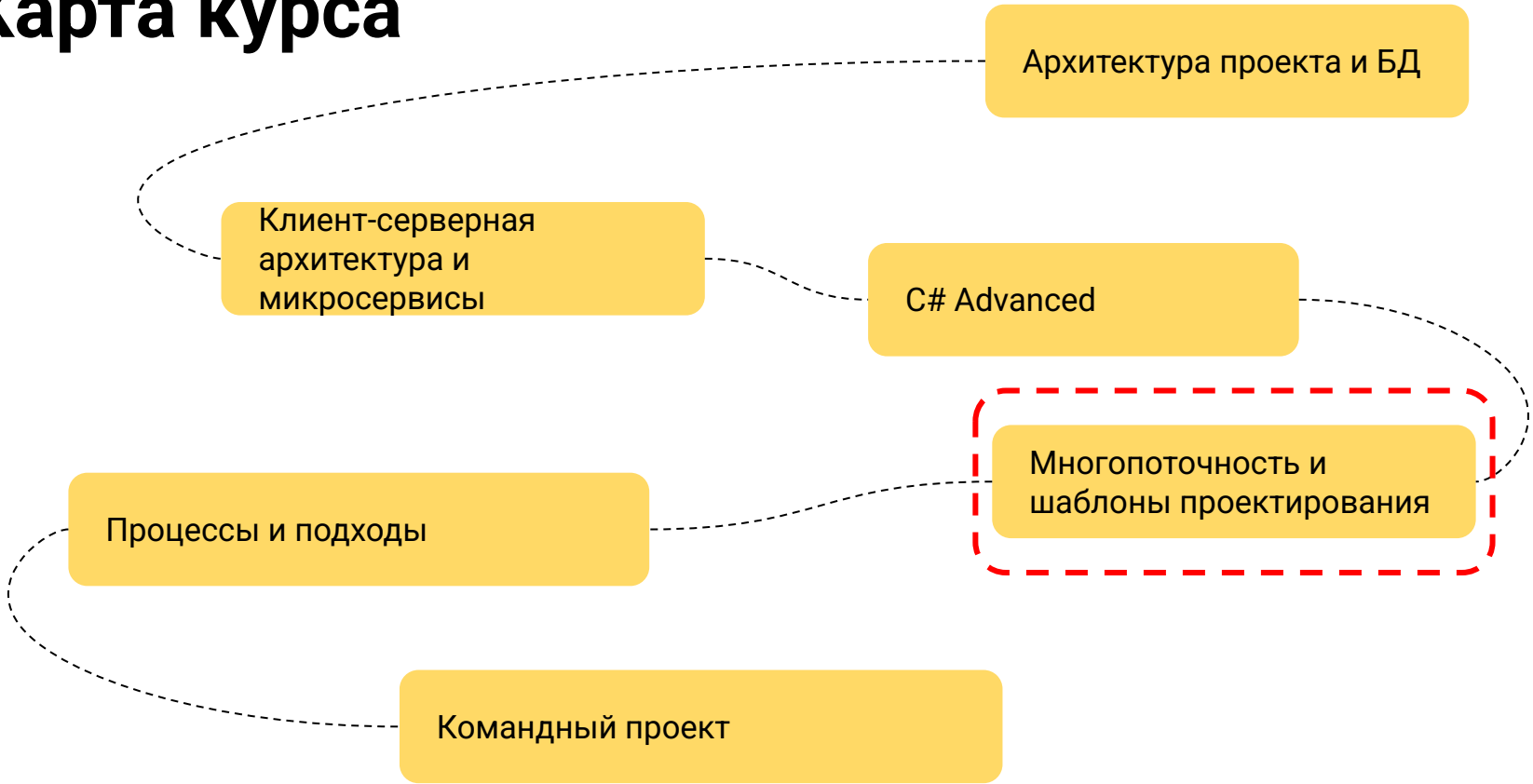
Активно
участвуем

Off-topic обсуждаем
в общем чате учебной группы
в telegram

Задаем вопрос
в чат или голосом

Вопросы вижу в чате,
могу ответить не сразу

Карта курса


Архитектура проекта и БД
Клиент-серверная архитектура и микросервисы
C# Advanced
Многопоточность и шаблоны проектирования
Процессы и подходы
Командный проект

Маршрут вебинара

- Знакомство
- Немного теории
- Проблема сложности
- Поведенческие шаблоны проектирования
- Примеры
- Рефлексия

Цели вебинара

К концу занятия вы сможете

1. Расширить знания о применении поведенческих шаблонах проектирования при разработке ПО
2. Применять на практике поведенческие шаблоны проектирования
3. Глубже понимать влияние дизайна на эффективность разработки

Смысл

Зачем вам это уметь

1. Управлять сложностью разрабатываемых вами решений
2. Снижать издержки на поддержку вашего ПО

Тестирование

Управление сложностью разработки ПО на уровне проектирования архитектуры

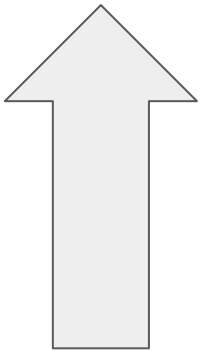
Проблема сложности при разработке ПО

Количество ресурсов , связей и компонентов, которые требуются для решения какой-либо задачи растет нелинейно по мере роста проекта.

Некоторые виды:

- алгоритмическая
- трудоемкость разработки
- информационная
- сложность тестирования

Уровни архитектуры



- инфраструктура (система) (паттерны межсервисного взаимодействия)
- компоненты системы (сервисы) (+ solid / patterns)
- модули проекта (+ solid / patterns)
- классы (+ solid / patterns)
- методы (+ clean code)

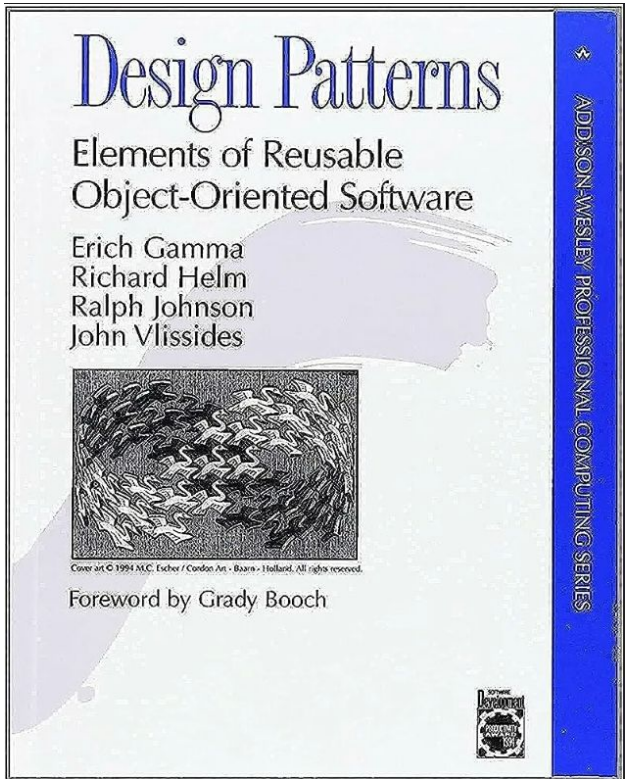
SOLID

SOLID – 5 основных принципов объектно-ориентированного программирования и проектирования:

- Принцип единственной ответственности (single responsibility principle, SRP)
- Принцип открытости/закрытости (open-closed principle, OCP)
- Принцип подстановки Лисков (Liskov substitution principle, LSP)
- Принцип разделения интерфейса (interface segregation principle, ISP)
- Принцип инверсии зависимостей (dependency inversion principle, DIP)

GoF Паттерны объектно-ориентированного проектирования

Design Patterns: Elements of Reusable Object-Oriented Software (1994)
Книгу написали
Эрих Гамма,
Ричард Хелм,
Ральф Джонсон и
Джон Влиссидес



Классификация паттернов ООП по GoF

- Порождающие паттерны дают возможность выполнять инициализацию объектов наиболее удобным и оптимальным способом.

- Структурные паттерны описывают взаимоотношения между различными классами или объектами, позволяя им совместно реализовывать поставленную задачу.

- Поведенческие паттерны позволяют грамотно организовать связь между сущностями для оптимизации и упрощения их взаимодействия.

