1. Методы `wait()`, `notify()` класса `Object`, интерфейсы `Lock` и `Condition`.

   wait(): освобождает монитор и переводит вызывающий поток в состояние ожидания до тех пор, пока другой поток не вызовет метод notify()

   notify(): продолжает работу потока, у которого ранее был вызван метод wait()

   notifyAll(): возобновляет работу всех потоков, у которых ранее был вызван метод wait**()**

   Все эти методы вызываются только из синхронизированного контекста - синхронизированного блока или метода.

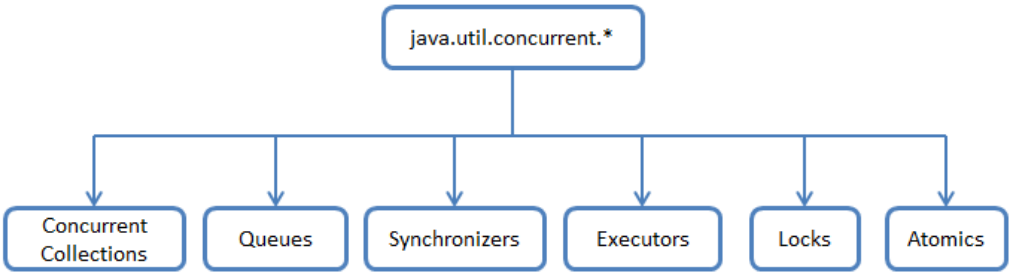
   Классы блокировок реализуют интерфейс **Lock**, который определяет следующие методы:

   - **void lock()**: ожидает, пока не будет получена блокировка
   - **void lockInterruptibly() throws InterruptedException**: ожидает, пока не будет получена блокировка, если поток не прерван
   - **boolean tryLock()**: пытается получить блокировку, если блокировка получена, то возвращает true. Если блокировка не получена, то возвращает false. В отличие от метода `lock()` не ожидает получения блокировки, если она недоступна
   - **void unlock()**: снимает блокировку
   - **Condition newCondition()**: возвращает объект Condition, который связан с текущей блокировкой

   Применение объектов Condition во многом аналогично использованию методов `wait/notify/notifyAll` класса Object, которые были рассмотрены в одной из прошлых тем. В частности, мы можем использовать следующие методы интерфейса Condition:

   - **await**: поток ожидает, пока не будет выполнено некоторое условие и пока другой поток не вызовет методы `signal/signalAll`. Во многом аналогичен методу `wait` класса Object
   - **signal**: сигнализирует, что поток, у которого ранее был вызван метод `await()`, может продолжить работу. Применение аналогично использованию методу `notify` класса Object
   - **signalAll**: сигнализирует всем потокам, у которых ранее был вызван метод `await()`, что они могут продолжить работу. Аналогичен методу `notifyAll()` класса Object

2. Классы-сихронизаторы из пакета `java.util.concurrent`.



В данном разделе представлены классы для активного управления потоков.

Semaphore Class 1.5 — Семафоры чаще всего используются для ограничения количества потоков при работе с аппаратными ресурсами или файловой системой. Доступ к общему ресурсу управляется с помощью счетчика. Если он больше нуля, то доступ разрешается, а значение счетчика уменьшается. Если счетчик равен нулю, то текущий поток блокируется, пока другой поток не освободит ресурс. Количество разрешений и «честность» освобождения потоков задается через конструктор. Узким местом при использовании семафоров является задание количества разрешений, т.к. зачастую это число приходится подбирать в зависимости от мощности «железа».

CountDownLatch Class 1.5 — Позволяет одному или нескольким потокам ожидать до тех пор, пока не завершится определенное количество операций, выполняющих в других потоках. Классический пример с драйвером довольно неплохо описывает логику класса: Потоки, вызывающие драйвер, будут висеть в методе await (с таймаутом или без), пока поток с драйвером не выполнит инициализацию с последующим вызовом метода countDown. Этот метод уменьшает счетчик count down на единицу. Как только счетчик становится равным нулю, все ожидающие потоки в await продолжат свою работу, а все последующие вызовы await будут проходить без ожиданий. Счетчик count down одноразовый и не может быть сброшен в первоначальное состояние.

