
# Шаблоны объектно-ориентированного проектирования

# Шаблоны проектирования – Паттерны ООП

- **Паттерн проектирования** — это часто встречающееся решение определённой 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.

# Из чего состоит паттерн?

- **Из чего состоит паттерн?**
- Описания паттернов обычно очень формальны и чаще всего состоят из таких пунктов:
- проблема, которую решает паттерн;
- мотивации к решению проблемы способом, который предлагает паттерн;
- структуры классов, составляющих решение;
- примера на одном из языков программирования;
- особенностей реализации в различных контекстах;
- связей с другими паттернами.

# История паттернов

- В 1995 году четвёрка авторов Эрих Гамм, Ричард Хелм, Ральф Джонсон, Джон Влиссидес написалиа книгу «**Приемы объектно-ориентированного проектирования. Паттерны проектирования**», в которую вошли 23 паттерна, решающие различные проблемы объектно-ориентированного дизайна.
- Название книги было слишком длинным, чтобы кто-то смог всерьёз его запомнить. Поэтому вскоре все стали называть её «book by the gang of four», то есть «книга от банды четырёх», а затем и вовсе «GOF book».
- Позже другими авторами были описаны дополнительные паттерны – сейчас их больше 100.

# Зачем знать паттерны?

- **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 просчётов 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.

# Классификации паттернов

- Два критерия классификации:
- Цель - отражает назначение паттерна
- Уровень - к чему обычно применяется паттерн: к объектам или класса

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

# Паттерны

- **Порождающие паттерны -** отвечают за удобное и безопасное создание новых объектов или даже целых семейств объектов

1. **Фабричный метод (Factory Method)**
2. **Абстрактная фабрика (*Abstract Factory*)**
3. **Строитель (Builder)**
4. **Прототип (*Prototype*)**
5. **Одиночка (*Singleton*)**

- **Структурные паттерны -** отвечают за построение удобных в поддержке иерархий классов.

1. **Адаптер (*Adapter*)**
2. **Мост (*Bridge*)**
3. **Компоновщик ( *Composite*)**
4. **Декоратор (*Обёртка, Decorator*)**
5. **Фасад (*Facade*)**
6. **Flyweight (*Кэш*)**
7. **Прокси (Proxy)**

# Паттерны (продолжение)

- **Поведенческие паттерны -** решают задачи эффективного и безопасного взаимодействия между объектами программы

1. **Цепочка обязанностей (*Chain of Responsibility*)**
2. **Команда (*Действие, Транзакция, Action, Command*)**
3. **Итератор (*Iterator*)**
4. **Посредник (*Mediator, Intermediary, Controller*)**
5. **Хранитель (*Memento, Снимок*)**
6. **Наблюдатель (*Издатель-Подписчик, Слушатель, Observer*)**
7. **Состояние (*State*)**
8. **Стратегия (*Strategy*)**
9. **Шаблонный метод (*Template Method*)**
10. **Посетитель (*Visitor*)**

# Использование интерфейсов, а не классов

- Сигнатура операции (метода): имя операции, параметры, и возвращаемое значение. Эту триаду называют сигнатурой операции.
- Интерфейс (объекта) - множество сигнатур всех определенных для объекта операций.
- Тип — это имя, используемое для обозначения конкретного интерфейса.
- Интерфейсы могут содержать другие интерфейсы в качестве подмножеств (наследование).
- Вызов операции (метода) полиморфен, т.е. до момента выполнения конкретная реализация интерфейса неизвестна (а клиенту и не важна – важен только интерфейс).
- Принцип объектно-ориентированного проектирования для повторного использования: **программируйте в соответствии с интерфейсом, а не с реализацией.**

# Композиция vs Наследование

- Есть два наиболее распространенных приема повторного использования функциональности в объектно-ориентированных системах - это **наследование класса** и **композиция объектов**.
- Наследование класса определяет реализацию одного класса в терминах другого. Прозрачный ящик (whitebox reuse) - внутреннее устройство родительских классов видимо подклассам.
- Композиция объектов - более сложная функциональность получается объединением или композицией объектов. Требуется, чтобы объединяемые объекты имели четко определенные интерфейсы. Черным ящиком (black-box reuse) - детали внутреннего устройства объектов остаются скрытыми.

# Композиция vs Наследование (плюсы и минусы)

- Наследование класса определяется статически на этапе компиляции, его проще использовать, оно напрямую поддержано языком программирования. Но нельзя изменить унаследованную от родителя реализацию во время выполнения программы, т.к. само наследование фиксировано на этапе компиляции. Подклассу доступны детали реализации родительского класса - что наследование нарушает инкапсуляцию. Реализации подкласса и родительского класса настолько тесно связаны, что любые изменения последней требуют изменять и реализацию подкласса.
- Композиция объектов определяется динамически во время выполнения за счет того, что объекты получают ссылки на другие объекты. Композицию можно применить, если объекты соблюдают интерфейсы друг друга. Доступ к объектам осуществляется только через их интерфейсы - нет нарушения инкапсуляции. Во время выполнения программы любой объект можно заменить другим, того же тип. Реализации объекта кодируются прежде всего его интерфейсы, значит и зависимость от реализации резко снижается. Классы и их иерархии остаются небольшими, и вероятность их разрастания до неуправляемых размеров невелика.
- Второе правило объектно-ориентированного проектирования: **предпочитайте композицию наследованию класса.**

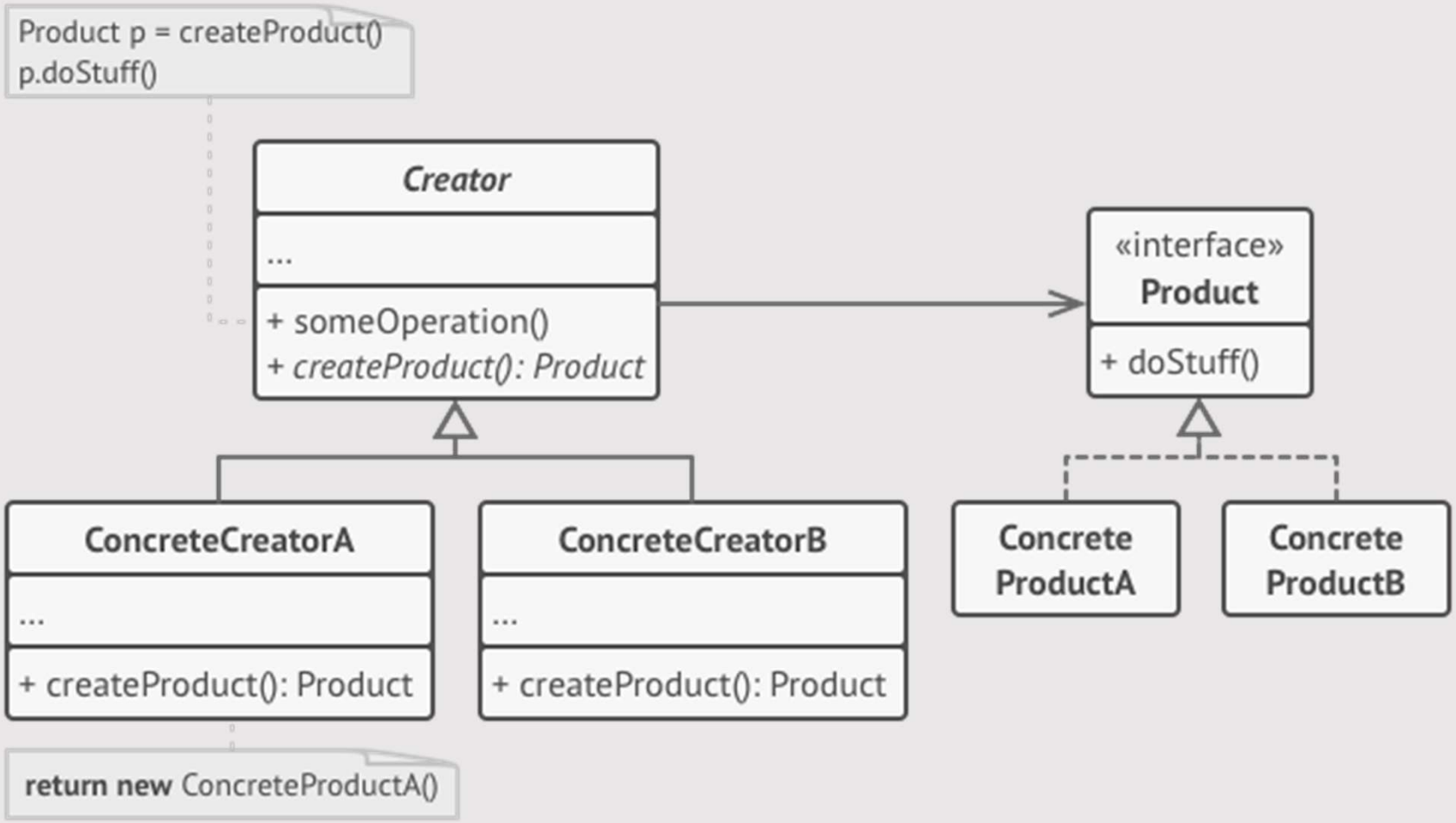
# Порождающие паттерны (Creational)

- Порождающие паттерны проектирования абстрагируют процесс инстанцирования. Они помогут сделать систему независимой от способа создания, композиции и представления объектов. Паттерн, порождающий классы, использует наследование, чтобы варьировать инстанцируемый класс, а паттерн, порождающий объекты, делегирует инстанцирование другому объекту
- Эти паттерны инкапсулируют знания о конкретных классах, которые применяются в системе.
- Скрывают детали того, как эти классы создаются и стыкуются.

# Фабричный метод
# Factory Method (Virtual Constructor)

- Определяет интерфейс для создания объекта, но оставляет подклассам решение о том, какой класс инстанцировать. Фабричный метод позволяет классу делегировать инстанцирование подклассам.
- Фабричный метод отделяет код производства продуктов от остального кода, который эти продукты использует.
- Создание объектов внутри класса с помощью фабричного метода всегда оказывается более гибким решением, чем непосредственное создание. Фабричный метод создает в подклассах операции-зацепки (hooks) для предоставления расширенной версии объекта.
- Соединение параллельных иерархий, которые возникают в случае, когда класс делегирует часть своих обязанностей другому классу, не являющемуся производным от него (фигуры – манипуляторы фигуры).

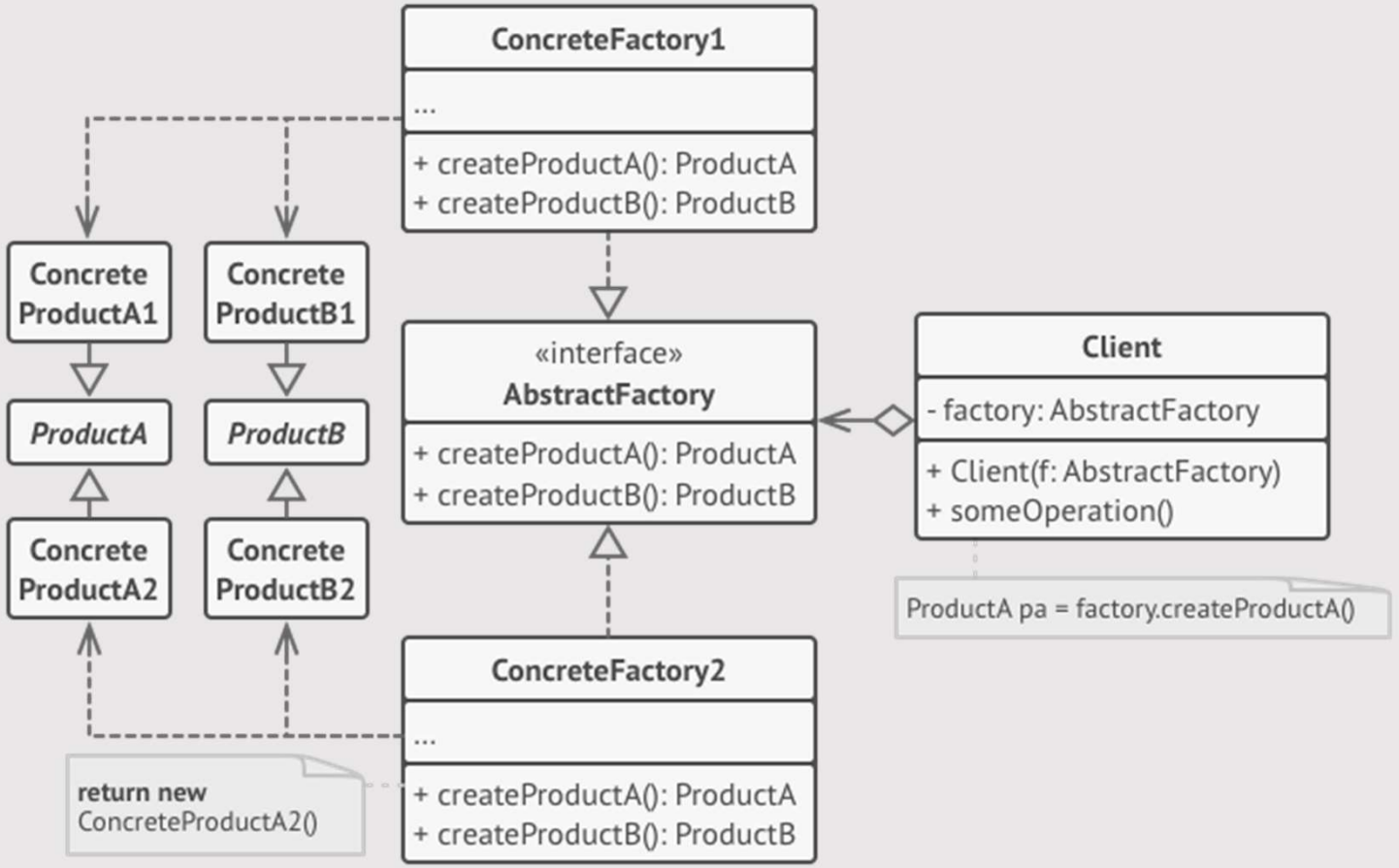
# Фабричный метод - реализация



# Абстрактная фабрика Abstract Factory (Kit)

- позволяет создавать семейства связанных объектов, не привязываясь к конкретным классам создаваемых объектов.
- Плюсы:
- изолирует конкретные классы
- упрощает замену семейств продуктов
- гарантирует сочетаемость продуктов
- Минусы:
- поддержать новый вид продуктов трудно