Цель / Уровень	Порождающие паттерны	Структурные паттерны	Паттерны поведения
Класс	Фабричный метод (135)	Адаптер (171)	Интерпретатор (287) Шаблонный метод (373)
Объект	Абстрактная фабрика (113) Одиночка (157) Прототип (146) Строитель (124)	Адаптер (171) Декоратор (209) Заместитель (246) Компоновщик (196) Мост (184) Приспособленец (231) Фасад (221)	Итератор (302) Команда (275) Наблюдатель (339) Посетитель (379) Посредник (319) Состояние (352) Стратегия (362) Хранитель (330) Цепочка обязанностей (263)

Поведенческие паттерны проектирования

Поведенческие паттерны проектирования

Паттерны поведения связаны с алгоритмами и распределением обязанностей между объектами. Речь в них идет не только о самих объектах и классах, но и о типичных схемах взаимодействия между ними. Паттерны поведения характеризуют сложный поток управления, который трудно проследить во время выполнения программы. Внимание акцентируется не на схеме управления как таковой, а на связях между объектами

При этом могут использоваться следующие механизмы:

- В паттернах поведения уровня класса для распределения поведения между разными классами используется наследование

- В паттернах поведения уровня объектов используется не наследование, а композиция

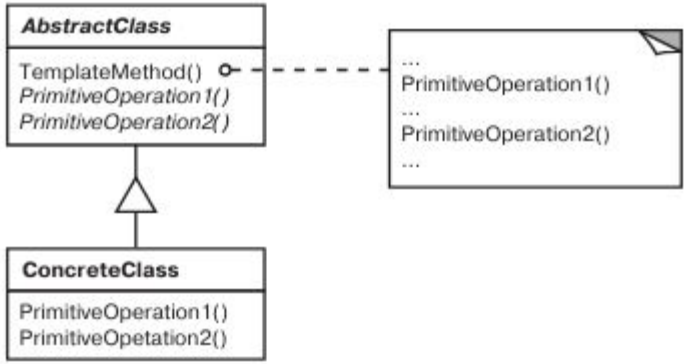
Поведенческие паттерны проектирования

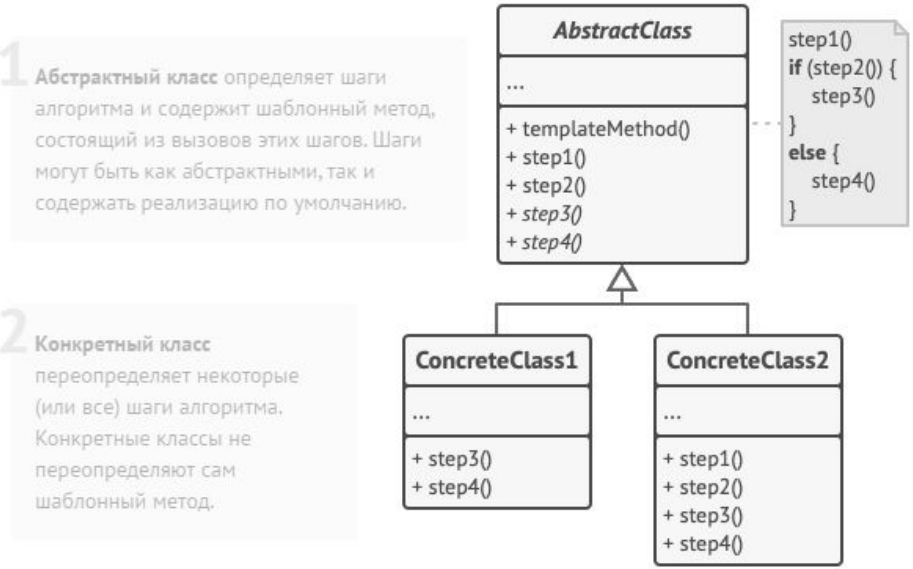
Список поведенческих шаблонов проектирования:

- Template method(Шаблонный метод)
- Стратегия (Strategy)
- Команда(Command)
- Наблюдатель(Observer)
- Посредник (Mediator)
- Посетитель (Visitor)
- Итератор (Iterator)
- Состояние (State)
- Снимок (Memento)
- Цепочка обязанностей (Chain of Responsibility)
- Интерпретатор (Interpreter)*

Template method (Шаблонный метод)

Шаблонный метод определяет основу алгоритма и позволяет подклассам переопределить некоторые шаги алгоритма, не изменяя его структуру в целом.



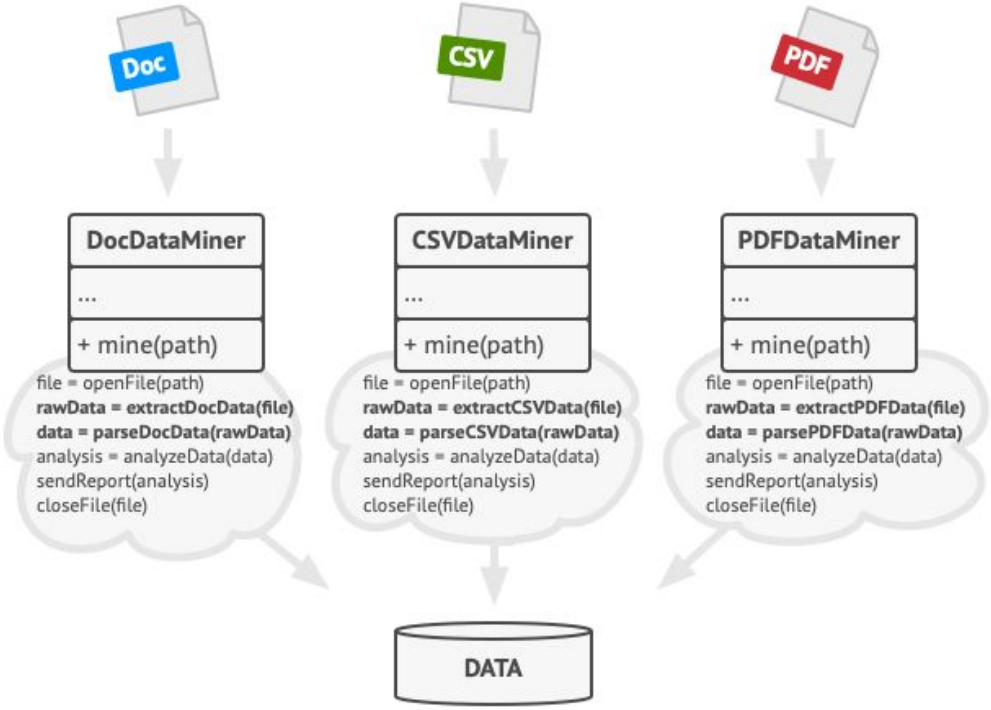


Template method(Шаблонный метод)

Основные условия для применения паттерна шаблонный метод:

- однократное использование инвариантных частей алгоритма, при этом реализация изменяющегося поведения остается на усмотрение подклассов;
- необходимость вычленить и локализовать в одном классе поведение, общее для всех подклассов, чтобы избежать дублирования кода. Это хороший пример техники «вынесения за скобки с целью обобщения». Сначала выявляются различия в существующем коде, которые затем выносятся в отдельные операции.В конечном итоге различающиеся фрагменты кода заменяются шаблонным методом, из которого вызываются новые операции
- управление расширениями подклассов. Шаблонный метод можно определить так, что он будет вызывать операции-зацепки (hooks) в определенных точках, разрешив тем самым расширение только в этих точках.

Template method (Шаблонный метод)

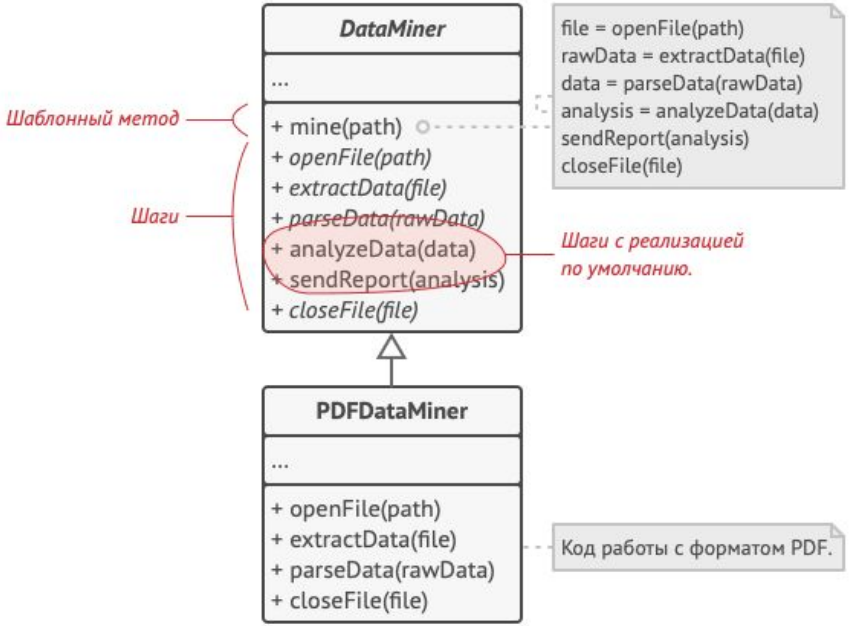


Классы дата-майнинга содержат много дублирования.

Вы пишете программу для дата-майнинга в офисных документах. Пользователи будут загружать в неё документы в разных форматах (PDF, DOC, CSV), а программа должна извлекать из них полезную информацию.