CyclicBarrier Class 1.5 — Может использоваться для синхронизации заданного количества потоков в одной точке. Барьер достигается когда N-потоков вызовут метод await(...) и заблокируются. После чего счетчик сбрасывается в исходное значение, а ожидающие потоки освобождаются. Дополнительно, если нужно, существует возможность запуска специального кода до разблокировки потоков и сброса счетчика. Для этого через конструктор передается объект с реализацией Runnable интерфейса.

Exchanger<V> Class 1.5 — Как видно из названия, основное предназначение данного класса — это обмен объектами между двумя потоками. При этом, также поддерживаются null значения, что позволяет использовать данный класс для передачи только одного объекта или же просто как синхронизатор двух потоков. Первый поток, который вызывает метод exchange(...) заблокируется до тех пор, пока тот же метод не вызовет второй поток. Как только это произойдет, потоки обменяются значениями и продолжат свою работу.

Phaser Class 1.7 — Улучшенная реализация барьера для синхронизации потоков, которая совмещает в себе функционал CyclicBarrier и CountDownLatch, вбирая в себя самое лучшее из них. Так, количество потоков жестко не задано и может динамически меняться. Класс может повторно переиспользоваться и сообщать о готовности потока без его блокировки. Более подробно можно почитать в хабратопике тут.

Concurrent Collections — набор коллекций, более эффективно работающие в многопоточной среде нежели стандартные универсальные коллекции из java.util пакета. Вместо базового враппера Collections.synchronizedList с блокированием доступа ко всей коллекции используются блокировки по сегментам данных или же оптимизируется работа для параллельного чтения данных по wait-free алгоритмам.

Queues — неблокирующие и блокирующие очереди с поддержкой многопоточности. Неблокирующие очереди заточены на скорость и работу без блокирования потоков. Блокирующие очереди используются, когда нужно «притормозить» потоки «Producer» или «Consumer», если не выполнены какие-либо условия, например, очередь пуста или перепонена, или же нет свободного «Consumer»'a.

3. **Volatile** - переменная

   Она всегда будет атомарно читаться и записываться. Даже если это 64-битные `double` или `long`.

   Java-машина не будет помещать ее в кэш. Так что ситуация, когда 10 потоков работают со своими локальными копиями исключена.

4.

Executor	**ExecutorService**
Executor - основной интерфейс пула потоков Java, используемый для одновременного выполнения представленных задач.	ExecutorService - это расширение интерфейса Executor, предоставляющее методы для асинхронного выполнения и закрытия пула потоков.
Предоставить метод execute () для отправки задач	Предоставить метод submit () для отправки задач
Метод execute () не имеет возвращаемого значения	Метод submit () возвращает объект Future, который можно использовать для получения результата выполнения задачи.
Невозможно отменить задачу	Вы можете отменить задачу в ожидании через Future.cancel ()
Не существует метода, связанного с закрытием пула потоков	Предоставляет метод для закрытия пула потоков

```java
public static void main(String[] args) {

    // create the pool
    ExecutorService service = Executors.newFixedThreadPool( nThreads: 10);

    // submit the tasks for execution
    for (int i = 0; i < 100; i++) {
        service.execute(new Task());
    }
    System.out.println("Thread Name: " + Thread.currentThread().getName());
}

static class Task implements Runnable {
    public void run() {
        System.out.println("Thread Name: " + Thread.currentThread().getName());
    }
}
```

Интерфейс Callable<V>

Интерфейс Callable<V> очень похож на интерфейс Runnable. Объекты, реализующие данные интерфейсы, исполняются другим потоком. Однако, в отличие от Runnable, интерфейс Callable использует Generic'и для определения типа возвращаемого объекта. Runnable содержит метод run(), описывающий действие потока во время выполнения, а Callable – метод call().

