Паттерны проектирования

План

1. Что такое паттерны проектирования.
2. Причины возникновения паттернов проектирования.
3. Понятие паттерна проектирования.
4. Принципы применения паттернов проектирования.
5. Принципы выбора паттернов проектирования.
6. Принципы разделения паттернов на категории.
7. Введение в UML.
8. Использование UML при анализе паттернов проектирования.
9. Порождающие паттерны.
10. Структурные паттерны.
11. Паттерны поведения.

Что такое паттерны проектирования. Понятие паттерна проектирования

Паттерн проектирования — это часто встречающееся решение определенной проблемы при проектировании архитектуры программ.

В отличие от готовых функций или библиотек, паттерн нельзя просто взять и скопировать в программу. Паттерн представляет собой не какой-то конкретный код, а общую концепцию решения той или иной проблемы, которую нужно будет ещё подстроить под нужды вашей программы.

Паттерны часто путают с алгоритмами, ведь оба понятия описывают типовые решения каких-то известных проблем. Но если алгоритм — это четкий набор действий, то паттерн — это высокоуровневое описание решения, реализация которого может отличаться в двух разных программах.

Если привести аналогии, то алгоритм — это кулинарный рецепт с четкими шагами, а паттерн — инженерный чертеж, на котором нарисовано решение, но не конкретные шаги его реализации.

Причины возникновения паттернов проектирования.

Паттерны это не какие-то супер-оригинальные решения, а наоборот, часто встречающиеся, типовые решения одной и той же проблемы.

Концепцию паттернов впервые описал Кристофер Александер в книге «**Язык шаблонов. Города. Здания. Строительство**». В книге описан «язык» для проектирования окружающей среды, единицы которого — шаблоны/паттерны (у *patterns*) — отвечают на архитектурные вопросы например, какой высоты сделать окна, сколько этажей должно быть в здании, какую площадь в микрорайоне отвести под деревья и газоны.

Идея показалась заманчивой четвёрке авторов: Эриху Гамме, Ричарду Хелму, Ральфу Джонсону, Джону Влиссидесу. В 1994 году они написали книгу «**Приемы объектно-ориентированного проектирования. Паттерны проектирования**», в которую вошли 23 паттерна и с тех пор были найдены десятки других объектных паттернов. «Паттерновый» подход стал популярен и в других областях программирования, поэтому сейчас можно встретить всевозможные паттерны и за пределами объектного проектирования.

Принципы применения и выбора паттернов проектирования

Если в распоряжение проектировщика предоставлен каталог из более чем 20 паттернов, трудно решать, какой паттерн лучше всего подходит для решения конкретной задачи проектирования. Ниже представлены разные подходы к выбору подходящего паттерна:

1. Подумайте, как паттерны решают проблемы проектирования. В разделе 1.6 обсуждается то, как с помощью паттернов можно найти подходящие объекты, определить нужную степень их детализации, специфицировать их интерфейсы. Здесь же говорится и о некоторых иных подходах к решению задач с помощью паттернов;

2. Пролистайте разделы каталога, описывающие назначение паттернов. В разделе 1.4,перечислены назначения всех представленных паттернов. Ознакомьтесь с целью каждого паттерна, когда будете искать тот, что в наибольшей степени относится к вашей проблеме. Чтобы сузить поиск, воспользуйтесь схемой в таблице 1.1;

3. Изучите взаимосвязи паттернов. На рис. 1.1 графически изображены соотношения между различными паттернами проектирования. Данная информация поможет вам найти нужный паттерн или группы паттернов;

4. Проанализируйте паттерны со сходными целями. Каталог состоит из трех частей: порождающие паттерны, структурные паттерны и паттерны поведения. Каждая часть начинается со вступительных замечаний о паттернах соответствующего вида и заканчивается разделом, где они сравниваются друг с другом;

5. Разберитесь в причинах, вызывающих перепроектирование. Взгляните на перечень причин, приведенный выше. Быть может, в нем упомянута ваша проблема? Затем обратитесь к изучению паттернов, помогающих устранить эту причину;

Зачем знать паттерны?

Вы можете вполне успешно работать, не зная ни одного паттерна. Более того, вы могли уже не раз реализовать какой-то из паттернов, даже не подозревая об этом.

- Проверенные решения. Вы тратите меньше времени, используя готовые решения, вместо повторного изобретения велосипеда. До некоторых решений вы смогли бы додуматься и сами, но многие могут быть для вас открытием.
- Стандартизация кода. Вы делаете меньше просчетов при проектировании, используя типовые унифицированные решения, так как все скрытые проблемы в них уже давно найдены.
- Общий программистский словарь. Вы произносите название паттерна, вместо того, чтобы час объяснять другим программистам, какой крутой дизайн вы придумали и какие классы для этого нужны.

Принципы разделения паттернов на категории

Паттерны отличаются по уровню сложности, детализации и охвата проектируемой системы. Проводя аналогию со строительством, вы можете повысить безопасность перекрёстка, поставив светофор, а можете заменить перекрёсток целой автомобильной развязкой с подземными переходами.

Самые низкоуровневые и простые паттерны — *идиомы*. Они не универсальны, поскольку применимы только в рамках одного языка программирования.

Самые универсальные — *архитектурные паттерны*, которые можно реализовать практически на любом языке. Они нужны для проектирования всей программы, а не отдельных ее элементов.

Кроме того, паттерны отличаются и предназначением. Рассмотрим три основные группы элементов:

- Порождающие паттерны беспокоятся о гибком создании объектов без внесения в программу лишних зависимостей.
- Структурные паттерны показывают различные способы построения связей между объектами.
- Поведенческие паттерны заботятся об эффективной коммуникации между объектами.

Каждую группу мы рассмотрим немного позже сначала нужно понять как мы будем их описывать

Введение в UML. Использование UML при анализе паттернов проектирования.

Unified Modeling Language (UML) — унифицированный язык моделирования. Расшифруем: *modeling* подразумевает создание модели, описывающей объект. *Unified* (универсальный, единый) — подходит для широкого класса проектируемых программных систем, различных областей приложений, типов организаций, уровней компетентности, размеров проектов. UML описывает объект в едином заданном синтаксисе, поэтому где бы вы не нарисовали диаграмму, ее правила будут понятны для всех, кто знаком с этим графическим языком

Для чего используется UML?

Одна из задач UML — служить средством коммуникации внутри команды и при общении с заказчиком. Давайте рассмотрим возможные варианты использования диаграмм.

Для чего используется UML?

Одна из задач UML — служить средством коммуникации внутри команды и при общении с заказчиком. Давайте рассмотрим возможные варианты использования диаграмм.