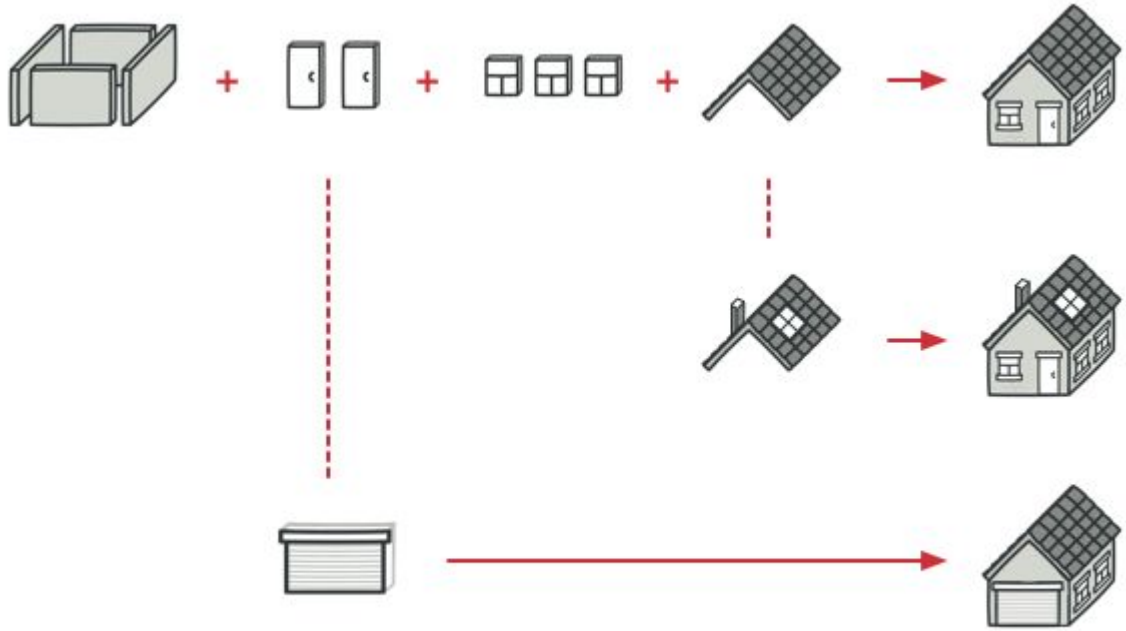
В первой версии вы ограничились только обработкой DOC-файлов. В следующей версии добавили поддержку CSV. А через месяц прикрутили работу с PDF-документами.

Template method (Шаблонный метод)



Шаблонный метод разбивает алгоритм на шаги, позволяя подклассам переопределить некоторые из них.

Template method (Шаблонный метод)

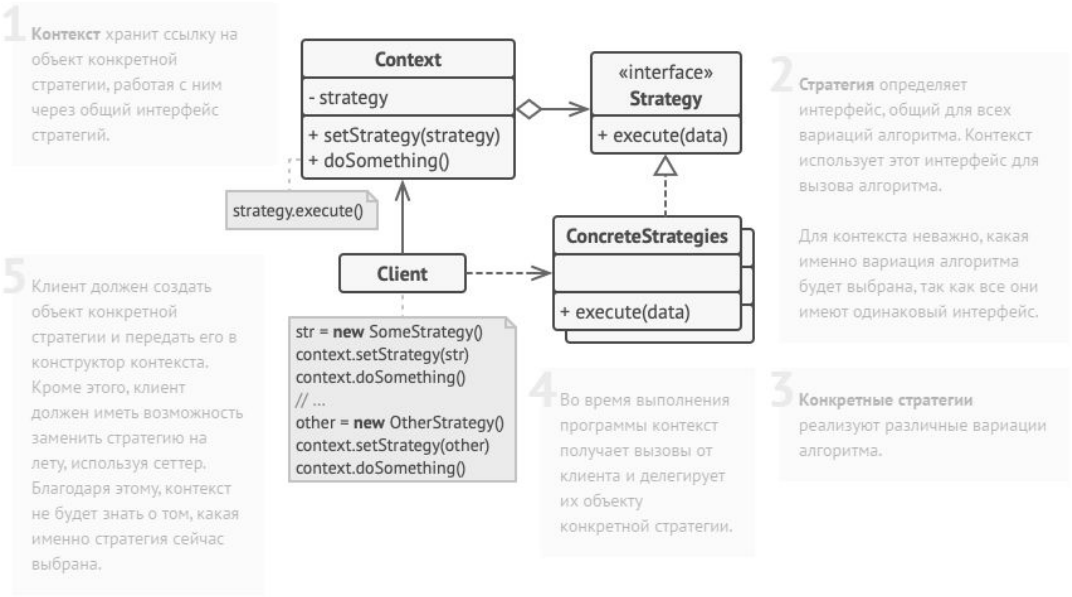


Проект типового дома могут немного изменить по желанию клиента.

LIVE

Стратегия (Strategy)

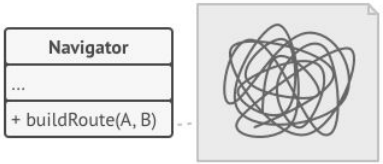
Стратегия определяет семейство схожих алгоритмов и помещает каждый из них в собственный класс, после чего алгоритмы можно взаимозаменять прямо во время исполнения программы.



Условия применимости:

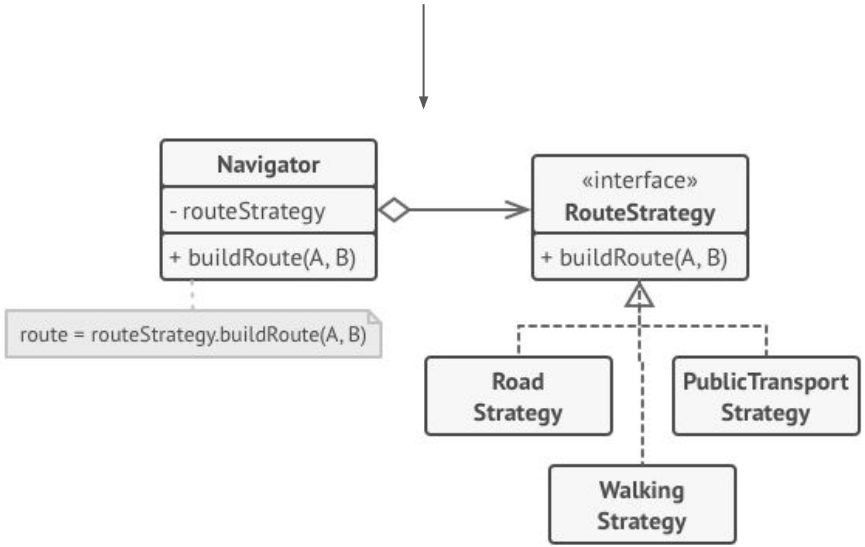
- наличие множества родственных классов, отличающихся только поведением. Стратегия позволяет настроить класс одним из многих возможных вариантов поведения;
- наличие нескольких разновидностей алгоритма. Например, можно определить два варианта алгоритма, один из которых требует больше времени, а другой — больше памяти.
- в алгоритме содержатся данные, о которых клиент не должен «знать»
- в классе определено много вариантов поведения, представленных разветвленными условными операторами

Стратегия (Strategy)



Код навигатора становится слишком раздутым.

Навигатор - построение маршрутов для различных способов передвижения.



Стратегии построения пути.

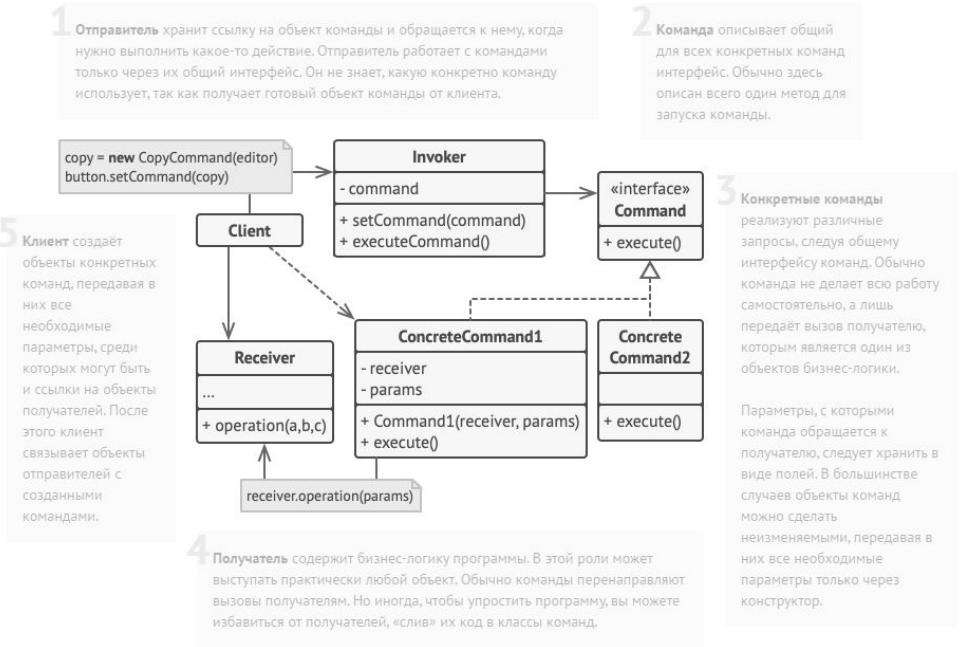
Хотя каждый класс будет прокладывать маршрут по-своему, для навигатора это не будет иметь никакого значения, так как его работа заключается только в отрисовке маршрута. Навигатору достаточно подать в стратегию данные о начале и конце маршрута, чтобы получить массив точек маршрута в оговоренном формате.