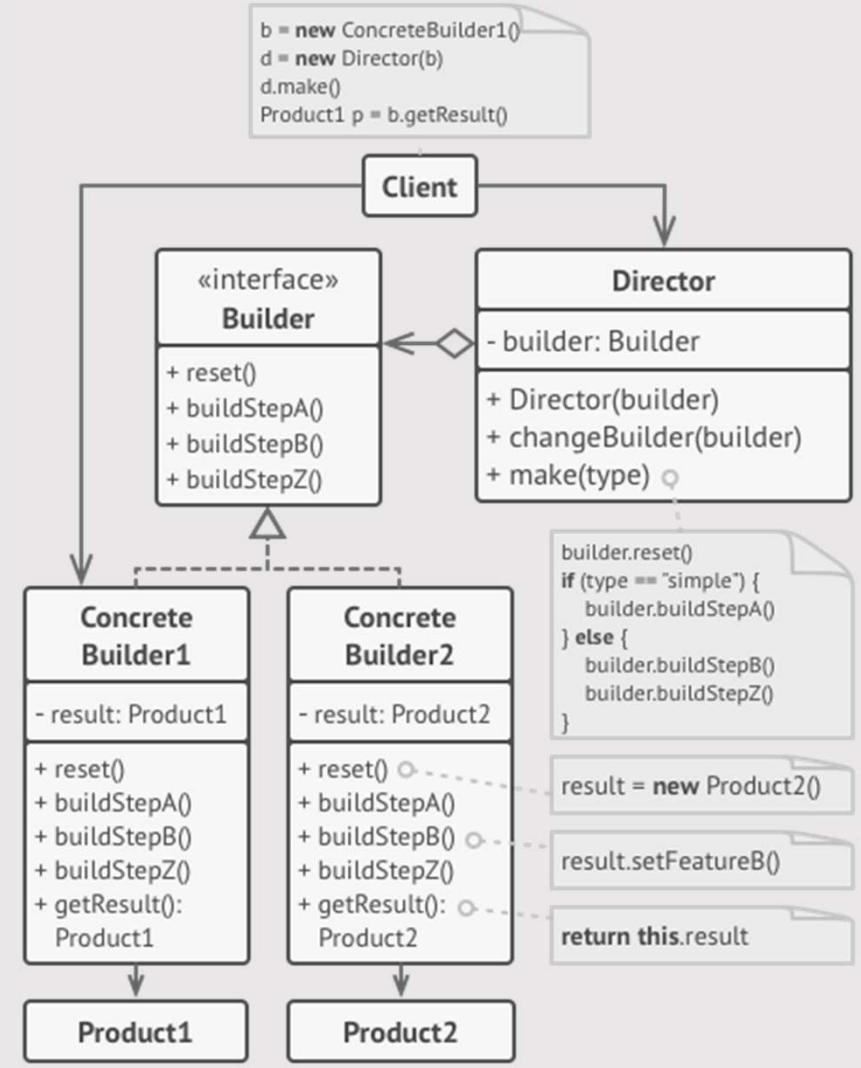
# Абстрактная фабрики - реализация



# Строитель - Builder

- Отделяет конструирование сложного объекта от его представления, так что в результате одного и того же процесса конструирования могут получаться разные представления.
- Конструирование объекта осуществляется внешними по отношению к нему сущностями, называемыми строителями.
- Позволяет избавиться от конструктора со множеством опциональных параметров.

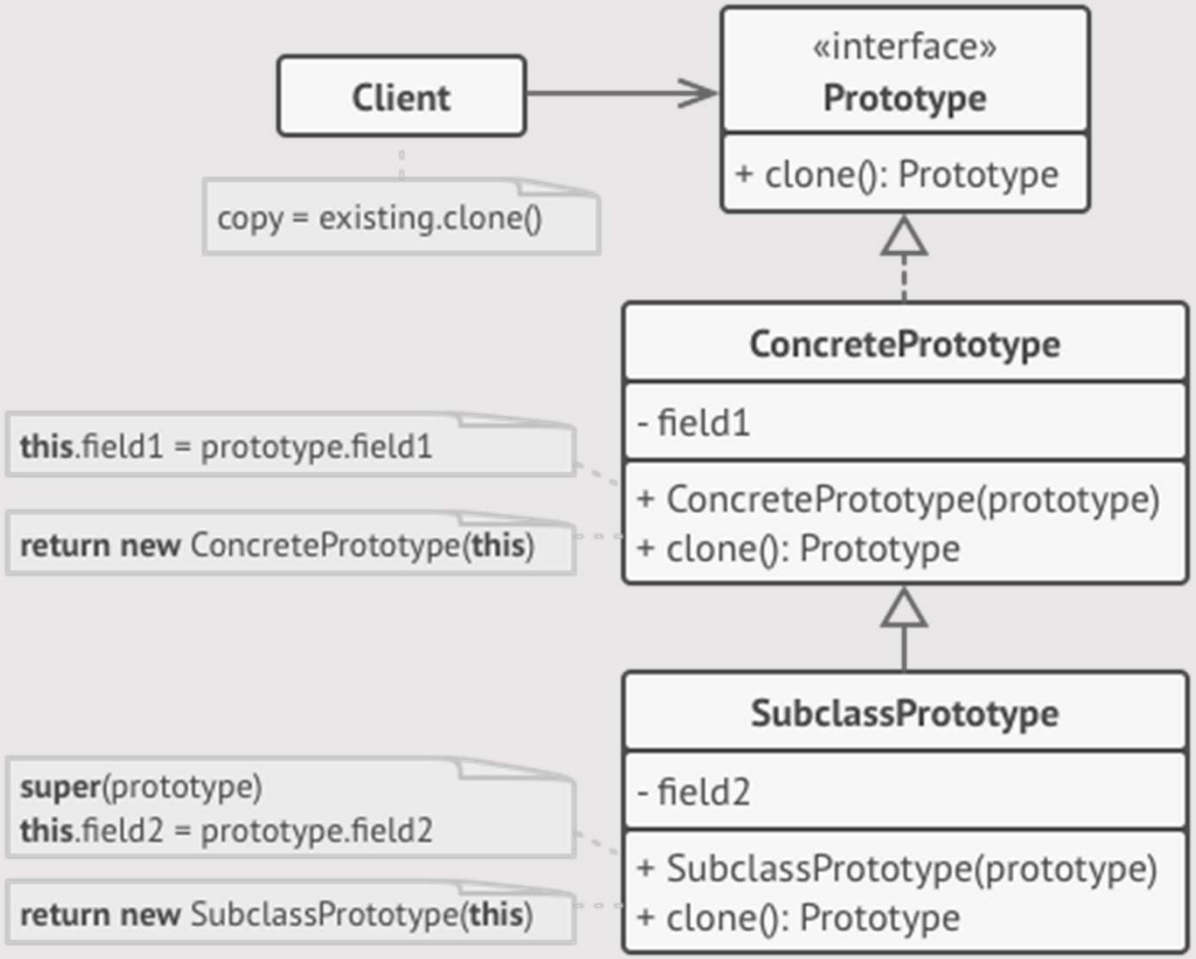
# Строитель - реализация



# Прототип - Prototype

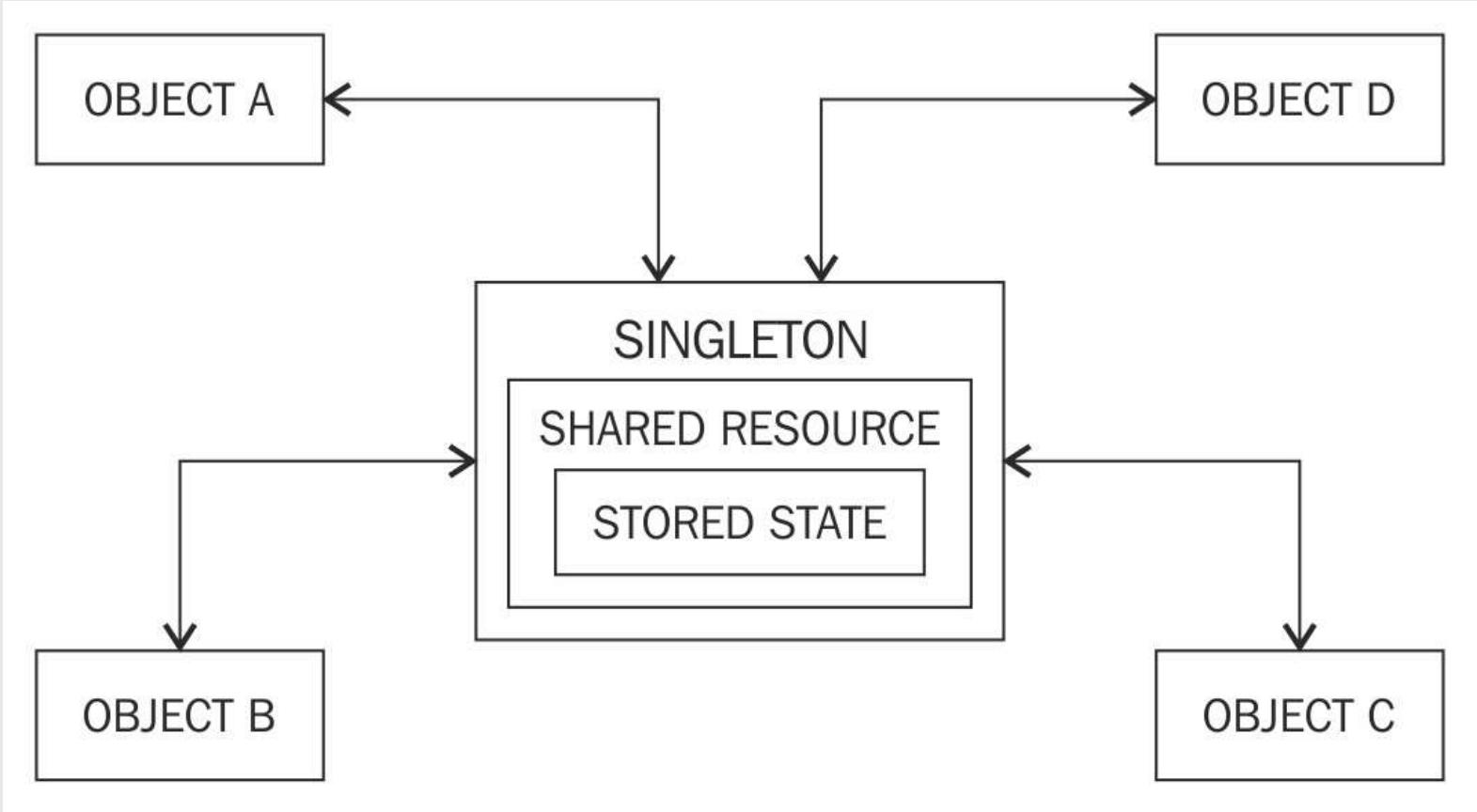
- Задает виды создаваемых объектов с помощью экземпляра-прототипа и создает новые объекты путем копирования этого прототипа.
- Паттерн прототип предлагает использовать набор прототипов, вместо создания подклассов для описания популярных конфигураций объектов.
- У прототипа те же самые результаты, что у абстрактной фабрики и строителя: он скрывает от клиента конкретные классы продуктов, уменьшая тем самым число известных клиенту имен. Кроме того, все эти паттерны позволяют клиентам работать со специфичными для приложения классами без модификаций.
- Основной недостаток паттерна прототип заключается в том, что каждый подкласс класса Prototype должен реализовывать операцию Clone, а это далеко не всегда просто. Например, сложно добавить операцию Clone, когда рассматриваемые классы уже существуют. Проблемы возникают и в случае, если во внутреннем представлении объекта есть другие объекты или наличествуют циклические ссылки

# Прототип - реализация

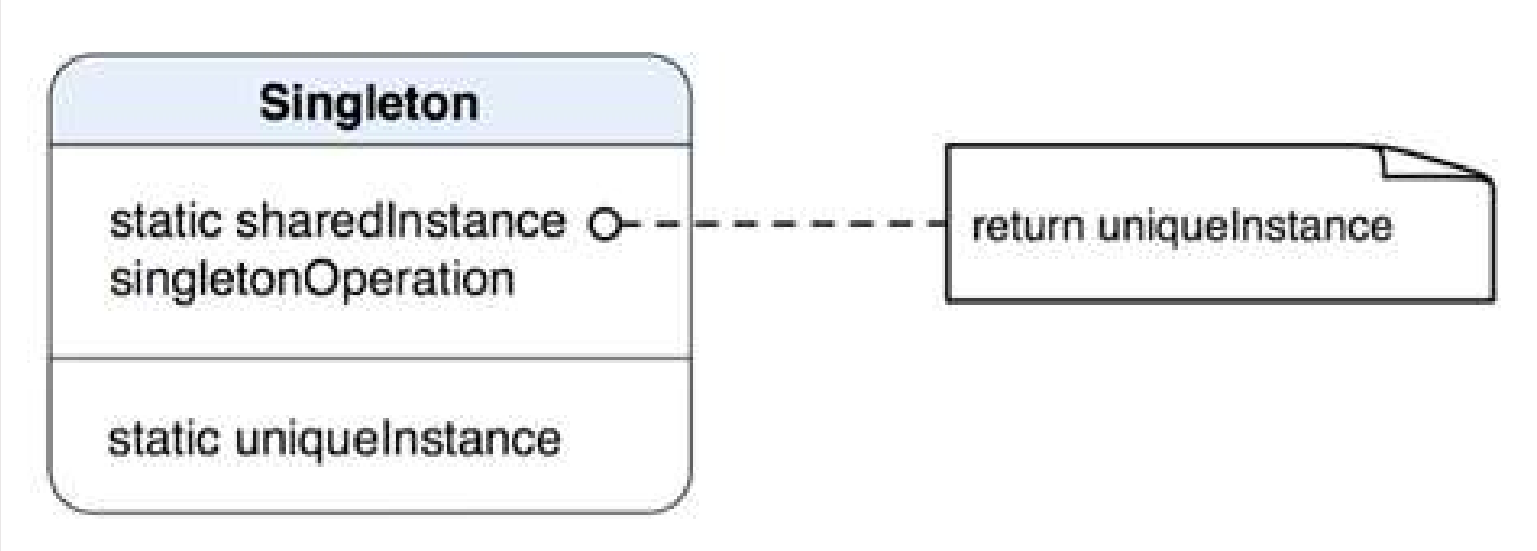


# Синглтон - Singleton

Гарантирует, что у класса есть только один экземпляр, и предоставляет к нему глобальную точку доступа.