Интерфейс Future<V>

Интерфейс Future также, как и интерфейс Callable, использует Generic'и. Методы интерфейса можно использовать для проверки завершения работы потока, ожидания завершения и получения результата. Результат выполнения может быть получен методом get, если поток завершил работу. Прервать выполнения задачи можно методом cancel. Дополнительные методы позволяют определить завершение задачи : нормальное или прерванное. Если задача завершена, то прервать ее уже невозможно.

Использование PreparedStatement

PreparedStatement предварительно компилирует запросы, которые могут содержать входные параметры обозначенные символом '?'

Пример использования PreparedStatement

```java
PreparedStatement pstmt = null;
//~~~ Чтение таблицы БД ~~~
pstmt = connection.prepareStatement(
        "SELECT * FROM GOODS where id > ? and id < ?");
// Определяем значения параметров
pstmt.setInt(1, 2);
pstmt.setInt(2, 10);
// Выполнение запроса
ResultSet rs = preparedStatement.executeQuery();

// Вывод результата запроса
while (rs.next()) {
   System.out.println("" + rs.getRow() + ". " +
                      "id = " + result2.getInt("id") +
                                ", name = '" + rs.getString("name"));
}
//~~~ Запись в таблицу БД ~~~
pstmt = connection.prepareStatement(
       "INSERT INTO GOODS(name) values(?)");
pstmt.setString(1, "Кофе");
pstmt.executeUpdate();
```

DriverManager - The basic service for managing a set of JDBC drivers.

Connection - A connection (session) with a specific database. SQL statements are executed and results are returned within the context of a connection.

Statement - The object used for executing a static SQL statement and returning the results it produces.

PreparedStatement - A SQL statement is precompiled and stored in a PreparedStatement object.

ResultSet - A table of data representing a database result set, which is usually generated by executing a statement that queries the database.

A ResultSet object maintains a cursor pointing to its current row of data. Initially the cursor is positioned before the first row. The next method moves the cursor to the next row, and because it returns false when there are no more rows in the ResultSet object, it can be used in a while loop to iterate through the result set.

RowSet - Объекты типа RowSet являются альтернативой объектам типа ResultSet. Интерфейс RowSet расширяет интерфейс ResultSet и определяет дополнительные методы, которые позволяют работать с компонентной архитектурой JavaBean. Кроме того, объекты данного типа могут использоваться при отсутствии постоянного соединения с базой данных