Класс навигатора будет иметь метод для установки стратегии, позволяя изменять стратегию поиска пути на лету. Такой метод пригодится клиентскому коду навигатора, например, переключателям типов маршрутов в пользовательском интерфейсе.

LIVE

Команда(Command, Действие, Транзакция, Action)

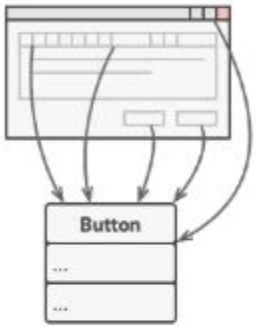
Паттерн **команда** преобразует запросы в объекты, позволяя передавать их как аргументы при вызове методов, ставить запросы в очередь, логировать их, а также поддерживать отмену операций.

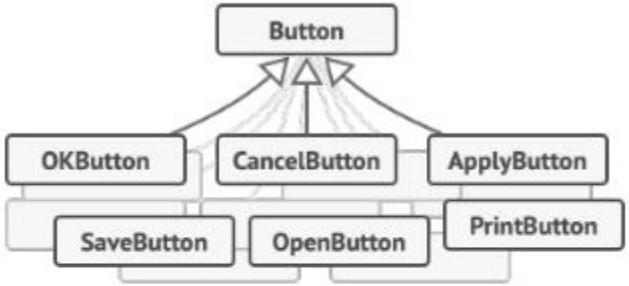


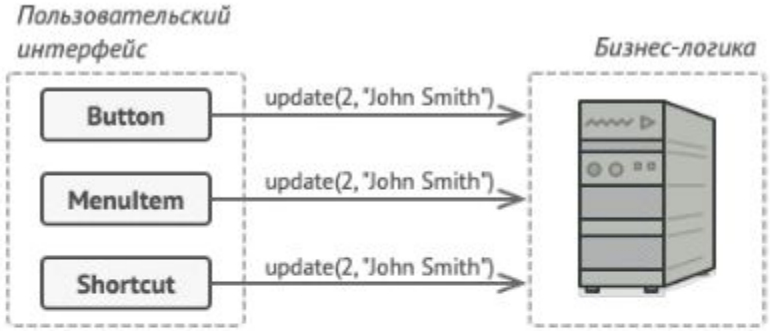
Условия применимости:

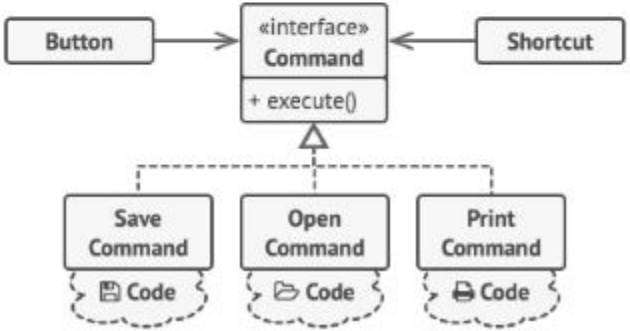
- параметризация объектов выполняемым действием(пользовательский интерфейс)
- определение, постановка в очередь и выполнение запросов в разное время.
- поддержка отмены операций.
- поддержка протоколирования изменений, чтобы их можно было выполнить повторно после сбоя в системе
- структурирование системы на основе высокоуровневых операций, построенных из примитивных (Макросы)

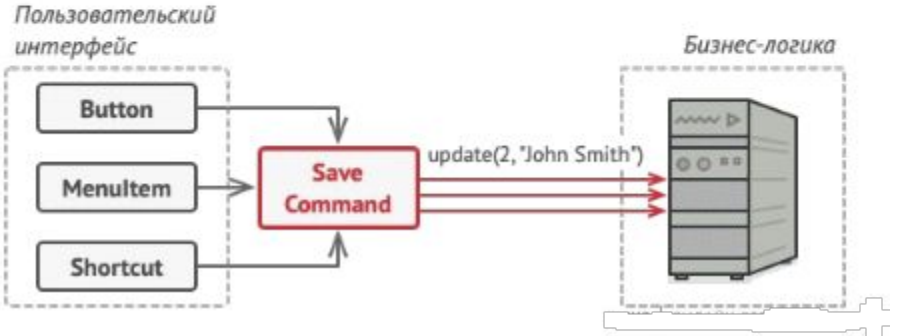
Команда(Command, Действие, Транзакция, Action)











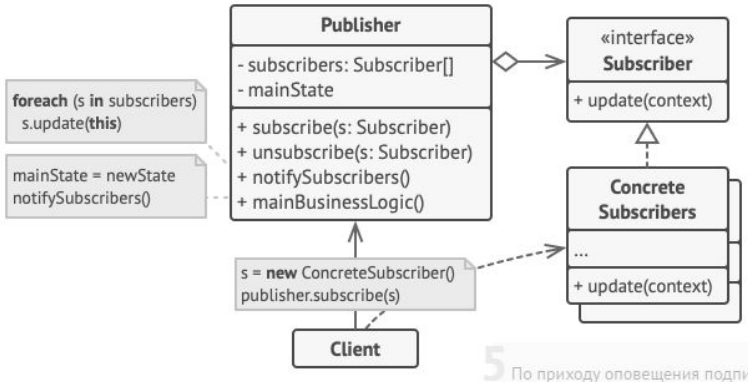
LIVE

Наблюдатель(Observer, Издатель-Подписчик, Слушатель)

Паттерн **наблюдатель** создает механизм подписки, позволяющий одним объектам следить и реагировать на события, происходящие в других

1 **Издатель** владеет внутренним состоянием, изменение которого интересно отслеживать подписчикам. Издатель содержит механизм подписки: список подписчиков и 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 общем интерфейсе подписчиков.



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.

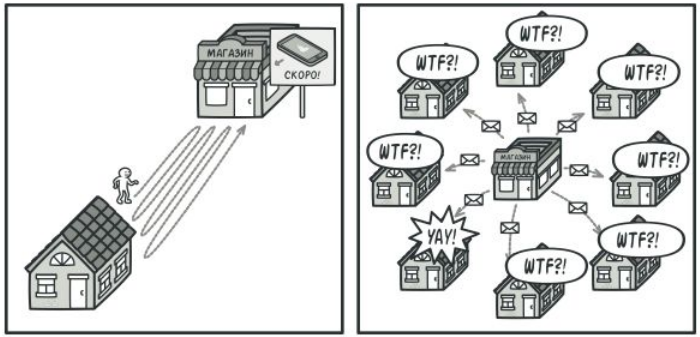
5 По приходу оповещения подписчику нужно получить обновлённое состояние издателя. Издатель может передать это состояние через параметры метода оповещения. Более гибкий вариант — передавать через параметры весь объект издателя, чтобы подписчик мог сам получить требуемые данные. Как вариант, подписчик может постоянно хранить ссылку на объект издателя, переданный ему в конструкторе.

6 **Клиент** создаёт объекты издателей и подписчиков, а затем регистрирует подписчиков на обновления в издателях.

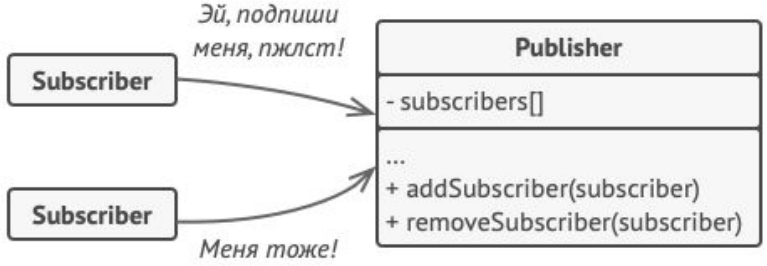
Условия применимости:

- у абстракции есть два аспекта, один из которых зависит от другого. Инкапсуляции этих аспектов в разные объекты позволяют изменять и повторно использовать их независимо;
- при модификации одного объекта требуется изменить другие, и вы не знаете, сколько именно объектов нужно изменить;
- один объект должен оповещать других, не делая предположений об объектах-наблюдателях. Другими словами, объекты не должны быть тесно связаны между собой.

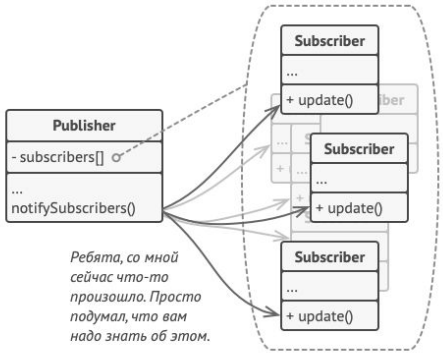
Наблюдатель(Observer, Издатель-Подписчик, Слушатель)



Постоянное посещение магазина или спам?



Подписка на события.