# Синглтон - реализация



# Фабричные паттерны

- Два способа параметризировать систему классами создаваемых ей объектов:
- Первый - порождение подклассов от класса, создающего объекты (фабричный метод). Недостаток: требуется создавать новый подкласс лишь для того, чтобы изменить класс продукта.
- Второй - основан на композиции объектов. Вы определяете объект, которому известно о классах объектов-продуктов, и делаете его параметром системы (абстрактная фабрика, строитель и прототип). Характерно создание «фабричного объекта», который изготавливает продукты. В абстрактной фабрике фабричный объект производит объекты разных классов. Фабричный объект строителя постепенно создает сложный продукт, следуя специальному протоколу. Фабричный объект прототипа изготавливает продукт путем копирования объекта-прототипа. В последнем случае фабричный объект и прототип - это одно и то же, поскольку именно прототип отвечает за возврат продукта.

# Лабораторная работа 1

- Создать набор классов моделирующих графические элементы на плоскости (Point, Line, Circle) и абстрактный класс графических элементов (GraphObject). Реализовать прототипирование.
- Создать класс графической сцены (Scene), как множества разнотипных графических объектов. Сделать сцену одиночкой.
- Обеспечить автоматическое добавление графических объектов в сцену (через hook'и фабричных методов)
- Реализовать создание объектов через абстрактную фабрику. Сделать отдельно фабрику для создания цветной и черно-белой сцены.
- Сделать два Строителя сцены – одни для реального построения тестовой сцены из нескольких объектов, другой для расчета занимаемой ими памяти.

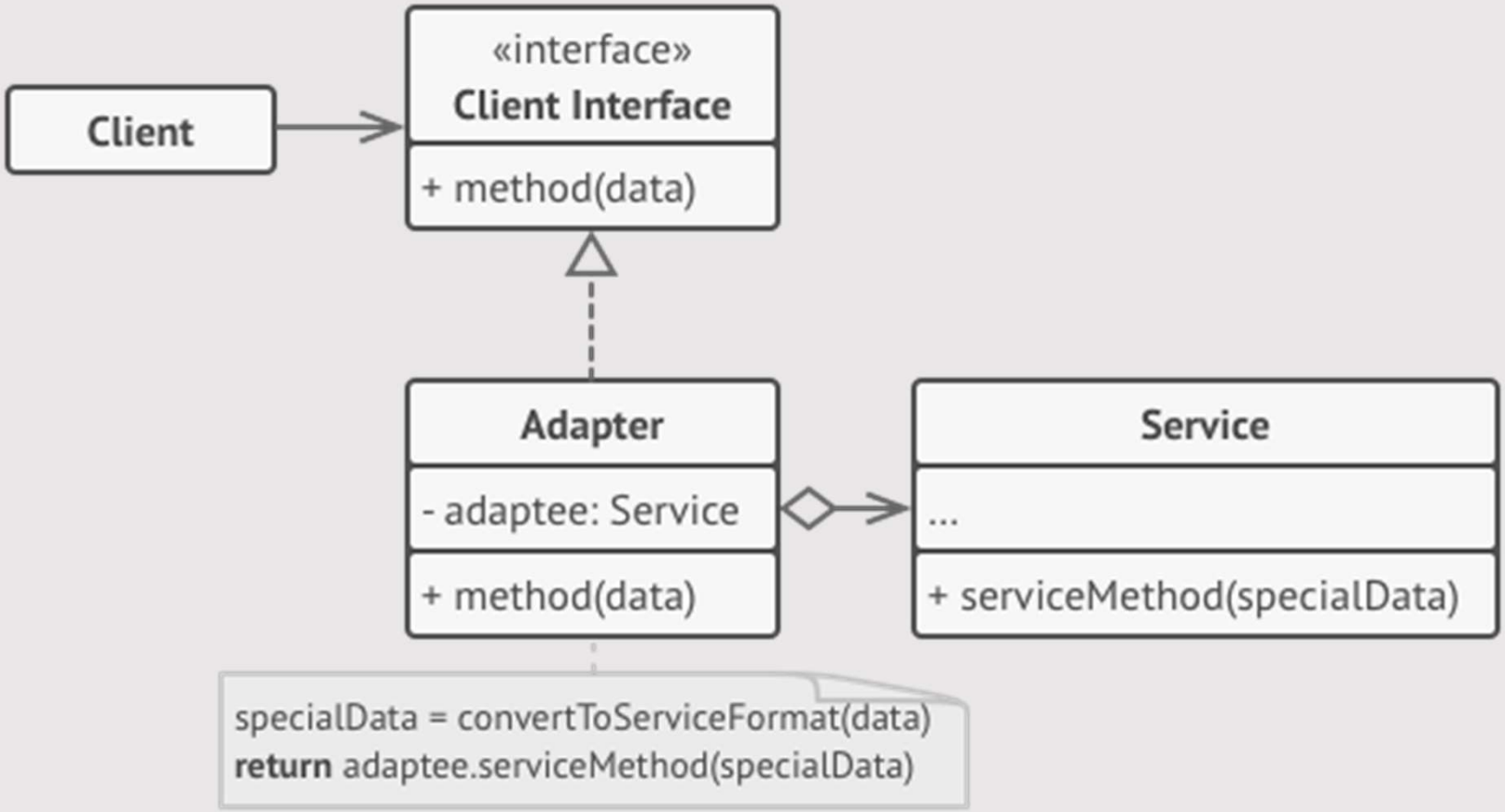
# Структурные паттерны (Structural)

- Образуют из классов и объектов более крупные структуры.
- Паттерны уровня класса используют наследование для составления композиций из интерфейсов и реализаций.
- Паттерны уровня объекта компонуют 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.

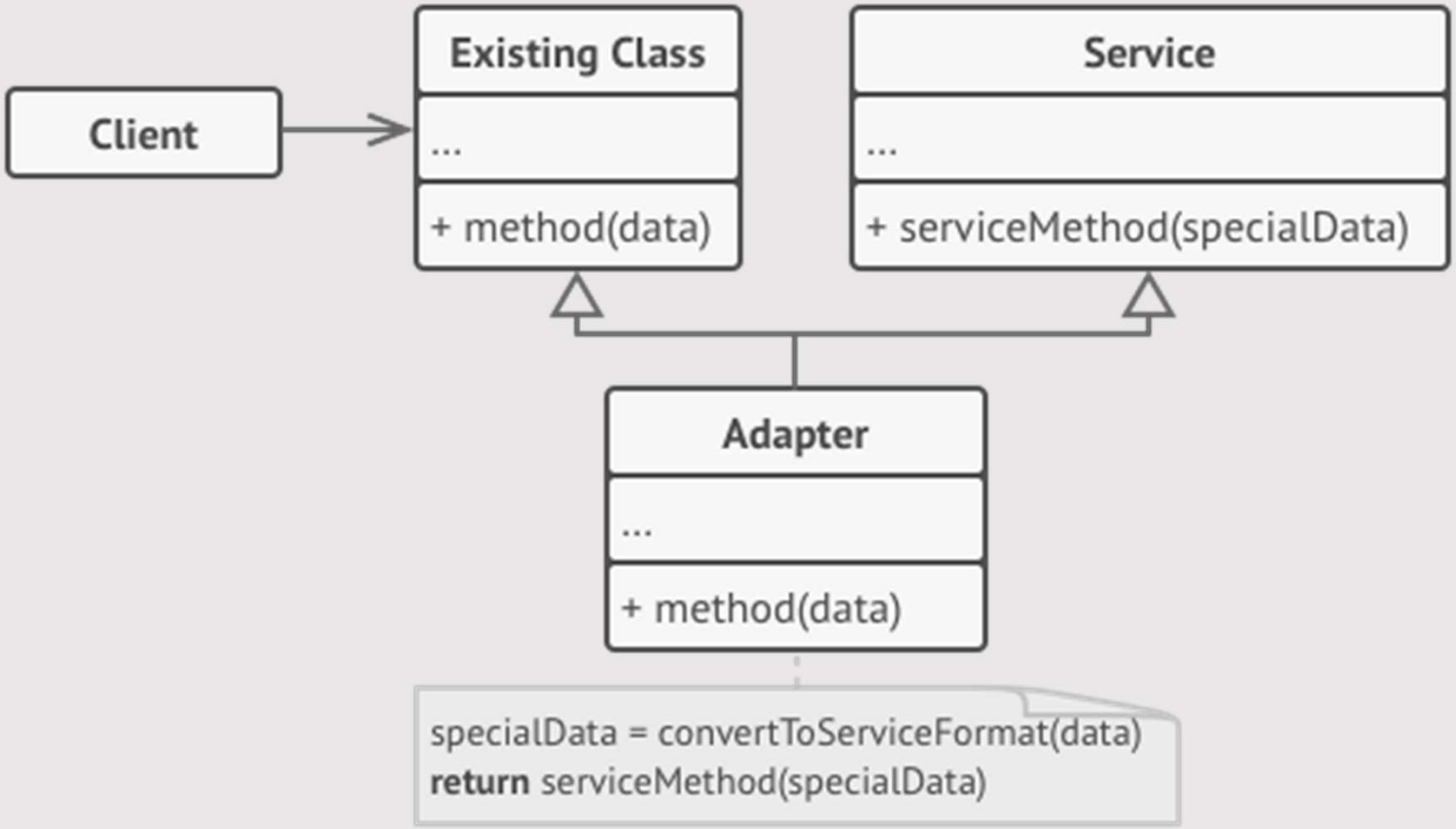
# Адаптер – Adapter (Обёртка - Wrapper)

- Позволяет использовать сторонний класс, интерфейс которого не соответствует остальному коду приложения.
- Преобразует интерфейс одного класса в интерфейс другого, который ожидают клиенты. Адаптер обеспечивает совместную работу классов с несовместимыми интерфейсами, которая без него была бы невозможна.
- Есть два подхода в реализации, которые соответствуют вариантам паттерна адаптер в его объектной и классовой ипостасях.

# Адаптер – реализация для объекта (композиция)



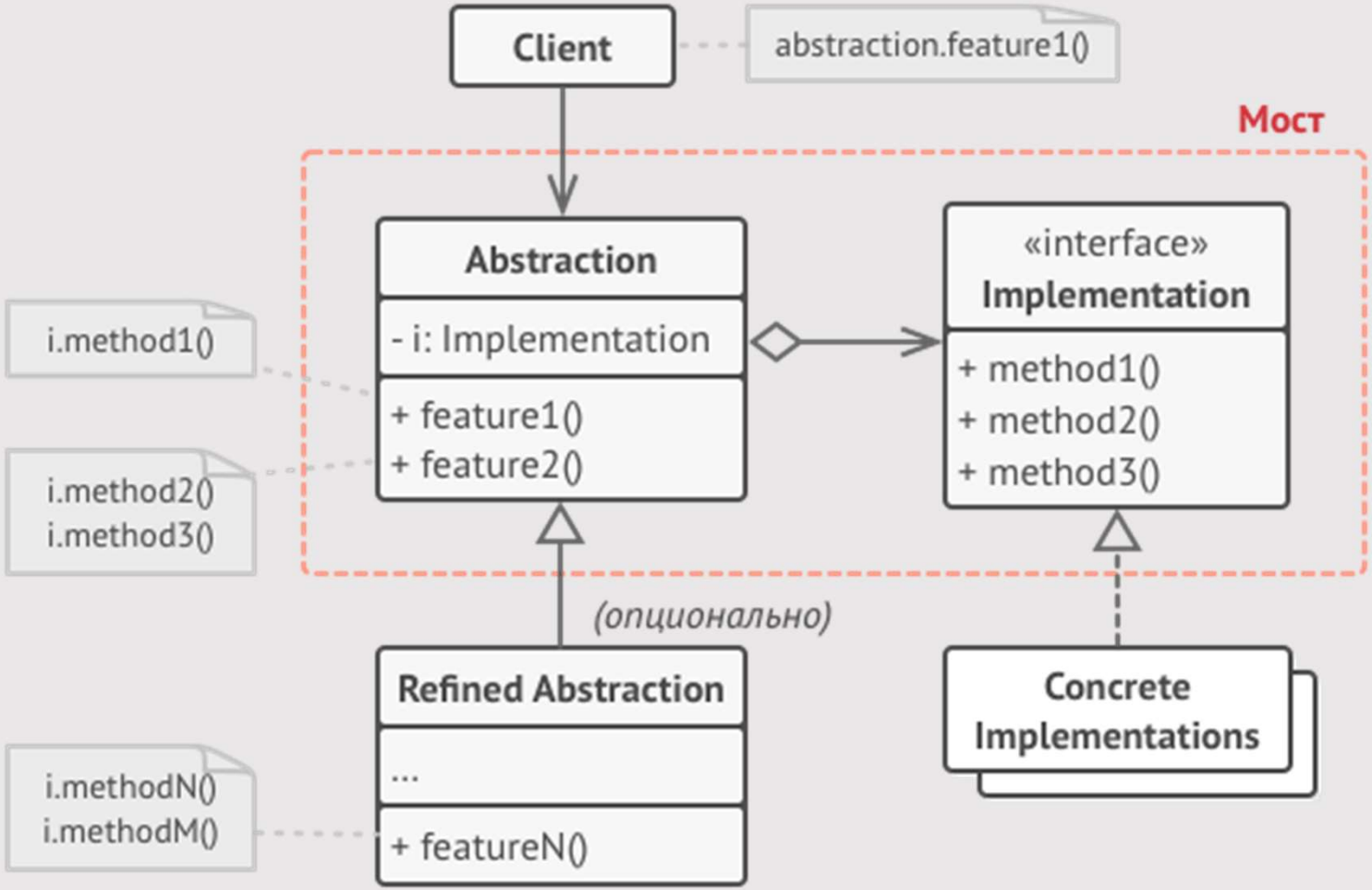
# Адаптер – реализация для класса (множественное наследование)