PATTERNS

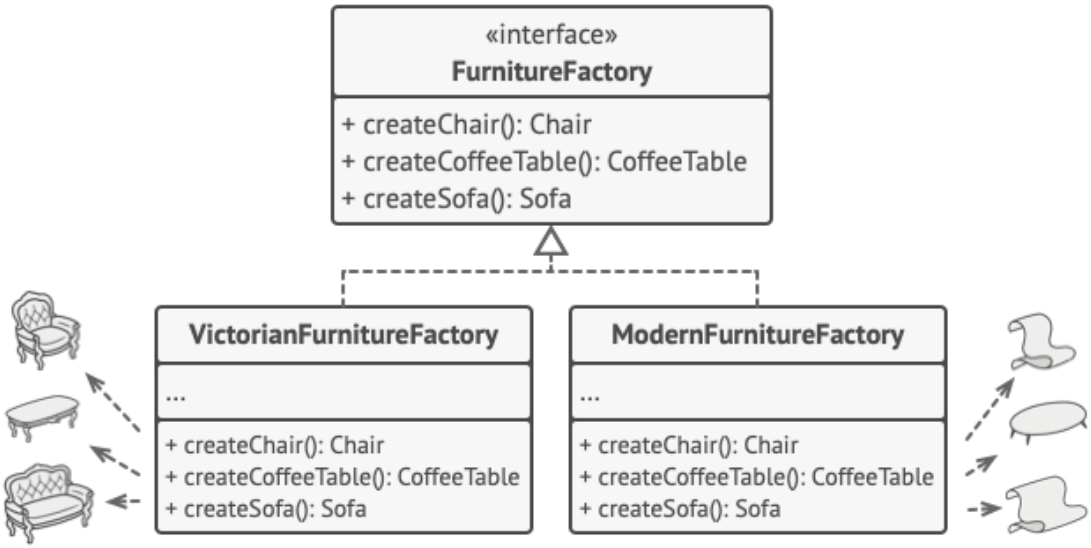
Порождающие

1. Factory

   Выносим создание объектов с общим интерфейсом в отдельный метод

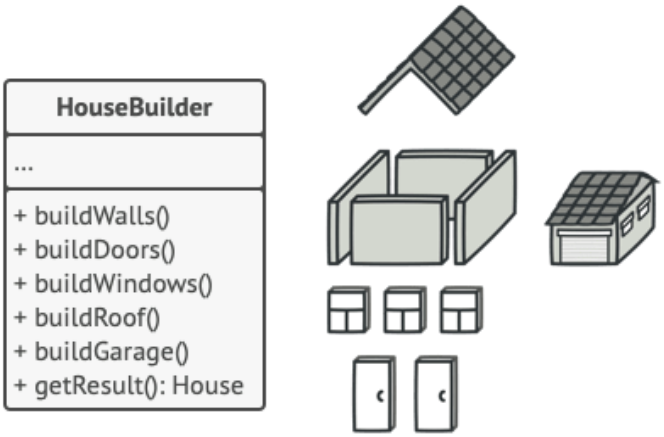
2. Abstract Factory

   Абстрактная фабрика предлагает выделить общие интерфейсы для отдельных продуктов, составляющих семейства.



3. Builder

Паттерн Строитель предлагает вынести конструирование объекта за пределы его собственного класса, поручив это дело отдельным объектам, называемым строителями.



4. Prototype

Паттерн Прототип поручает создание копий самим копируемым объектам. Он вводит общий интерфейс для всех объектов, поддерживающих клонирование. Это позволяет копировать объекты, не привязываясь к их конкретным классам.

5. Singletone

Одиночка — это порождающий паттерн проектирования, который гарантирует, что у класса есть только один экземпляр, и предоставляет к нему глобальную точку доступа.

Структурные

6. Adapter

Это объект-переводчик, который трансформирует интерфейс или данные одного объекта в такой вид, чтобы он стал понятен другому объекту.

При этом адаптер оборачивает один из объектов, так что другой объект даже не знает о наличии первого. Например, вы можете обернуть объект, работающий в метрах, адаптером, который бы конвертировал данные в футы.

Адаптеры могут не только переводить данные из одного формата в другой, но и помогать объектам с разными интерфейсами работать сообща. Это работает так:

Адаптер имеет интерфейс, который совместим с одним из объектов.Поэтому этот объект может свободно вызывать методы адаптера.

Адаптер получает эти вызовы и перенаправляет их второму объекту, но уже в том формате и последовательности, которые понятны второму объекту.

Иногда возможно создать даже двухсторонний адаптер, который работал бы в обе стороны.

7. Bridge

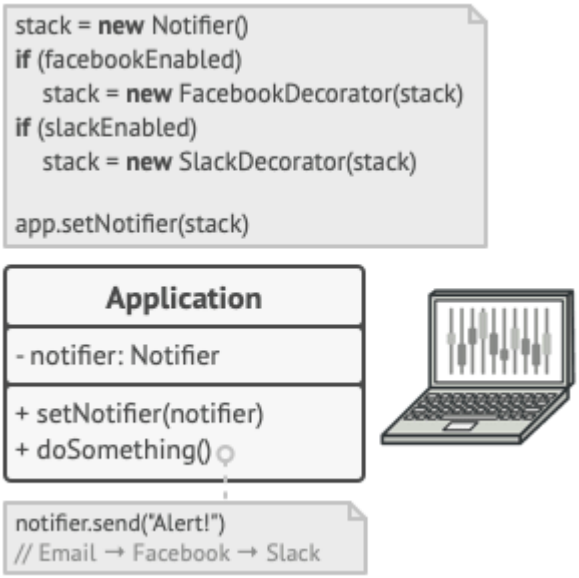
Мост — это структурный паттерн проектирования, который разделяет один или несколько классов на две отдельные иерархии — абстракцию и реализацию, позволяя изменять их независимо друг от друга.

