Паттерны объектно-ориентированного программирования. ч.2

Ключевые принципы для повторного использования

(интерфейсы, композиция)

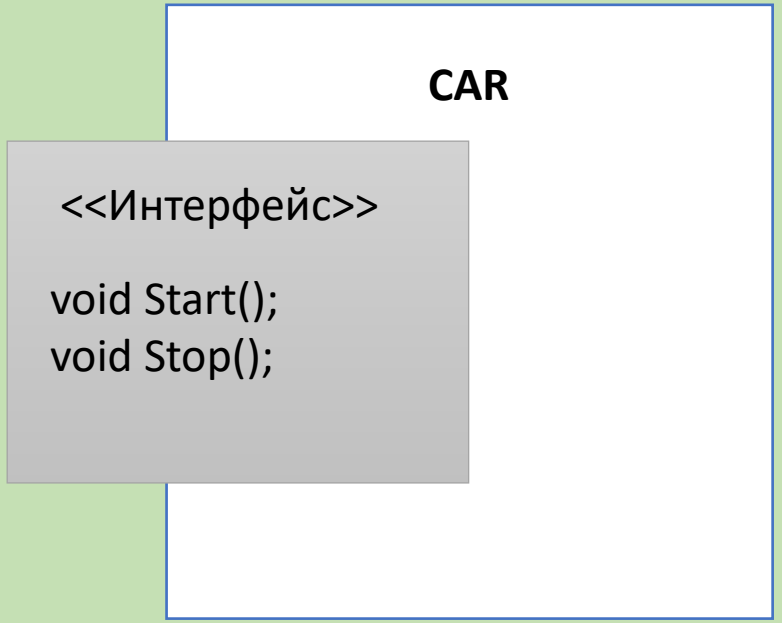
Порождающие паттерны: фабричный метод, абстрактная фабрика, строитель, синглтон

Структурные паттерны: адаптер, прокси

Интерфейс

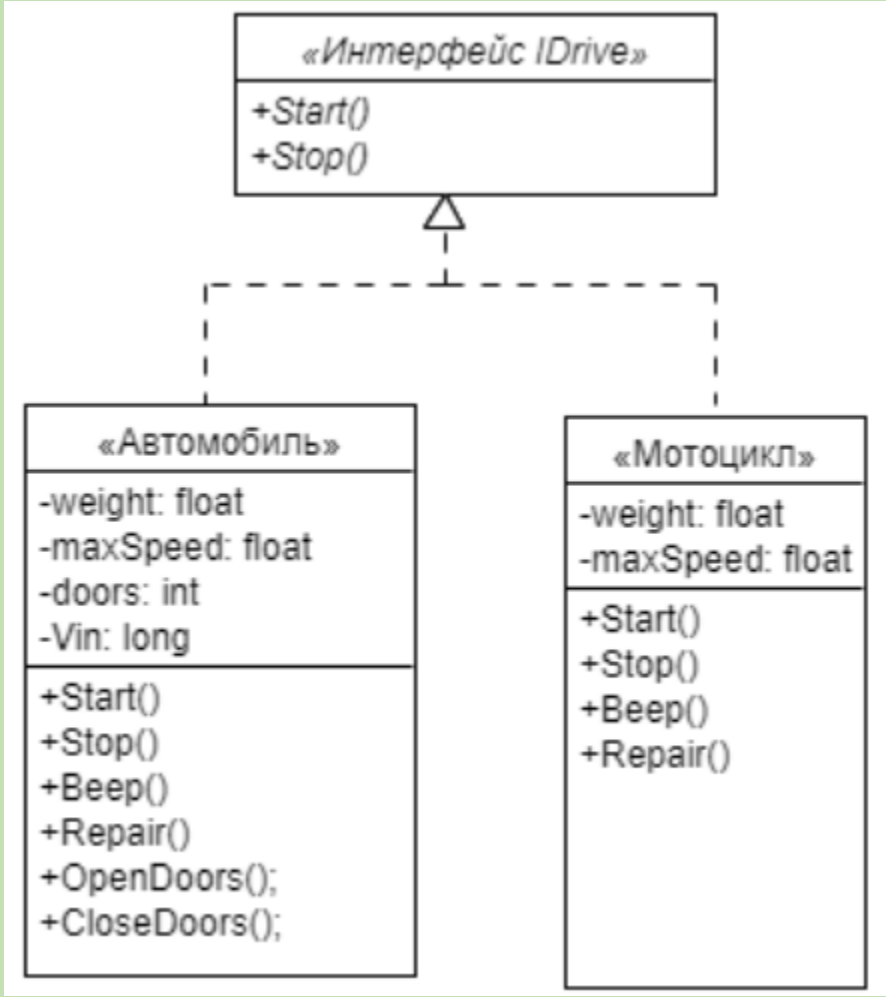
- *«Интерфейс* описывает множество запросов, которые можно отправить объекту» (GoF), интерфейс формализует способы взаимодействия с объектом

- Интерфейс не определяет реализацию операций, *не конкретизирует,* что должен делать объект в ответ на запросы.
- Разные объекты могут вправе реализовывать одни и те же запросы интерфейса по-разному.
- Два объекта с разными реализациями могут иметь одинаковые интерфейсы
- За счет динамического полиморфизма (связывания) работая с объектом через интерфейс, можно получать разное поведение в зависимости от конкретного типа объекта



Наследование интерфейса

- Интерфейсы позволяют унифицировано работать с разными объектами (разных классов), которые имеют разные реализации
- Для этого необходимо вынести интерфейсную часть объекта в отдельную сущность (класс) и ввести наследование от таких классов.
- В C++ для максимальной «чистоты» базового интерфейсного класса применяются чистые виртуальные функции: в этом случае интерфейсный класс *исключительно заявляет* об операциях (методах), которые должны быть у производного класса, но не определяет их
- Такие «чистые абстракции» (без полей и определений) в разных языках (н-р, C#, Java) выделяют в отдельный тип сущностей (**interface**)



Наследование класса vs. наследование «интерфейса»

- Наследование класса = наследование реализации
- Наследование класса моделирует отношение «является» или «A is B»: «Автомобиль» является «Транспортным средством»

Базовый класс: Animal

Производные классы:Cat,Mouse,Horse, Rat

- Наследование интерфейса = реализация функциональности, которая заявлена в интерфейсе
- Наследование интерфейса моделирует отношение «может» или «A can B»:
- «Автомобиль» может «Ездить»

Базовый класс: IMove

Производные классы: Cat, Mouse, Horse, Car, Train, Man, Wind, ..

«Программируйте в соответствии с интерфейсами, а не с реализацией»

- У манипулирования объектами строго через интерфейс абстрактного класса есть следующие преимущества
- Клиенту не нужно располагать информацией о конкретных типах объектов, которыми он пользуется, при условии что все они имеют ожидаемый клиентом интерфейс
- Такой подход существенно уменьшает число зависимостей между подсистемами
- Не объявляйте переменные как экземпляры конкретных классов. Вместо этого придерживайтесь интерфейса, определенного абстрактным классом.

```
class Client{
public:
// клиент не зависит
// от реализации IDrive
 void Action(IDrive drivable)
 {
    drivable.Start();
  }
};
```

Механизмы повторного использования

- Наследование («прозрачный ящик»), отношение «is»
  - Наследование класса определяется статически на этапе компиляции
  - Нарушение инкапсуляции родителя для потомков
  - Тесная связь родителя и потомков
  - Нельзя изменить унаследованную от родителя реализацию во время выполнения программы
- Композиция* («черный ящик»), отношение «has»
  - Композиция объектов – это альтернатива наследованию класса
  - Для композиции требуется, чтобы объединяемые объекты имели четко определенные интерфейсы
  - Композиция объектов определяется динамически во время выполнения, следовательно, реализация может меняться во время выполнения
- Обобщенное программирование