# Мост - Bridge

- Разделяет один или несколько классов на две отдельные иерархии — абстракцию и реализацию, позволяя изменять их независимо друг от друга.
- Аналогичен структуре адаптера, но имеет иное назначение. Отделяет интерфейс от реализации, чтобы то и другое можно было изменять независимо. Адаптер же призван изменить интерфейс существующего объекта
- Если на необходимо создать несколько разных абстракций (интерфейсов) с разными вариантами реализаций (платформами), которым делегировано выполнение работы то при использовании наследования количество дочерних классов быстро растёт. Проблема в попытке расширения классов по нескольким независимым «плоскостям».
- Паттерн Мост предлагает заменить наследование композицией. Для этого нужно выделить одну из таких «плоскостей» в отдельную иерархию и ссылаться на объект этой иерархии, вместо хранения его состояния и поведения внутри одного класса.

# Мост - реализация



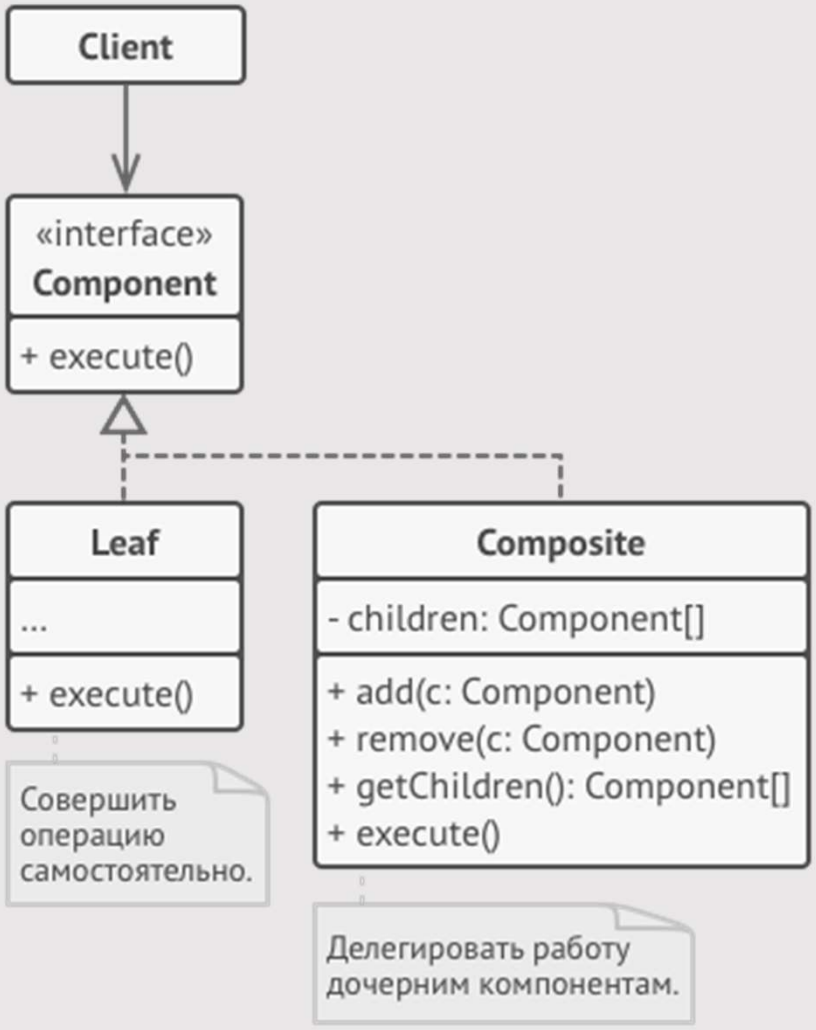
# Связи Моста

- Паттерн абстрактная фабрика может создать и сконфигурировать мост. Для обеспечения совместной работы не связанных между собой классов прежде всего предназначен паттерн адаптер. Обычно он применяется в уже готовых системах. Мост же участвует в проекте с самого начала и призван поддержать возможность независимого изменения абстракций и их реализаций.

# Компоновщик - Composite

- Компонует объекты в древовидные структуры для представления иерархий часть-целое. Позволяет клиентам единообразно трактовать индивидуальные и составные объекты.
- Имеет смысл только тогда, когда основная модель вашей программы может быть структурирована в виде дерева.
- Упрощает архитектуру клиента при работе со сложным деревом компонентов.
- Облегчает добавление новых видов компонентов
- **Строитель** можно использовать для последовательного создания дерева **Компоновщика**.

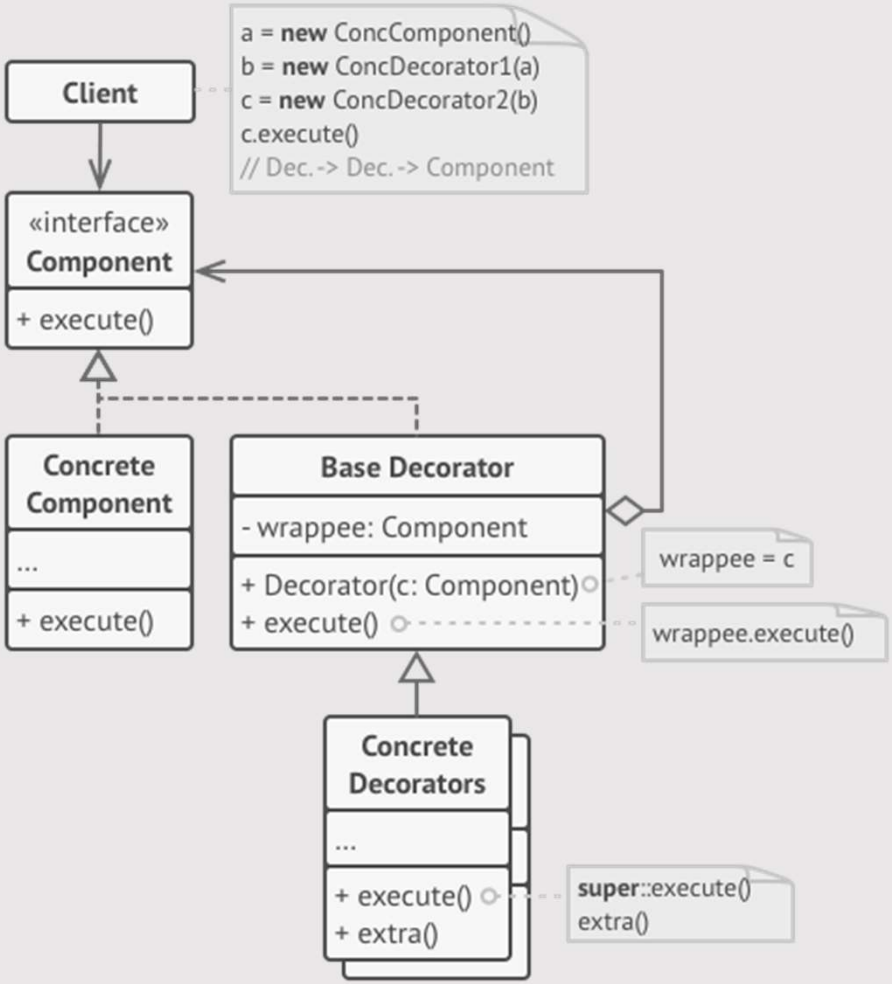
# Компоновщик - реализация



# Декоратор (Обёртка) - Decorator

- Динамически добавляет объекту новые обязанности. Является гибкой альтернативой порождению подклассов с целью расширения функциональности.
- Динамически добавляет объектам новую функциональность, оборачивая их в полезные «обёртки», реализующие тот же интерфейс что и «оборачиваемые» объекты.
- Декоратор использует композицию вместо наследования для расширения функционала.
- Декоратор содержит ссылку на другой объект и делегирует ему работу, вместо того чтобы самому наследовать его поведение.

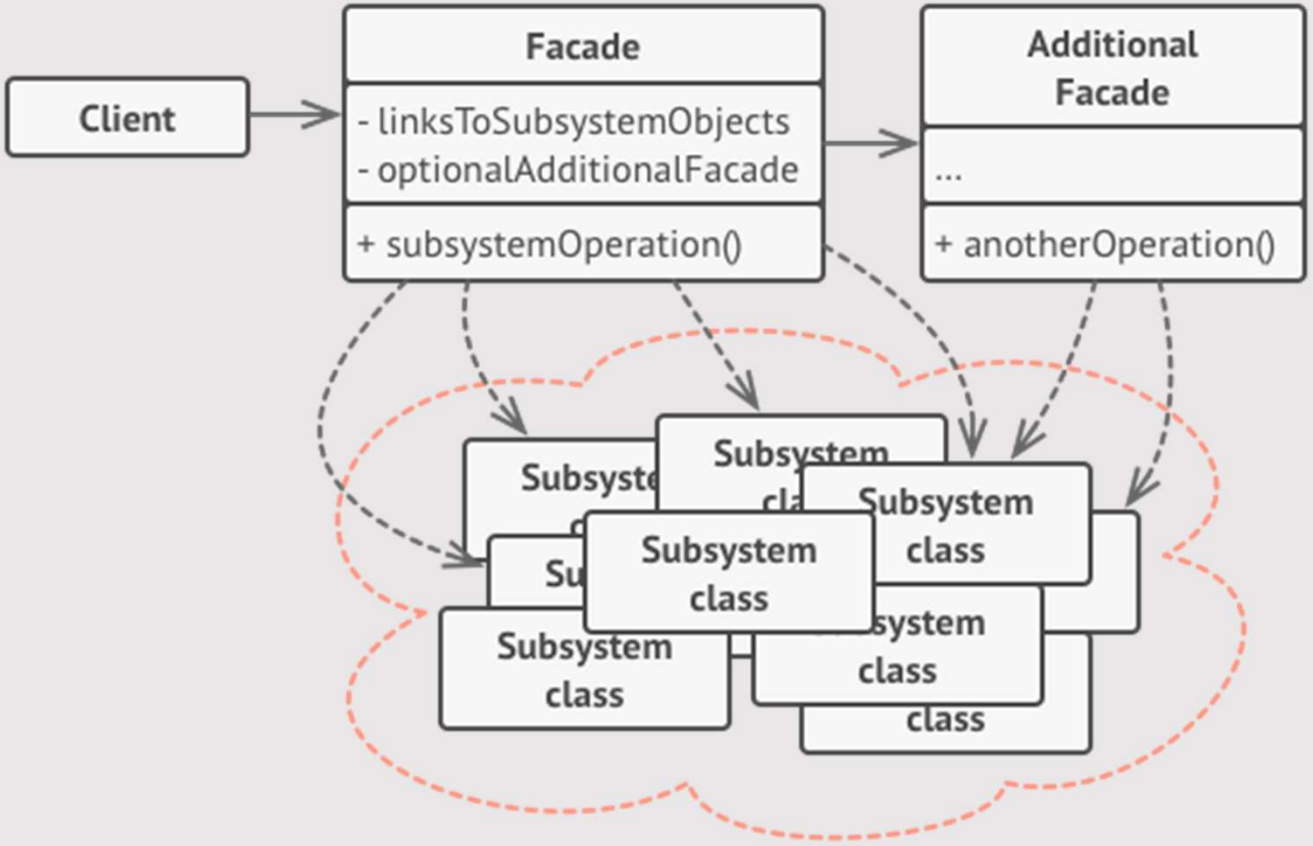
# Декоратор - реализация



# Фасад - Facade

- Предоставляет унифицированный интерфейс вместо набора интерфейсов некоторой подсистемы. Фасад определяет интерфейс более высокого уровня, который упрощает использование подсистемы.
- Если коду приходится работать с большим количеством объектов некой сложной библиотеки или фреймворка, приходится самостоятельно инициализировать эти объекты, следить за правильным порядком зависимостей и так далее.
- Фасад — это простой интерфейс для работы со сложной подсистемой, содержащей множество классов.