Оповещения о событиях.

Подписка на газеты и их доставка.

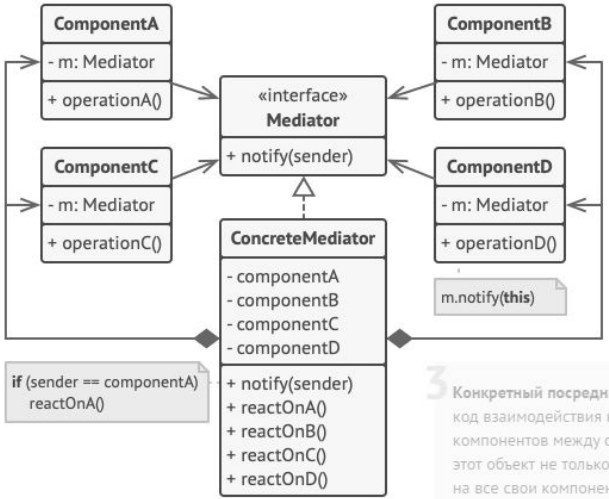
LIVE

Посредник (Mediator, Intermediary, Controller)

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

1 **Компоненты** — это разнородные объекты, содержащие бизнес-логику программы. Каждый компонент хранит ссылку на объект посредника, но работает с ним только через абстрактный интерфейс посредников. Благодаря этому, компоненты можно повторно использовать в другой программе, связав их с посредником другого типа.

2 **Посредник** определяет интерфейс для обмена информацией с компонентами. Обычно хватает одного метода, чтобы оповещать посредника о событиях, произошедших в компонентах. В параметрах этого метода можно передавать детали события: ссылку на компонент, в котором оно произошло, и любые другие данные.



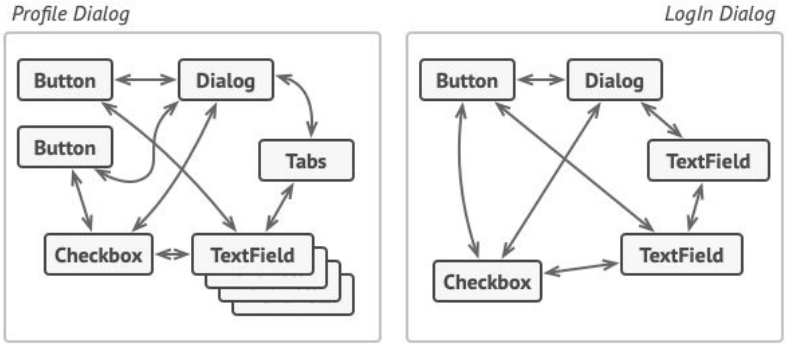
3 **Конкретный посредник** содержит код взаимодействия нескольких компонентов между собой. Зачастую этот объект не только хранит ссылки на все свои компоненты, но и сам их создаёт, управляя дальнейшим жизненным циклом.

4 Компоненты не должны общаться друг с другом напрямую. Если в компоненте происходит важное событие, он должен оповестить своего посредника, а тот сам решит — касается ли событие других компонентов, и стоит ли их оповещать. При этом компонент-отправитель не знает кто обработает его запрос, а компонент-получатель не знает кто его прислал.

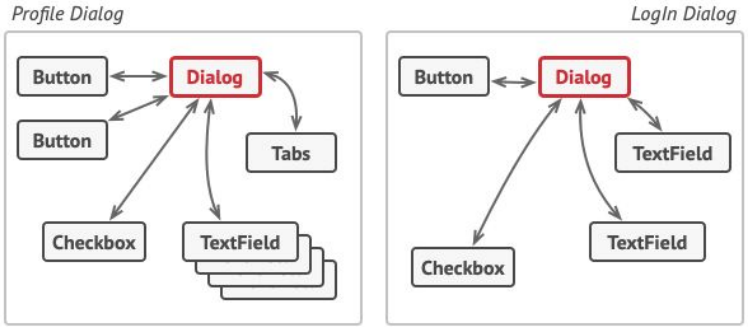
Условия применимости:

- существование объектов, связи между которыми сложны и четко определены. Получающиеся при этом взаимозависимости не структурированы и трудны для понимания;
- повторное использование объекта затруднено, поскольку он обменивается информацией со многими другими объектами;
- поведение, распределенное между несколькими классами, должно настраиваться без порождения множества подклассов.

Посредник (Mediator, Intermediary, Controller)

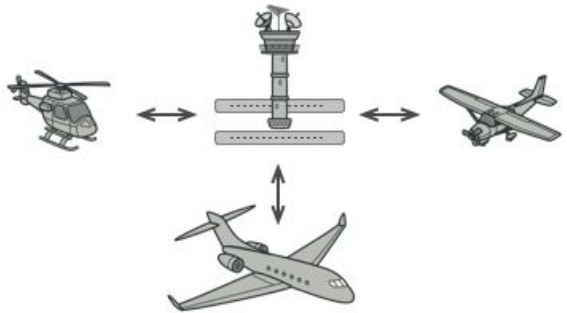


Беспорядочные связи между элементами пользовательского интерфейса.



Элементы интерфейса общаются через посредника.

Код элементов нужно трогать при изменении каждого диалога.

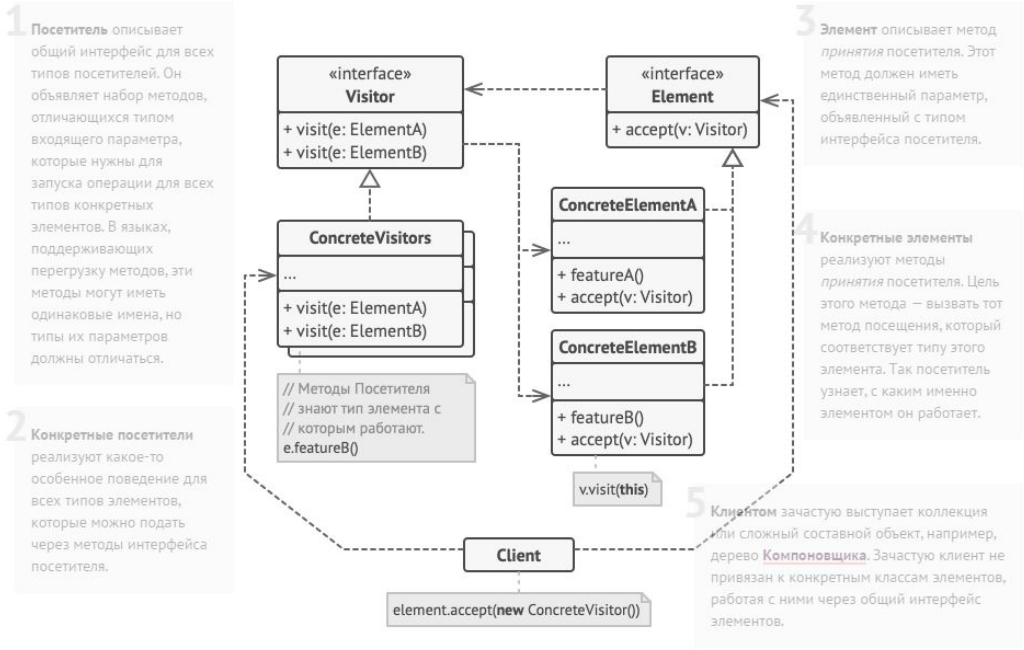


Пилоты самолётов общаются не напрямую, а через диспетчера.

LIVE

Посетитель (Visitor)

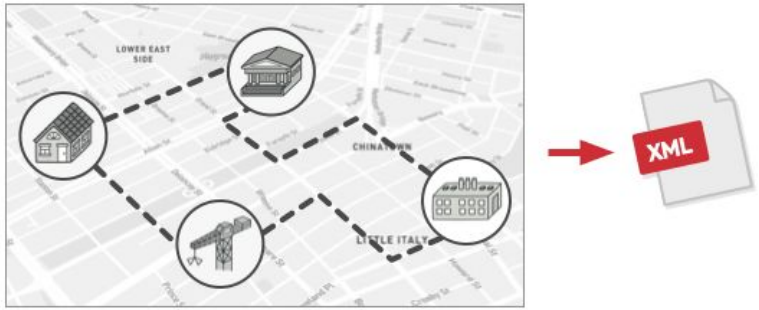
Паттерн **Visitor** описывает операцию, выполняемую с каждым объектом из некоторой структуры. Позволяет определить новую операцию, не изменяя классы этих объектов



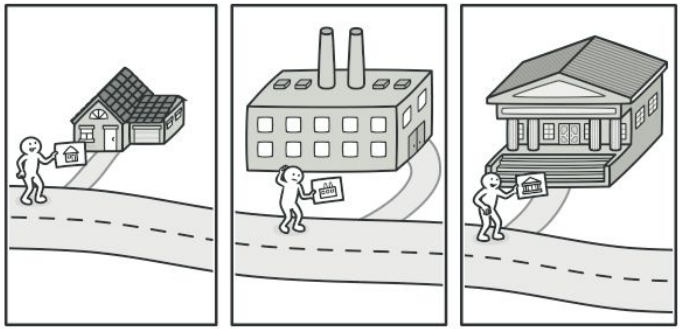
Условия применимости:

- в структуре присутствуют объекты многих классов с различными интерфейсами, и вы хотите выполнять над ними операции, зависящие от конкретных классов;
- над объектами, входящими в состав структуры, должны выполняться разнообразные, не связанные между собой операции и вы не хотите «засорять» классы такими операциями. Посетитель позволяет объединить родственные операции, поместив их в один класс. Если структура объектов является общей для нескольких приложений, то паттерн посетитель позволит в каждое приложение включить только относящиеся к нему операции;
- классы, определяющие структуру объектов, изменяются редко, но новые операции над этой структурой добавляются часто. При изменении классов, представленных в структуре, придется переопределить интерфейсы всех посетителей, а это может вызвать затруднения. Поэтому если классы меняются достаточно часто, то, вероятно, лучше определить операции прямо в них.