8. Composite

Компоновщик — это структурный паттерн, который позволяет создавать дерево объектов и работать с ним так же, как и с единичным объектом. Компоновщик давно стал синонимом всех задач, связанных с построением дерева объектов. Все операции компоновщика основаны на рекурсии и «суммировании» результатов на ветвях дерева.

9. Decorator

Декоратор — это структурный паттерн проектирования, который позволяет динамически добавлять объектам новую функциональность, оборачивая их в полезные «обёртки».

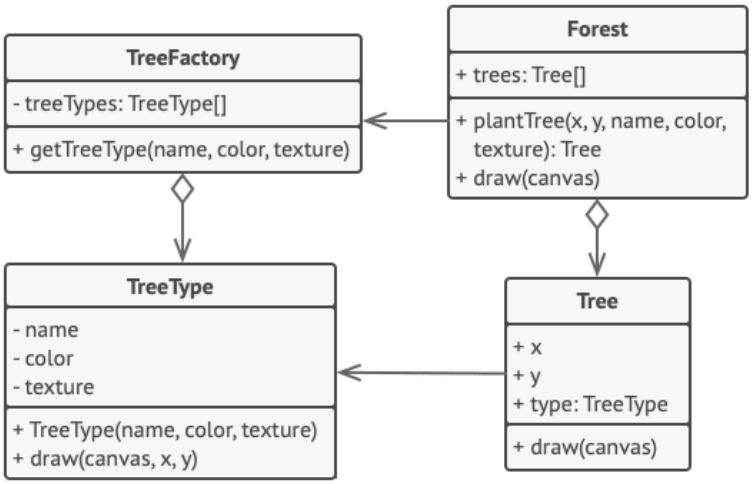


10. Facade

Фасад — это простой интерфейс для работы со сложной подсистемой, содержащей множество классов. Фасад может иметь урезанный интерфейс, не имеющий 100% функциональности, которой можно достичь, используя сложную подсистему напрямую. Но он предоставляет именно те фичи, которые нужны клиенту, и скрывает все остальные.

11. Flyweight

Структурный паттерн проектирования, который позволяет вместить бóльшее количество объектов в отведённую оперативную память. Легковес экономит память, разделяя общее состояние объектов между собой, вместо хранения одинаковых данных в каждом объекте.



12. Proxy

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.

Поведенческие

13. Chain of Responsibility

Цепочка обязанностей — это поведенческий паттерн проектирования, который позволяет передавать запросы последовательно по цепочке обработчиков. Каждый последующий обработчик решает, может ли он обработать запрос сам и стоит ли передавать запрос дальше по цепи.

14. Command

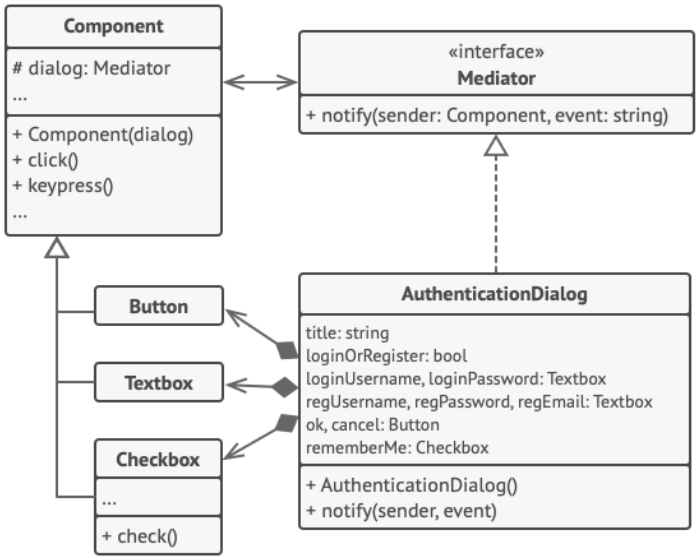
Поведенческий паттерн проектирования, который превращает запросы в объекты, позволяя передавать их как аргументы при вызове методов, ставить запросы в очередь, логировать их, а также поддерживать отмену операций.