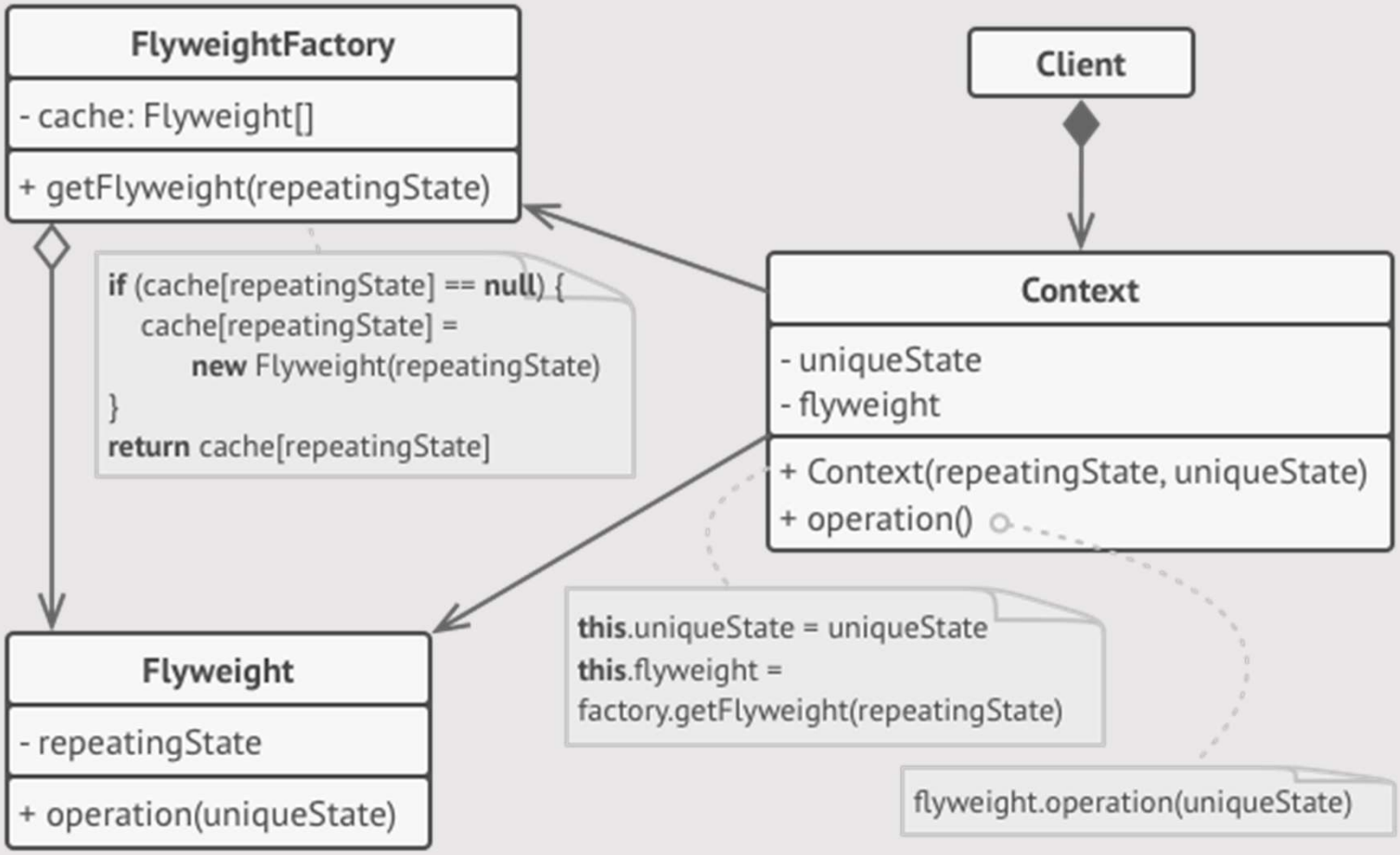
# Фасад - реализация



# Кэш (Приспособленец, Легковес) - Flyweight

- Использует разделение для эффективной поддержки множества мелких объектов.
- Позволяет вместить бóльшее количество объектов в отведённую оперативную память.
- Экономит память, разделяя общее состояние объектов между собой, вместо хранения одинаковых данных в каждом объекте.

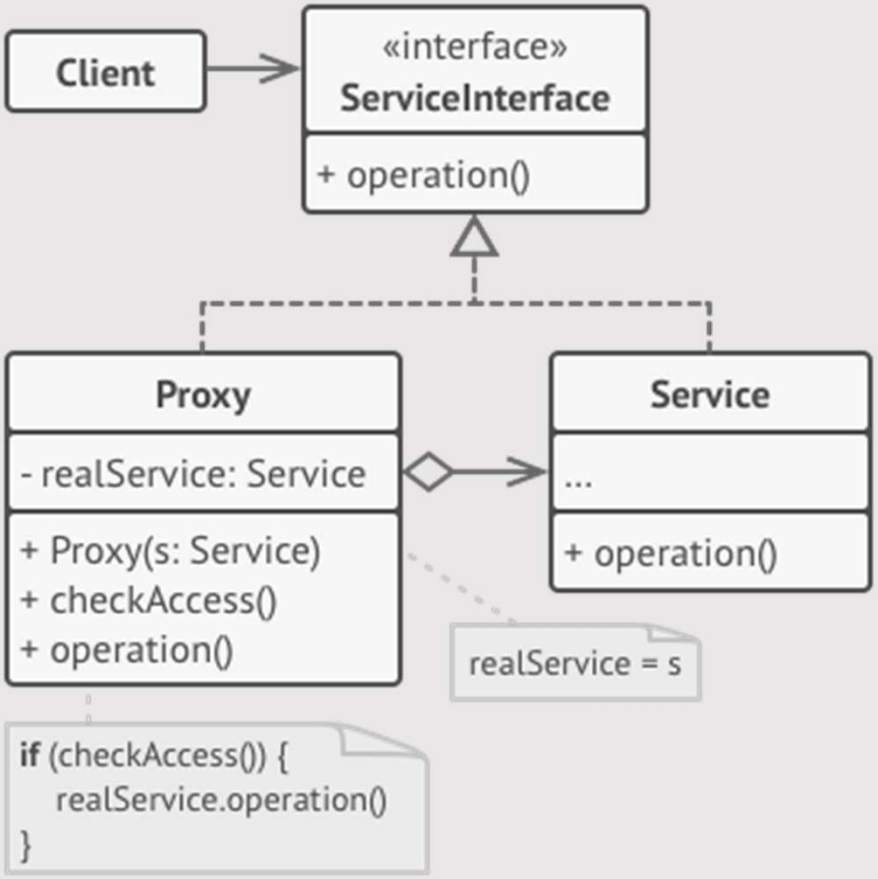
# Кэш - реализация



# Прокси (Заместитель) – Proxy (Surrogate)

- Является суррогатом другого объекта и контролирует доступ к нему.
- Перехватывает вызовы к оригинальному объекту, позволяя сделать что-то до или после передачи вызова оригиналу.
- Подставляется на место реальных объектов, т.к. реализует тот же интерфейс.

# Прокси - реализация



# Структурные паттерны

- Все структурные паттерны основаны на небольшом множестве языковых механизмов структурирования кода и объектов (одиночном и множественном наследовании для паттернов уровня класса и композиции для паттернов уровня объектов).
- **Адаптер** и **мост**: есть несколько общих атрибутов. Повышают гибкость, вводя дополнительный уровень косвенности при обращении к другому объекту. Оба перенаправляют запросы другому объекту, используя иной интерфейс. Различие между адаптером и мостом в их назначении. **Адаптера** устраняет несовместимость между двумя существующими интерфейсами. **Мост** связывает абстракцию с ее, возможно, многочисленными реализациями.
- **Компоновщик, Декоратор:** основаны на рекурсивной композиции и предназначены для организации заранее неопределенного числа объектов. Однако назначение **Декоратора** - добавить новые обязанности объекта без порождения подклассов. **Компоновщик** должен так структурировать классы, чтобы различные взаимосвязанные объекты удавалось трактовать единообразно, а несколько объектов рассматривать как один.
- **Прокси** в отличие от **Декоратора** не нужно динамически добавлять и отбирать свойства, он не предназначен для рекурсивной композиции. Он должен предоставить стандартную замену субъекту, когда прямой доступ к нему неудобен или нежелателен.

# Лабораторная работа 2

- Используя проект первой лабораторной, добавьте новую графическую фигуру - Треугольник, не унаследованную от GraphObject (вы её получили из сторонней библиотеке). Попробуйте встроить эту фигуру в вашу графическую сцену, используя адаптер.
- Сделайте частью графической сцены композитный элемент, который может состоять из других простых или композитных графических элементов, используя компоновщик.
- Добавьте функционал закрашивания фигуры с помощью декоратора.
- Разработайте фасад для работы с графической сценой и фигурами.

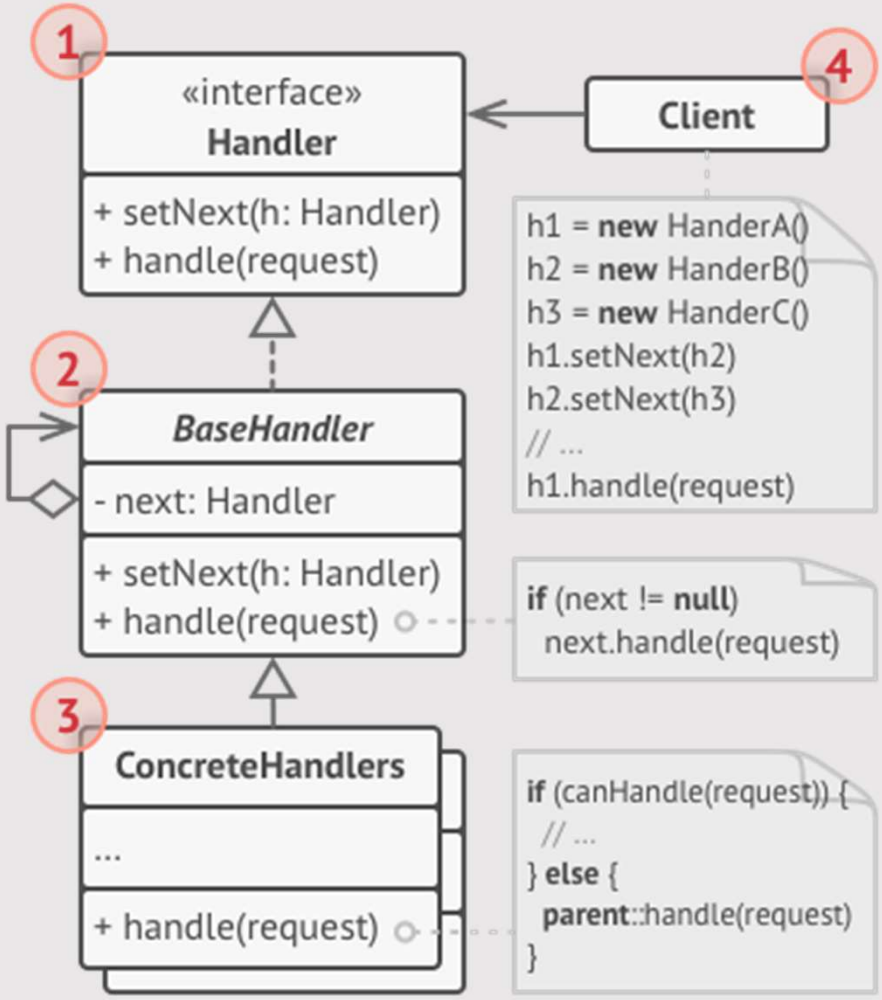
# Поведенческие паттерны (Behavioral)

- Решают задачи эффективного и безопасного взаимодействия между объектами программы
- Связаны с алгоритмами и распределением обязанностей между объектами. Речь в них идет не только о самих объектах и классах, но и о типичных способах взаимодействия. Паттерны поведения характеризуют сложный поток управления, который трудно проследить во время выполнения программы.
- В паттернах поведения уровня класса используется наследование - чтобы распределить поведение между разными классами.
- В паттернах поведения уровня объектов используется не наследование, а композиция. Некоторые из них описывают, как с помощью кооперации множество равноправных объектов справляется с задачей, которая ни одному из них не под силу.

# Цепочка обязанностей – Chain of Responsibility

- 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.
- Базируется на том, чтобы превратить отдельные поведения в объекты
- Передавая запросы в первый обработчик цепочки можно гарантировать, что все объекты в цепи смогут его обработать. При этом длина цепочки не имеет никакого значения.