Способы взаимосвязи объектов

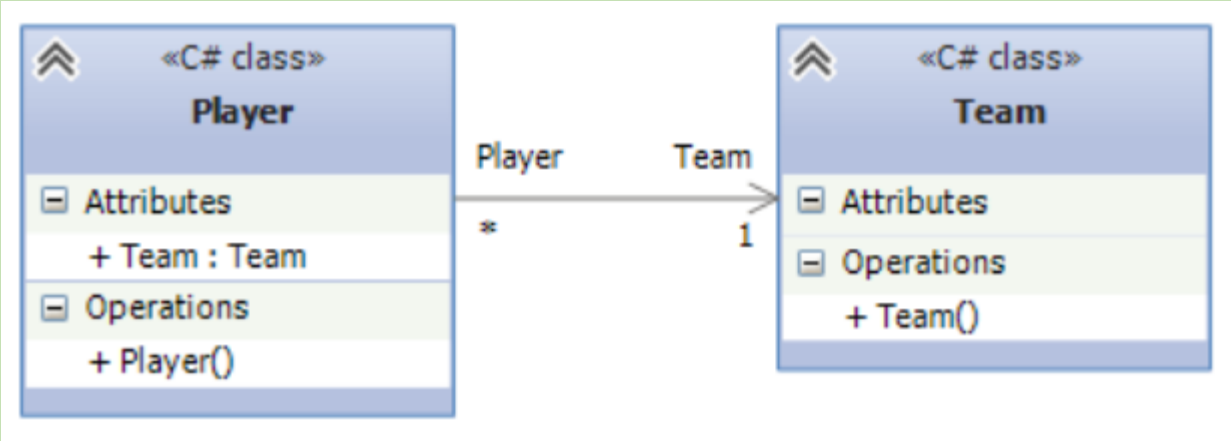
- Взаимосвязь объектов (ассоциация) может выражаться через *агрегирование* или *композицию*
- GoF оперировали терминами «осведомленность» и «агрегирование»:
- Агрегирование
  - Один объект владеет другим или несет за него ответственность
  - Агрегат и его составляющие имеют одинаковое время жизни
- Осведомленность
  - Одному объекту известно о другом объекте
  - Осведомленные объекты не несут никакой ответственности друг за друга
- Осведомленность более слабое отношение, чем агрегирование

Ассоциация

- Ассоциация - это отношение, при котором объекты одного типа каким-то образом связаны с объектами другого типа (содержат или используют).
- Моделирует отношение «имеет», «состоит из», «использует»:

  автомобиль имеет двигатель

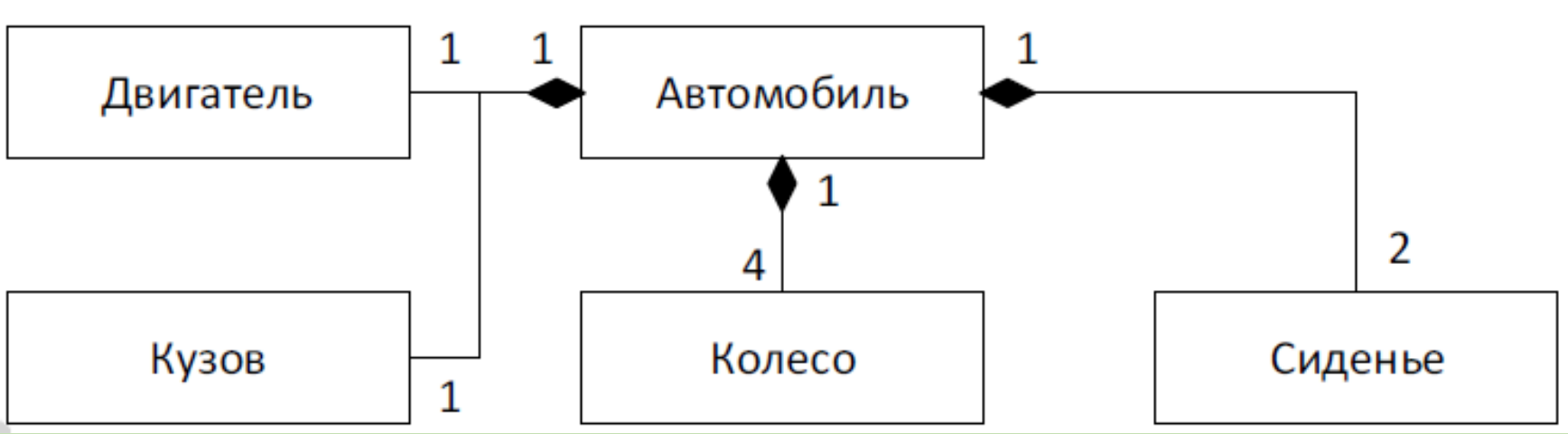
  футбольный клуб состоит из игроков



- Связь между сущностями может быть достаточно жесткой или слабой (объекты существуют сами по себе)
- Частные случаи ассоциации, которая реализует отношение “has ”: *композиция* и *агрегация*

Композиция

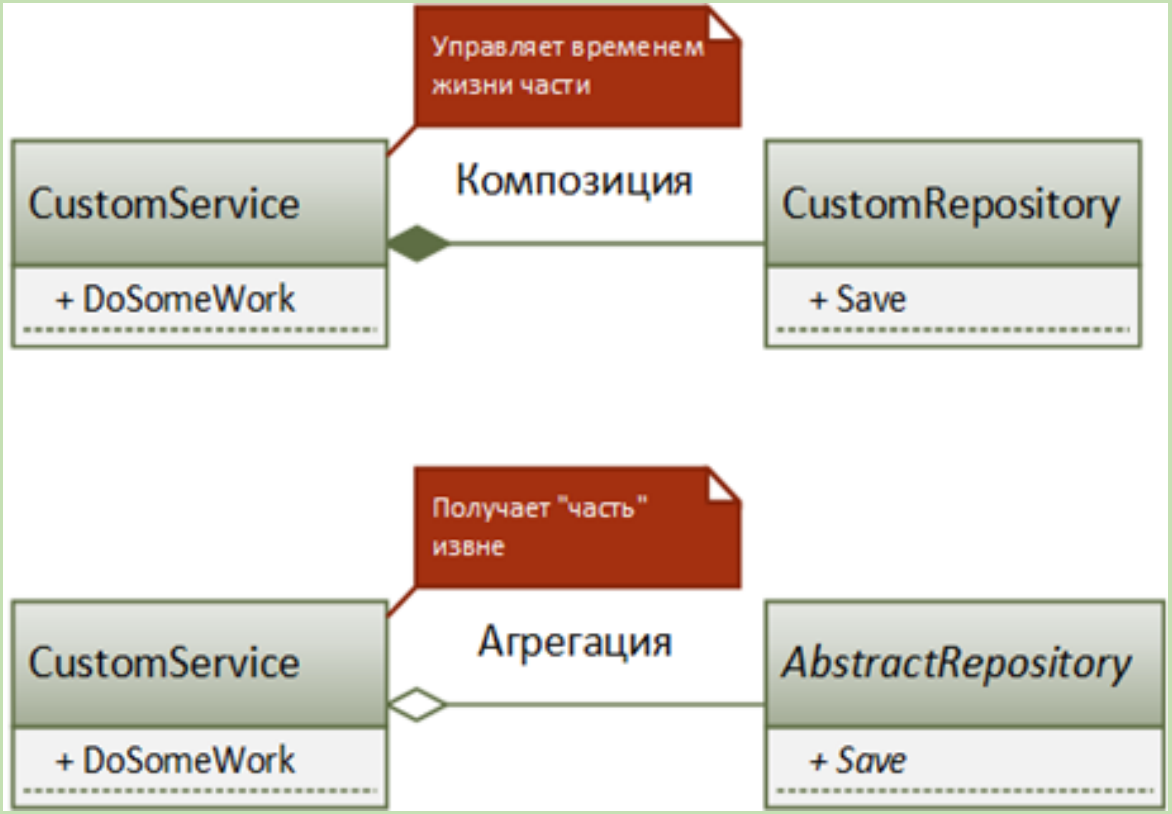
- Форма агрегации, при которой делается акцент на том, что часть не может существовать в отрыве от целого. Уничтожается целое – уничтожается и часть.



