



5. Шаблоны объектно-ориентированного проектирования

Одно из ключевых различий между архитектурными паттернами Кристофера Александера и паттернами проектирования программного обеспечения заключается в том, что паттерны программного обеспечения поощряют повторное использование и модификацию:

- объектов;
- моделей;
- классов.

Архитектурные паттерны в основном поощряют повторное использование чертежей, то есть классов, но не реальных зданий, парков и городов, построенных на основе этих чертежей. Гораздо проще снести и заменить часть программного пакета по мере необходимости, чем расширять дом.

Паттерн позволяет динамически изменять состояние объекта.

Иерархии объектов и паттерны проектирования

Объектно-ориентированное программирование — это нечто большее, чем просто инкапсуляция в объект некоторых данных и процедур для манипулирования этими данными.

Объектно-ориентированные методы также имеют дело с классификацией объектов и рассматривают отношения между различными классами объектов. Основным средством для выражения отношений между классами объектов считается деривация — процесс создания новых классов, получаемых из существующих классов.

Что делает деривацию такой полезной, так это понятие наследования. Производные классы наследуют характеристики классов, от которых они получены. Унаследованную функциональность можно переопределить, а в производном классе — определить дополнительную функциональность. Особенность этой книги — то, что практически все структуры данных представлены в контексте одной иерархии классов. По сути, иерархия классов представляет собой таксономию структур данных. Различные реализации этой



абстрактной структуры данных представляют собой производные от одного и того же абстрактного базового класса.

Связанные базовые классы, в свою очередь, являются производными от классов, которые абстрагируют и инкапсулируют общие черты этих классов. В дополнение к работе с иерархически связанными классами, опытные объектно-ориентированные дизайнеры также очень тщательно рассматривают взаимодействие между несвязанными классами. С опытом хороший проектировщик обнаруживает повторяющиеся паттерны взаимодействия между объектами. Научившись использовать эти паттерны, ваши объектно-ориентированные проекты станут более гибкими и пригодными для повторного использования.

В этом тексте используются следующие паттерны объектно-ориентированного проектирования:

1. Порождающие.

Эти паттерны решают проблемы обеспечения гибкости создания объектов. Отвечают за удобное и безопасное создание новых объектов или даже целых семейств объектов. Обычная форма создания объекта часто приводит к проблемам проектирования или увеличивает сложность конструкции. Порождающие шаблоны проектирования решают эту проблему, контролируя процесс создания объекта.

1. Фабричный Метод (Factory Method).
2. Абстрактная фабрика (Abstract Factory).
3. Строитель (Builder).
4. Пул объектов (Object Pool).
5. Прототип (Prototype).
6. Одиночка (Singleton).
7. Пул одиночек (Multiton).

2. Структурные.

Отвечают за построение удобных в поддержке иерархий классов. Такие паттерны уровня класса используют наследование для составления композиций из интерфейсов и реализаций. Вместо композиции интерфейсов или реализаций, структурные паттерны уровня объекта компонуют объекты для получения новой функциональности.

1. Адаптер (Adapter).



2. Мост (Bridge).
3. Компоновщик (Composite).
4. Декоратор (Decorator).
5. Фасад (Facade).
6. Приспособленец (Flyweight).
7. Заместитель (Proxy).

3. Поведенческие.

Эти паттерны решают проблемы эффективного взаимодействия между объектами. Паттерны поведения связаны с алгоритмами и распределением обязанностей между объектами. Речь в них идёт не только о самих объектах и классах, но и о типичных схемах взаимодействия между ними.


Эти паттерны характеризуют сложный поток управления, который трудно проследить во время выполнения программы. Внимание акцентируется не на схеме управления как таковой, а на связях между объектами. Поведенческие паттерны выполняют задачи эффективного и безопасного взаимодействия между объектами программы.

К поведенческим паттернам относятся:

1. Цепочка ответственности (Chain of responsibility).
2. Команда (Command).
3. Итератор (Iterator).
4. Посредник (Mediator).
5. Хранитель (Memento).
6. Наблюдатель (Observer).
7. Состояние (State).
8. Стратегия (Strategy).
9. Шаблонный метод (Template method).
- 10.Посетитель (Visitor).

6. Порождающие шаблоны



 **Порождающие шаблоны** (*Creational patterns*) — шаблоны проектирования, которые имеют дело с процессом создания объектов. Они позволяют сделать систему независимой от способа создания, композиции и представления объектов. Шаблон, порождающий классы, использует наследование, чтобы изменять наследуемый класс, а шаблон, порождающий объекты, делегирует инстанцирование другому объекту.

Порождающие шаблоны инкапсулируют знания о конкретных классах, которые применяются в системе, то есть скрывают детали того, как такие классы создаются и стыкуются. Единственная информация об объектах, известная системе — это их интерфейсы, определённые посредством абстрактных классов. Следовательно, порождающие шаблоны обеспечивают большую гибкость при решении вопроса о том, что создаётся, кто это создаёт, как и когда. Можно собрать систему из «готовых» объектов с самой различной структурой и функциональностью статически (на этапе компиляции) или динамически (во время выполнения).

Иногда допустимо выбирать между тем или иным порождающим шаблоном. Например, есть случаи, когда с пользой для дела используется как прототип, так и абстрактная фабрика. В других ситуациях порождающие шаблоны дополняют друг друга. Так, применяя строитель, можно использовать другие шаблоны для решения вопроса о том, какие компоненты надо строить, а прототип часто реализуется вместе с одиночкой. Порождающие шаблоны тесно связаны друг с другом, их рассмотрение лучше проводить совместно, чтобы лучше были видны их сходства и различия.

Порождающие

Эти паттерны решают проблемы обеспечения гибкости создания объектов. Отвечают за удобное и безопасное создание новых объектов или даже целых семейств объектов.

Обычная форма создания объекта часто приводит к проблемам проектирования или увеличивает сложность конструкции. Порождающие шаблоны проектирования решают эту проблему, контролируя процесс создания объекта.

1. Одиночка (Singleton).
2. Фабричный Метод (Factory Method).
3. Абстрактная фабрика (Abstract Factory).



4. Строитель (Builder).
5. Пул объектов (Object Pool).
6. Пул одиночек (Multiton).
7. Прототип (Prototype).

6.1. Одиночка (Singleton)

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

Допустим, в городе требуется организовать связь между жителями. Мы можем связать всех жителей между собой, протянув между ними кабели телефонных линий. Такая система неверна.

Например, очень затратно будет добавить одного жителя в связи: протянуть по ещё одной линии к каждому жителю. Чтобы этого избежать, создадим телефонную станцию, которая и станет нашим «одиночкой». Она всегда одна, и если кому-то потребуется связаться с кем-то, это можно сделать через такую телефонную станцию: все обращаются только к ней.

Соответственно, для добавления нового жителя надо изменить только записи на самой телефонной станции. Один раз создав телефонную станцию, все станут пользоваться только ей. В свою очередь, эта станция помнит всё, что с ней происходило с момента создания. Каждый может воспользоваться этой информацией, даже если он только приехал в город.

Основной смысл «одиночки»: когда говорим «Нужна телефонная станция», то отвечают «Она уже построена там-то», а не «Давай её сделаем заново». «Одиночка» всегда один.

Проблема

Одиночка решает сразу две проблемы, нарушая принцип единственной ответственности класса.

Гарантирует наличие единственного экземпляра класса

Представим, что мы создали объект, а через некоторое время пробуем создать ещё один. В этом случае хотелось бы получить старый объект, вместо создания



нового. Такое поведение невозможно реализовать через обычный конструктор, так как конструктор класса всегда возвращает новый объект.

Предоставляет глобальную точку доступа

Это не просто глобальная переменная, через которую можно достигаться к определённому объекту. Глобальные переменные не защищены от записи, поэтому любой код способен подменять их значения без нашего ведома.

Решение

Все реализации одиночки сводятся к тому, чтобы скрыть конструктор по умолчанию и создать публичный статический метод, который и будет контролировать жизненный цикл объекта-одиночки.

Если у нас есть доступ к классу одиночки, значит, появится доступ и к этому статическому методу. Из какой бы точки кода его ни вызвали, он всегда будет отдавать один и тот же объект.

Структура

Singleton
- instance : Singleton
- Singleton() + static getInstance() : Singleton

Одиночка определяет статический метод `getInstance()`, который возвращает единственный экземпляр своего класса. Конструктор одиночки должен быть скрыт от клиентов.

Задача вызова метода `getInstance` — стать единственным способом получить объект этого класса.

Применимость



Паттерн «Одиночка» используется, когда:

- должен быть ровно один экземпляр некоторого класса, легкодоступный всем клиентам;
- единственный экземпляр расширяется путём порождения подклассов, и клиентам надо иметь возможность работать с расширенным экземпляром без модификации своего кода.