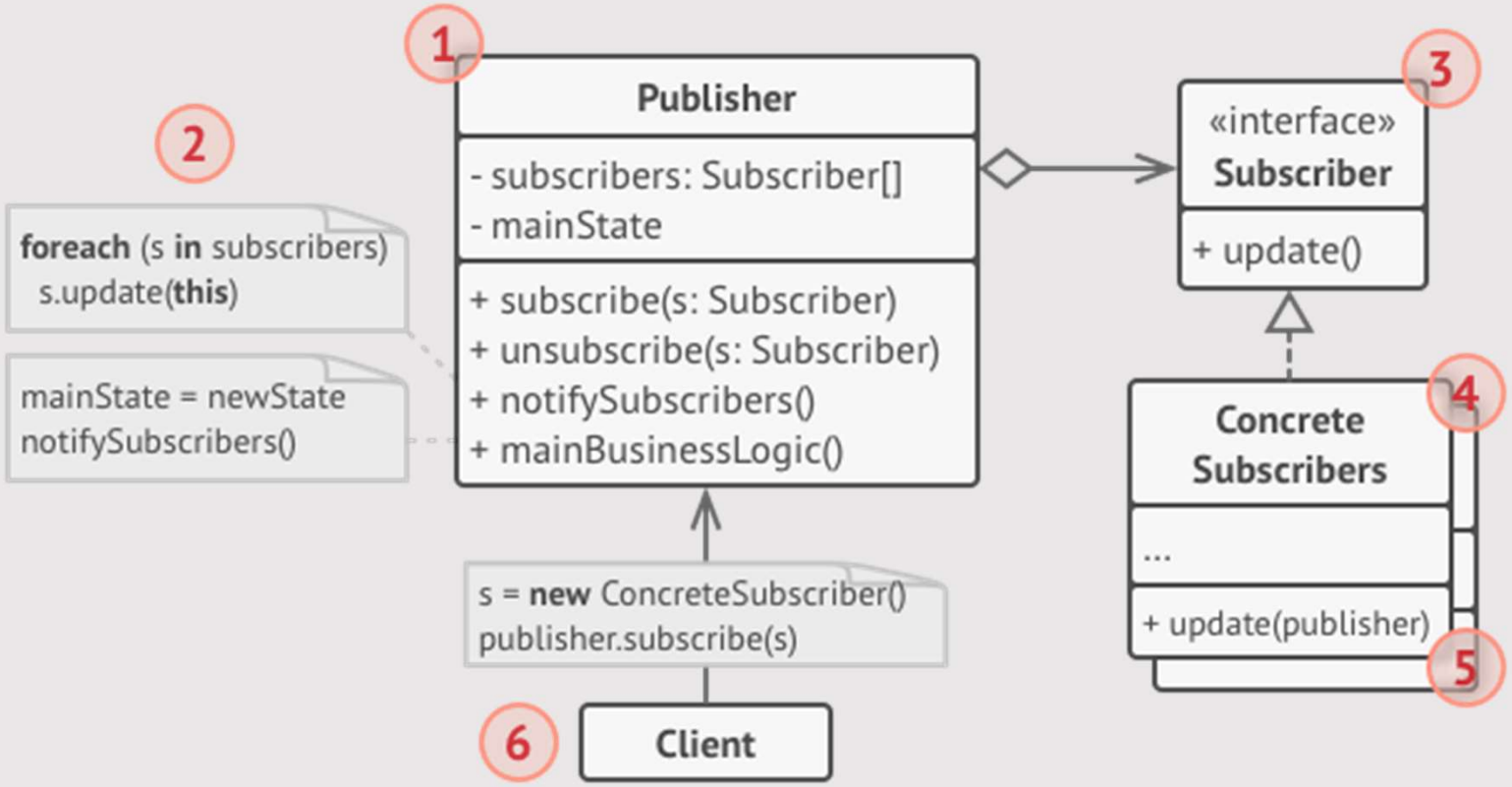
# Цепочка обязанностей – реализация



# Наблюдатель (Слушатель, издатель-подписчик) – Observer Publish-Subscribe)

- Определяет зависимость типа «один ко многим» между объектами таким образом, что при изменении состояния одного объекта все зависящие от него оповещаются об этом и автоматически обновляются.
- Создаёт механизм подписки, позволяющий одним объектам следить и реагировать на события, происходящие в других объектах.
- Обеспечивают меньшую связанность объектов (Loosely coupled component).
- Позволяет любому объекту с интерфейсом подписчика зарегистрироваться на получение оповещений о событиях, происходящих в объектах-издателях.

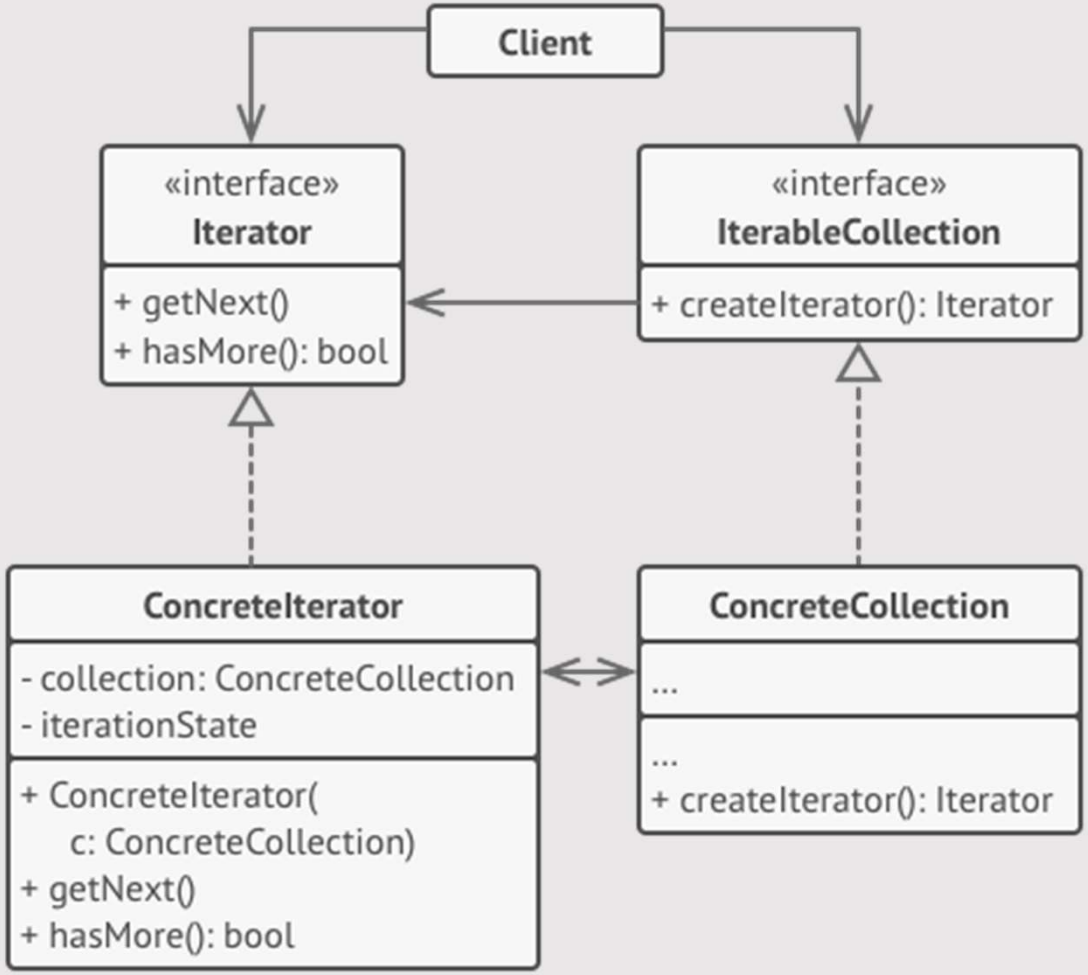
# Наблюдатель – реализация



# Итератор (Курсор) - Iterator (Cursor)

- Предоставляет способ последовательного доступа ко всем элементам составного объекта, не раскрывая его внутреннего представления.
- Отделение алгоритма переборки данных от самих данных.
- Можно сделать несколько итераторов, обеспечивающих разный порядок обхода.
- Объект-итератор может отслеживать состояние обхода, текущую позицию в коллекции и сколько элементов ещё осталось обойти.
- Не оправдан, если можно обойтись простым циклом.

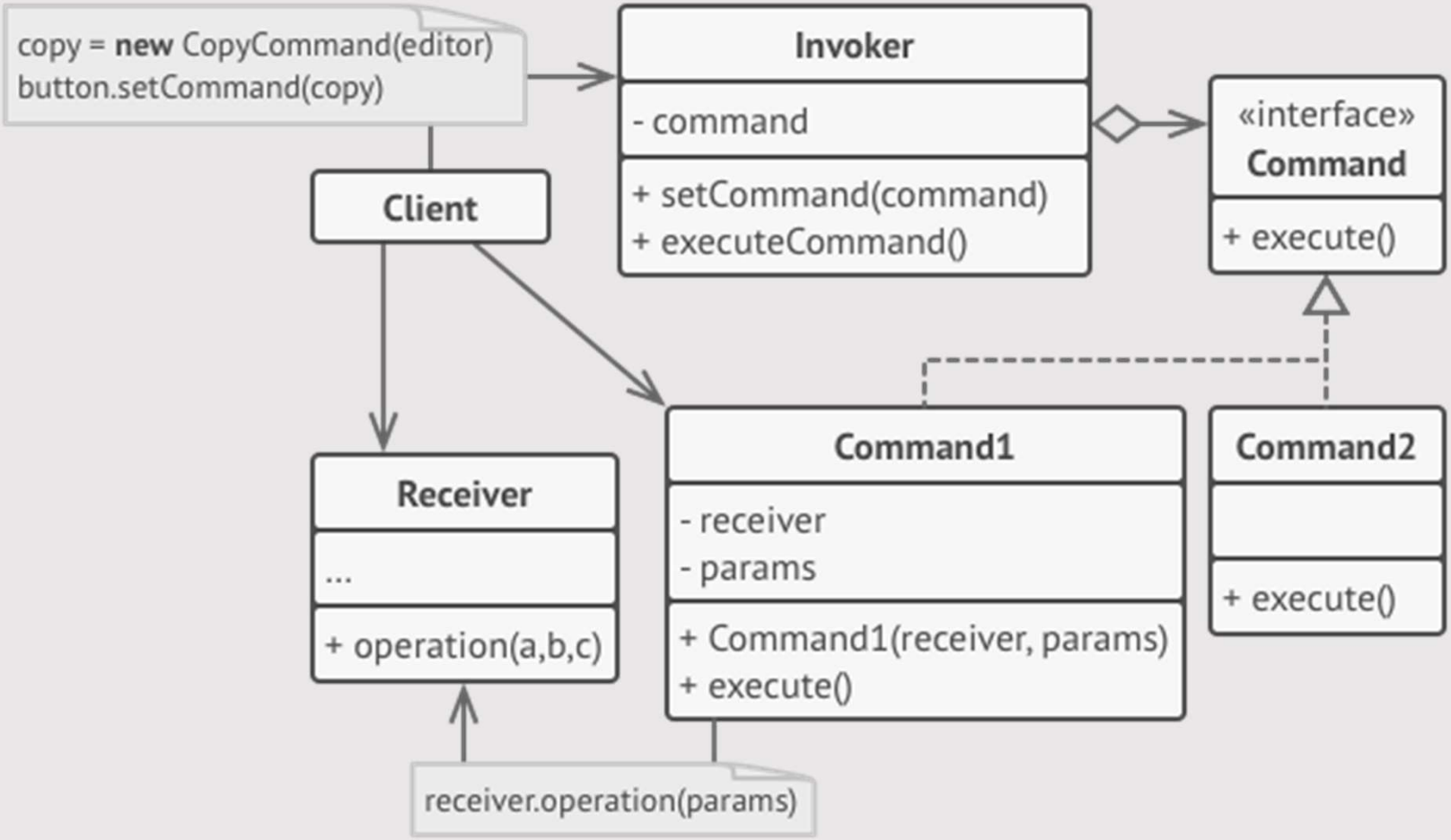
# Итератор - реализация



# Команда – Command (Action)

- Инкапсулирует запрос как объект, позволяя тем самым задавать параметры клиентов для обработки соответствующих запросов, ставить запросы в очередь или протоколировать их, а также поддерживать отмену операций.
- Паттерн команда позволяет библиотечным объектам отправлять запросы не известным объектам приложения, преобразовав сам запрос в объект. Этот объект можно хранить и передавать, как и любой другой.
- Команда превращает операции в объекты. А объекты можно передавать, хранить и менять внутри других объектов.
- Команды можно сериализовать, то есть превратить в поток, чтобы потом сохранить в файл или базу данных. Затем в любой удобный момент её можно достать обратно, снова превратить в объект команды и выполнить. Таким же образом команды можно передавать по сети, логировать или выполнять на удалённом сервере.