Агрегация

- Агрегация - это тип отношений, когда один объект является частью другого. Агрегация образует слабую связь между объектами. Все зависимые классы инициализируются вне основного объекта.

- В случае композиции целое явно контролирует время жизни своей составной части (часть не существует без целого), а в случае агрегации целое хоть и содержит свою составную часть, время их жизни не связано (например, составная часть передается через параметры конструктора).



Наследование vs. Агрегация vs. Композиция ..

- Агрегация объектов в большинстве случаев предпочтительнее наследования классов

- Предпочитайте агрегацию вместо композиции, поскольку первая стимулирует использование абстракций, а не конкретных классов.

Классификация GoF-паттернов

- **Порождающие паттерны**
  - Отвечают за создание объектов, позволяя системе оставаться независимой как от самого процесса порождения, так и от типов порождаемых объектов
- **Структурные паттерны**
  - Организуют структуру классов (на этапе разработки) или объектов (на этапе выполнения программы)
- **Паттерны поведения**
  - Характеризуют, как классы и объекты взаимодействуют между собой

Пространство паттернов GoF

Цель / Уровень	Порождающие паттерны	Структурные паттерны	Паттерны поведения
Класс	Фабричный метод	Адаптер (класса)	Интерпретатор
			Шаблонный метод
Объект	Абстрактная фабрика	Адаптер (объекта)	Итератор
	Одиночка	Декоратор	Команда
	Прототип	Заместитель	Наблюдатель
	Строитель	Компоновщик	Посетитель
		Мост	Посредник
		Приспособленец	Состояние
		Фасад	Стратегия
			Хранитель
			Цепочка обязанностей

Порождающие паттерны

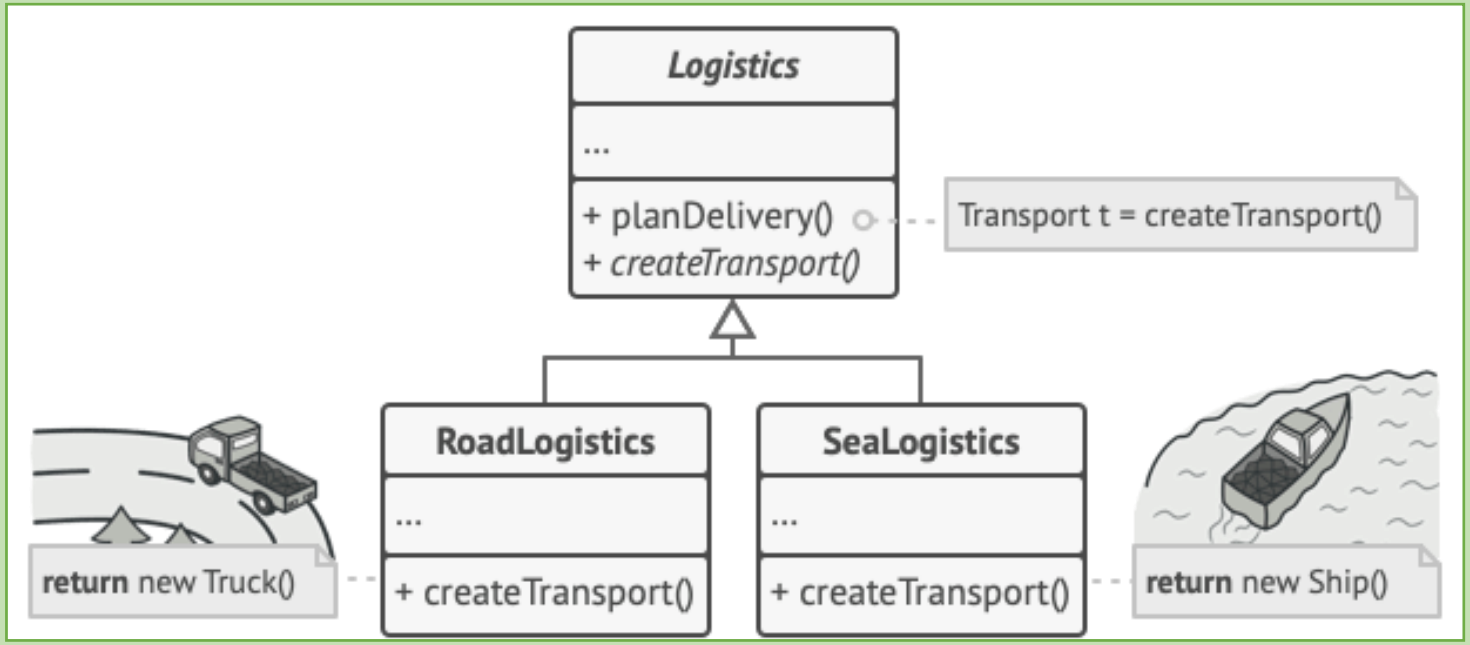
- Порождающие паттерны делегируют операцию создания объектов методу или группе методов
- Класс, отвечающий за создание объектов, называют **фабрикой**.

- Задачи, которые решают паттерны:

  - Взаимодействие объектов должно проходить строго через интерфейс, т.к. клиенту не нужно знать информацию о конкретных типах объектов
  - Такой подход может кардинально уменьшить число зависимостей между подсистемами
  - Но, т.к. клиенты не знают, объекты какого конкретного класса будут созданы, следовательно самостоятельно создать эти объекты не могут; здесь приходят на помощь порождающие паттерны

Фабричный метод

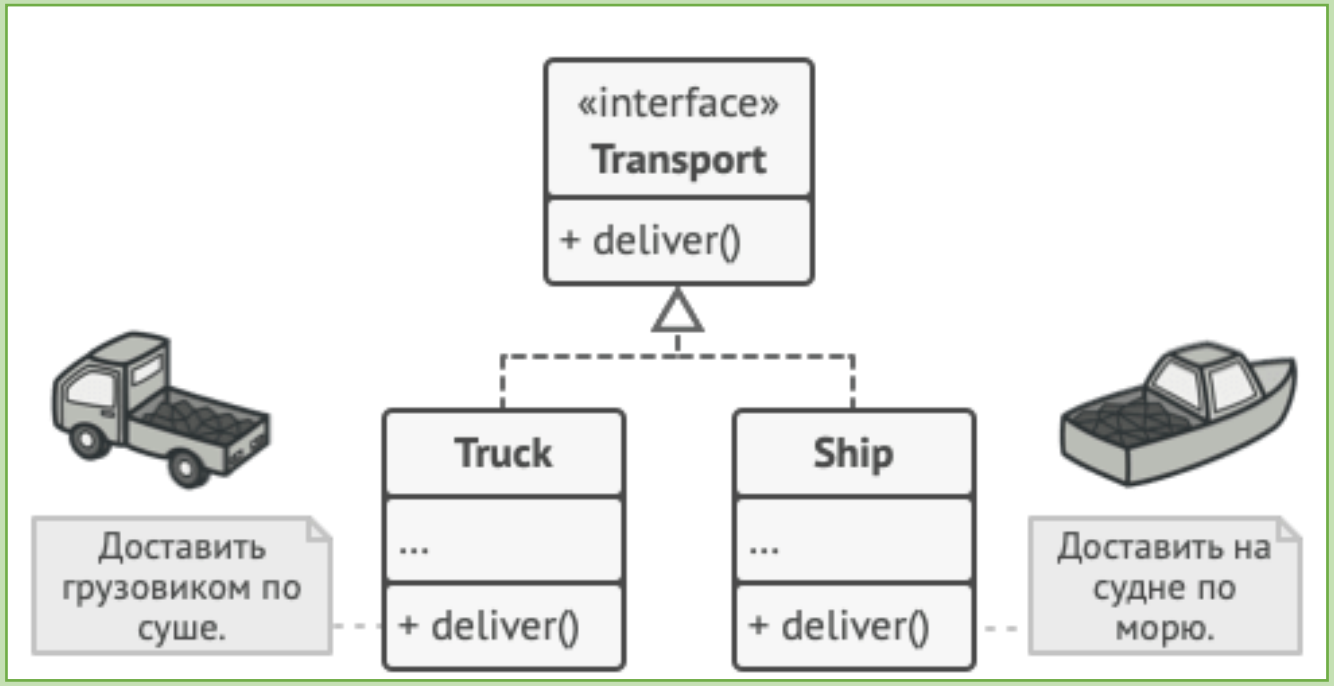
- Паттерн Фабричный метод позволяет создавать объекты не напрямую, а через вызов специального *фабричного метода.*
- Паттерн позволяет в процессе развития программной системы изменять и добавлять **способы** создания объектов и **типы** создаваемых объектов