Преимущества и недостатки

Преимущества	Недостатки
<ol style="list-style-type: none">1. Контролируемый доступ к единственному экземпляру.2. Допускает уточнение операций и представления.3. Допускает переменное число экземпляров.	<ol style="list-style-type: none">1. Нарушает принцип единственной ответственности класса.2. Усложняет модульное тестирование.

6.2. Фабричный метод (Factory Method)

Это порождающий паттерн проектирования, который определяет общий интерфейс для создания объектов в суперклассе, позволяя подклассам изменять тип создаваемых объектов.

Метод фабрики считается креативным паттерном проектирования, то есть связанным с созданием объектов. В паттерне «Фабрика» мы создаём объект, не раскрывая логики создания клиенту, а клиент использует тот же общий интерфейс для создания нового типа объекта.

Паттерн «Фабрика» вводит свободную связь между классами, что считается наиболее важным принципом, который необходимо учитывать и применять при проектировании архитектуры приложения. Свободная связь может быть введена в архитектуру приложения путём программирования на основе абстрактных сущностей, а не конкретных реализаций. Это делает нашу архитектуру не только более гибкой, но и менее хрупкой.



Назначение

Определяет интерфейс для создания объекта, но оставляет подклассам решение, какой класс инстанцировать. Фабричный метод позволяет классу делегировать инстанцирование подклассам.

Проблема

Представим, что требуется создать программу управления грузовыми перевозками. Изначально перевозки осуществляются на грузовых автомобилях, поэтому весь код работает с объектами класса **«Грузовик»**.

В какой-то момент программой заинтересовались морские перевозчики. Но большая часть кода привязана к классу **«Грузовик»**. Чтобы добавить в программу классы морских судов, надо исправить большую часть кода. Изменения будут требоваться каждый раз при добавлении нового вида транспорта.

Таким образом, получится очень большой код, наполненный условными операторами, определяющими тип транспорта.

Применимость

Паттерн «Фабричный метод» стоит использовать:

- когда заранее неизвестны типы и зависимости объектов, с которыми работает код;
- когда хотим дать возможность пользователям расширять части нашего фреймворка или библиотеки;
- когда требуется экономить системные ресурсы, повторно используя уже созданные объекты, вместо порождения новых.

Преимущества и недостатки

Преимущества	Недостатки
--------------	------------



<ol style="list-style-type: none">1. Избавляет класс от привязки к конкретным классам продуктов.2. Выделяет код производства продуктов в одно место, упрощая поддержку кода.3. Упрощает добавление новых продуктов в программу.4. Реализует принцип открытости/закрытости.	<ol style="list-style-type: none">5. Иногда приводит к созданию больших параллельных иерархий классов, так как для каждого класса продукта надо создать свой подкласс создателя.
---	--

6.3. Абстрактная фабрика (Abstract Factory)

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

Проблема

Допустим, требуется создать симулятор мебельного магазина.

В коде содержится:

1. Семейство зависимых продуктов: стол, шкаф, диван.
2. Вариации этого семейства. Представим, что мебель представлена в разных стилях: барокко, модерн, прованс.

Требуется такой способ создавать объекты продуктов, чтобы они сочетались с другими продуктами семейства. Помимо этого, не следует вносить изменения в существующий код при добавлении новых продуктов или семейств в программу.

Применимость

Паттерн «Абстрактная фабрика» стоит использовать, когда:



- система не зависит от того, как создаются, компонуются и представляются входящие в неё объекты;
- входящие в семейство взаимосвязанные объекты используются вместе, и нам требуется обеспечить выполнение этого ограничения;
- система должна конфигурироваться одним из семейств составляющих её объектов;
- мы хотим предоставить библиотеку объектов, раскрывая только их интерфейсы, но не реализацию.

Преимущества и недостатки

Преимущества	Недостатки
<ol style="list-style-type: none">1. Гарантирует сочетаемость создаваемых продуктов.2. Избавляет клиентский код от привязки к конкретным классам продуктов.3. Выделяет код производства продуктов в одно место, упрощая поддержку кода.4. Упрощает добавление новых продуктов в программу.5. Реализует принцип открытости/закрытости.	<ol style="list-style-type: none">1. Усложняет код программы из-за введения множества дополнительных классов.2. Требуется наличие всех типов продуктов в каждой вариации.

6.4. Строитель (Builder)

Строитель — это порождающий паттерн проектирования, который позволяет создавать сложные объекты пошагово. Он даёт возможность использовать один и тот же код строительства для получения разных представлений объектов.

Проблема



Представим сложный объект, требующий кропотливой пошаговой инициализации множества полей и вложенных объектов. Код инициализации таких объектов обычно спрятан внутри монструозного конструктора с десятком параметров.

Например, надо создать объект «Дом». Чтобы построить стандартный дом, требуется поставить четыре стены, установить двери, вставить окна и положить крышу. Но что, если дом нужен больше, светлее?

Применимость

Паттерн «Строитель» используется, когда:

- алгоритм создания сложного объекта не зависит от того, из каких частей состоит объект, и как они стыкуются между собой;
- процесс конструирования обеспечивает различные представления конструируемого объекта.

Преимущества и недостатки

Преимущества	Недостатки
<ol style="list-style-type: none">1. Позволяет изменять внутреннее представление продукта.2. Изолирует код, реализующий конструирование и представление.3. Даёт более тонкий контроль над процессом конструирования.	<ol style="list-style-type: none">1. При существенных различиях типов конструируемых объектов может усложняться интерфейс строителя.

6.5. Пул объектов (Object Pool)

Объектный пул — порождающий шаблон проектирования, набор инициализированных и готовых к использованию объектов. Когда системе требуется объект, он не создаётся, а берётся из пула. Если объект больше не нужен, он не уничтожается, а возвращается в пул.



Пул объектов требуется для улучшения производительности и эффективности использования памяти благодаря повторному использованию объектов из фиксированного пула вместо их индивидуального выделения и освобождения.

Применимость

Объектный пул применяется, когда:

- программа может создать только ограниченное число экземпляров некоторого класса;
- надо часто создавать и удалять объекты;
- объекты одного размера;
- выделение объектов из кучи работает медленно или может привести к фрагментации памяти;
- каждый объект инкапсулирует ресурс типа базы данных или сетевого соединения, который сложно получать и можно использовать повторно.

Преимущества и недостатки

Преимущества	Недостатки
<ol style="list-style-type: none">1. Централизованное управление ресурсами.2. В процессе работы ресурсы не тратятся.	<ol style="list-style-type: none">1. После того как объект возвращён, он должен вернуться в состояние, пригодное для дальнейшего использования. Для этого потребуется дополнительное программирование.2. Если в объекте есть секретные данные, то после его использования надо позаботиться об удалении этих данных.

6.6. Пул одиночек (Multiton)

Пул одиночек (англ. **Multiton**) похож на шаблон **Одиночка**. Позволяет создавать и содержать несколько **одиночек**, доступ к которым можно получить по уникальному «ключу».



Мультитон обобщает шаблон «Одиночка». В то время как «Одиночка» разрешает создание лишь одного экземпляра класса, мультитон позволяет создавать несколько экземпляров, которые управляются через ассоциативный массив. Создаётся лишь один экземпляр для каждого ключа ассоциативного массива, что позволяет контролировать уникальность объекта по какому-либо признаку.

6.7. Прототип (Prototype)

Прототип — это порождающий паттерн проектирования, который позволяет копировать объекты, не вдаваясь в подробности их реализации.

Проблема

Допустим, есть объект, который надо скопировать. Чтобы это сделать, надо создать пустой объект такого же класса, а затем поочерёдно скопировать значения всех полей из старого объекта в новый. Но часть его состояния может быть приватной, а значит — недоступной для остального кода программы, вследствие чего не каждый объект получится скопировать.

Помимо этого, копирующий код станет зависеть от классов копируемых объектов. Чтобы перебрать все поля объекта, надо привязаться к его классу. Из-за этого не получится копировать объекты, зная только их интерфейсы, а не конкретные классы.

Применимость

Паттерн «Прототип» используется, когда:

- код не зависит от классов копируемых объектов;
- есть большое число подклассов, которые отличаются начальными значениями полей.

Преимущества и недостатки

Преимущества	Недостатки
1. Позволяет клонировать объекты, не привязываясь к их	1. Сложно клонировать составные объекты, имеющие



конкретным классам. 2. Меньше повторяющегося кода инициализации объектов. 3. Ускоряет создание объектов. 4. Альтернатива созданию подклассов для конструирования сложных объектов.	ссылки на другие объекты.
---	---------------------------

7. Структурные шаблоны

Эти паттерны решают проблемы эффективного построения связей между объектами. Структурные паттерны отвечают за построение удобных в поддержке иерархий классов. Такие паттерны уровня класса используют наследование для составления композиций из интерфейсов и реализаций. Вместо композиции интерфейсов или реализаций, структурные паттерны уровня объекта komponуют объекты для получения новой функциональности.