Посетитель (Visitor)

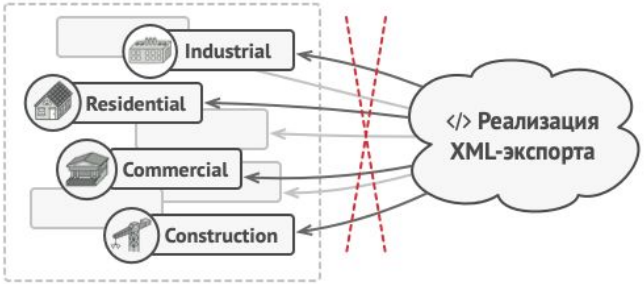


Экспорт геоузлов в XML.

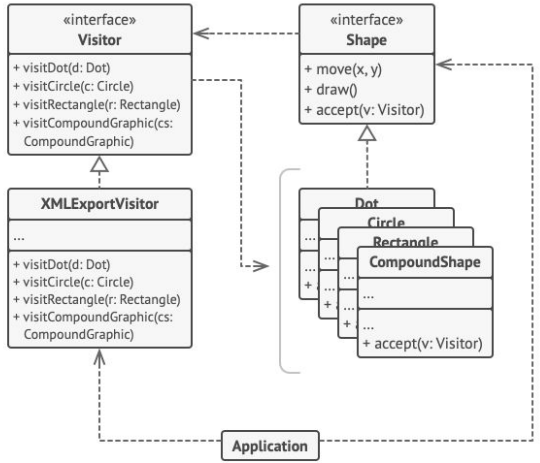


У страхового агента приготовлены полисы для разных видов организаций.

Существующие классы



Код XML-экспорта придётся добавить во все классы узлов, а это слишком накладно.

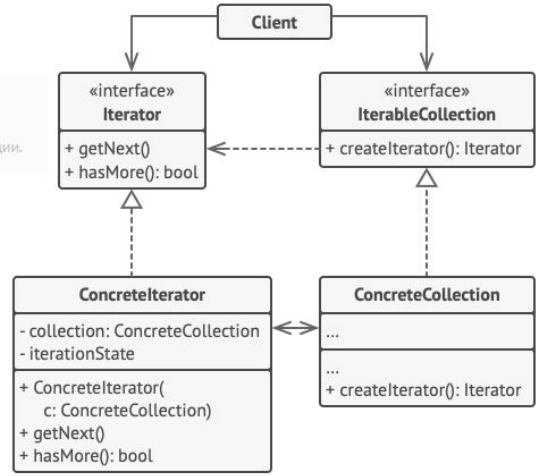


Пример организации экспорта объектов в XML через отдельный класс-посетитель.

LIVE

Итератор (Iterator)

Паттерн Iterator предоставляет способ последовательного обращения ко всем элементам составного объекта без раскрытия его внутреннего представления.



1 **Итератор** описывает интерфейс для доступа и обхода элементов коллекции.

2 **Конкретный итератор** реализует алгоритм обхода какой-то конкретной коллекции. Объект итератора должен сам отслеживать текущую позицию при обходе коллекции, чтобы отдельные итераторы могли обходить одну и ту же коллекцию независимо.

3 **Коллекция** описывает интерфейс получения итератора из коллекции. Как мы уже говорили, коллекции не всегда являются списком. Это может быть и база данных, и удалённое API, и даже дерево Компоновщика. Поэтому сама коллекция может создавать итераторы, так как она знает, какие именно итераторы способны с ней работать.

4 **Конкретная коллекция** возвращает новый экземпляр определённого конкретного итератора, связав его с текущим объектом коллекции. Обратите внимание, что сигнатура метода возвращает интерфейс итератора. Это позволяет клиенту не зависеть от конкретных классов итераторов.

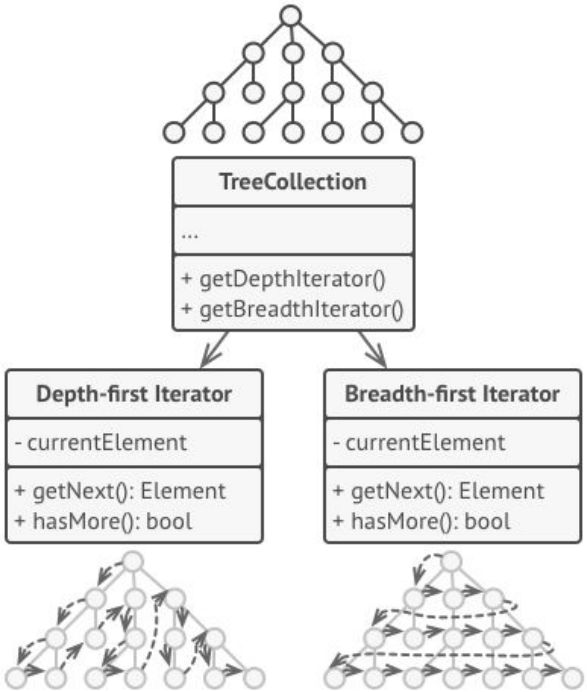
5 **Клиент** работает со всеми объектами через интерфейсы коллекции и итератора. Так клиентский код не зависит от конкретных классов, что позволяет применять различные итераторы, не изменяя существующий код программы.

В общем случае клиенты не создают объекты итераторов, а получают их из коллекций. Тем не менее, если клиенту требуется специальный итератор, он всегда может создать его самостоятельно.

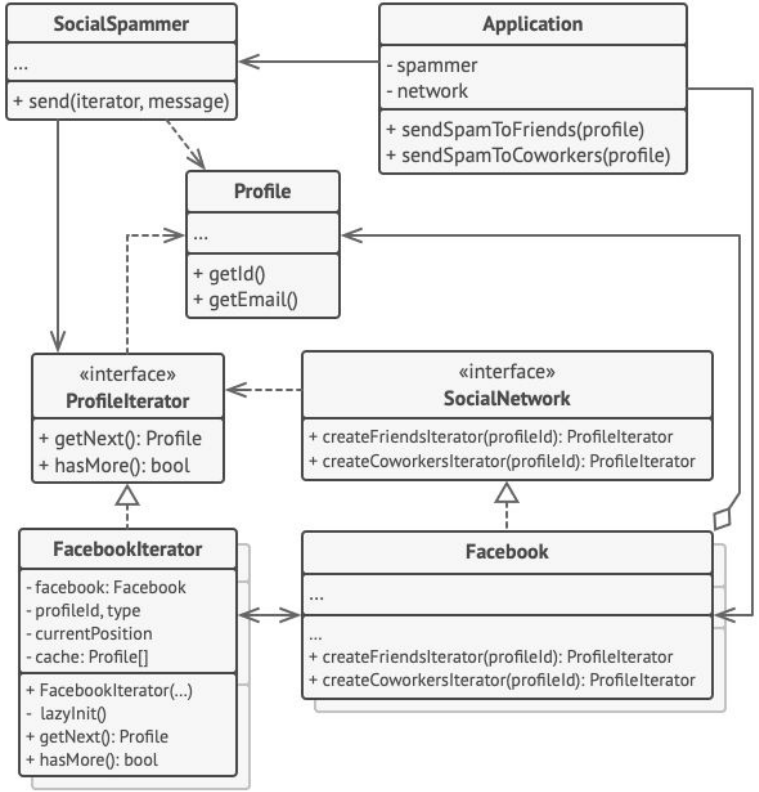
Условия применимости:

- обращение к содержимому агрегированных объектов без раскрытия их внутреннего представления;
- поддержка нескольких активных обходов одного и того же агрегированного объекта;
- предоставление единообразного интерфейса для обхода различных агрегированных структур (то есть для поддержки полиморфной итерации).

Итератор (Iterator)



Итераторы содержат код обхода коллекции. Одну коллекцию могут обходить сразу несколько итераторов.