- Проектирование. UML-диаграммы помогут при моделировании архитектуры больших проектов, в которой можно собрать как крупные, так и более мелкие детали и нарисовать каркас (схему) приложения. По нему впоследствии будет строиться код.
- Реверс-инжиниринг — создание UML-модели из существующего кода приложения, обратное построение. Может применяться, например, на проектах поддержки, где есть написанный код, но документация неполная или отсутствует.
- Из моделей можно извлекать текстовую информацию и генерировать относительно удобочитаемые тексты — документировать. Текст и графика будут дополнять друг друга.

Нотация UML для описания логики проекта

Как и любой другой язык, UML имеет собственные правила оформления моделей и синтаксис. С помощью графической нотации UML можно визуализировать систему, объединить все компоненты в единую структуру, уточнять и улучшать модель в процессе работы. На общем уровне графическая нотация UML содержит 4 основных типа элементов:

- фигуры;
- линии;
- значки;
- надписи.

UML-нотация является де-факто отраслевым стандартом в области разработки программного обеспечения, ИТ-инфраструктуры и бизнес-систем.

Виды UML-диаграмм

В языке UML есть 12 типов диаграмм:

- 4 типа диаграмм представляют **статическую структуру** приложения;
- 5 типов представляют **поведенческие аспекты** системы;
- 3 представляют **физические аспекты** функционирования системы (диаграммы реализации).

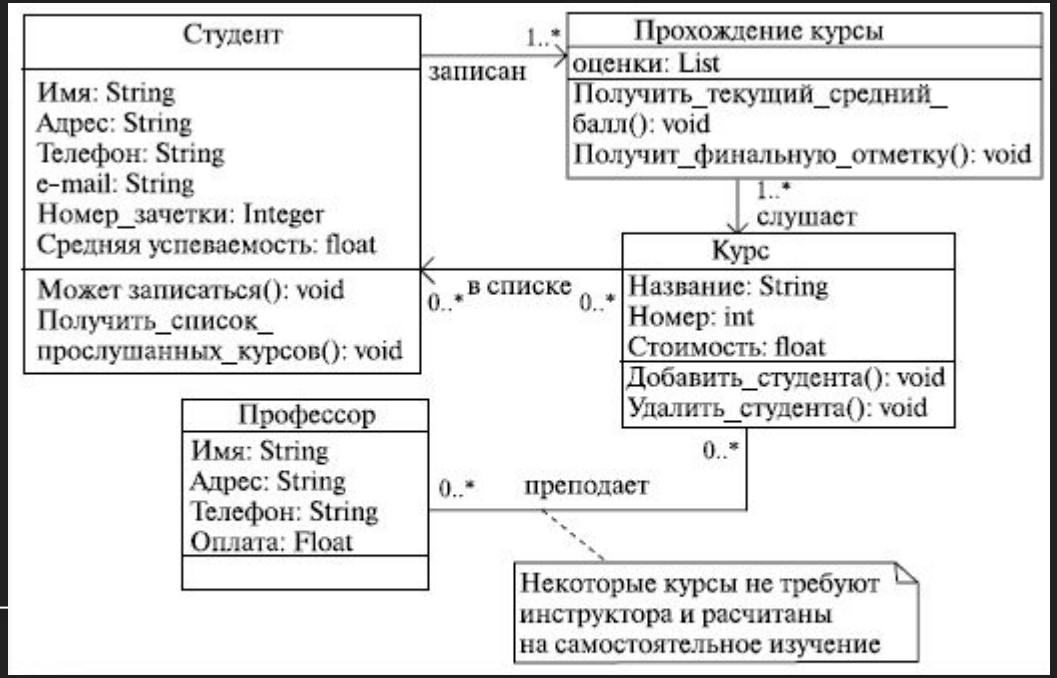
Некоторые из видов диаграмм специфичны для определенной системы и приложения. Самыми доступными из них являются:

- Диаграмма прецедентов (Use-case diagram);
- Диаграмма классов (Class diagram);
- Диаграмма активностей (Activity diagram);
- Диаграмма последовательности/взаимодействия (Sequence diagram);
- Диаграмма развёртывания (Deployment diagram);
- Диаграмма сотрудничества (Collaboration diagram);
- Диаграмма объектов (Object diagram);
- Диаграмма состояний (Statechart diagram).

Далее рассмотрим по нашей тематике это три вида диаграмм - классов, взаимодействия, объектов

Диаграмма классов — Class diagram

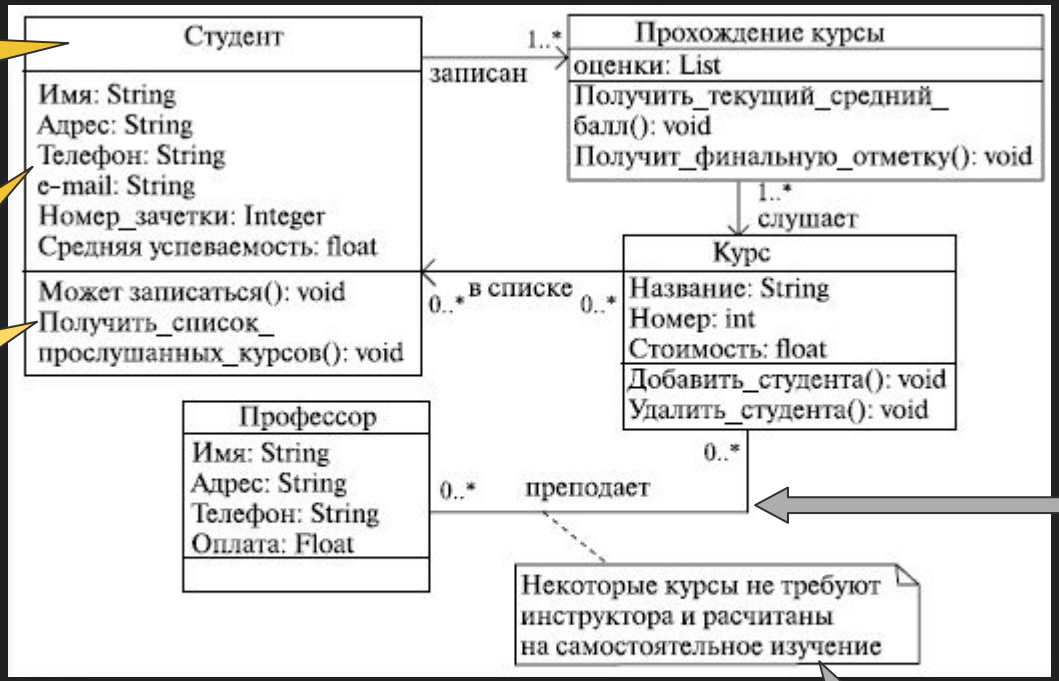
Класс (class) — категория вещей, которые имеют общие атрибуты и операции. Сама диаграмма классов являет собой набор статических, декларативных элементов модели. Она дает нам наиболее полное и развернутое представление о связях в программном коде, функциональности и информации об отдельных классах. Приложения генерируются зачастую именно с диаграммы классов.