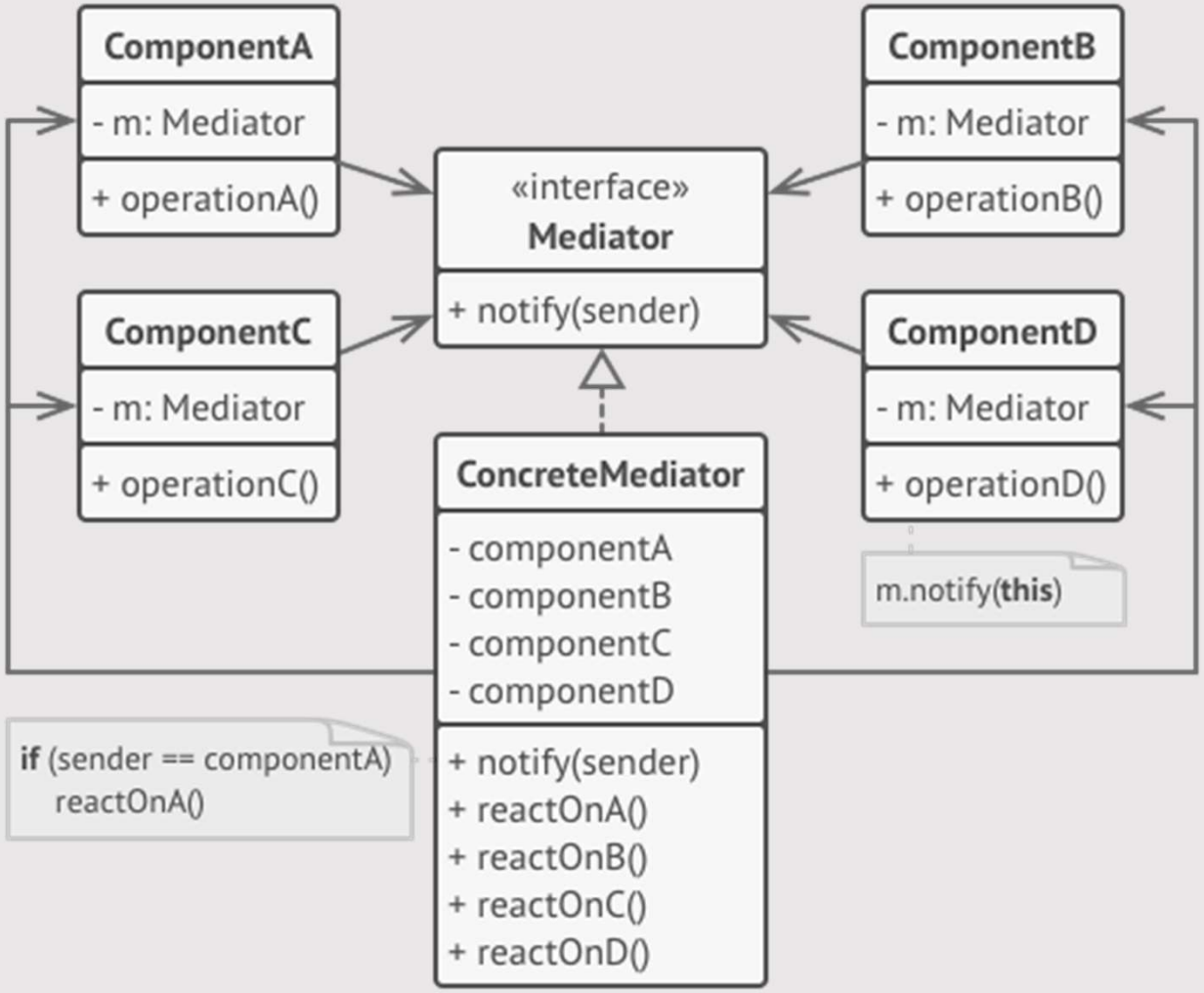
# Команда – реализация



# Посредник - Mediator, Controller

- Определяет объект, инкапсулирующий способ взаимодействия множества объектов. Посредник обеспечивает слабую связанность системы, избавляя объекты от необходимости явно ссылаться друг на друга и позволяя тем самым независимо изменять взаимодействия между ними.
- Перемещает связи в один класс-посредник и централизует управление в одном месте.
- После применения паттерна компоненты теряют прежние связи с другими компонентами, а всё их общение происходит косвенно, через объект-посредник.
- Если раньше изменение отношений в одном компоненте могли повлечь за собой лавину изменений во всех остальных компонентах, то теперь достаточно создать подкласс посредника и поменять в нём связи между компонентами.

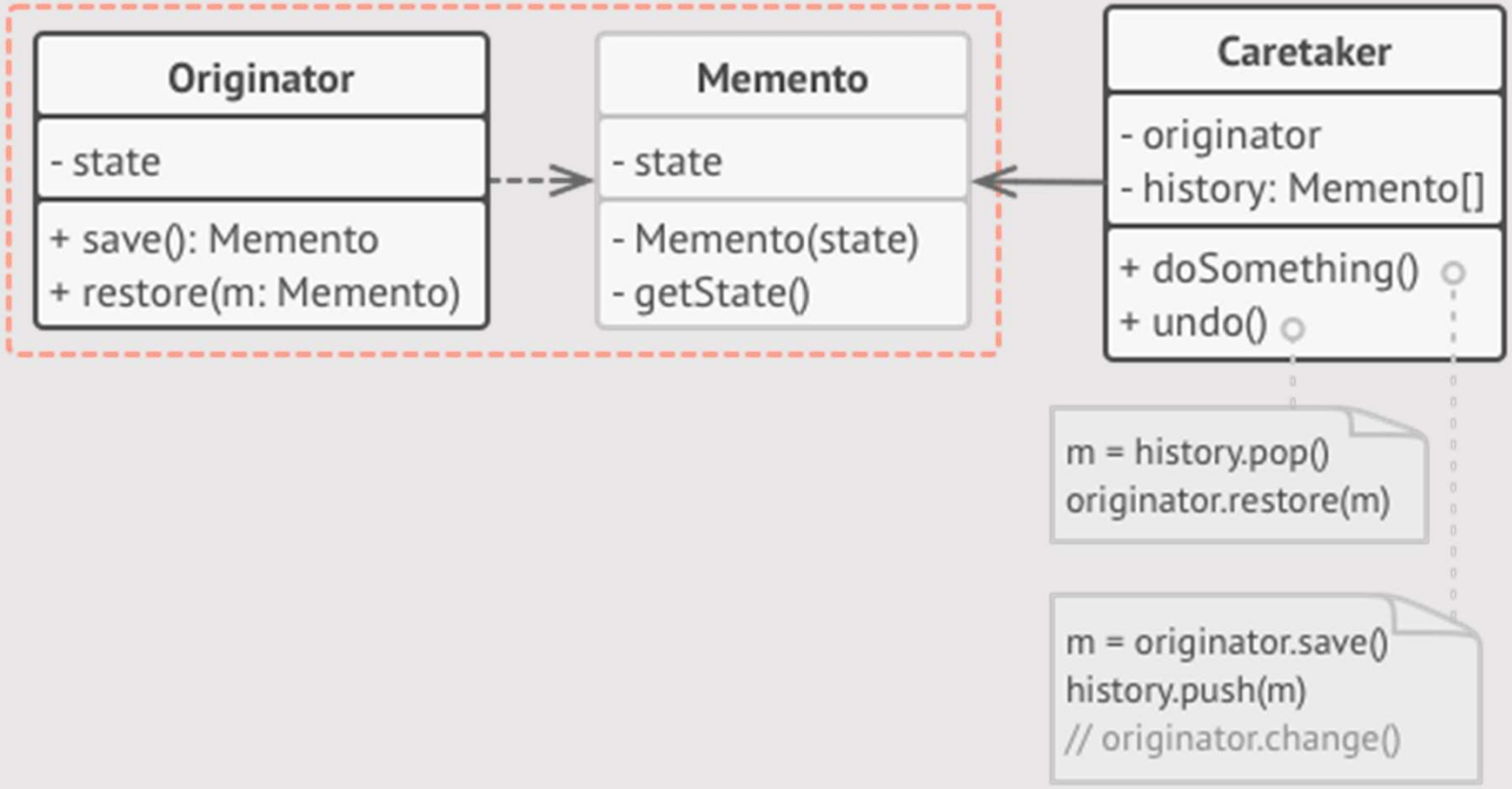
# Посредник - реализация



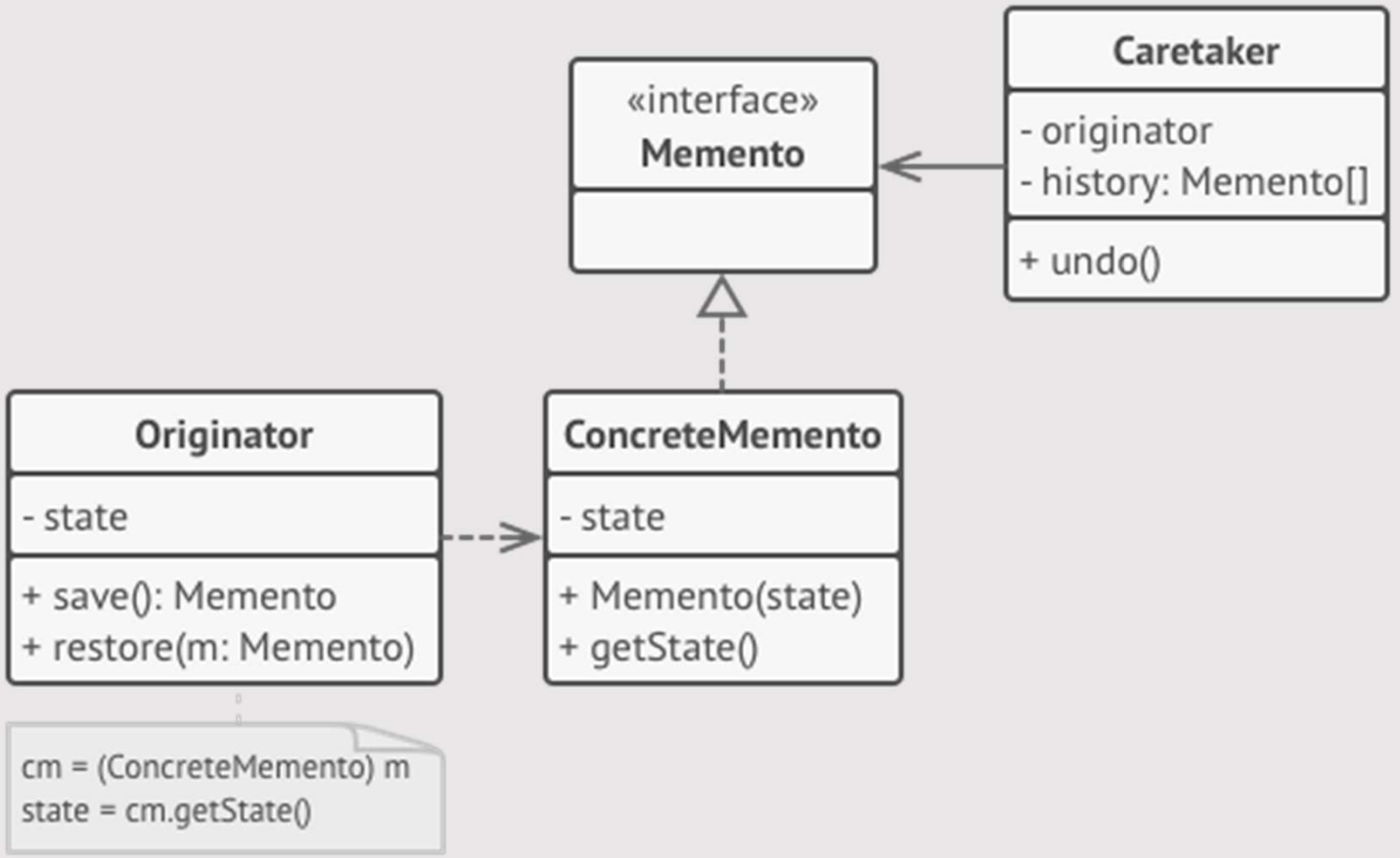
# Хранитель (Снимок) - Memento

- Позволяет сохранять и восстанавливать прошлые состояния объектов, не раскрывая подробностей их реализации.
- Поручает создание копии состояния объекта самому объекту, который этим состоянием владеет.
- Предлагает держать копию состояния в специальном объекте-снимке с ограниченным интерфейсом, закрытым для посторонних и открытым для того, кто его создал.
- Классическая реализация паттерна полагается на механизм вложенных классов, который доступен лишь в некоторых языках программирования (C++, C#, Java).
- Реализация с пустым промежуточным интерфейсом подходит для языков, не имеющих механизма вложенных классов (например, PHP).
- Внешняя сущность Опекун может хранить снимке, но не имея доступа к данным, не может их изменить.

# Хранитель – реализация (вложенные классы - C++, C#, Java)



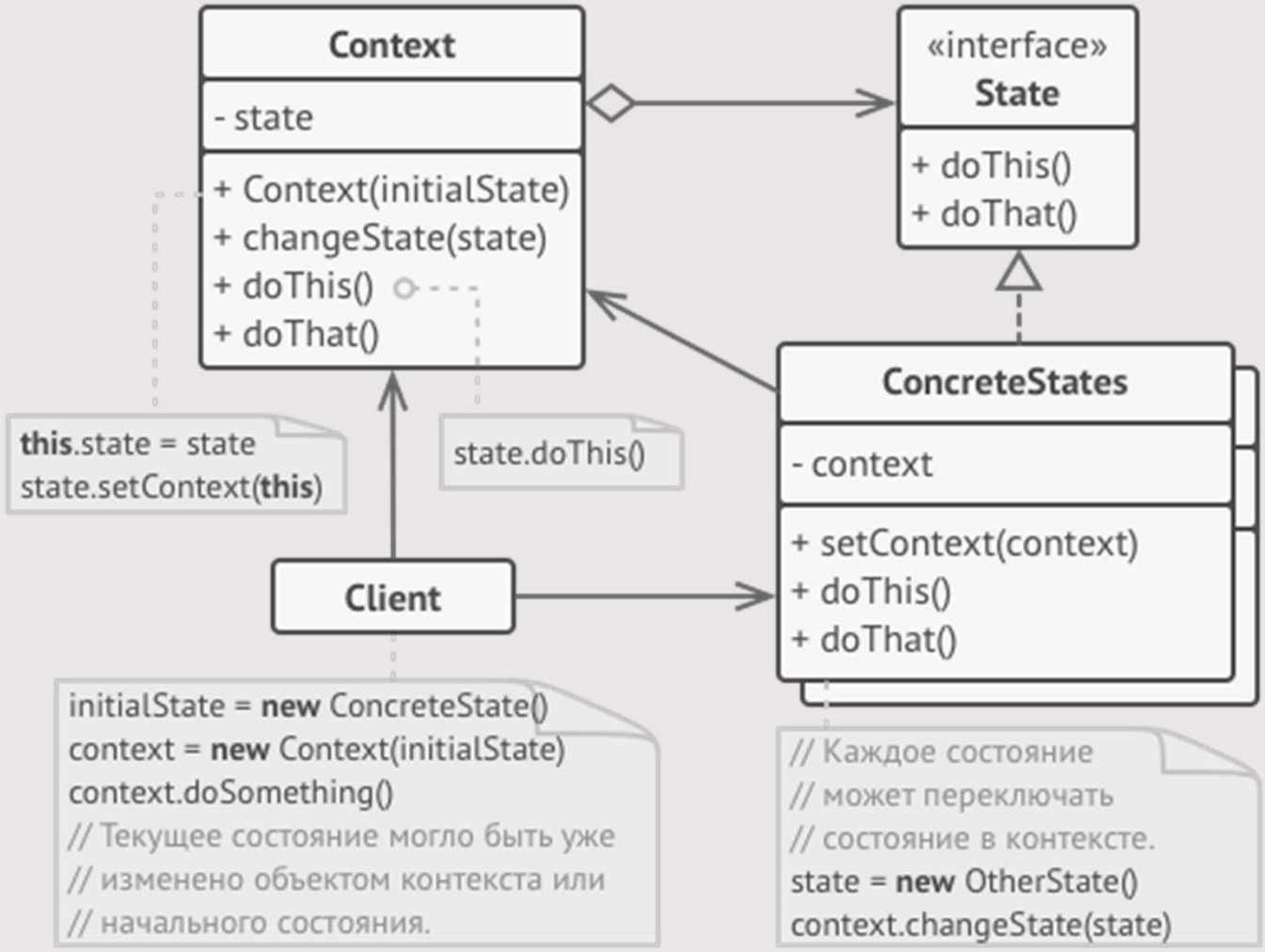
# Хранитель – реализация (пустой промежуточный класс - PHP)