Дополнительная гибкость в этом случае связана с возможностью изменить композицию объектов во время выполнения, что недопустимо для статической композиции классов.

К **структурным паттернам** относятся:

1. Адаптер (Adapter).
2. Мост (Bridge).
3. Компоновщик (Composite).
4. Декоратор (Decorator).
5. Фасад (Facade).
6. Приспособленец (Flyweight).
7. Заместитель (Proxy).

7.1. Адаптер (Adapter)



Адаптер — это структурный паттерн проектирования, который позволяет объектам с несовместимыми интерфейсами работать вместе.

Адаптер преобразует интерфейс, ожидаемый клиентом, одного класса в интерфейс другого. Этот паттерн обеспечивает совместную работу классов с несовместимыми интерфейсами, которая без него была бы невозможна.

Например, летя в первый раз за границу, нас могут удивить стандарты розеток в разных странах. Европейская зарядка будет бесполезна в США без специального адаптера, позволяющего подключиться к розетке другого типа.

Проблема

Допустим, требуется создать приложение для торговли на бирже. Приложение скачивает биржевые котировки из нескольких источников в XML, а затем рисует графики.

В какой-то момент для улучшения приложения принято решение применить стороннюю библиотеку аналитики. Но библиотека использует только формат данных JSON, который не поддерживается приложением.

Изменение библиотеки может повлечь нарушения в работе существующего кода. Не всегда при этом есть доступ к исходному коду библиотеки.

Решение

Для решения проблемы создаётся адаптер — объект-переводчик, который трансформирует интерфейс или данные одного объекта в такой вид, чтобы он стал понятен другому объекту. Адаптер оборачивает один из объектов так, что другой объект даже не знает о наличии первого.

Адаптеры не только переводят данные из одного формата в другой, но и позволяют объектам с разными интерфейсами работать сообща.

Например, 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.

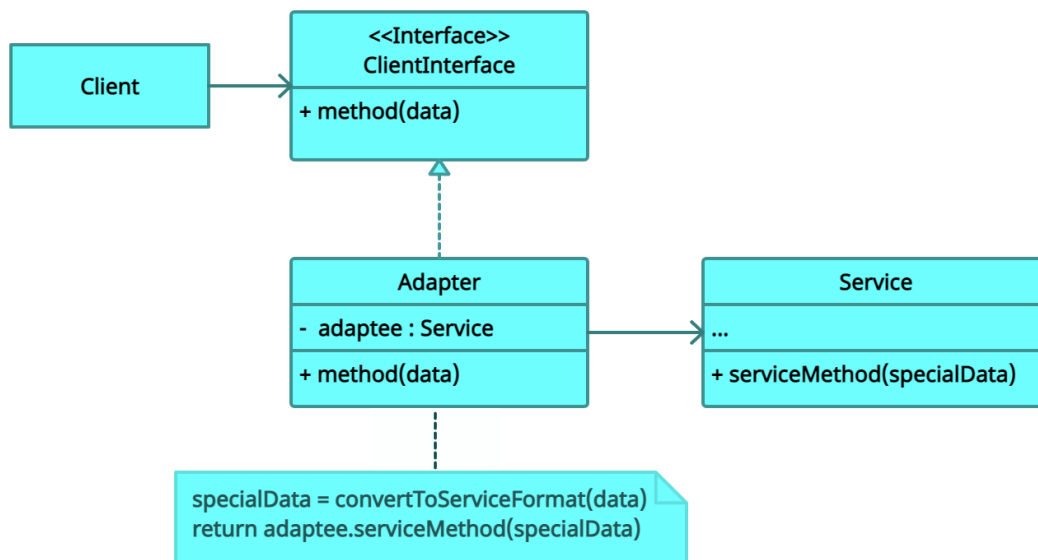
Иногда создаётся двухсторонний адаптер для работы в обе стороны.

Таким образом, в приложении биржевых котировок можно создать класс, который бы оборачивал объект того или иного класса библиотеки аналитики.



Код посылал бы адаптеру запросы в формате XML, а адаптер сначала транслировал входящие данные в формат JSON, а затем передавал их методам обёрнутого объекта аналитики.

Структура



Клиент (Client) — это класс, который содержит действующую бизнес-логику программы.

Клиентский интерфейс (ClientInterface) описывает протокол, через который клиент может работать с другими классами.

Сервис (Service) — полезный сторонний класс. Клиент не может использовать этот класс напрямую, так как сервис имеет непонятный ему интерфейс.

Адаптер (Adapter) — это класс, который одновременно работает и с клиентом, и с сервисом. Он реализует клиентский интерфейс и содержит ссылку на объект сервиса. Адаптер получает вызовы от клиента через методы клиентского интерфейса, а затем переводит их в вызовы методов обёрнутого объекта в правильном формате.

Применимость

Применять паттерн «Адаптер» следует, когда:

1. Хотим использовать действующий класс, но его интерфейс не соответствует нашим потребностям.



2. Собираемся создать повторно используемый класс, который должен взаимодействовать с заранее неизвестными или не связанными с ним классами, имеющими несовместимые интерфейсы.
3. **Для адаптера объектов!** Надо использовать несколько действующих подклассов, но непрактично адаптировать их интерфейсы путём порождения новых подклассов от каждого. В этом случае адаптер объектов приспособливает интерфейс их общего родительского класса.

Преимущества и недостатки

Преимущества	Недостатки
1. Отделяет и скрывает от клиента подробности преобразования различных интерфейсов.	1. Усложняет код программы из-за введения дополнительных классов.

7.2. Мост (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.

Проблема

Допустим, требуется создать класс геометрических фигур, который имеет подклассы «Круг» и «Квадрат». Мы хотим расширить иерархию фигур по цвету, то есть иметь красные и синие фигуры. Но чтобы всё это объединить, придётся создать четыре комбинации подклассов: «КрасныеКруги», «СиниеКруги», «КрасныеКвадраты», «СиниеКвадраты».

При добавлении новых видов фигур и цветов количество комбинаций будет расти в геометрической прогрессии. То есть если добавить фигуру треугольник и цвет жёлтый, то комбинаций будет уже девять: «КрасныеКруги», «СиниеКруги», «ЖёлтыеКруги», «КрасныеКвадраты», «СиниеКвадраты», «ЖёлтыеКвадраты», «КрасныеТреугольники», «СиниеТреугольники», «ЖёлтыеТреугольники».

Применимость



Паттерн «Мост» используется в следующих случаях:

1. Мы хотим избежать постоянной привязки абстракции к реализации.
2. Требуется, чтобы и абстракции, и реализации расширились новыми подклассами. В таком случае паттерн «Мост» позволяет комбинировать разные абстракции и реализации и изменять их независимо.
3. Важно, чтобы изменения в реализации абстракции не сказывались на клиентах, и клиентский код не перекомпилировался.
4. **Только для C++!** Если мы хотим полностью скрыть от клиентов реализацию абстракции, в C++ представление класса демонстрируется через его интерфейс.
5. Число классов начинает быстро расти. Это признак того, что иерархию следует разделить на две части.
6. Мы хотим разделить одну реализацию между несколькими объектами, и этот факт требуется скрыть от клиента.

Преимущества и недостатки

Преимущества	Недостатки
<ol style="list-style-type: none">1. Позволяет строить платформно-независимые программы.2. Скрывает лишние или опасные детали реализации от клиентского кода.3. Реализует принцип открытости/закрытости.	<ol style="list-style-type: none">1. Усложняет код программы из-за введения дополнительных классов.

7.3. Компоновщик (Composite)

Компоновщик — это структурный паттерн проектирования, который позволяет группировать множество объектов в древовидную структуру, а затем работать с ней так, будто это единичный объект.



Проблема

Паттерн «Компоновщик» имеет смысл только тогда, когда основная модель программы структурируется в виде дерева.

Например, есть два объекта: продукт и коробка. Коробка может содержать несколько продуктов и других коробок поменьше. Те, в свою очередь, тоже содержат либо продукты, либо коробки и так далее.

Теперь предположим, что наши продукты и коробки — часть заказов. Каждый заказ содержит как простые продукты без упаковки, так и составные. Наша задача — узнать цену всего заказа.

Если выполнять задачу прямо, то потребуется открыть коробки заказа, перебрать продукты и посчитать их общую стоимость. Но это слишком хлопотно, так как не все типы коробок и их содержимое известны. Мы также не знаем количество уровней вложенности коробок, поэтому перебрать коробки простым циклом не выйдет.

