Оглавление

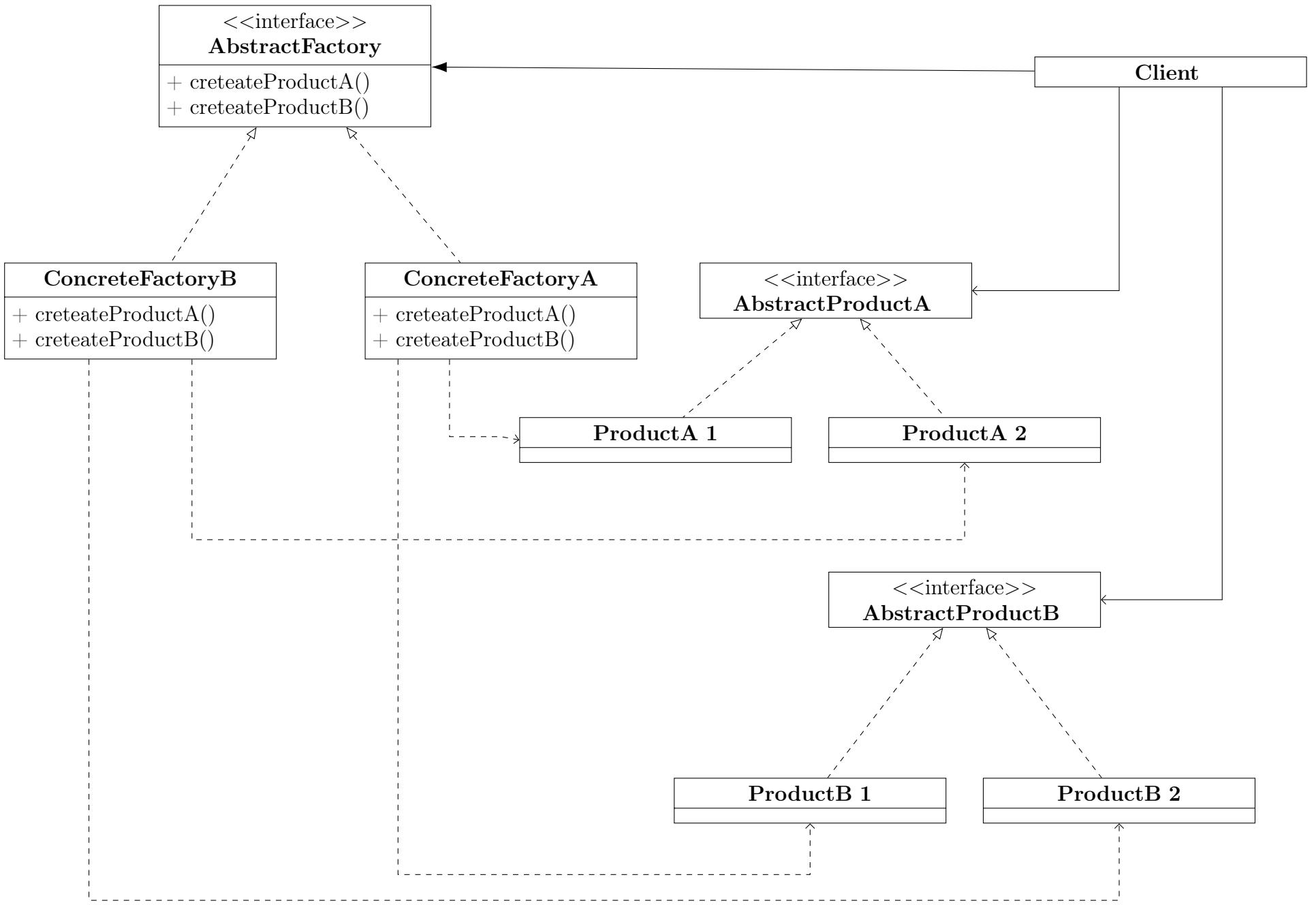
Глава 1

Design Patterns

1.1 Creational patterns

1.1.1 Abstract Factory

Abstract Factory — паттерн, порождающий объекты
Назначение: предоставляет интерфейс для создания семейств взаимосвязанных или взаимозависимых объектов, не специфицируя их конкретных классов.



Описание структуры:
- AbstractFactory — объявляет интерфейс для операций, создающих абстрактные объекты-продукты;
- ConcreteFactory — реализует операции, создающие конкретные объекты продукты;
- AbstractProduct — объявляет интерфейс для типа объекта продукта;
- ConcreteProduct — определяет объект продукт, создаваемый соответствующей конкретной фабрикой;
- Client — пользуется исключительно интерфейсами, которые объявлены в классах AbstractFactory и AbstractProduct.