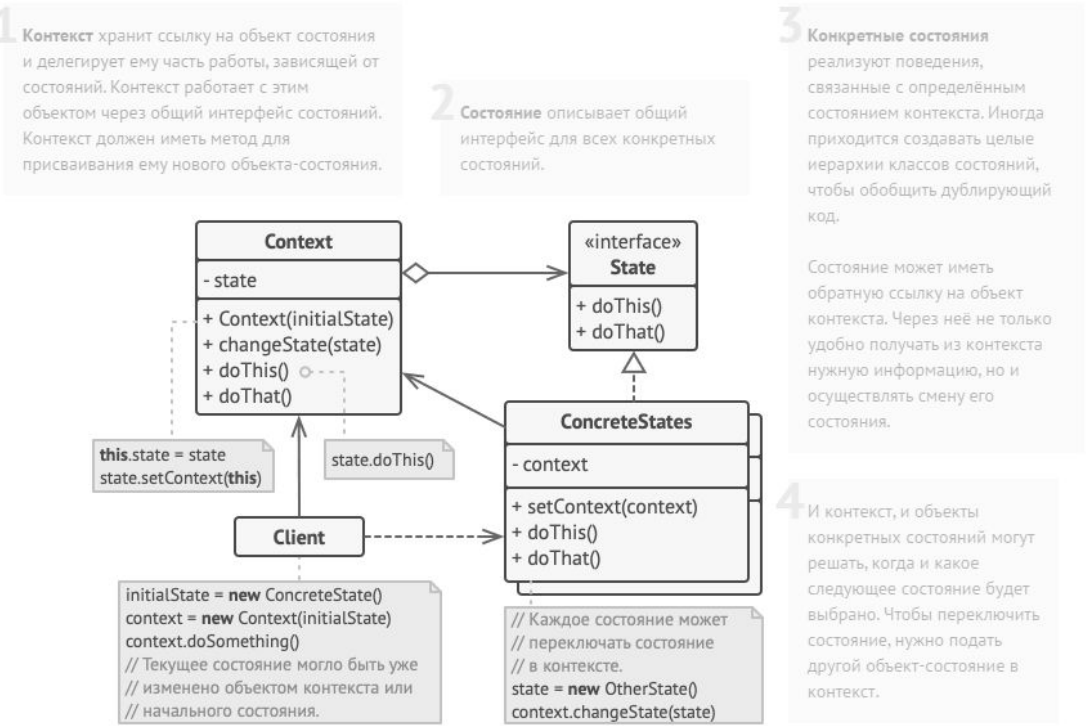


Пример обхода социальных профилей через итератор.

LIVE

Состояние (State)

Паттерн State позволяет объектам менять поведение в зависимости от своего состояния. Извне создается впечатление, что изменился класс объекта.

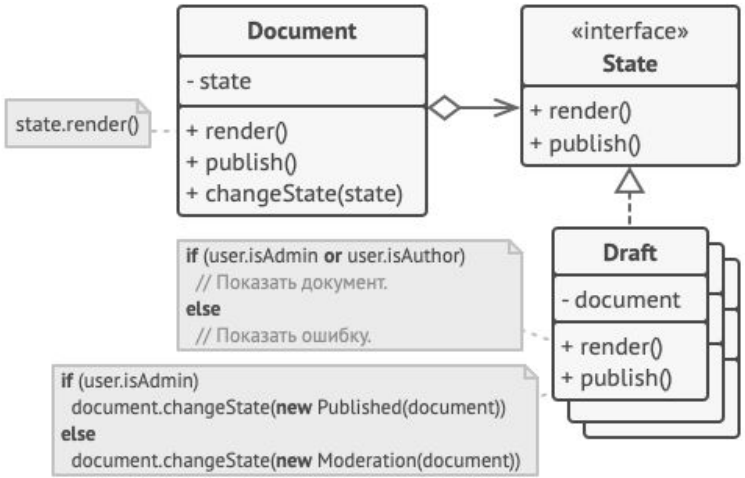


Условия применимости:

- поведение объекта зависит от его состояния и должно изменяться во время выполнения;
- когда в коде операций встречаются состоящие из многих ветвей условные операторы, в которых выбор ветви зависит от состояния. Обычно в таком случае состояние представлено перечисляемыми константами. Часто одна и та же структура условного оператора повторяется в нескольких операциях. Паттерн состояние предлагает поместить каждую ветвь в отдельный класс. Это позволяет трактовать состояние объекта как самостоятельный объект, который может изменяться независимо от других.

Состояние (State)

Возможные состояния документа и переходы между ними.



Документ делегирует работу своему активному объекту-состоянию.

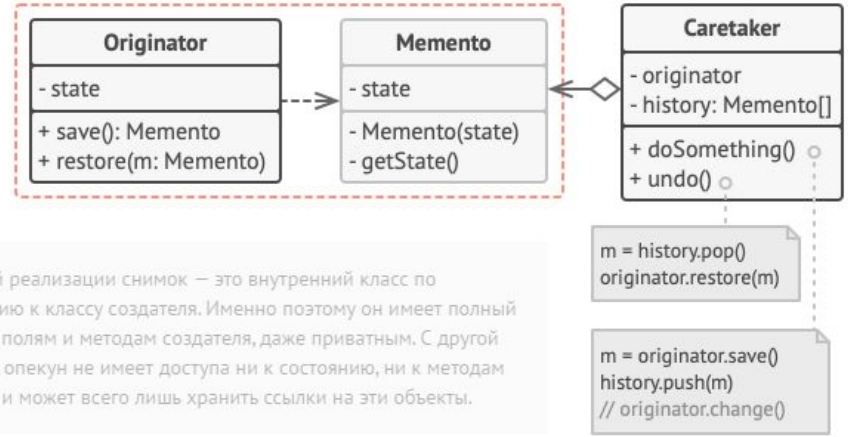
LIVE

Снимок (Memento)

Паттерн снимок не нарушая инкапсуляции, фиксирует и выносит за пределы объекта его внутреннее состояние, так чтобы позднее можно было восстановить в

1 **Создатель** может производить снимки своего состояния, а также воспроизводить прошлое состояние, если подать в него готовый снимок.

2 **Снимок** — это простой объект данных, содержащий состояние создателя. Надёжнее всего сделать объекты снимков неизменяемыми, передавая в них состояние только через конструктор.



3 **Опекун** должен знать, когда делать снимок создателя и когда его нужно восстанавливать.

Опекун может хранить историю прошлых состояний создателя в виде стека из снимков. Когда понадобится отменить выполненную операцию, он возьмёт «верхний» снимок из стека и передаст его создателю для восстановления.

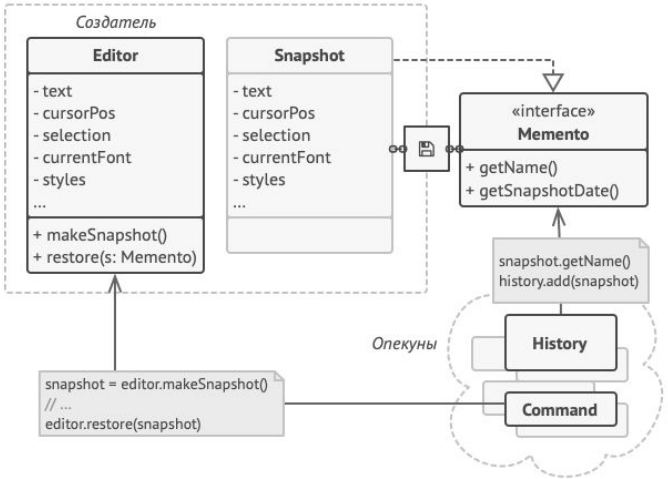
4 В данной реализации снимок — это внутренний класс по отношению к классу создателя. Именно поэтому он имеет полный доступ к полям и методам создателя, даже приватным. С другой стороны, опекун не имеет доступа ни к состоянию, ни к методам снимков и может всего лишь хранить ссылки на эти объекты.

Условия применимости:

- необходимость сохранения снимка состояния объекта (или его части),чтобы впоследствии объект можно было восстановить в том же состоянии
- прямой интерфейс для получения этого состояния привел бы к раскрытию подробностей реализации и нарушению инкапсуляции объекта.

Снимок (Memento)

Перед выполнением команды вы можете сохранить копию состояния редактора, чтобы потом иметь возможность отменить операцию.



Снимок полностью открыт для создателя, но лишь частично открыт для опекунов.

LIVE

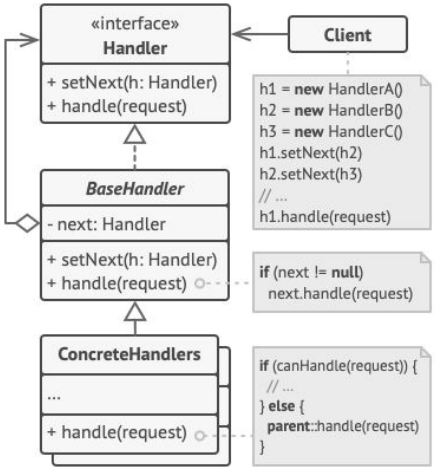
Цепочка обязанностей (Chain of Responsibility)

Паттерн цепочка обязанностей позволяет избежать привязки отправителя запроса к его получателю, предоставляя возможность обработать запрос нескольким объектам. Связывает объекты-получатели в цепочку и передает запрос по этой цепочке, пока он не будет обработан.