# Состояние - State

- Позволяет объектам менять поведение в зависимости от своего состояния. Извне создаётся впечатление, что изменился класс объекта.
- Реализует машину состояние (state machine) – конечный автомат
- Реализация раскладывается на множество классов-состояний

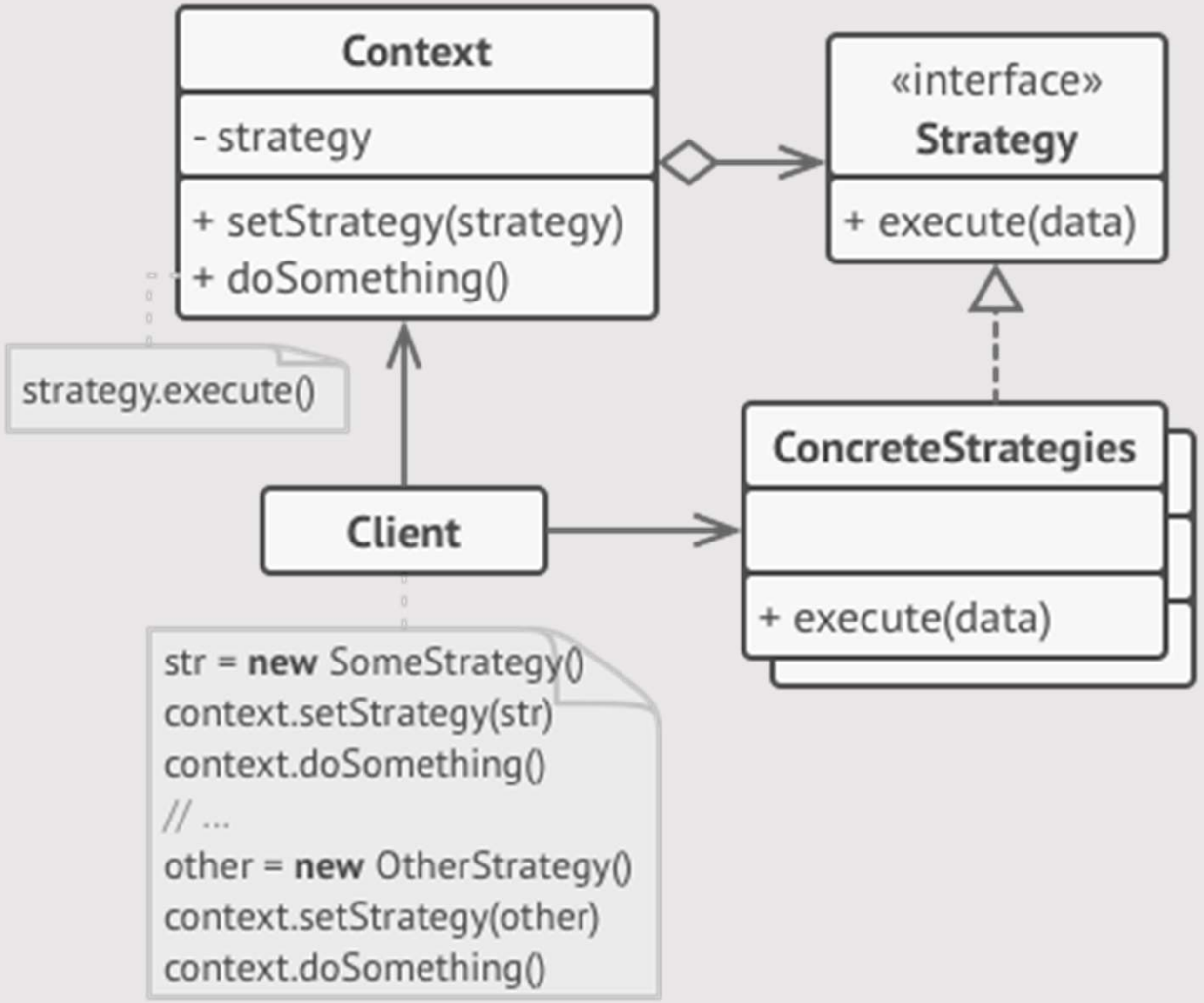
# Состояние - реализация



# Стратегия - Strategy

- Определяет семейство алгоритмов, инкапсулирует каждый из них и делает их взаимозаменяемыми. Стратегия позволяет изменять алгоритмы независимо от клиентов, которые ими пользуются.
- Алгоритмы инкапсулируется классами, которые и являются стратегиями.
- Классы стратегий реализуют общий интерфейс.
- Контекст содержит ссылку на конкретную стратегию и вызывает ей при необходимости.

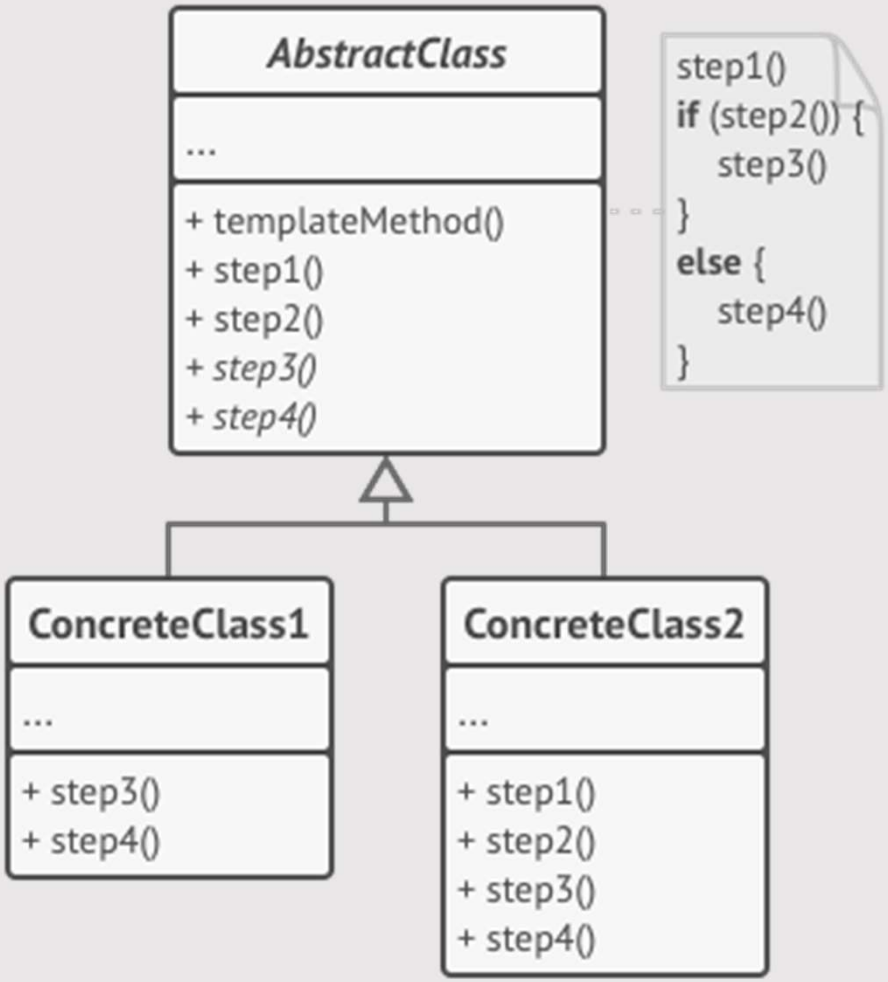
# Стратегия - реализация



# Шаблонный метод – Template Method

- Шаблонный метод определяет основу алгоритма и позволяет подклассам переопределить некоторые шаги алгоритма, не изменяя его структуру в целом.
- Фабричные методы часто вызываются из шаблонных
- Важной целью при проектировании шаблонных методов является сокращение числа примитивных операций, которые должны быть замещены в подклассах.
- Шаблонный метод использует наследование, чтобы расширять части алгоритма. Стратегия использует делегирование, чтобы изменять выполняемые алгоритмы на лету. Шаблонный метод работает на уровне классов. Стратегия позволяет менять логику отдельных объектов.

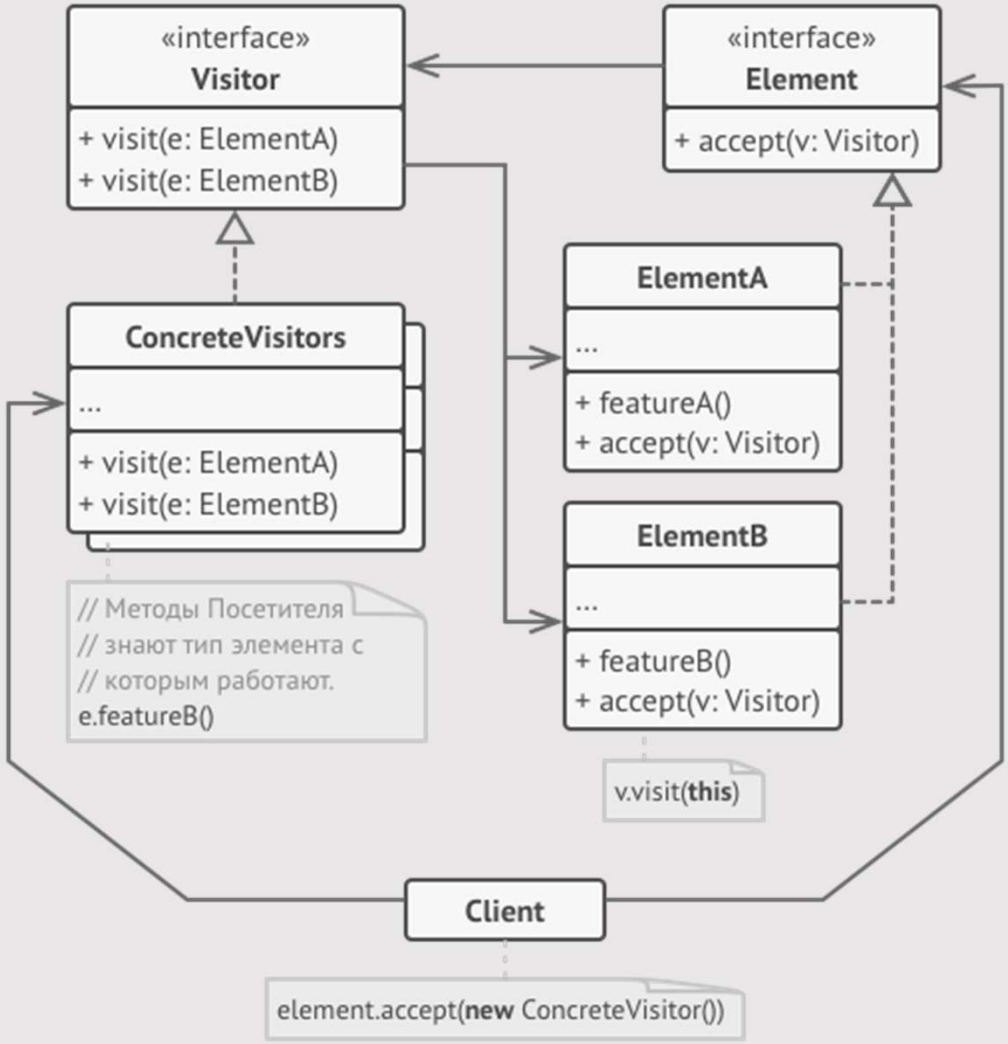
# Шаблонный метод – реализация



# Посетитель - Visitor

- Описывает операцию, выполняемую с каждым объектом из некоторой структуры.
- Позволяет добавлять в программу новые операции, не изменяя (почти) классы объектов, над которыми эти операции могут выполняться.
- Использует механизм двойной диспетчеризации.

# Посетитель - реализация



# Паттерны поведения

- Инкапсуляция вариаций - элемент многих паттернов поведения. Если определенная часть программы подвержена периодическим изменениям, эти паттерны позволяют определить объект для инкапсуляции такого аспекта. Другие части программы, зависящие от данного аспекта, могут кооперироваться с ним. Обычно паттерны поведения определяют абстрактный класс, с помощью которого описывается инкапсулирующий объект. Своим названием паттерн обязан этому объекту.
- Объект **стратегия** инкапсулирует алгоритм
- Объект **состояние** инкапсулирует поведение, зависящее от состояния
- Объект **посредник** инкапсулирует протокол общения между объектами
- Объект **итератор** инкапсулирует способ доступа и обхода компонентов составного объекта.

# Лабораторная 3

- В библиотеке текстовых псевдоокон совместите реализации Цепочки обязанностей и Наблюдателя, используя код из примера.
- Сделайте выгрузку графической сцены используя Посетителя в формате XML

# Вопросы?

# Выбирайте Центр «Специалист» – крупнейший учебный центр России!

info@specialist.ru

+7 (495) 232-32-16