Для класса "студент" есть таблица, содержащая атрибуты: имя, адрес, телефон, e-mail, номер зачетки, средняя успеваемость. И также показаны связи данной сущности с другими: прохождением курса, какой курс слушает, кто профессор. В этом примере также добавляются функции, которые могут быть применены к сущности "студент".

Диаграмма классов. Структура

Ассоциация – представляет собой отношения между экземплярами классов



Название класса

Поля класса

Методы класса

Комментарий

Каждый конец ассоциации обладает кратностью которая показывает, сколько объектов, расположенных с соответствующего конца ассоциации, может участвовать в данном отношении. В примере на рисунке каждый Курс имеет сколь угодно Профессоров и каждый Профессор имеет сколько угодно Курсов. В общем случае кратность может быть задана любым множеством. Ассоциации может быть присвоено имя. В качестве имени обычно выбирается глагол или глагольное словосочетание, сообщающие смысл и назначение связи. Также на концах ассоциации под кратностью может указываться имя роли, т.е. какую роль выполняют объекты, находящиеся с данного конца

Перейдем к примеру…

Диаграмма последовательности/взаимодействия (Sequence diagram)

В нотации UML взаимодействие элементов рассматривается в информационном аспекте их коммуникации. Другими словами, взаимодействующие объекты обмениваются между собой некоторой информацией.

1. Создать диаграмму последовательностей. Например, «Создание нового заказа».
2. На данной диаграмме должно быть отражено взаимодействие объектов классов. Например, «Менеджер», «Списковая форма товаров на складе», «Списковая форма заказов», «Форма редактирования заказов», «Заказ», «Позиция Заказов»

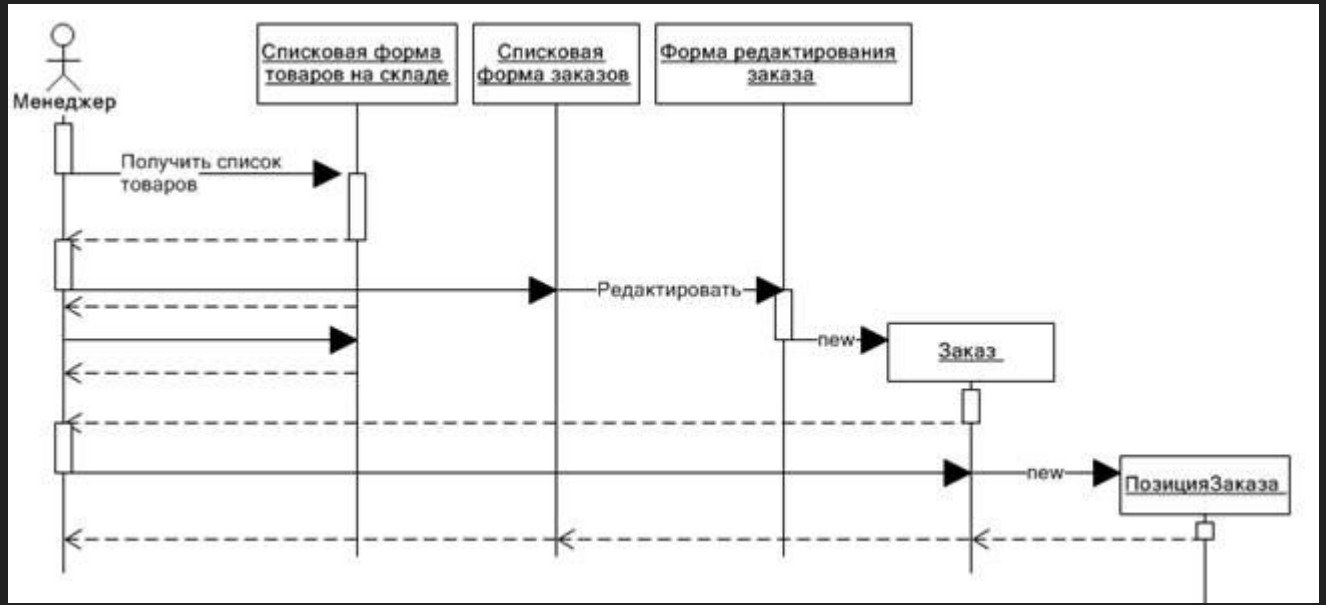
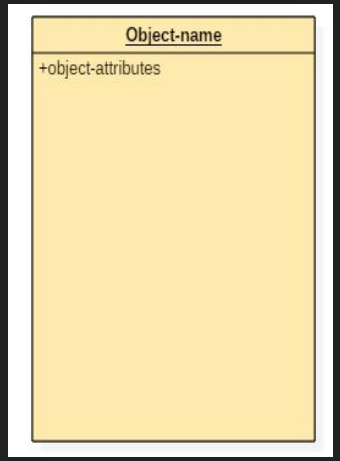
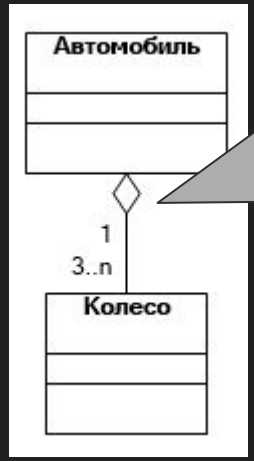


Диаграмма объектов (Object diagram)

Разница между диаграммой классов и объектов заключается в том, что диаграмма классов в основном представляет собой вид с высоты птичьего полета на систему, которая также называется абстрактной. Диаграмма объектов описывает экземпляр класса. Он визуализирует определенные функциональные возможности системы. дают нам примеры того, как структуры данных выглядят в конкретный момент времени. Если статичную структуру удобно представить в виде диаграммы классов, то диаграммы объектов, как контрольные примеры, позволяют судить, все ли ее компоненты на месте. Из диаграммы объектов также можно почерпнуть полезную информацию об элементах модели и их ссылках.