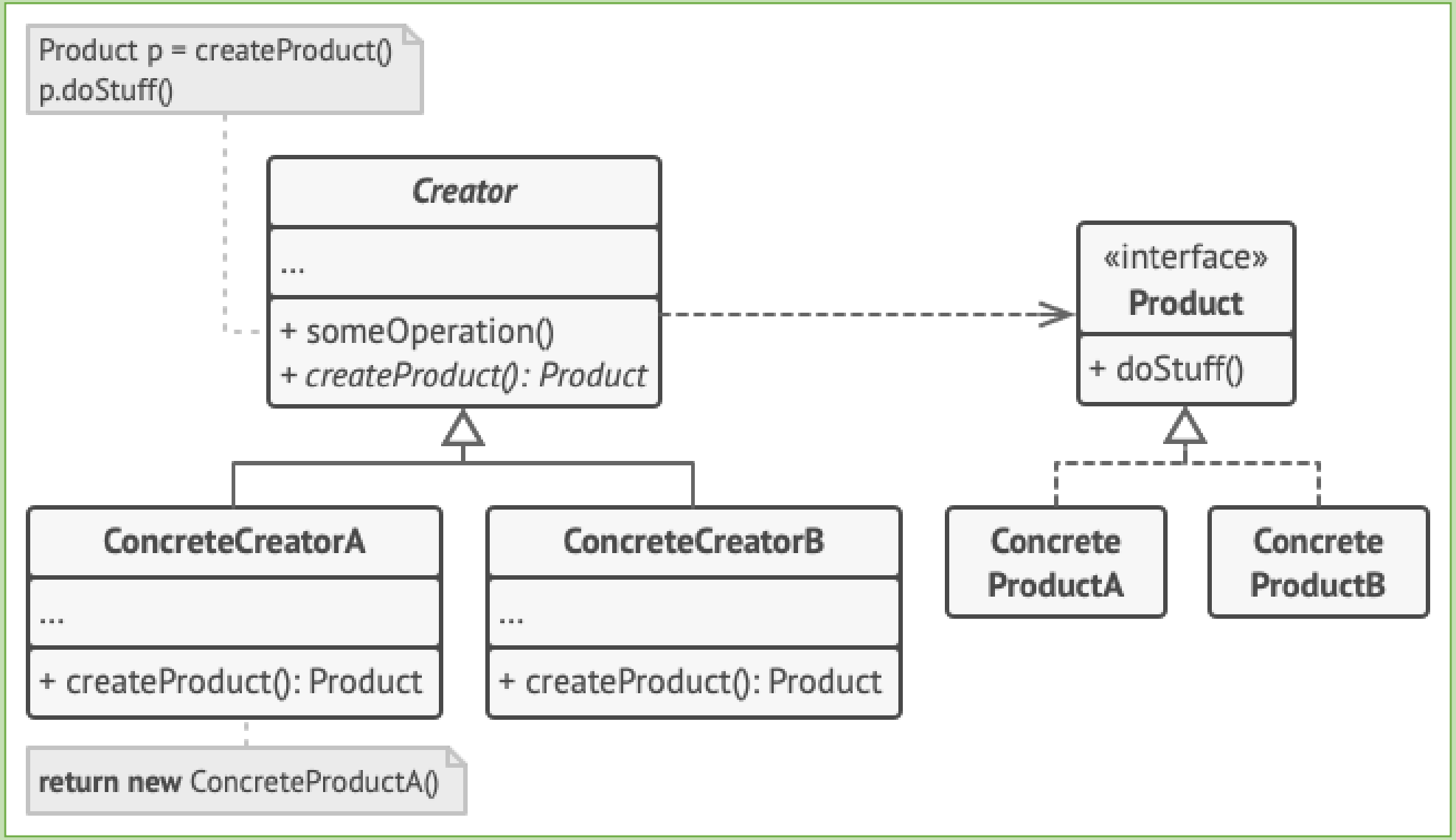
https://**refactoring.guru**/ru/design-patterns/factory-method

Фабричный метод: порождаемые объекты

- все возвращаемые объекты должны иметь общий интерфейс. Подклассы смогут производить объекты различных классов, следующих одному и тому же интерфейсу.



Фабричный метод: общая схема



Статический фабричный метод

- Упрощение схемы фабричного метода, когда создание объектов разных типов реализуется в одном статическом методе в зависимости от переданных аргументов

```
Figure figure = Geometry::GetFigure(points, size);
```

- Основное отличие статического метода от классического фабричного метода – тип создаваемого объекта определяется не типом наследника, а аргументами, переданными методу

Фабричный метод vs конструктор

- «Именованные конструкторы»: фабричные методы позволяют обойти ограничение конструкторов и использовать разные способы (методы) создания при одинаковых аргументах

```
class TimeSpan {
    public:
        TimeSpan(double ticks);
        static TimeSpan FromMilliseconds(double value);
        static TimeSpan FromSeconds(double value);
        static TimeSpan FromMinutes(double value);
};
```

```
Color c1 = Color::FromName("red");
Color c2 = Color::GetGreen();
Color c3 = Color::FromARGB(0, 0, 255, 255);
```

Фабричный метод vs конструктор

- **Тяжеловесный процесс создания**. Конструктор отвечает за корректную инициализацию объекта, который должен быть готов для использования клиентами. Обычно логика инициализации относительно простая и выполняется только конструктором. Тяжеловесную логику лучше вынести в статический фабричный метод, чем программировать **Bloated constructor**

- **Кэширование объектов.** Вместо создания новых объектов фабричный метод может возвращать уже созданные объекты. Используется при работе с тяжелыми ресурсоемкими объектами (подключение к базе данных, файлы).

- **Асинхронный конструктор.** При длительной процедуре создания объектов статичный шаблонный метод можно сделать асинхронным.