15. Iterator

Это поведенческий паттерн проектирования, который даёт возможность последовательно обходить элементы составных объектов, не раскрывая их внутреннего представления.

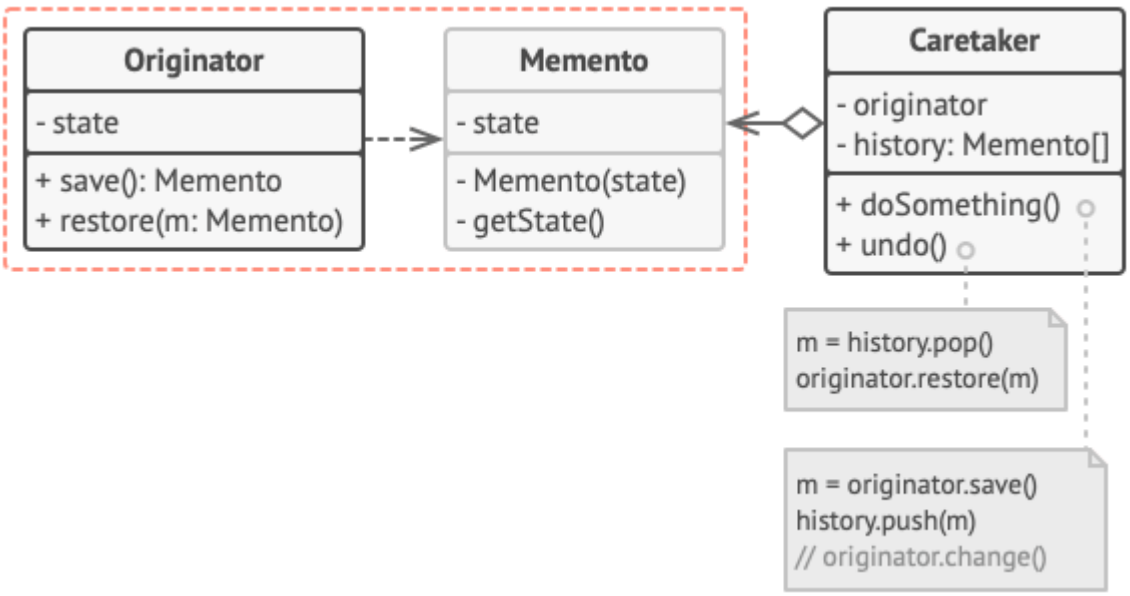
16. Mediator

Поведенческий паттерн проектирования, который позволяет уменьшить связанность множества классов между собой, благодаря перемещению этих связей в один класс-посредник.