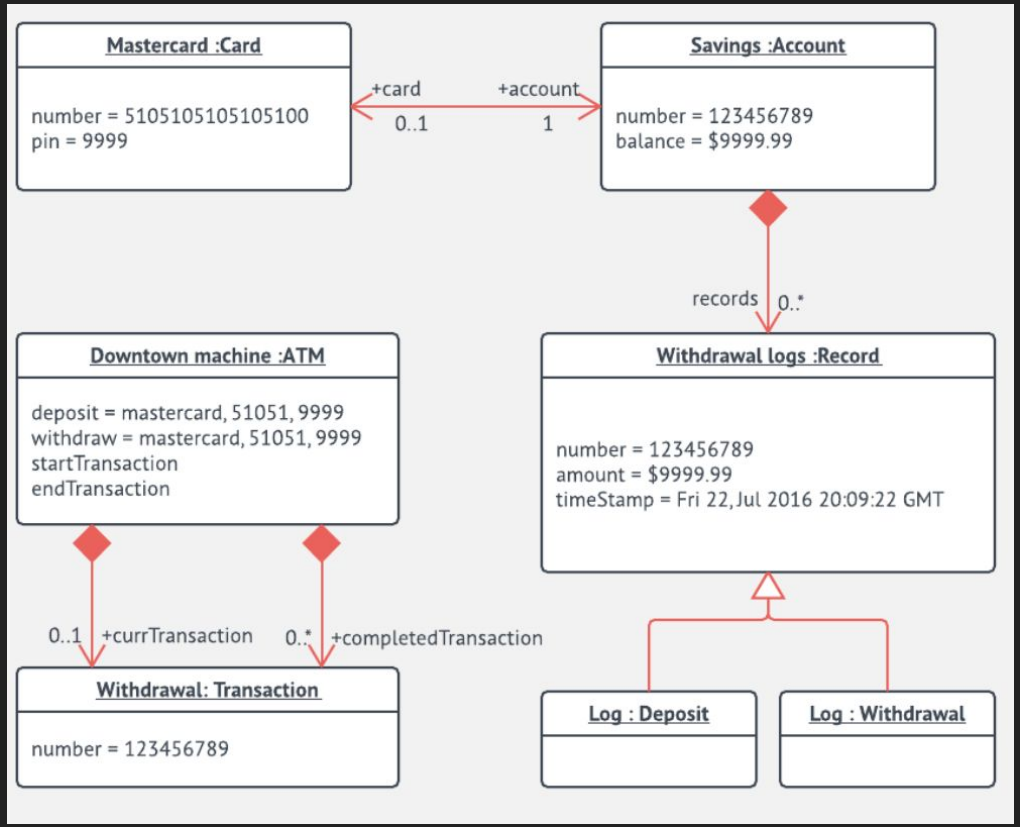


Агрегация (aggregation) – это ассоциация типа «целое-часть». Агрегация в UML представляется в виде прямой с ромбом на конце. Ромб на связи указывает, какой класс является агрегирующим (т.е. «состоящим из»); класс с противоположного конца — агрегированным (т.е. те самые «части»).

Композиция (composition) – это такая агрегация, где объекты-части не могут существовать сами по себе и уничтожаются при уничтожении объекта агрегирующего класса. Композиция изображается так же, как ассоциация, только ромбик закрашен.

Диаграмма объектов (Object diagram)

Важно понимать разницу между агрегацией и композицией: при агрегации объекты-части могут существовать сами по себе, а при композиции — нет. Пример агрегации: автомобиль—колесо, пример композиции: дом—комната.



Порождающие паттерны.

Отвечают за удобное и безопасное создание новых объектов или даже целых семейств объектов.

- Factory Method
- Abstract Factory
- Builder
- Prototype
- Singleton

Рассмотрим каждый из них …

Порождающие паттерны. Factory Method

Фабричный метод — это порождающий паттерн проектирования, творческий шаблон. Это один из наиболее часто используемых шаблонов в программировании.
Создание объектов — сложная задача. Обычно конструкторы принимают множество параметров при создании объекта. Порождающие шаблоны проектирования и, следовательно, фабричный шаблон облегчают работу.

Бизнес-объекты будут необходимы во многих местах проекта. Следовательно, фрагменты кода или конструкторы, создающие связанные объекты, будут вызываться во всей системе. По этой причине будет много повторов копипаста и Фабричный шаблон решает проблему, где создавать объекты. Он абстрагирует процесс создания объекта. Его подклассы берут на себя управление созданием объектов с разной специализацией, но одного происхождения. Процесс создания объекта отделен от использования созданных объектов с помощью шаблона Factory. Это метод, который ослабляет отношения между творением и много использует.

Ответственность за получение информации о состоянии в процессе создания объекта максимально возлагается на методы создателя. Вместо того, чтобы передавать параметры извне, по возможности создатель должен получить эту информацию сам. Это снижает сложность создания объектов.

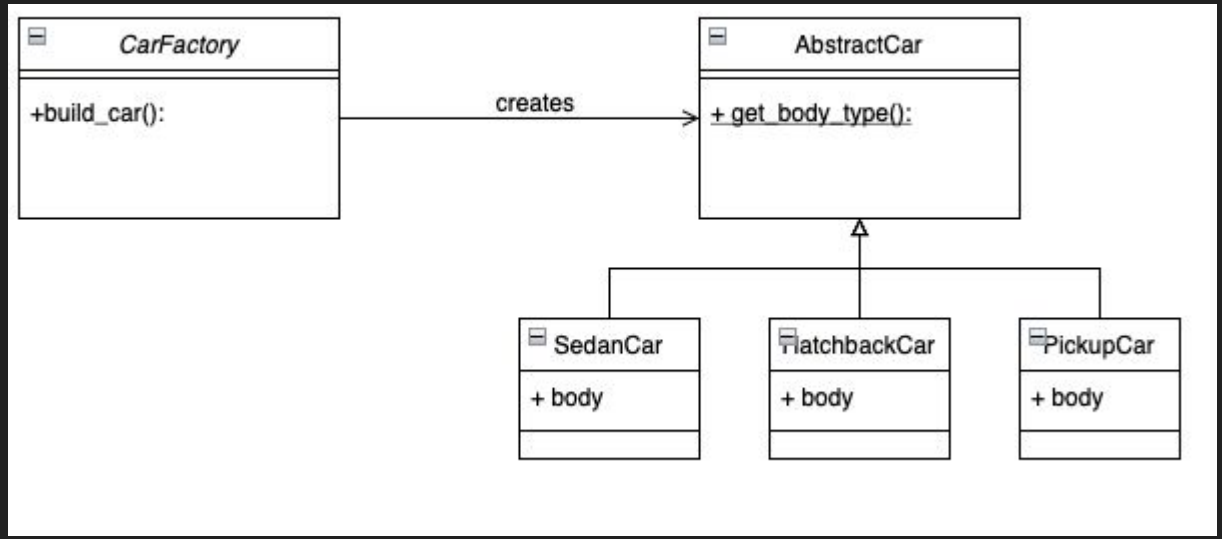
Порождающие паттерны. Factory Method

Пример по проблеме

Для ясности давайте рассмотрим в этом примере воображаемую бизнес-задачу. Компания-производитель автомобилей выпускает автомобили с 3 типами кузова: седан, хэтчбек и пикап. Заводская производственная линия должна быть готова производить эти различные типы автомобилей всякий раз, когда из отдела планирования производства отправляется новый производственный план.