Применимость

Паттерн «Компоновщик» используется в следующих случаях:

1. Требуется представить иерархию объектов вида «часть-целое».
2. Мы хотим, чтобы клиенты единообразно трактовали составные и индивидуальные объекты.

Преимущества и недостатки

Преимущества	Недостатки
<ol style="list-style-type: none">1. Упрощает архитектуру клиента при работе со сложным деревом компонентов.2. Облегчает добавление новых видов компонентов.	<ol style="list-style-type: none">1. Создаёт слишком общий дизайн классов.

7.4. Декоратор (Decorator)



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

Проблема

Представим, что мы работаем над библиотекой оповещений, которая подключается к разным программам, чтобы получать уведомления о важных событиях.

Основа библиотеки — класс `Notifier` с методом `send`, который принимает на вход строку-сообщение и отправляет её всем администраторам по электронной почте. Задача сторонней программы — создать и настроить этот объект, указав, кому отправлять оповещения, а затем использовать его каждый раз, когда что-то случается.

В какой-то момент стало понятно, что одних email-оповещений пользователям мало. Некоторые из них хотели бы получать извещения о критических проблемах через SMS, другие — в виде сообщений и т. д.

Сначала мы добавили каждый тип оповещений в программу, унаследовав их от базового класса `Notifier`. Теперь пользователь выбирал один из типов оповещений, который и использовался в дальнейшем.



Но почему нельзя выбрать несколько типов оповещений сразу?

Если случилось что-то важное, лучше получить оповещения по всем каналам.

Попытка реализовать все возможные комбинации приводит к сильному увеличению кода.

Можно было бы применить наследование. Но механизм наследования имеет несколько проблем.

1. Он статичен. Нельзя изменить поведение действующего объекта. Для этого надо создать новый объект, выбрав другой подкласс.
2. Он не разрешает наследовать поведение нескольких классов одновременно. Из-за этого приходится создавать множество подклассов-комбинаций для получения совмещённого поведения.



Декоратор имеет альтернативное название — обёртка. Оно более точно описывает суть паттерна: мы помещаем целевой объект в другой объект-обёртку, который запускает базовое поведение объекта, а затем добавляет к результату что-то своё.

Применимость

Паттерн «Декоратор» следует использовать:

1. В случае динамического, прозрачного для клиентов добавления обязанностей объектам.
2. Для реализации обязанностей, которые могут быть сняты с объекта.
3. Когда расширение путём порождения подклассов по каким-то причинам неудобно или невозможно. Иногда приходится реализовывать много независимых расширений, так что порождение подклассов для поддержки всех возможных комбинаций приведёт к комбинаторному росту их числа. В других случаях определение класса может быть скрыто или почему-либо ещё недоступно, из-за чего породить от него подкласс нельзя.

Преимущества и недостатки

Преимущества	Недостатки
<ol style="list-style-type: none">1. Большая гибкость, чем у наследования.2. Можно добавлять несколько новых обязанностей сразу.3. Позволяет иметь несколько мелких объектов вместо одного объекта для всего.	<ol style="list-style-type: none">1. Трудно конфигурировать многократно обёрнутые объекты.2. Обилие маленьких классов.

7.5. Фасад (Facade)

Фасад — это структурный паттерн проектирования, который предоставляет простой интерфейс к сложной системе классов, библиотеке или фреймворку.



Проблема

Нашему коду приходится работать с множеством объектов некой сложной библиотеки или фреймворка. Мы должны самостоятельно инициализировать эти объекты, следить за правильным порядком зависимостей и так далее.

В результате бизнес-логика классов тесно переплетается с деталями реализации сторонних классов. Такой код сложно понимать и поддерживать.

Применимость

Паттерн «Фасад» используется в следующих случаях:

1. Мы хотим предоставить простой интерфейс к сложной подсистеме. Часто подсистемы усложняются по мере развития. Применение большинства паттернов приводит к появлению меньших классов, но в большей численности.

Такую подсистему проще повторно использовать и настраивать под конкретные нужды, но вместе с тем применять подсистему без настройки становится труднее.

Фасад предлагает некоторый вид системы по умолчанию, устраивающий большинство клиентов. И лишь те объекты, которым требуются более широкие возможности настройки, могут обратиться напрямую к тому, что находится за фасадом.

2. Между клиентами и классами реализации абстракции есть много зависимостей. Фасад позволит отделить подсистему как от клиентов, так и от других подсистем, что, в свою очередь, способствует повышению степени независимости и переносимости.
3. Мы хотим разложить подсистему на отдельные слои. Используем фасад для определения точки входа на каждый уровень подсистемы. Если подсистемы зависят друг от друга, то зависимость можно упростить, разрешив подсистемам обмениваться информацией только через фасады.

Преимущества и недостатки

Преимущества	Недостатки
--------------	------------



1. Изолирует клиентов от компонентов сложной подсистемы.	1. Фасад рискует стать антипаттерном, привязанным ко всем классам программы.
--	--

7.6. Приспособленец (Flyweight)

Приспособленец — это структурный паттерн проектирования, который позволяет вместить больше объектов в отведённую оперативную память. Этот паттерн экономит память, разделяя общее состояние объектов между собой, вместо хранения одинаковых данных в каждом объекте.

Проблема

Представим, что мы решили написать небольшую игру, в которой игроки перемещаются по карте и стреляют друг в друга. Важная составляющая игры — реалистичная система частиц. Пули, снаряды, осколки от взрывов должны красиво летать.

Игра отлично работала на мощном компьютере. Однако на другом компьютере игра начинает тормозить и вылетает через несколько минут после запуска. Мы обнаружили, что игра вылетает из-за недостатка оперативной памяти. Другой компьютер значительно менее «прокачанный», поэтому проблема у него проявляется быстрее.

Каждая частица представлена собственным объектом, имеющим множество данных. В определённый момент, когда побоище на экране достигает кульминации, новые объекты частиц уже не помещаются в оперативную память компьютера, и программа вылетает.

Применимость

Паттерн «Приспособленец» применяется в следующих случаях:

1. В приложении используется большое число объектов, из-за этого накладные расходы на хранение высоки.
2. Большая часть состояния объектов выносится вовне.
3. Многие группы объектов заменяются относительно небольшим числом разделяемых объектов, поскольку внешнее состояние вынесено.



4. Приложение не зависит от идентичности объекта. Поскольку объекты-приспособленцы могут разделяться, то проверка на идентичность возвратит «истину» для концептуально различных объектов.

Преимущества и недостатки

Преимущества	Недостатки
1. Экономит оперативную память.	1. Расходует процессорное время на поиск/вычисление контекста. 2. Усложняет код программы из-за введения множества дополнительных классов.

7.7. Заместитель (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.

Проблема

Представим, у нас есть внешний ресурсоёмкий объект, который требуется не всё время, а изредка. Мы могли бы создавать этот объект не в самом начале программы, а только когда он кому-то реально понадобится. Каждый клиент объекта получил бы некий код отложенной инициализации. Но это привело бы к множественному дублированию кода.

В идеале этот код требуется поместить прямо в служебный класс, но это не всегда возможно. Например, код класса может находиться в закрытой сторонней библиотеке.

Применимость

Паттерн «Заместитель» применяется в следующих случаях:



1. Защищающий заместитель. Когда в программе есть разные типы пользователей, и нам требуется защищать объект от неавторизованного доступа.
2. Удалённый заместитель. Когда настоящий сервисный объект находится на удалённом сервере.
3. Логирующий заместитель. Когда настоящий сервисный объект находится на удалённом сервере.
4. «Умная» ссылка. Когда требуется кешировать результаты запросов клиентов и управлять их жизненным циклом.

Преимущества и недостатки

Преимущества	Недостатки
<ol style="list-style-type: none">1. Виртуальный заместитель выполняет оптимизацию.2. Повышает быстродействие и безопасность кода.	<ol style="list-style-type: none">1. Усложняет код программы из-за введения дополнительных классов.2. Увеличивает время отклика.

8. Поведенческие шаблоны

Эти паттерны решают проблемы эффективного взаимодействия между объектами. Паттерны поведения связаны с алгоритмами и распределением обязанностей между объектами. Речь в них идёт не только о самих объектах и классах, но и о типичных схемах взаимодействия между ними.

Эти паттерны характеризуют сложный поток управления, который трудно проследить во время выполнения программы. Внимание акцентируется не на схеме управления как таковой, а на связях между объектами. Поведенческие паттерны выполняют задачи эффективного и безопасного взаимодействия между объектами программы.