1 **Обработчик** определяет общий для всех конкретных обработчиков интерфейс. Обычно достаточно описать единственный метод обработки запросов, но иногда здесь может быть объявлен и метод выставления следующего обработчика.

2 **Базовый обработчик** — опциональный класс, который позволяет избавиться от дублирования одного и того же кода во всех конкретных обработчиках.

Обычно этот класс имеет поле для хранения ссылки на следующий обработчик в цепочке. Клиент связывает обработчики в цепь, подавая ссылку на следующий обработчик через конструктор или сеттер поля. Также здесь можно реализовать базовый метод обработки, который бы просто перенаправлял запрос следующему обработчику, проверив его наличие.



4 **Клиент** может либо сформировать цепочку обработчиков единожды, либо перестраивать её динамически, в зависимости от логики программы. Клиент может отправлять запросы любому из объектов цепочки, не обязательно первому из них.

3 **Конкретные обработчики** содержат код обработки запросов. При получении запроса каждый обработчик решает, может ли он обработать запрос, а также стоит ли передать его следующему объекту.

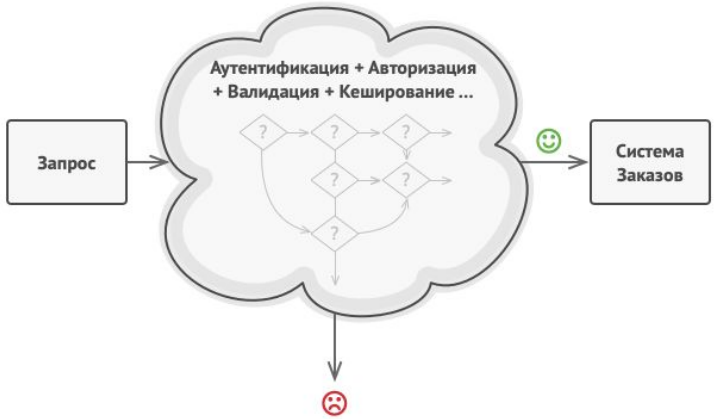
В большинстве случаев обработчики могут работать сами по себе и быть неизменяемыми, получив все нужные детали через параметры конструктора.

Условия применимости:

- запрос может быть обработан более чем одним объектом, причем настоящий обработчик заранее неизвестен и должен быть найден автоматически;
- запрос должен быть отправлен одному из нескольких объектов, без явного указания, какому именно;
- набор объектов, способных обработать запрос, должен задаваться динамически.

Цепочка обязанностей (Chain of Responsibility)

Запрос проходит ряд проверок перед доступом в систему заказов.



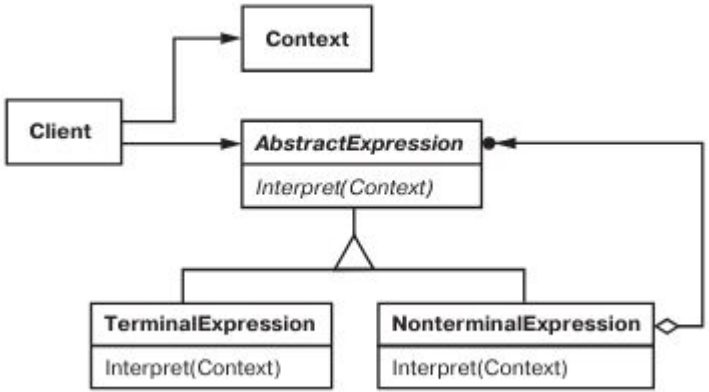
Со временем код проверок становится всё более запутанным.

Обработчики следуют в цепочке один за другим.

LIVE

Интерпретатор (Interpreter)*

Паттерн интерпретатор для заданного языка определяет представление его грамматики, а также интерпретатор предложений этого языка.



Условия применимости:

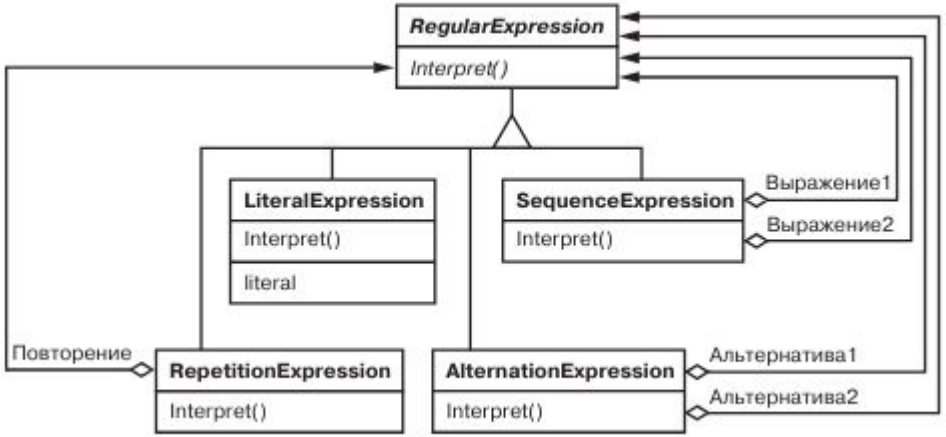
- Используйте паттерн интерпретатор в ситуациях, когда имеется интерпретируемый язык, конструкции которого можно представить в виде абстрактных синтаксических деревьев. Этот паттерн лучше всего работает в следующих случаях:
- грамматика проста. Для сложных грамматик иерархия классов становится слишком громоздкой и неуправляемой. В таких случаях лучше применять парсеры-генераторы, поскольку они могут интерпретировать выражения без построения абстрактных синтаксических деревьев, что экономит память (а возможно, и время)
- эффективность не является главным критерием. Наиболее эффективные интерпретаторы обычно не работают непосредственно с деревьями, а сначала транслируют их в другую форму. Так, регулярное выражение часто преобразуется в конечный автомат. Но даже в этом случае сам транслятор можно реализовать с помощью паттерна интерпретатор.

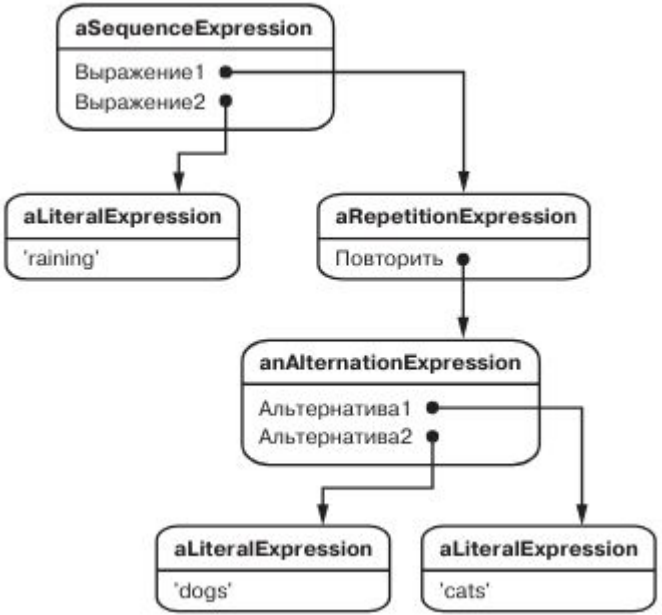
Интерпретатор (Interpreter)*

Допустим, они описываются следующей грамматикой:

```
expression ::= literal | alternation | sequence | repetition |
               '(' expression ')'
alternation ::= expression '|' expression
sequence ::= expression '&' expression
repetition ::= expression '*'
literal ::= 'a' | 'b' | 'c' | ... { 'a' | 'b' | 'c' | ... }*
```

где `expression` — начальный символ, а `literal` — терминальный символ, определяющий простые слова.





raining & (dogs | cats) *

LIVE

Список материалов для изучения

1. Гради Буч “Объектно-ориентированный анализ и проектирование”
2. Фредерик Брукс “Мифический человеко месяц”
3. Влиссидес Джон, Хелм Ричард “Паттерны объектно-ориентированного проектирования”
4. Крейг Ларман «Применение UML и шаблонов проектирования»
5. Сергей Тепляков “Паттерны проектирования на платформе .NET”
6. https://refactoring.guru/ru/design-patterns/behavioral-patterns
7. https://gitlab.com/otus-demo/behavioral-design-pattern/
8. https://learn.microsoft.com/en-us/dotnet/architecture/microservices/microservice-ddd-cqrs-patterns/microservice-application-layer-implementation-web-api
9. https://github.com/nemanjarogic/DesignPatternsLibrary
10. https://code-maze.com/design-patterns-csharp/
11. https://code-maze.com/csharp-visitor-design-pattern/
12. https://medium.com/codenx/implement-mediator-pattern-with-mediatr-in-c-8a271d7b9901

Вопросы?

Ставим “+”,
если вопросы есть

Ставим “–”,
если вопросов нет

Рефлексия

Заполните, пожалуйста, опрос о занятии по ссылке в чате

Спасибо за внимание!

Приходите на следующие вебинары

Михаил Дмитриев

Ведущий программист НИПК Электрон

Разрабатываю и поддерживаю приложения для работы с радиологическими комплексами

https://t.me/sf321