Перейдем к решению…

Давайте воспользуемся фабричным шаблоном проектирования, чтобы решить эту проблему. Существуют различные типы дочерних классов автомобиля, которые реализуют этот абстрактный класс автомобиля. Автозавод создает соответствующие классы автомобилей в соответствии с поступающим спросом.



Порождающие паттерны. Abstract Factory

Как говорили , шаблон проектирования Factory работает на основе производных дочерних классов путем наследования родительских классов. Это **метод** , который генерирует эти производные классы. Он отвечает за создание только одного объекта.

С другой стороны, Abstract Factory Pattern — это **класс** . Он отвечает за создание семейства объектов. Это семейство представляет собой объекты, предназначенные для совместного использования и к которым оно относится. Таким образом, он предоставляет несколько объектов для создания это можно сделать, потому что он содержит фабрики семейства объектов. То есть шаблон проектирования «Абстрактная фабрика» содержит множество шаблонов проектирования «Фабрика».

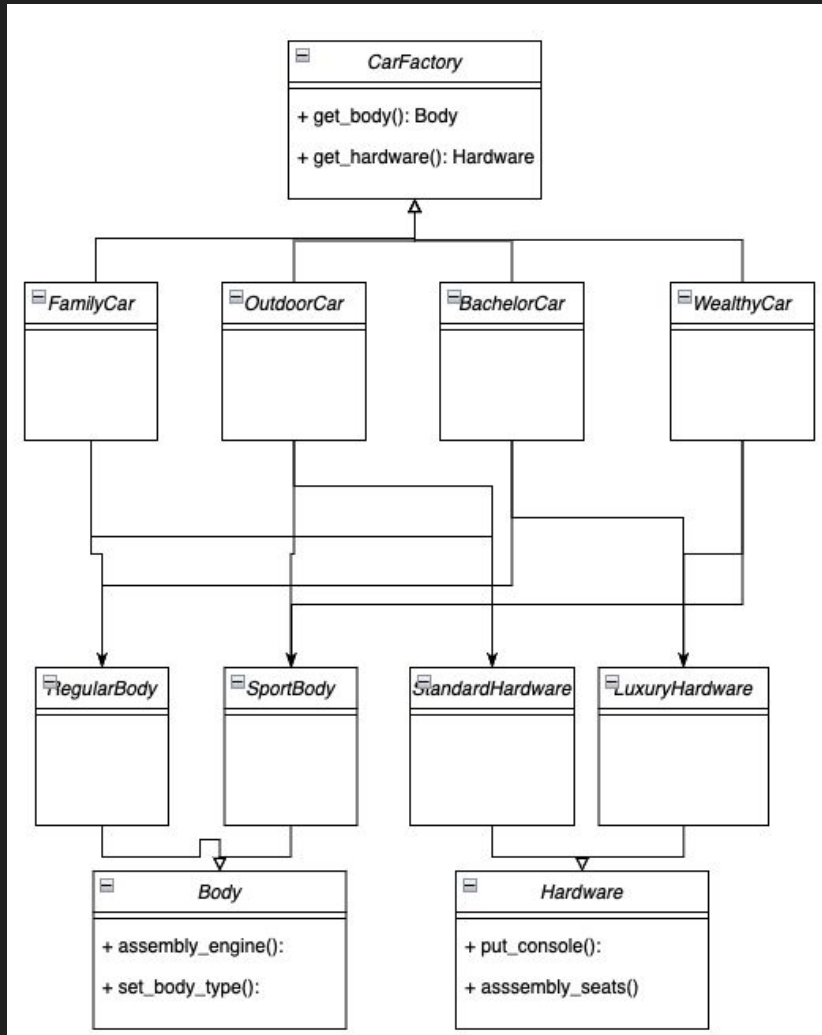
Эти два также можно рассматривать как один и тот же шаблон проектирования. Шаблон проектирования абстрактной фабрики можно рассматривать как более индивидуализированную версию другого

Порождающие паттерны. Abstract Factory

Проблема - Один и тот же производитель автомобилей создает разные модели автомобилей в соответствии с потребительскими сегментами. Маркетинговая группа разделила потенциальных клиентов на 4 группы: *семья, активный отдых, холостяк и богатые* .

С другой стороны, отдел разработки продуктов выбирает две основные части автомобиля при создании своих продуктов в ответ на эту сегментацию: *кузов* и *оборудование* .

Перейдем к решению



Порождающие паттерны. Builder

Шаблон построителя — это порождающий шаблон, цель которого состоит в том, чтобы отделить построение сложного объекта от его представления, чтобы вы могли использовать один и тот же процесс построения для создания различных представлений.
Шаблон Builder пытается решить,

Как можно упростить класс, включающий создание сложного объекта?

Как класс может создавать различные представления сложного объекта?

Шаблоны Builder и Factory очень похожи в том, что они оба создают экземпляры новых объектов во время выполнения. Разница заключается в том, что когда процесс создания объекта является более сложным, то вместо того, чтобы Factory возвращала новый экземпляр ObjectA, он вызывает метод конструктора директора строителей, ObjectA.construct()который проходит через более сложный процесс построения, включающий несколько шагов. Оба возвращают объект/продукт.

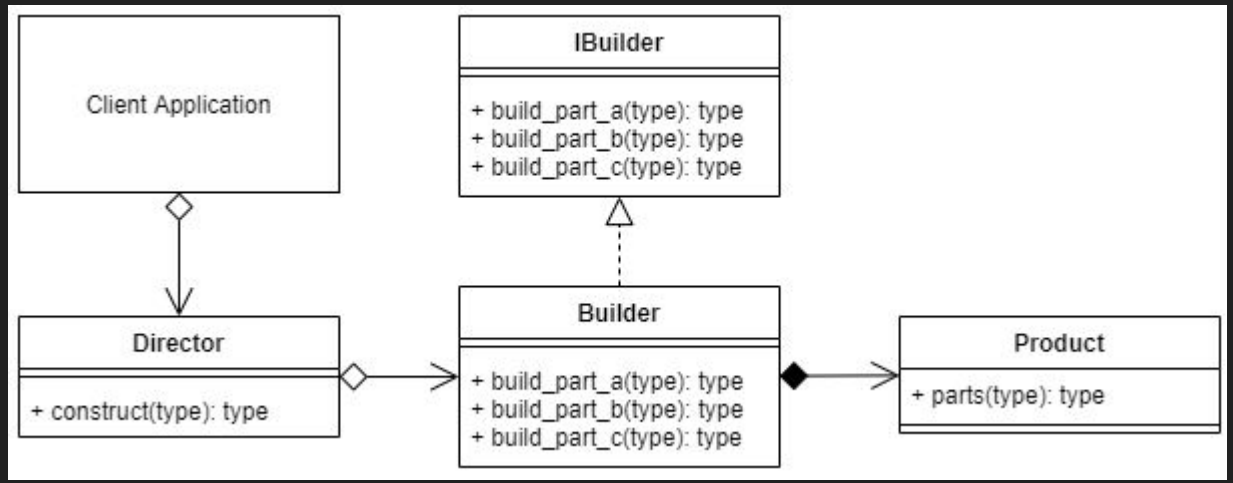
Порождающие паттерны. Builder

Product : создаваемый продукт.