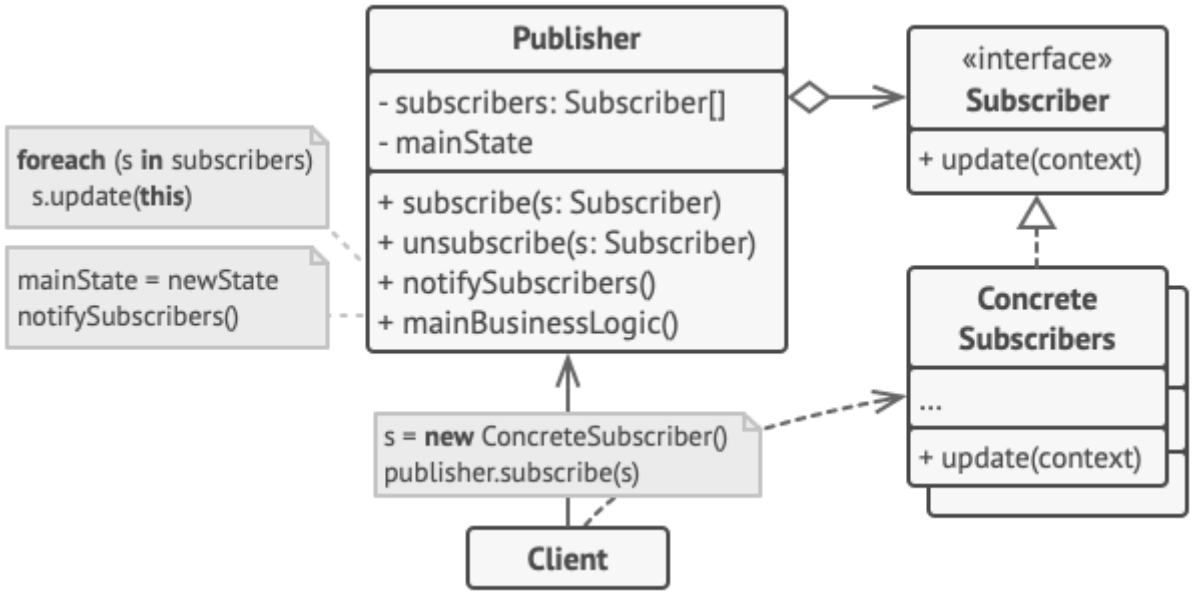
17. Memento

Поведенческий паттерн проектирования, который позволяет сохранять и восстанавливать прошлые состояния объектов, не раскрывая подробностей их реализации.



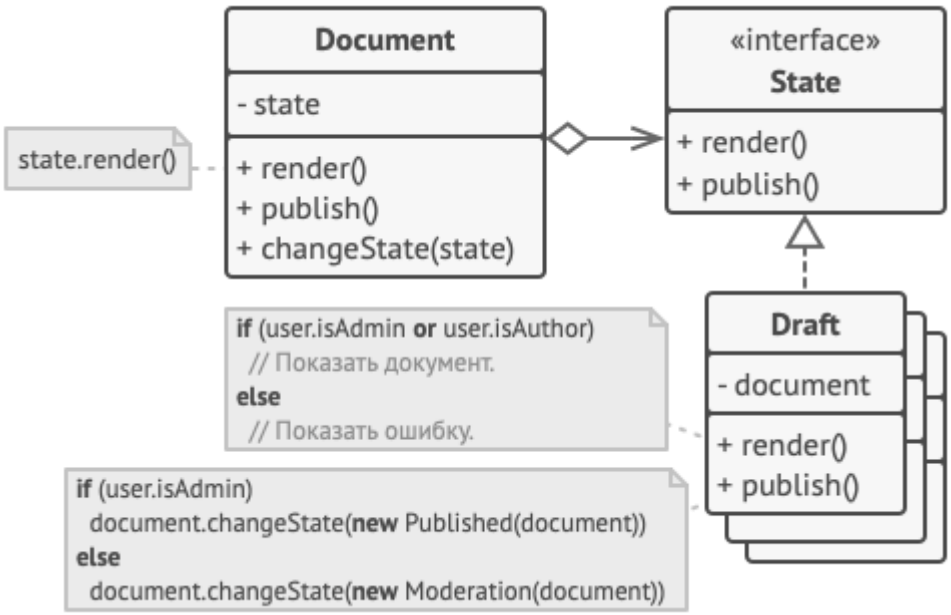
18. Observer

Поведенческий паттерн проектирования, который создаёт механизм подписки, позволяющий одним объектам следить и реагировать на события, происходящие в других объектах.