Эти шаблоны оказываются важны, когда система больше зависит от композиции объектов, чем от наследования классов. Получается так, что основной упор делается не на жёстком кодировании фиксированного набора поведений, а на определении небольшого набора фундаментальных поведений, через композицию которых можно получать любое число более



сложных. Таким образом, для создания объектов с конкретным поведением требуется нечто большее, чем простое инстанцирование класса.

К поведенческим паттернам относятся:

1. Цепочка ответственности (Chain of responsibility).
2. Команда (Command).
3. Итератор (Iterator).
4. Посредник (Mediator).
5. Хранитель (Memento).
6. Наблюдатель (Observer).
7. Состояние (State).
8. Стратегия (Strategy).
9. Шаблонный метод (Template method).
- 10.Посетитель (Visitor).

8.1. Цепочка обязанностей (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.

Проблема

Представим, что мы делаем систему приёма онлайн-заказов. Нам надо ограничить доступ к ней так, чтобы только авторизованные пользователи могли создавать заказы. Определённые пользователи, обладающие правами администратора, должны иметь полный доступ к заказам.

Эти проверки требуется выполнять последовательно. Ведь пользователя можно попытаться «залогинить» в систему, если его запрос содержит логин и пароль. Но если такая попытка не удалась, то проверять расширенные права доступа попросту не имеет смысла.

Допустим, впоследствии добавилось ещё несколько проверок.



С каждым новым правилом код проверок, содержащий несколько условных операторов, всё больше и больше увеличивался. При изменении одного правила приходилось трогать код всех проверок. А чтобы применить проверки к другим ресурсам, пришлось продублировать их код в других классах.

Применимость

Варианты применения цепочки обязанностей:

1. Запрос обрабатывается более, чем одним объектом, причём настоящий обработчик заранее неизвестен и находится автоматически.
2. Мы хотим отправить запрос одному из нескольких объектов, не указывая явно, какому именно.
3. Набор объектов, способных обработать запрос, задаётся динамически.

Преимущества и недостатки

Преимущества	Недостатки
<ol style="list-style-type: none">1. Уменьшает зависимость между клиентом и обработчиками.2. Реализует принцип единственной ответственности.3. Реализует принцип открытости/закрытости.	<ol style="list-style-type: none">1. Запрос может остаться никем не обработанным.

8.2. Команда (Command)

Команда — это 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.

Проблема

Представим, что мы работаем над программой текстового редактора. Дело как раз подошло к разработке панели управления. Мы создали класс красивых



кнопок и хотим использовать его для всех кнопок приложения, начиная от панели управления, заканчивая простыми кнопками в диалогах.

Все эти кнопки похожи, но делают разные вещи. Самым простым решением было бы создать подклассы для каждой кнопки и переопределить в них метод действия под разные задачи.

Но, во-первых, получается очень много подклассов. Во-вторых, код кнопок, относящийся к графическому интерфейсу, начинает зависеть от классов бизнес-логики, которая довольно часто меняется.

Некоторые операции, например, «Сохранить», вызываются из нескольких мест: нажав кнопку на панели управления, вызвав контекстное меню или просто нажав клавиши Ctrl+S. Когда в программе были только кнопки, код сохранения был лишь в подклассе SaveButton. Но теперь его придётся продублировать ещё в два класса.

Решение

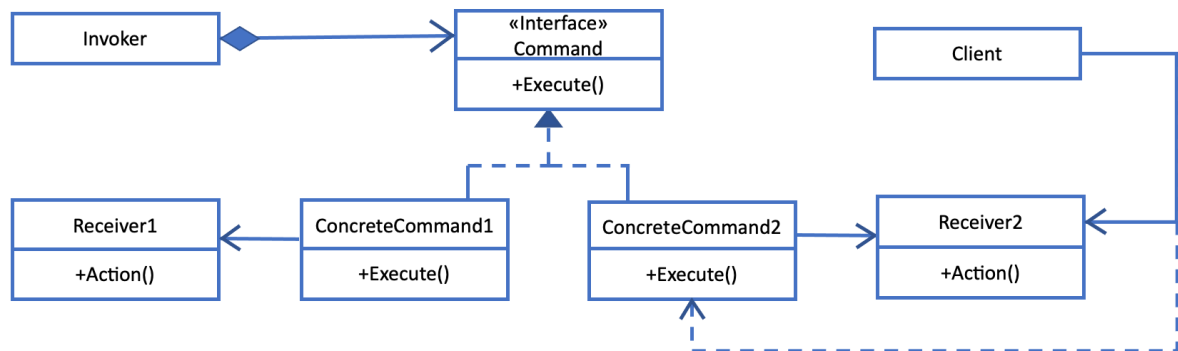
Паттерн «Команда» предлагает больше не отправлять вызовы напрямую. Вместо этого, каждый вызов, отличающийся от других, следует завернуть в собственный класс с единственным методом, который и осуществит вызов. Такие объекты называются командами.

К объекту интерфейса можно будет привязать объект команды, который знает, кому и в каком виде следует отправлять запросы. Чтобы передать запрос, объект интерфейса вызовет метод команды, а та — позаботится обо всём остальном.

Используя общий интерфейс команд, объекты кнопок будут ссылаться на объекты команд различных типов. При нажатии кнопки станут делегировать работу связанным командам, а команды — перенаправлять вызовы тем или иным объектам бизнес-логики.

То же с контекстным меню и с горячими клавишами. Они прикрепятся к тем же объектам команд, что и кнопки, избавляя классы от дублирования.

Таким образом, команды станут гибкой прослойкой между пользовательским интерфейсом и бизнес-логикой.



Команда (Command) объявляет интерфейс для выполнения операции.

Конкретная команда (ConcreteCommand) определяет связь между объектом-получателем **Receiver** и действием, реализует операцию **Execute** путём вызова соответствующих операций объекта **Receiver**.

Инициатор (Invoker) обращается к команде для выполнения запроса.

Клиент (Client) создаёт объект класса **ConcreteCommand** и устанавливает его получателя.

Получатель (Receiver) располагает информацией о способах выполнения операций, требуемых для удовлетворения запроса. В роли получателя может выступить любой класс.

Применимость

Паттерн «Команда» используется:

1. Мы хотим параметризовать объекты выполняемым действием, как в случае с пунктами меню **MenuItem**. В процедурном языке такая параметризация выражается через функцию обратного вызова, то есть такой функции, которая регистрируется, чтобы вызываться позднее. Команды представляют собой объектно-ориентированную альтернативу функциям обратного вызова.
2. Требуется определять, ставить в очередь и выполнять запросы в разное время. Время жизни объекта **Command** необязательно должно зависеть от времени жизни исходного запроса. Если получателя запроса удастся реализовать так, чтобы он не зависел от адресного пространства, то объект-команду можно передать другому процессу, который займётся его выполнением.



3. Надо поддерживать отмену операций. Операция Execute объекта Command сохраняет состояние, требуемое для отката действий, выполненных командой. В этом случае в интерфейсе класса Command должна быть дополнительная операция Unexecute, которая отменяет действия, выполненные предшествующим обращением к Execute. Выполненные команды хранятся в списке истории.

Для реализации произвольного числа уровней отмены и повтора команд надо обходить этот список в обратном и прямом направлениях, вызывая при посещении каждого элемента команду Unexecute или Execute.

4. Требуется поддерживать протоколирование изменений, чтобы выполнять их повторно после аварийной остановки системы. Дополнив интерфейс класса Command операциями сохранения и загрузки, появляется возможность вести протокол изменений во внешней памяти.

Для восстановления после сбоя следует загрузить сохранённые команды с диска и повторно выполнить их через операцию Execute.

5. Мы хотим структурировать систему на основе высокоуровневых операций, построенных из примитивных. Такая структура типична для информационных систем, поддерживающих транзакции. Транзакция инкапсулирует набор изменений данных. Паттерн «Команда» позволяет моделировать транзакции.

У всех команд есть общий интерфейс, что даёт возможность работать одинаково с любыми транзакциями. Через этот паттерн легко добавлять в систему новые виды транзакций.

Преимущества и недостатки

Преимущества	Недостатки
<ol style="list-style-type: none">1. Убирает прямую зависимость между объектами, вызывающими операции, и объектами, которые их выполняют прямо.2. Позволяет реализовать простую отмену и повтор операций.3. Позволяет реализовать	<ol style="list-style-type: none">1. Усложняет код программы из-за введения множества дополнительных классов.



отложенный запуск операций.	
4. Позволяет собирать сложные команды из простых.	
5. Реализует принцип открытости/закрытости.	

8.3. Итератор (Iterator)

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

Проблема

Коллекции — самая распространённая структура данных, встречающаяся в программировании. Это набор объектов, собранный в одну кучу по каким-то критериям.