Builder : строит конкретный продукт. Реализует **интерфейс строителя.**

Builder Interface: интерфейс, который должен реализовывать конструктор.

Director : имеет construct() метод, который при вызове создает индивидуальный продукт.



Порождающие паттерны. Prototype

Шаблон проектирования « **Прототип** » удобен, когда для создания новых объектов требуется больше ресурсов, чем вы хотите использовать или иметь в наличии. Вы можете сэкономить ресурсы, просто создав копию любого существующего объекта, который уже находится в памяти.

Например, файл, который вы загрузили с сервера, может быть большим, но поскольку он уже находится в памяти, вы можете просто клонировать его и работать с новой копией независимо от оригинала.В интерфейсе шаблонов прототипов вы создаете статический метод клонирования, который должен быть реализован всеми классами, использующими интерфейс. Как метод клонирования реализован в конкретном классе, зависит от вас. Вам нужно будет решить, требуется ли поверхностная или глубокая копия.

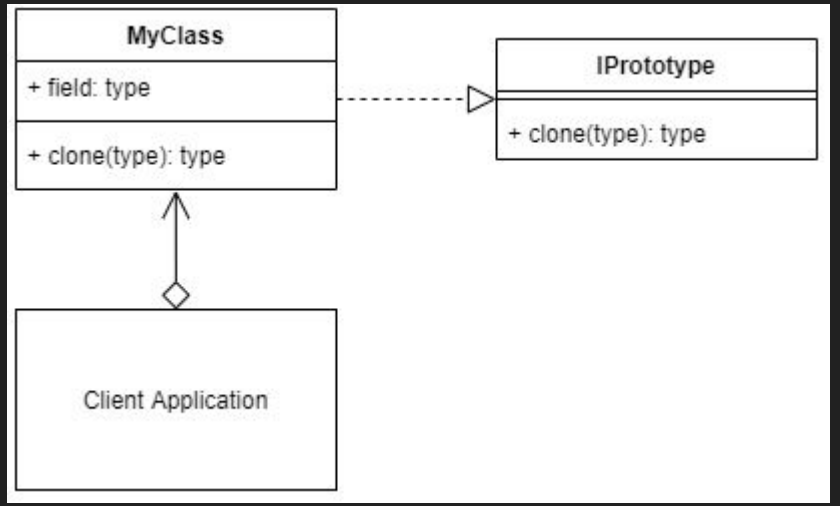
- Неглубокая копия копирует и создает новые ссылки на один уровень в глубину,
- Глубокая копия копирует и создает новые ссылки для всех уровней.

В Python у вас есть изменяемые объекты, такие как списки , словари , наборы и любые пользовательские объекты, которые вы могли создать. Неглубокая копия создаст новые копии объектов с новыми ссылками в памяти, но базовые данные, например, фактические элементы в списке, будут указывать на ту же ячейку памяти, что и исходный копируемый список/объект. Теперь у вас будет два списка, но элементы в списках будут указывать на одно и то же место в памяти. Таким образом, изменение любых элементов скопированного списка также повлияет на исходный список. Обязательно проверьте свою реализацию, чтобы используемый вами метод копирования работал должным образом. Поверхностные копии обрабатываются намного быстрее, чем глубокие, и глубокие копии не всегда необходимы, если вы не собираетесь получать от них выгоду.

Порождающие паттерны. Prototype

- **Интерфейс прототипа** : интерфейс, описывающий `clone()`метод.
- **Прототип** : объект/продукт, который реализует интерфейс прототипа.
- **Клиент** : клиентское приложение, которое использует и создает ProtoType.

По умолчанию он будет поверхностно копировать объект, который вы просили клонировать. Объект может быть любого типа, от числа до строки, от словаря до чего угодно, что вы создали.
В примере создано список чисел. На первый взгляд, при копировании этого списка будет казаться, что список был полностью клонирован. А вот внутренних пунктов списка не было. Они будут указывать на ту же ячейку памяти, что и исходный список, однако идентификатор памяти нового списка является новым и отличается от исходного.

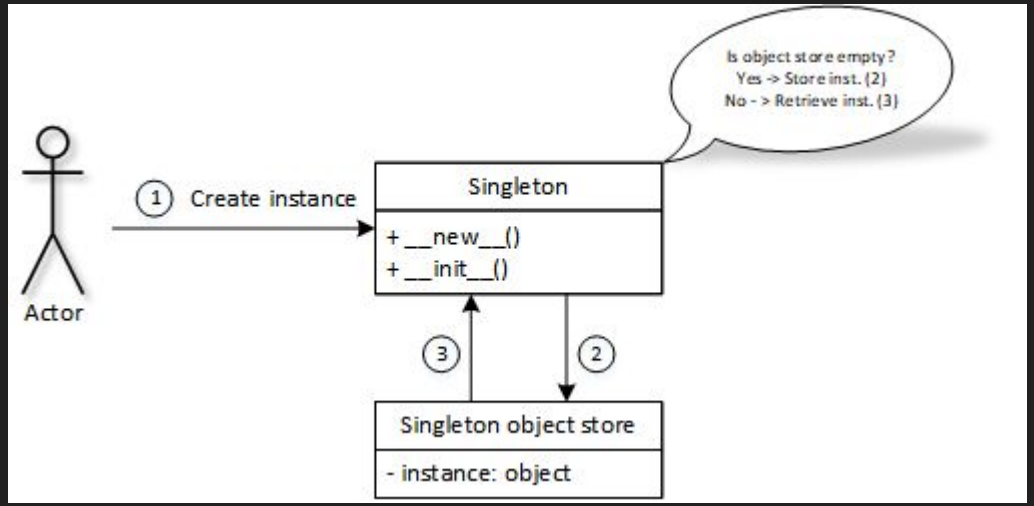


Порождающие паттерны. Singleton

Шаблон singleton — это шаблон, в котором только один экземпляр класса создается и используется во всей проектируемой системе. Имея единственный экземпляр класса, мы получаем следующие преимущества

- Общее состояние или ресурс в приложении — например, подключение к базе данных, регистратор и т. д.;
- Эффективность памяти — в основном потому, что мы гарантируем, что в памяти хранится только один объект;

Кто-то может спросить, чем это отличается от глобальной переменной?
Что ж, наличие глобальной переменной делает объект доступным для всего приложения, но не мешает пользователям создавать другие объекты. Однако шаблон singleton должен ограничивать пользователя в создании нескольких экземпляров класса. Как это достигается путем перехвата запроса, который создает объект и возвращает уже созданный экземпляр



Структурные паттерны

Структурные шаблоны описывают метод проектирования, определяя простой способ реализации отношений классов. Они помогают комбинировать объекты для большей структуры.