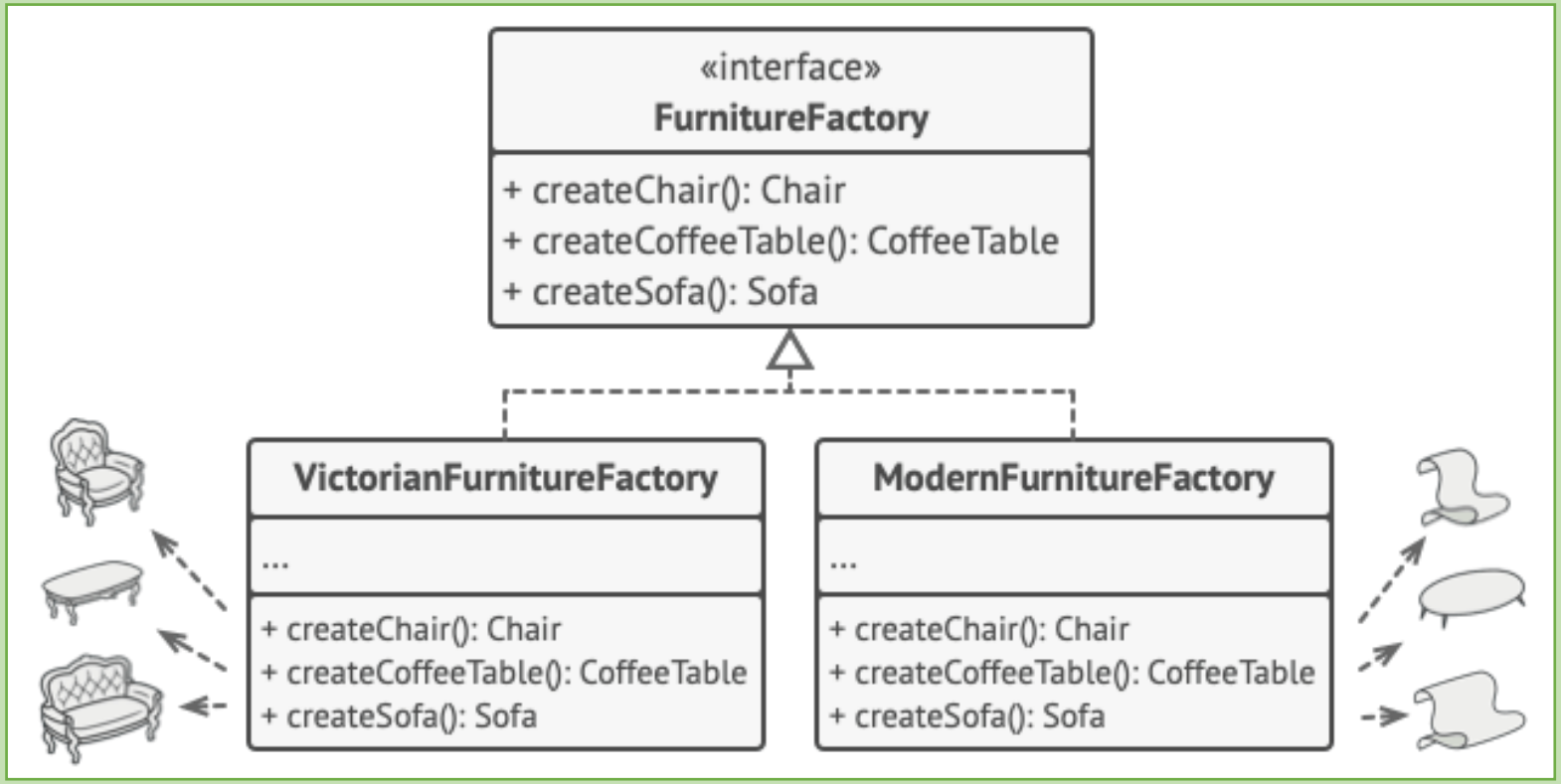
Фабричный метод vs конструктор

- **Когда заранее неизвестны типы и зависимости объектов, с которыми должен работать ваш код.** Код производства объектов отделен от кода использования объектов.

- **Когда вы хотите дать возможность пользователям расширять части вашего фреймворка или библиотеки.** Пользователи могут расширять классы вашего фреймворка через наследование.

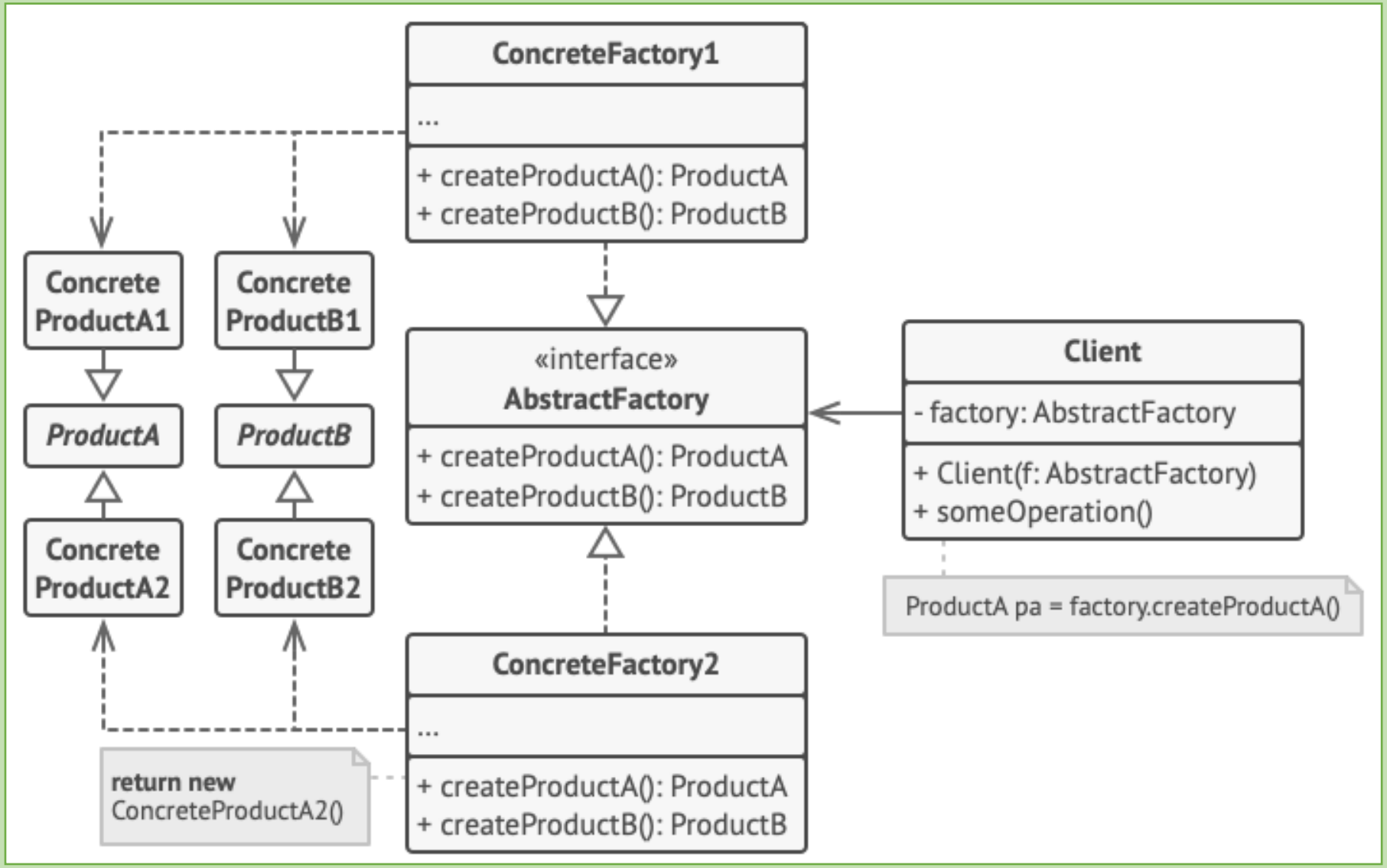
Абстрактная фабрика

- **Абстрактная фабрика** — это порождающий паттерн проектирования, который позволяет создавать **семейства связанных объектов**, не привязываясь к конкретным классам создаваемых объектов.