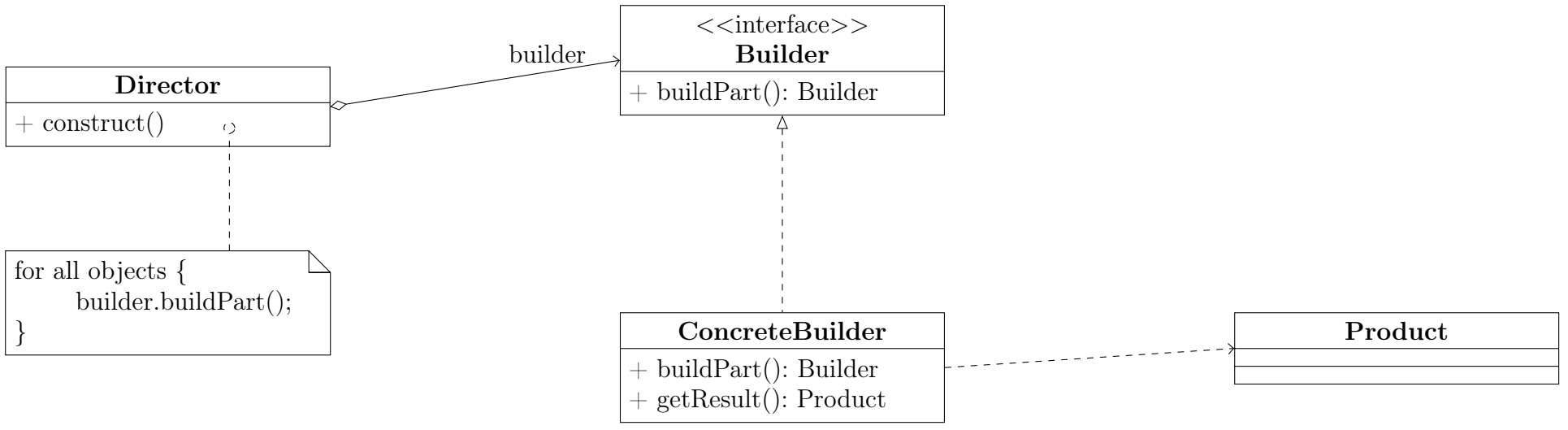
Варианты использования, когда:
- система не должна зависеть от того, как создаются, компилируются и представляются входящие в нее объекты;
- входящие в семейство взаимосвязные объекты должны использоваться вместе и вам необходимо обеспечить выполнение этого ограничения;
- система должна конфигурироваться одним из семейств составляющих ее объектов;
- вы хотите предоставить библиотеку объектов, раскрывая только их интерфейсы, но не реализацию.

Отношения
- Обычно во время выполнения создается единственный экземпляр класса ConcreteFactory. Эта конкретная фабрика создает объекты продукты, имеющие вполне определенную реализацию. Для создания других видов объектов клиент должен воспользоваться другой конкретной фабрикой;
- AbstractFactory передоверяет создание объектов продуктов своему под классу ConcreteFactory.

Плюсы/Минусы
+ изолирует конкретные классы
+ упрощает замену семейств продуктов
+ гарантирует сочетаемость продуктов
- поддержать новый вид продуктов трудно

1.1.2 Builder

Builder — паттерн, порождающий объекты.
Назначение: отделяет конструирование сложного объекта от его представления, так что в результате одного и того же процесса конструирования могут получаться разные представления.



Описание структуры:
- Builder — задает абстрактный интерфейс для создания частей объекта Product;
- ConcreteBuilder —
- Director — конструирует объект, пользуясь интерфейсом Builder;
- Product — представляет сложный конструируемый объект. ConcreteBuilder строит внутреннее представление продукта и определяет процесс его сборки;