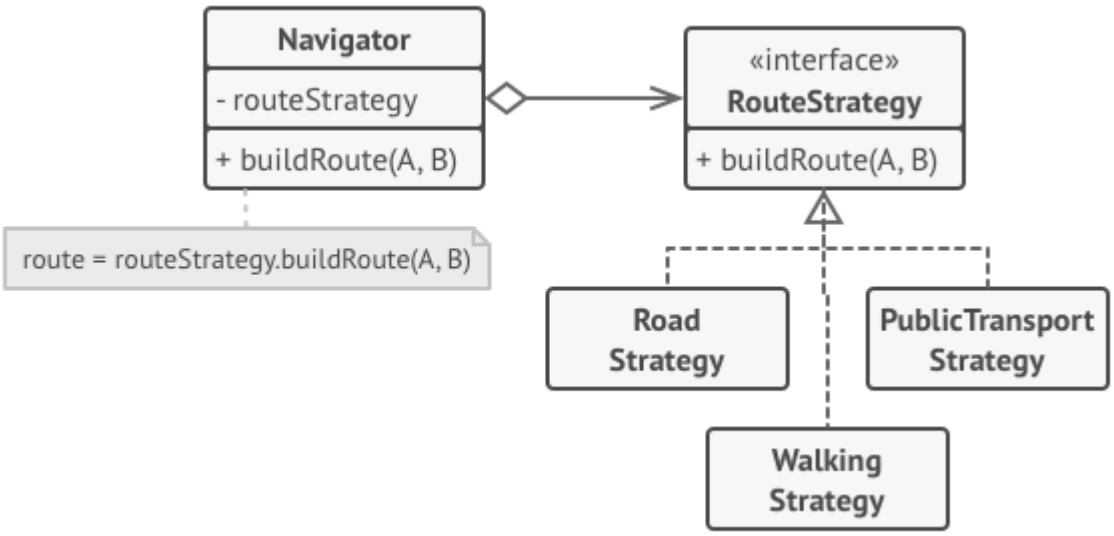
19. State

Поведенческий паттерн проектирования, который позволяет объектам менять поведение в зависимости от своего состояния. Извне создаётся впечатление, что изменился класс объекта.



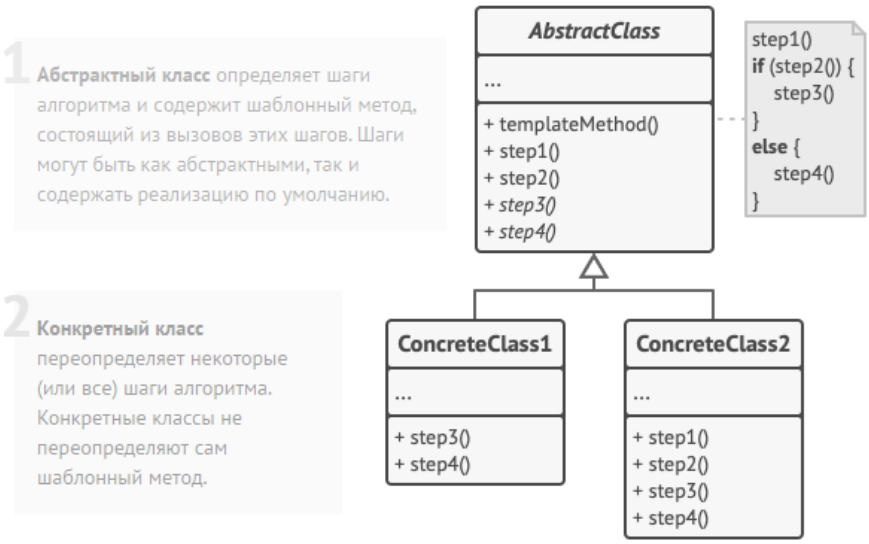
20. Strategy

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



21. Template method

Поведенческий паттерн проектирования, который определяет скелет алгоритма, перекладывая ответственность за некоторые его шаги на подклассы. Паттерн позволяет подклассам переопределять шаги алгоритма, не меняя его общей структуры.



22. Visitor

Поведенческий паттерн проектирования, который позволяет добавлять в программу новые операции, не изменяя классы объектов, над которыми эти операции могут выполняться.