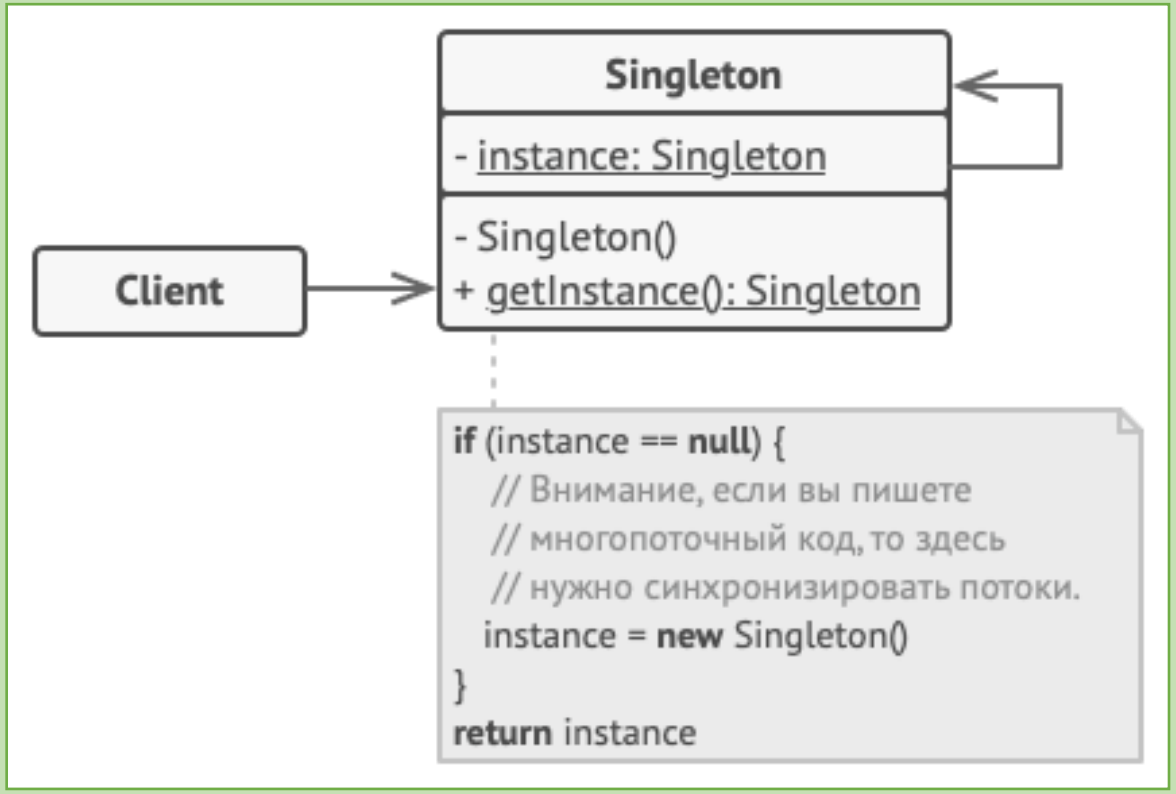
https://refactoring.guru/ru/design-patterns/abstract-factory

Абстрактная фабрика



Паттерн «Синглтон»

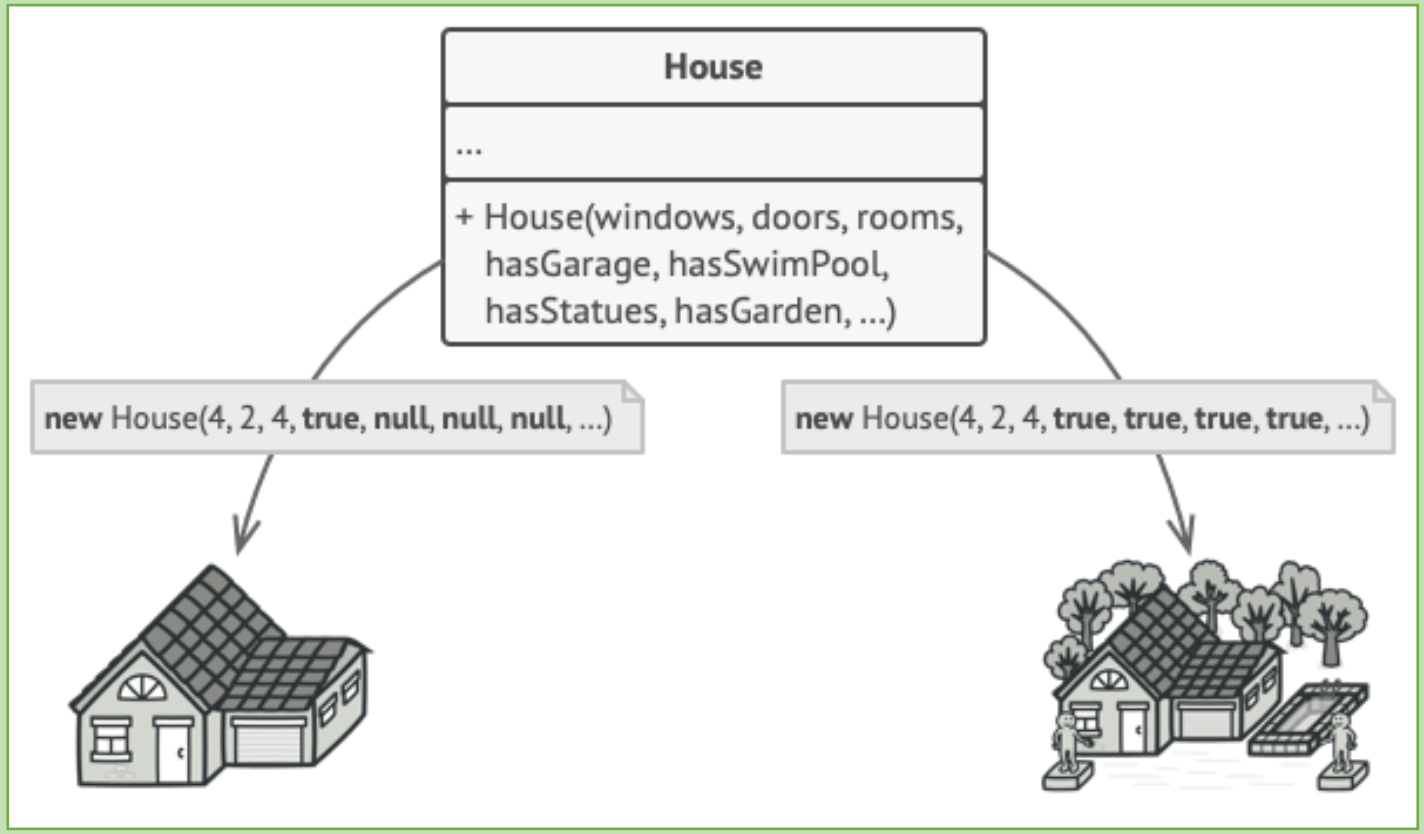
- **Одиночка** — это порождающий паттерн проектирования, который гарантирует, что у класса есть только один экземпляр, и предоставляет к нему глобальную точку доступа.
- Чаще всего это полезно для доступа к какому-то общему ресурсу, например, базе данных.
- «Объектно-ориентированная глобальная переменная»
- Синглтон гарантирует, что никакой другой код не заменит созданный экземпляр класса, поэтому вы всегда уверены в наличии лишь одного объекта-одиночки.