Большинство коллекций выглядит как обычный список элементов. Но есть и экзотические коллекции, построенные на основе деревьев, графов и других сложных структур данных.

Но как бы ни была структурирована коллекция, пользователь должен иметь возможность последовательно обходить её элементы, чтобы проделывать с ними какие-то действия.

Иногда требуется обходить список по-разному в зависимости от задачи. Например, дерево можно обходить как в глубину, так и в ширину или в случайном порядке. Но вряд ли мы захотим засорять интерфейс класса `Tree` операциями для различных вариантов обхода, даже если все их можно предусмотреть заранее. Порой надо, чтобы в один и тот же момент действовало несколько активных обходов списка.

Таким образом, добавляя всё новые алгоритмы в код коллекции, мы понемногу «размываем» её основную задачу, которая заключается в эффективном хранении данных. Некоторые алгоритмы и вовсе слишком «заточены» под определённое приложение и смотрятся странно в общем классе коллекции.



Применимость

Паттерн «Итератор» используется:

1. Для доступа к содержимому агрегированных объектов без раскрытия их внутреннего представления.
2. Для поддержки нескольких активных обходов одного и того же агрегированного объекта.
3. Для предоставления единообразного интерфейса, чтобы обойти различные агрегированные структуры, то есть для поддержки полиморфной итерации.

Преимущества и недостатки

Преимущества	Недостатки
<ol style="list-style-type: none">1. Упрощает классы хранения данных.2. Позволяет реализовать различные способы обхода структуры данных.3. Позволяет одновременно перемещаться по структуре данных в разные стороны.	<ol style="list-style-type: none">1. Неоправдан, если можно обойтись простым циклом.

8.4. Посредник (Mediator)

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

Проблема

Предположим, что у нас есть диалог создания профиля пользователя. Он состоит из всевозможных элементов управления — текстовых полей, чекбоксов, кнопок.



Задача отдельных элементов диалога — взаимодействовать друг с другом. Например, чекбокс открывает скрытое поле для ввода, а клик по кнопке отправки запускает проверку значений всех полей формы.

Прописав эту логику прямо в коде элементов управления, мы не сможем повторно использовать их в других местах приложения. Они станут слишком тесно связанными с элементами диалога редактирования профиля, которые не нужны в других контекстах. Поэтому мы сможем использовать либо все элементы сразу, либо не одного.

Применимость

Паттерн «Посредник» следует использовать, когда:

1. Есть объекты, связи между которыми сложны и чётко определены. Получающиеся при этом взаимозависимости не структурируются и трудны для понимания.
2. Нельзя повторно использовать объект, поскольку он обменивается информацией со многими другими объектами.
3. Поведение, распределённое между несколькими классами, должно поддаваться настройке без порождения множества подклассов.

Преимущества и недостатки

Преимущества	Недостатки
<ol style="list-style-type: none">1. Устраняет зависимости между компонентами, позволяя использовать их повторно.2. Упрощает взаимодействие между компонентами.3. Централизует управление в одном месте.	<ol style="list-style-type: none">1. Посредник может сильно увеличить код.

8.5. Хранитель (Memento)



Хранитель (снимок) — это поведенческий паттерн проектирования, который позволяет сохранять и восстанавливать прошлые состояния объектов, не раскрывая подробностей их реализации.

Проблема

Предположим, что мы пишем программу текстового редактора. Помимо обычного редактирования, наш редактор позволяет менять форматирование текста, вставлять картинки и прочее.

В какой-то момент мы решили сделать все эти действия отменяемыми. Для этого требуется сохранять текущее состояние редактора перед тем, как выполнить какое-либо действие. Если потом пользователь решит отменить своё действие, мы достанем копию состояния из истории и восстановим старое состояние редактора.

Чтобы сделать копию состояния объекта, достаточно скопировать значение его полей. Таким образом, если мы сделали класс редактора достаточно открытым, то любой другой класс сможет скопировать его состояние. Но если решено провести рефакторинг — убрать или добавить парочку полей в класс редактора — придётся менять код всех классов, которые способны копировать состояние редактора.

Получается, нам придётся либо открыть классы для всех желающих, испытывая массу хлопот с поддержкой кода, либо оставить классы закрытыми, отказавшись от идеи отмены операций.

Применимость

Паттерн «Хранитель» следует использовать, когда:

1. Требуется сохранить мгновенный снимок состояния объекта (или его части), чтобы впоследствии восстановить объект в том же состоянии.
2. Прямое получение этого состояния раскрывает детали реализации и нарушает инкапсуляцию объекта.

Преимущества и недостатки

Преимущества	Недостатки
1. Не нарушает инкапсуляции	1. Требует много памяти, если клиенты слишком часто создают



исходного объекта. 2. Упрощает структуру исходного объекта. Ему не нужно хранить историю версий своего состояния.	снимки. 2. Может повлечь дополнительные издержки памяти, если объекты, хранящие историю, не освобождают ресурсы, занятые устаревшими снимками. 3. В некоторых языках, например, PHP, Python, JavaScript, сложно гарантировать, чтобы только исходный объект имел доступ к состоянию снимка.
--	---

8.6. Null Object

Null Object — это объект с определённым нейтральным (null) поведением. Шаблон проектирования Null Object описывает использование таких объектов и их поведение (или отсутствие такового).

Проблема

В некоторых языках объекты имеют значение NULL. Ссылки на такие объекты нуждаются в проверке на NULL-значение перед использованием, так как методы класса «нулевого» объекта, как правило, не вызываются.

Рассмотрим, например, простую заставку, отображающая шары, которые перемещаются по экрану и имеют специальные цветовые эффекты. Это легко достигается путём создания класса «Шар», чтобы представлять шары и использовать:

- шаблон стратегии для управления движением шара;
- и другой шаблон стратегии для управления цветом шара.

Тогда было бы тривиально писать стратегии для многих различных типов движений и цветовых эффектов, а также создавать шары с любой их комбинацией. Однако для начала нам надо создать самые простые стратегии, чтобы убедиться, что всё работает.



В этом случае самая простая стратегия — это ничего не делать, не двигаться и не менять цвет. Однако паттерн стратегии требует, чтобы у шара были объекты, которые реализуют интерфейсы стратегии.

Применимость

Этот шаблон проектирования рекомендуется использовать, когда:

1. Объект требует взаимодействия с другими объектами. Null Object не устанавливает нового взаимодействия — он использует уже установленное взаимодействие.
2. Какие-то из взаимодействующих объектов должны бездействовать.
3. Требуется абстрагирование «общения» с объектами, имеющими NULL-значение.

Преимущества и недостатки

Преимущества	Недостатки
<ol style="list-style-type: none">1. Отказ от условных операторов.2. Уменьшает шанс исключений из-за нулевых указателей.	<ol style="list-style-type: none">1. Ещё один новый класс.

8.7. Наблюдатель (Observer)

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

Проблема

Представим, что мы имеем два объекта: покупатель и магазин. В магазин должны завезти новый товар, который интересен покупателю.

Покупатель каждый день ходит в магазин, чтобы проверить наличие товара. Но при этом он злится, без толку тратя своё драгоценное время.



С другой стороны, магазин рассылает спам каждому своему покупателю. Многих это расстроит, так как товар специфический и не всем нужен.

Получается конфликт: либо покупатель тратит время на периодические проверки, либо магазин тратит ресурсы на бесполезные оповещения.

Применимость

Паттерн «Наблюдатель» используется, когда:

1. У абстракции есть два аспекта, один из которых зависит от другого. Инкапсуляции этих аспектов в разные объекты позволяют изменять и повторно использовать их независимо.
2. При модификации одного объекта требуется изменить другие, и неизвестно, сколько именно объектов надо изменить.
3. Один объект должен оповещать других, не делая предположений об уведомляемых объектах. Другими словами, мы не хотим, чтобы объекты были тесно связаны между собой.

Преимущества и недостатки

Преимущества	Недостатки
<ol style="list-style-type: none">1. Издатели не зависят от конкретных классов подписчиков и наоборот.2. Мы можем подписывать и отписывать получателей на лету.3. Реализует принцип открытости/закрытости.	<ol style="list-style-type: none">1. Подписчики оповещаются в случайном порядке.

8.8. Состояние (State)

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



Проблема

Программа может находиться в одном из нескольких состояний, которые всё время сменяют друг друга. Набор этих состояний, а также переходов между ними, предопределён и конечен. Находясь в разных состояниях, программа может по-разному реагировать на одни и те же события, которые с ней происходят.

Такой подход применяется и к отдельным объектам. Например, объект «Документ» принимает три состояния: «Черновик», «Модерация» или «Опубликован». В каждом из этих состояний метод «Опубликовать» будет работать по-разному.