Есть несколько видов:

- Adapter/Wrapper
- Bridge
- Facade
- Proxy

Структурные паттерны. Adapter/Wrapper

Иногда классы были написаны, и у вас нет возможности изменить их интерфейс в соответствии с вашими потребностями. Это происходит, если метод, который вы вызываете, находится в другой системе в сети, в библиотеке, которую вы можете импортировать, или вообще в чем-то, что невозможно изменить напрямую для ваших конкретных нужд.

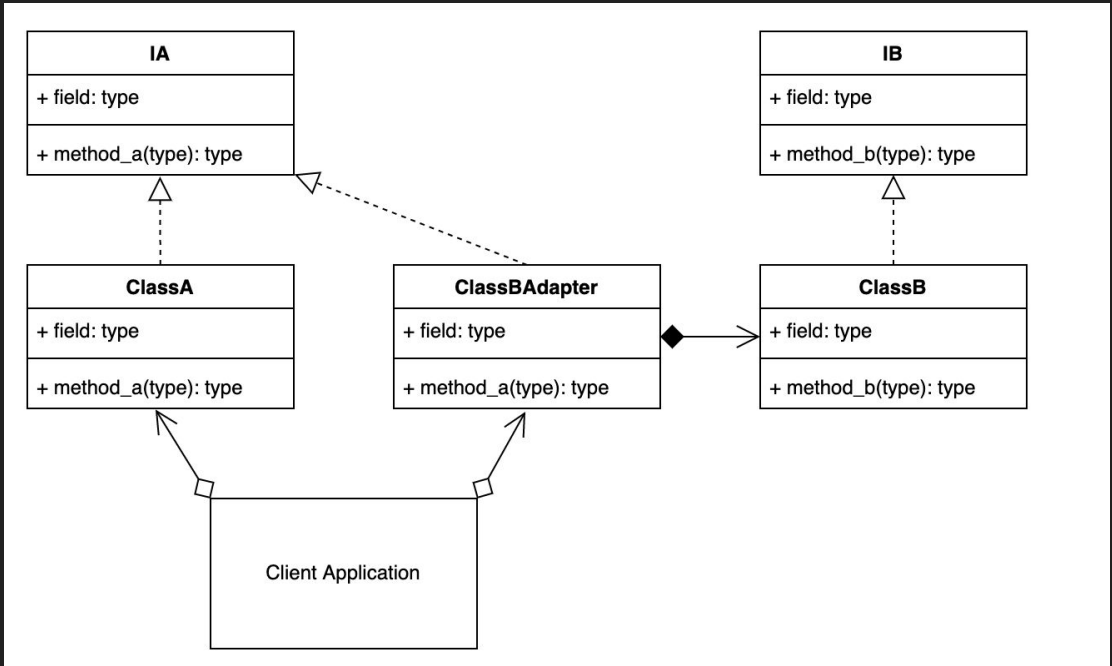
Шаблон проектирования **адаптера** решает следующие проблемы:

- Как можно повторно использовать класс, у которого нет интерфейса, необходимого клиенту?
- Как классы с несовместимыми интерфейсами могут работать вместе?
- Как можно предоставить альтернативный интерфейс для класса?

У вас могут быть два похожих класса, но у них разные сигнатуры методов, поэтому вы создаете адаптер поверх одной из сигнатур метода, чтобы его было проще реализовать и расширить в клиенте.

Адаптер похож на декоратор тем, что он действует как оболочка объекта. Он также используется во время выполнения; однако он не предназначен для рекурсивного использования.

Структурные паттерны. Adapter/Wrapper



Структурные паттерны. Bridge

Шаблон « **Мост»** аналогичен шаблону « Адаптер », за исключением того, что вы его разработали.

Мост — это подход к рефакторингу уже существующего кода, тогда как адаптер создает интерфейс поверх существующего кода с помощью существующих доступных средств без рефакторинга какого-либо существующего кода или интерфейсов.

Например, у вас может быть один класс автомобилей, который производит очень хороший автомобиль. Но вы хотели бы немного изменить дизайн или передать ответственность за создание различных компонентов на аутсорсинг.

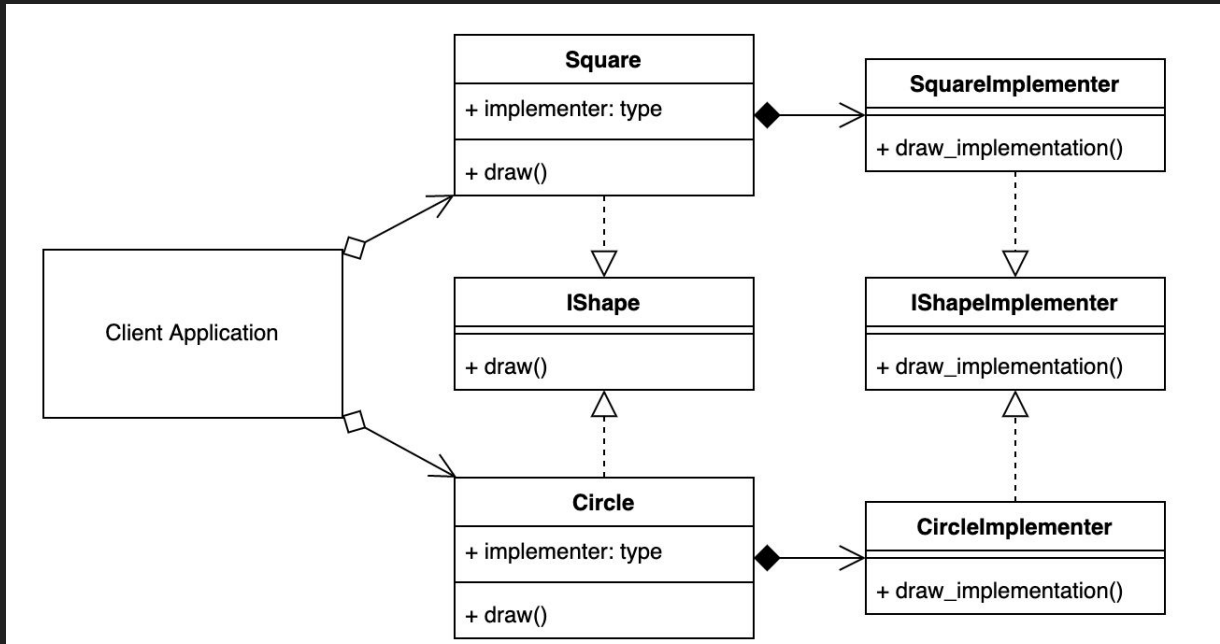
Шаблон Bridge — это процесс разделения абстракции и реализации, так что это даст вам множество новых способов использования ваших классов.