Паттерн «Строитель»

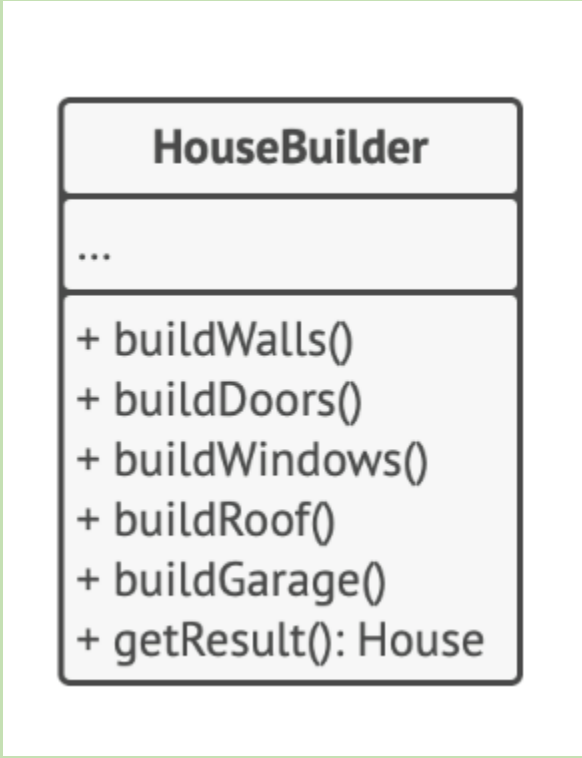
- Паттерн отделяет конструирование сложного объекта из нескольких этапов от его представления. Последовательность шагов с разными параметрами может приводить к разным объектам.

- Паттерн «Строитель» позволяет собирать объекты пошагово, вызывая только те шаги, которые вам нужны.
- Позволяет избежать «нагруженных» конструкторов с десятками опций



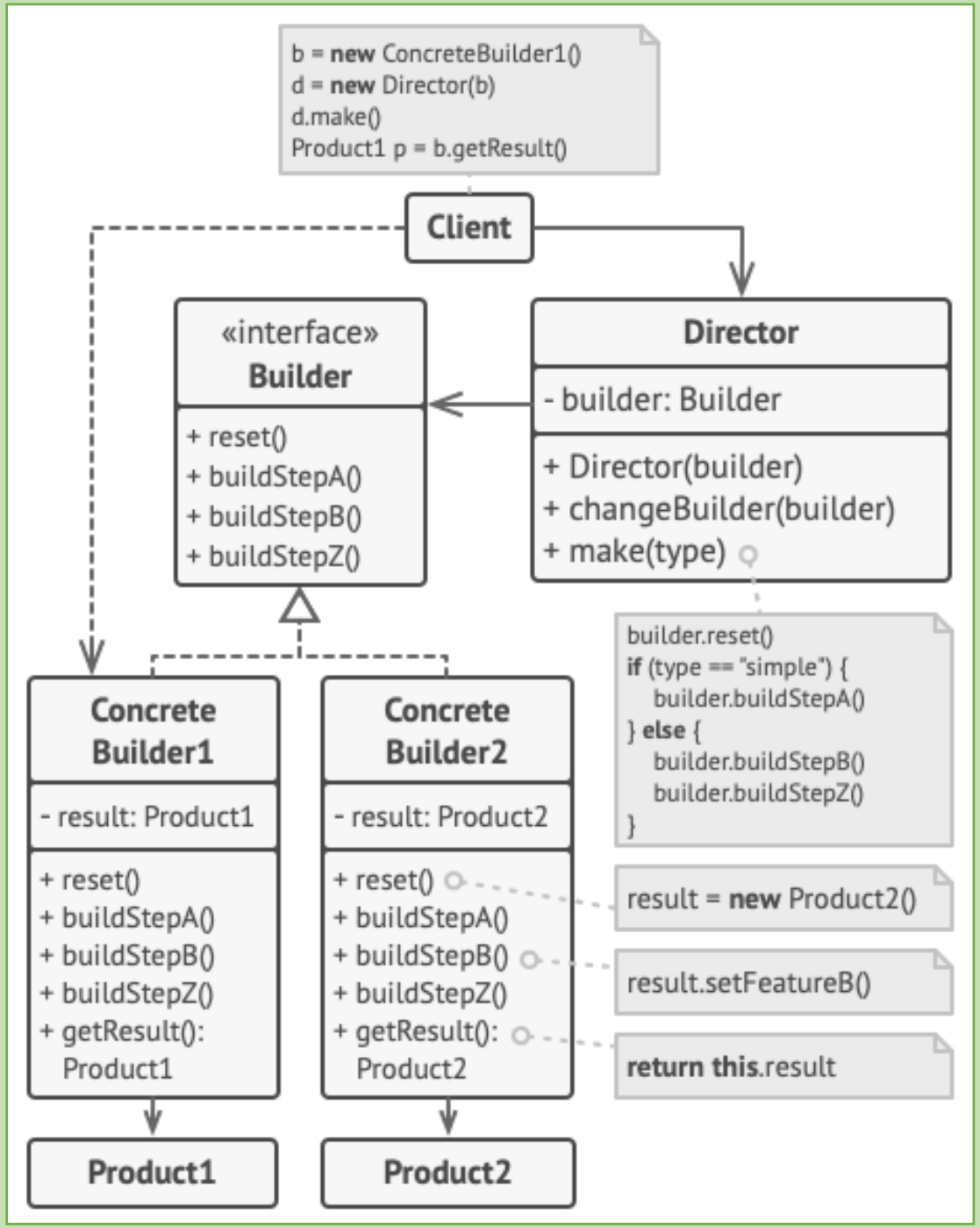
Паттерн «Строитель»

- **Паттерн Строитель** предлагает вынести конструирование объекта за пределы его собственного класса, поручив это дело отдельным объектам, называемым **строителями**.
- Процесс конструирования разбивается на отдельные шаги
- Чтобы создать объект, вам нужно поочерёдно вызывать методы строителя. Причём не нужно запускать все шаги, а только те, что нужны для производства объекта определённой конфигурации.



«Строитель» общая схема

- Можно выделить вызовы методов строителя в отдельный класс, называемый *директором*.
- Директор задает порядок шагов строительства, а строитель — их выполняет.
- Для возможности конструирования разных объектов с разной логикой этапов необходимо ввести *интерфейс строителя* и конкретных строителей.



Реализация «Строителя» с fluent api

- *Текучий интерфейс* (fluent api) – специальная форма реализации строителя
- Каждый метод (этап конструирования) возвращает объект, над которым выполняется текущая операция.
- Это позволяет связывать методы в цепочку и повышает «читабельность» кода.
- В C++ возможна специфическая реализация fluent api через перегрузку операторов

```
auto mail = new MailMessageBuilder()
    .From("nobody@nowhere.com")
    .To("somebody@somewhere.com")
    .Cc("boss@somewhere.com")
    .Subject("bad news")
    .Body("You're fired!")
    .Build();

SmtpClient::Send(mail);
```

```
cout << "one" << 2 << x << endl;
```