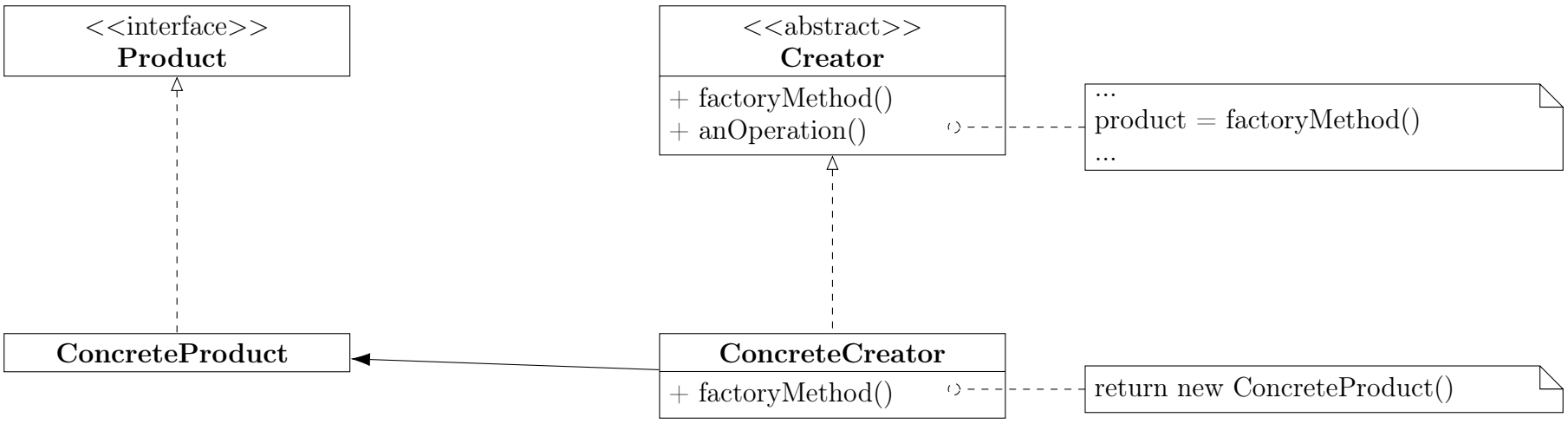
Варианты использования, когда:
- алгоритм создания сложного объекта не должен зависеть от того, из каких частей состоит объект и как они стыкуются между собой;
- процесс конструирования должен обеспечивать различные представления конструируемого объекта.

Отношения
- клиент создает объект распорядитель Director и конфигурирует его нужным объектом строителем Builder;
- распорядитель уведомляет строителя о том, что нужно построить очередную часть продукта;
- строитель обрабатывает запросы распорядителя и добавляет новые части к продукту;
- клиент забирает продукт у строителя.

Плюсы/Минусы
+ позволяет изменять внутреннее представление продукта;
+ изолирует код, реализующий конструирование и представление;
+ дает более тонкий контроль над процессом конструирования;

1.1.3 Factory method

Factory method — паттерн, порождающий классы..
Назначение: Определяет интерфейс для создания объекта, но оставляет подклассам решение о том, какой класс инстанцировать. Factory method позволяет классу делегировать инстанцирование подклассам.



Описание структуры:
- Product — определяет интерфейс объектов, создаваемых фабричным методом;
- ConcreteProduct — реализует интерфейс Product;
- Creator —
- ConcreteCreator — замещает фабричный метод, возвращающий объект ConcreteProduct.

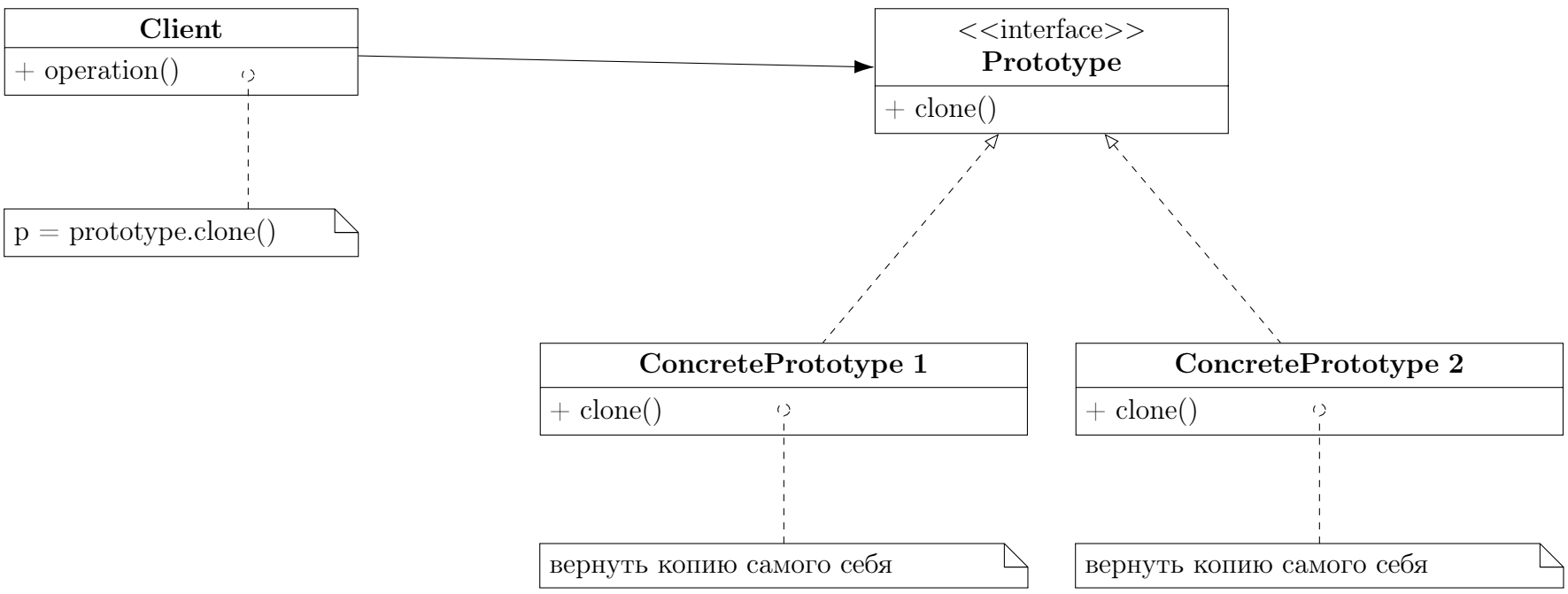
Варианты использования, когда:
- классу заранее неизвестно, объекты каких классов ему нужно создавать;
- класс спроектирован так, чтобы объекты, которые он создает, специфицировались подклассами;
- класс делегирует свои обязаности одному из нескольких вспомагательных подклассов, и вы планиурете локализовать знание о том, какой класс принимает эти обязаности на себя.