Если в «Документ» добавить ещё десяток состояний, то каждый метод будет состоять из увесистого условного оператора, перебирающего доступные состояния. Такой код крайне сложно поддерживать. Малейшее изменение логики переходов заставит перепроверять работу всех методов, которые содержат условные операторы.

Применимость

Паттерн «Состояние» используется, когда:

1. Поведение объекта зависит от его состояния и изменяется во время выполнения.
2. В коде операций встречаются состоящие из многих ветвей условные операторы, в которых выбор ветви зависит от состояния. Обычно в таком случае состояние представляется 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 изменяется независимо от других.

Преимущества и недостатки

Преимущества	Недостатки
1. Избавляет от больших условных операторов машины состояний.	1. Может неоправданно усложнить код, если состояний мало и они



2. Концентрирует в одном месте код, связанный с определённым состоянием.	редко меняются.
3. Упрощает код контекста.	

8.9. Стратегия (Strategy)

Стратегия — это поведенческий паттерн проектирования, который определяет семейство похожих алгоритмов и помещает каждый из них в собственный класс, после чего алгоритмы взаимозаменяются прямо во время исполнения программы.

Проблема

Представим приложение, которое демонстрирует красивую и удобную карту, позволяющую с лёгкостью ориентироваться в незнакомом городе.

Одна из самых востребованных функций — поиск и прокладывание маршрутов. Пребывая в неизвестном городе, пользователь должен иметь возможность указать начальную точку и пункт назначения, а навигатор проложит оптимальный путь.

Первая версия навигатора могла прокладывать маршрут лишь по дорогам, поэтому отлично подходила для путешествий на автомобиле. Потом добавился навигатор для прокладывания пеших маршрутов. Через некоторое время появилась опция прокладывания пути на общественном транспорте. Есть возможность добавить прокладывание маршрутов по велодорожкам, интересные маршруты посещения достопримечательностей и другое.

С каждым алгоритмом основной класс увеличивается, что делает код очень запутанным.

Применимость

Паттерн «Стратегия» используется, когда:



1. Есть много родственных классов, отличающихся только поведением. Стратегия позволяет сконфигурировать класс, задав одно из возможных поведений.
2. Мы хотим иметь несколько разных вариантов алгоритма. Например, можно определить два варианта алгоритма, один из которых требует больше времени, а другой — больше памяти. Стратегии разрешается применять, когда варианты алгоритмов реализованы в виде иерархии классов.
3. В алгоритме содержатся данные, о которых клиент не должен «знать». Следует использовать паттерн «Стратегия», чтобы не раскрывать сложные, специфичные для алгоритма, структуры данных.
4. В классе определено много поведений, что представлено разветвлёнными условными операторами. В этом случае проще перенести код из ветвей в отдельные классы стратегий.

Преимущества и недостатки

Преимущества	Недостатки
<ol style="list-style-type: none">1. Изолирует код и данные алгоритмов от остальных классов.2. Уход от наследования к делегированию.3. Реализует принцип открытости/закрытости.	<ol style="list-style-type: none">1. Усложняет программу из-за дополнительных классов.2. Клиент должен знать, в чём разница между стратегиями, чтобы выбрать подходящую.

8.10. Шаблонный метод (Template method)

Шаблонный метод — это поведенческий паттерн проектирования, который определяет скелет алгоритма, перекидывая ответственность за некоторые его шаги на подклассы. Паттерн позволяет подклассам переопределять шаги алгоритма, не меняя его общей структуры.



Проблема

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

В первой версии доступна только обработка DOC-файлов. В следующей версии добавилась поддержка CSV. А через месяц появилась работа с PDF-документами.

Код для каждого класса имеет много общего. Остальной код, работающий с объектами этих классов, постоянно проверяет тип обработчика. Было бы здорово избавиться от повторной реализации алгоритма извлечения данных в каждом классе.

Применимость

Паттерн «Шаблонный метод» следует применять:

1. Чтобы однократно использовать инвариантные части алгоритма, оставляя реализацию изменяющегося поведения на усмотрение подклассов.
2. Когда надо вычлениить и локализовать в одном классе поведение, общее для всех подклассов, дабы избежать дублирования кода. Это хороший пример техники «вынесения за скобки для обобщения», описанной в работе Уильяма Опдайка и Ральфа Джонсона.

Сначала идентифицируются различия в действующем коде, а затем они выносятся в отдельные операции. В результате различающиеся фрагменты кода заменяются шаблонным методом, из которого вызываются новые операции.

3. Для управления расширениями подклассов. Можно определить шаблонный метод так, что он будет вызывать операции-зацепки (hooks).

Преимущества и недостатки

Преимущества	Недостатки
1. Облегчает повторное использование кода.	1. Мы жёстко ограничены скелетом действующего алгоритма. 2. Мы можем нарушить принцип



	<p>подстановки Барбары Лисков, изменяя базовое поведение одного из шагов алгоритма через подкласс.</p> <p>3. С ростом количества шагов шаблонный метод становится слишком сложно поддерживать.</p>
--	--

8.11. Посетитель (Visitor)

Посетитель — это поведенческий паттерн проектирования, который позволяет добавлять в программу новые операции, не изменяя классы объектов, над которыми эти операции выполняются.

Проблема

Представим, наша команда разрабатывает приложение, работающее с геоданными в виде графа. Узлы графа — городские локации: памятники, театры, рестораны, важные предприятия и прочее. Каждый узел имеет ссылки на другие, ближайшие к нему узлы. Каждому типу узлов соответствует свой класс, а каждый узел представлен отдельным объектом.

Наша задача — сделать экспорт этого графа в XML.

Если бы мы могли редактировать классы узлов, достаточно было бы добавить метод экспорта в каждый тип узла. Затем, перебирая узлы графа — вызывать этот метод для каждого узла.

К сожалению, классы узлов изменить не удалось. Экспорт в XML неуместен в рамках этих классов. Их основная задача была связана с геоданными, а экспорт выглядит в рамках этих классов чужеродно. Если на следующей неделе понадобился бы экспорт в какой-то другой формат данных, эти классы снова пришлось бы менять.

Применимость



Паттерн «Посетитель» используется, когда:

1. В структуре есть объекты многих классов с различными интерфейсами, и мы хотим выполнять над ними операции, зависящие от конкретных классов.
2. Над объектами, входящими в состав структуры, надо выполнять разнообразные, не связанные между собой операции. Мы не хотим «засорять» классы такими операциями. Посетитель позволяет объединить родственные операции, поместив их в один класс. Если структура объектов — общая для нескольких приложений, то паттерн «Посетитель» позволит в каждое приложение включить только относящиеся к нему операции.
3. Классы, устанавливающие структуру объектов, изменяются редко, но новые операции над этой структурой добавляются часто. При изменении классов, представленных в структуре, надо переопределить интерфейсы всех посетителей, что может вызвать затруднения. Поэтому если классы меняются достаточно часто, то, вероятно, лучше определить операции прямо в них.

Преимущества и недостатки

Преимущества	Недостатки
<ol style="list-style-type: none">1. Упрощает добавление операций, работающих со сложными структурами объектов.2. Объединяет родственные операции в одном классе.3. Посетитель может накапливать состояние при обходе структуры элементов.	<ol style="list-style-type: none">1. Паттерн неоправдан, если иерархия элементов часто меняется.2. Может привести к нарушению инкапсуляции элементов.

9. Антипаттерны проектирования

Сложности с восприятием паттернов, опасность бездумного применения паттернов. Паттерны — инструмент, который надо грамотно применить. Он



подбирается для конкретной задачи. Главное — оценить, какую проблему решает тот или иной паттерн, и для чего он появился. Если подходящих паттернов подобрать не удаётся, лучше отказаться от их использования. В противном случае, пытаясь улучшить реализацию путём использования шаблонов, вполне вероятно вместо пользы нанести серьёзный вред.

Стоит проявить внимательность при реализации того или иного паттерна в соответствии с описанием. Не всегда реализация, которая приводится в литературе, подходит для конкретной ситуации. Нередко приходится использовать различные вырожденные случаи паттернов.

Использовать паттерны, безусловно, надо. За долгие годы они доказали свою эффективность. Однако применение того или иного паттерна должно быть обоснованным и адекватным. Иначе — это реализация «паттернов ради паттернов», которая не несёт в себе никакой пользы. Бездумное применение паттернов внесёт лишь дополнительную сложность в решение.

Антипаттерн — подход к решению задач, считающийся неэффективным. Рассмотрение антипаттерна включает в себя не только неправильное решение проблемы, но и нахождение правильного решения.

Противоположным подходом к решению задач, наиболее эффективными решениями, называется «паттерн». Хорошая практика программирования — избегание антипаттернов. Использование антипаттернов влечёт негативные последствия.