Термин введен Эриком Эвансом и Мартином Фаулером

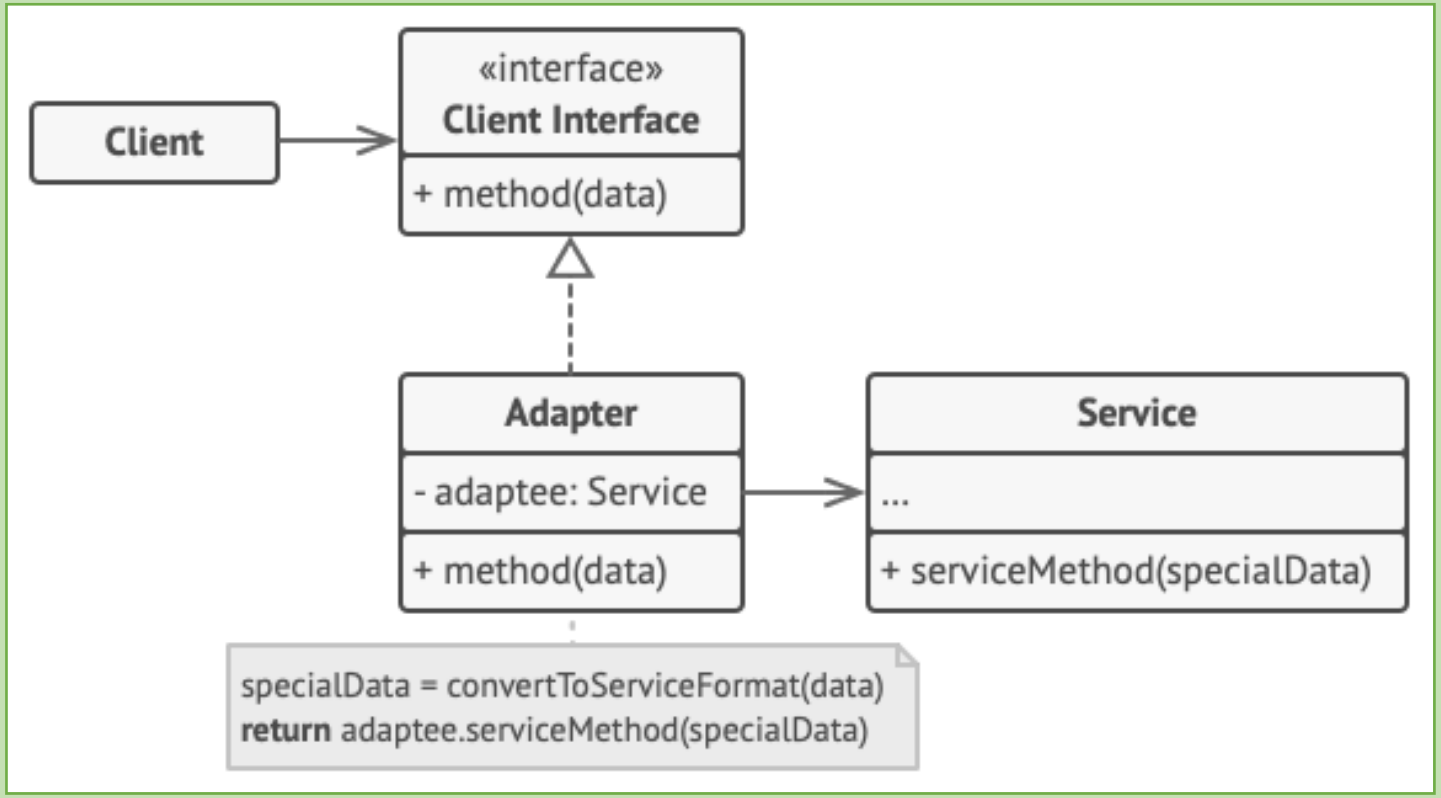
Структурные паттерны

- Эти паттерны отвечают за построение удобных в поддержке иерархий классов.
- Используется, чтобы адаптировать интерфейс некоторого объекта к необходимому (ожидаемому) интерфейсу (*адаптер*)
- Реализует возможность динамического добавления функционала к объекту (*декоратор*).
- Предоставляет унифицированный интерфейс вместо набора интерфейсов некоторой подсистемы. Определяет интерфейс более высокого уровня, который упрощает использование подсистемы (*фасад*).
- Используется для создания иерархических структур данных, отражающих отношение «часть-целое» (*компоновщик*).
- Отделить абстракцию от ее реализации так, чтобы то и другое можно было изменять независимо (*мост*)

Паттерн «Адаптер»

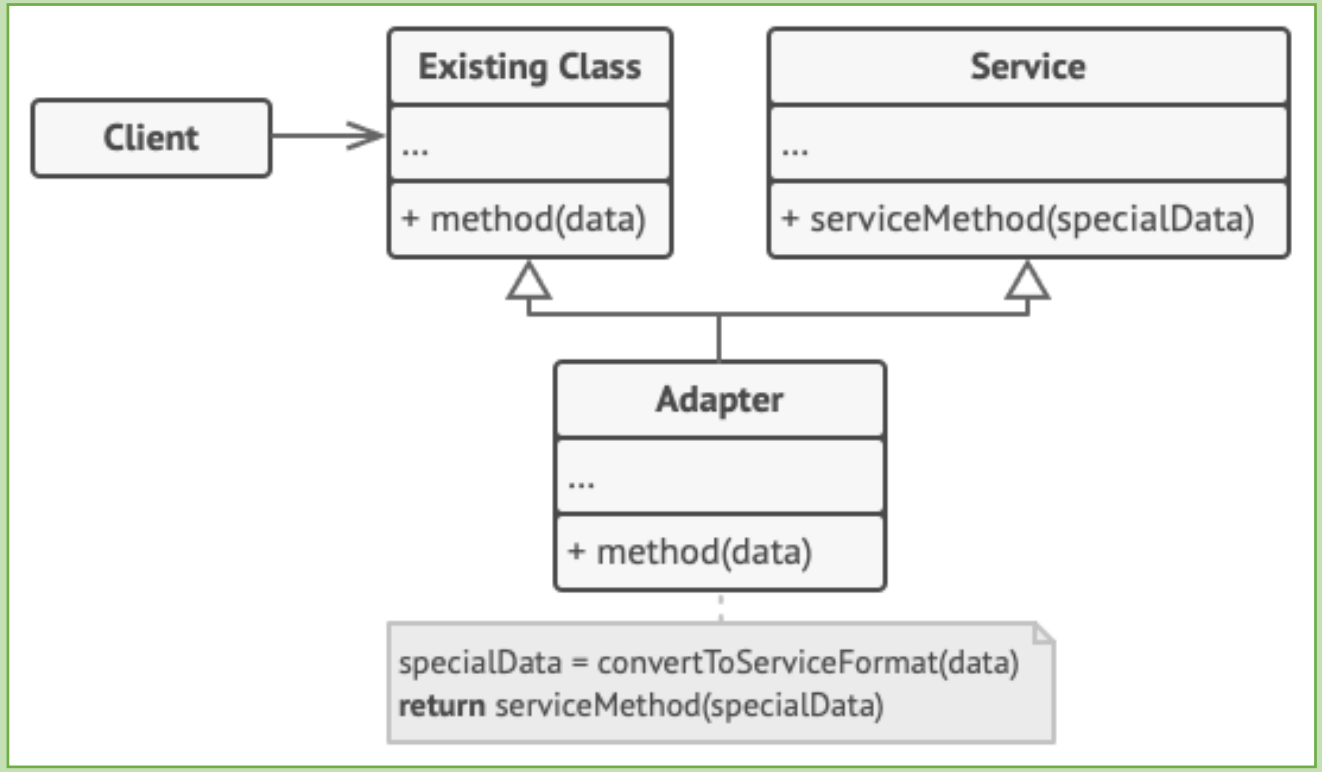
- Адаптер - это структурный паттерн проектирования, который позволяет объектам с несовместимыми интерфейсами работать вместе (*«переводчик»*).

- Например, клиент работает с данными в csv-формате, а сторонняя библиотека с json-форматом. Для взаимодействия можно ввести адаптер, который скрывает механизм работы с библиотекой, обеспечивая опосредованную работу клиента и библиотеки



Паттерн «Адаптер»

- В C++ реализация паттерна возможна и за счёт множественного наследования: адаптер наследует оба интерфейса одновременно.



Паттерн Proxy

- Заместитель — это структурный паттерн проектирования, который позволяет подставлять вместо реальных объектов специальные объекты-заменители. Эти объекты перехватывают вызовы к оригинальному объекту, позволяя сделать что-то до или после передачи вызова оригиналу.