Шаблон моста аналогичен шаблону адаптера, за исключением того, что вы его разработали. Мост — это подход к рефакторингу уже существующего кода, тогда как адаптер адаптируется к существующему коду с помощью существующих интерфейсов и методов без изменения внутренних компонентов.

Структурные паттерны. Bridge

В этом примере рисуем квадрат и круг. Оба они могут быть классифицированы как формы. Форма настроена как интерфейс абстракции. Утонченные абстракции, `Square`и `Circle`, реализующие `IShape` интерфейс.

При создании объектов «Квадрат» и «Круг» им также назначаются соответствующие исполнители, такие как `SquareImplementer` и `CircleImplementer`. Когда вызывается метод каждой формы, `draw` вызывается эквивалентный метод в их реализации. Square и Circle соединены мостом, и над каждым реализатором и абстракцией можно работать независимо.



Поведенческие паттерны

Поведенческие паттерны включают в себя общение между объектами, то, как объекты взаимодействуют и выполняют данную задачу. Всего в Python существует 11 поведенческих паттернов: *Цепочка ответственности, Команда, Интерпретатор, Итератор, Посредник, Память, Наблюдатель, Состояние, Стратегия, Шаблон, Посетитель.*

Поведенческие паттерны. Command

Шаблон **Command** — это поведенческий шаблон проектирования, в котором существует абстракция между объектом, который вызывает команду, и объектом, который ее выполняет.

Например, кнопка вызовет **Invoker** , который вызовет предварительно зарегистрированную **Command** , которую выполнит **Receiver .**

Конкретный класс будет делегировать запрос объекту команды вместо того, чтобы реализовывать запрос напрямую.

Шаблон команды является хорошим решением для реализации функций UNDO/REDO в вашем приложении

Может использоваться например в :

- Кнопки графического интерфейса, меню
- Запись макроса
- Многоуровневая отмена/повтор
- Сеть — отправляйте целые объекты команд по сети, даже в виде пакетов

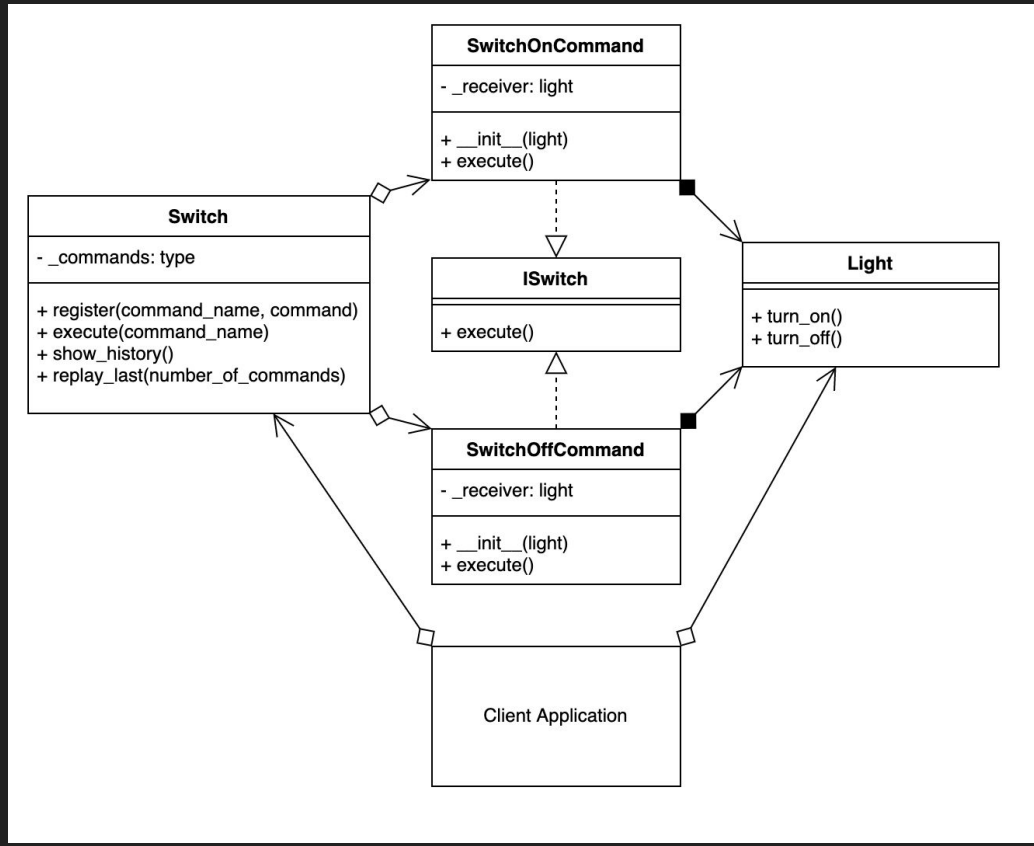
Поведенческие паттерны. Command

Это будет умный выключатель света.

Этот выключатель света будет хранить историю каждого вызова одной из его команд.

И он может воспроизводить свои команды.

В будущем интеллектуальный выключатель света может быть расширен, чтобы его можно было вызывать удаленно или автоматически, в зависимости от датчиков.



Поведенческие паттерны. Iterator

Преимущества использования шаблона Iterator заключаются в том, что клиент может перемещаться по набору агрегатов/объектов/колекции без необходимости понимать их внутренние представления и/или структуры данных.

Итератор обычно содержит два метода, реализующих следующие концепции.

- **next** : возвращает следующий объект в совокупности (коллекция, объект).
- **has_next** : возвращает логическое значение, указывающее, находится ли Iterable в конце итерации или нет.

Преимущества и недостатки

- Упрощает классы хранения данных.
- Позволяет реализовать различные способы обхода структуры данных.
- Позволяет одновременно перемещаться по структуре данных в разные стороны.
- Не оправдан, если можно обойтись простым циклом.

Поведенческие паттерны. Iterator