Развитие программирования и увеличение количества создаваемых программ привело к появлению различных однотипных и похожих проблем, возникающих в процессе разработки. Это привело к появлению шаблонов проектирования (design patterns) — изученных и проверенных на практике способов решения однотипных проблем.

Однако даже использование паттернов проектирования, в непредназначенных для применения ситуациях, порождает появление ещё больших проблем, чем до их использования. Ситуация обусловлена тем, что у начинающих программистов, как и у работников любой другой сферы, изначально недостаточно базы знаний и опыта работы. Отсюда появление проблем вытекает из неспособности в сжатые сроки создать программу, что ведёт к неэффективной разработке и некачественно написанному программному коду. Казалось бы, всё нормально, программа работает, но при добавлении какой-либо новой функциональности мы упрёмся в какое-то ограничение.



Использование паттернов проектирования начинается не только с момента продумывания архитектуры программы, но и применяется при последующей поддержке программы на всём этапе её жизненного цикла. Например, в процессе поддержки программы выявляется проблема, допущенная на начальном этапе, и верным решением исправления проблемы будет использование одного из подходящих для этой ситуации паттерна проектирования.

Рассмотрение антипаттерна принято разделять на три типа:

- архитектура (антипаттерны архитектора);
- разработка (антипаттерны разработчика);
- руководство (антипаттерны менеджера).

Обозначим некоторые антипаттерны архитектора и разработчика.

Архитектурные антипаттерны

Архитектурные антипаттерны (architectural antipatterns) — это проблемы, возникающие в результате неправильно принятых решений архитекторов программного обеспечения. Структура программы, взаимосвязь её внутренних компонентов продумана крайне неэффективно, либо вообще не продумана. Отсюда при разработке будет возникать множество вопросов по архитектуре того или иного компонента программы.

Типичные проблемы, связанные с архитектурой системы:

1. Инверсия абстракции (Abstraction inversion) — сокрытие части функциональности от внешнего использования с расчётом на то, что никто не будет его использовать.
2. Неопределённая точка зрения (Ambiguous viewpoint) — представление модели без спецификации её точки рассмотрения.
3. Большой комок грязи (Big ball of mud) — программа с нераспознаваемой структурой.
4. Бензиновая фабрика (Gas factory) — необязательная сложность дизайна.
5. Затычка на ввод данных (Input kludge) — забывчивость в спецификации и выполнении поддержки неверного ввода.



6. Раздувание интерфейса (Interface bloat) — разработка интерфейса очень мощным и очень сложным для реализации.
7. Волшебная кнопка (Magic pushbutton) — выполнение результатов действий пользователя в виде неподходящего интерфейса.
8. Состояние гонки (Race hazard, Race condition) — непредвидение возможности наступления событий в неожиданном порядке.
9. Мышиная возня — создание множества мелких и абстрактных классов для решения одной конкретной задачи более высокого уровня.
10. Сохранение или смерть (Save or die) — сохранение изменений в конфигурации на жёсткий диск только при завершении приложения.

Антипаттерны разработки

Антипаттерны разработки (development antipatterns) — это проблемы, возникающие в результате неправильно принятых решений разработчиков программного обеспечения. Даже если архитектура программы очень хорошо продумана, при её реализации могут возникнуть некоторые вопросы, неправильное решение которых способно привести к некачественно созданной программе.

Рассмотрим антипаттерны разработки. Проблем, с которыми сталкиваются программисты, может быть много:

1. Базовый класс-утилиты (BaseBean) — наследование функциональности из класса, вместо делегирования к нему.
2. Божественный объект (God object) — большое число методов в одном классе.
3. Проблема йо-йо (Yo-yo problem) — размытость сильно связанного кода по иерархии классов, например, кода, выполняемого по порядку.
4. Одиночество (Singletonitis) — неуместное использование паттерна «Одиночка».
5. Приватизация (Privatisation) — сокрытие функциональности в приватной секции (private), затрудняющее его расширение в классах-потомках.
6. Каша из интерфейсов (Interface soup) — объединение нескольких интерфейсов в один по принципу изоляции интерфейсов (Interface segregation).
7. Висящие концы — интерфейс, методы которого бессмысленны и реализуются «пустышками».



Антипаттерны в кодировании

1. Ненужная сложность (Accidental complexity) — ненужная сложность в решении.
2. Действие на расстоянии (Action at a distance) — взаимодействие между широко разделёнными частями программы.
3. Накопить и запустить (Accumulate and fire) — установка параметров подпрограмм в глобальных переменных.
4. Слепая вера (Blind faith) — недостаточная проверка исправления ошибки или правильной работы подпрограммы.
5. Лодочный якорь (Boat anchor) — сохранение более неиспользуемой части системы.
6. Активное ожидание (Busy spin, busy waiting) — потребление ресурсов процессорного времени во время ожидания события посредством постоянно повторяемой проверки, вместо того, чтобы использовать асинхронное программирование.
7. Таинственный код (Cryptic code) — использование аббревиатур вместо понятных имён.
8. Жёсткое кодирование (Hard code) — внедрение предположений об окружении программы в большом числе точек её реализации.
9. Мягкое кодирование (Soft code) — боязнь жёсткого кодирования, приводящая к настраиванию абсолютно всего.
10. Поток лавы (Lava flow) — сохранение лишнего или низкокачественного кода, потому что его удаление слишком дорого или имеет непредсказуемые последствия.
11. Волшебные числа (Magic numbers) — использование числовых констант без объяснения их смысла.
12. Процедурный код (Procedural code) — когда другая парадигма считается более подходящей.
13. Спагетти-код (Spaghetti code) — код, запутанный как макароны.



14. Лазанья-код (Lasagnia code) — использование большого числа уровней абстракции.
15. Равиоли-код (Ravioli code) — объекты склеены между собой как пельмени, что не позволяет делать рефакторинг.
16. Мыльный пузырь (Soap bubble) — объект, инициализированный мусором и притворяющийся, что содержит данные.

Методологические антипаттерны

1. Программирование методом копирования-вставки (Copy and paste programming) — копирование и использование кусков уже имеющегося кода.
2. Дефакторинг (De-Factoring) — уничтожение функциональности и замены её документацией.
3. Золотой молоток (Golden hammer) — уверенность, что любимое решение универсально и его можно использовать везде.
4. Фактор невероятности (Improbability factor) — предположение о невозможности того, что сработает известная ошибка.
5. Преждевременная оптимизация (Premature optimization) — оптимизация кода, когда это ещё не требуется.
6. Программирование методом подбора (Programming by permutation) — подход к программированию небольшими изменениями.
7. Два тоннеля (Two tunnel) — добавление новой функциональности в отдельное приложение вместо расширения имеющегося.
8. Коммит-убийца (Commit assassin) — внесение изменений в систему контроля версий без проверки влияния на другие части программы.
9. Раздувание ПО (Software bloat) — последующие версии системы требовать всё больше и больше ресурсов.

Антипаттерны управления

Организационные антипаттерны (managerial antipatterns) — это проблемы, возникающие в результате неправильно принятых решений руководителем проекта — менеджером. Менеджер проекта отвечает за взаимодействие



между собой всех, кто занимается разработкой программы. Напрямую от него зависит принятие решений по срокам проекта и выделяемым ресурсам на его разработку, что влияет на правильность работы всего проекта.

Проблемы, с которыми сталкиваются менеджеры:

1. Аналитический паралич (Analysis paralysis) — большие затраты на анализ и проектирование, что приводит к закрытию проекта до начала его реализации.
2. Дойная корова (Cash cow) — в продукт, приносящий выгоду без вложений, не вкладываются средства в развитие и разработку новых продуктов.
3. Продолжительное устаревание (Continuous obsolescence) — приложение больших усилий на портирование программы в новые окружения.
4. Сваливание расходов (Cost migration) — перенос расходов на проект к уязвимому отделу или бизнес-партнёру.
5. Раздутый улучшизм (Creeping featurism) — добавление новых улучшений в ущерб суммарному качеству программы.
6. Раздутый элитизм (Creeping elegance) — непропорциональное улучшение красоты кода в ущерб функциональности и качеству системы.
7. Разработка комитетом (Design by committee) — разработка проекта без централизованного управления либо при некомпетентном руководстве.
8. Неуёмная преданность (Escalation of commitment) — продолжение реализации решения после того, как доказана его ошибочность.
9. Я тебе это говорил (I told you so) — игнорирование мнения профессионала.
10. Управление, основанное на числах (Management by numbers) — излишнее внимание к численным показателям, имеющим не очень важное значение.
11. Драконовские меры (Management by perkele) — неоправданно жёсткий стиль управления.