Отношения
- Создатель «полагается» на свои подклассы в определении фабричного метода, который будет возвращать экземпляр подходящего конкретного продукта.

Плюсы/Минусы
+ предоставляет подклассам операции зацепки (hooks);
+ соединяет параллельные иерархии;

1.1.4 Prototype

Prototype — паттерн, порождающий объекты.
Назначение: задает виды создаваемых объектов с помощью экземпляра прототипа и создает новые объекты путем копирования этого прототипа.



Описание структуры:
- Prototype — объявляет интерфейс для клонирования самого себя;
- ConcretePrototype — реализует операцию клонирования себя;
- Client — создает новый объект, обращаясь к прототипу с запросом клонировать себя.

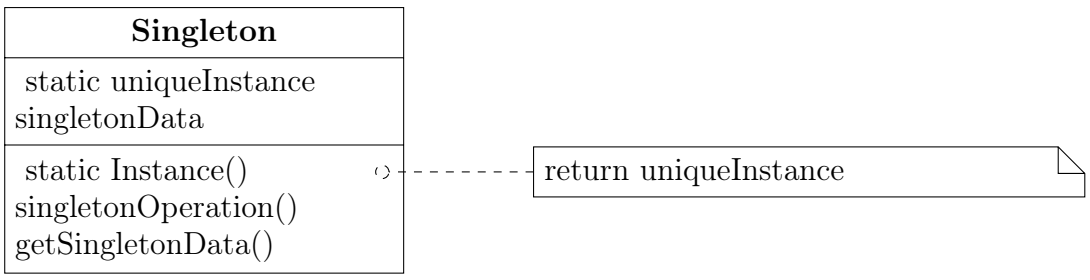
Варианты использования, когда система не должна зависеть от того, как в ней создаются, компонуются и представляются продукты:
- инстацируемые классы определяются во время выполнения, например с помощью динамической загрузки;
- для того чтобы избежать построения иерархий классов или фабрик, паралельных иерархий классов продуктов;
- экземпляры класса могут находиться в одном из не очень большого числа различных состояний. Может оказаться удобнее установить соответствующее число прототипов и клонировать их, а не инстанцировать каждый раз класс вручную в подходящем состоянии.

Отношения
- Клиент обращается к прототипу, чтобы тот создал свою копию.

Плюсы/Минусы
+ добавление и удаление продуктов во время выполнения
+ спецификация новых объектов путем изменения значений
+ специфицирование новых объектов путем изменения структуры
+ уменьшение числа подклассов
+ динамическое конфигурирование приложения классами

1.1.5 Singleton

Singleton — паттерн, порождающий объекты.
Назначение: гарантирует, что у класса есть только один экземпляр, и предоставляет к нему глобальную точку доступа.



Описание структуры:
- Singleton — определяет операцию Instance, которая позволяет клиентам получать доступ к единственному экземпляру.

Варианты использования, когда:
- должен быть ровно один экземпляр некоторого класса, легко доступный всем клиентам;
- единственный экземпляр должен расширяться путем порождения подклассов, и клиентам нужно иметь возможность работать с расширенными экземпляром без модификации своего кода.

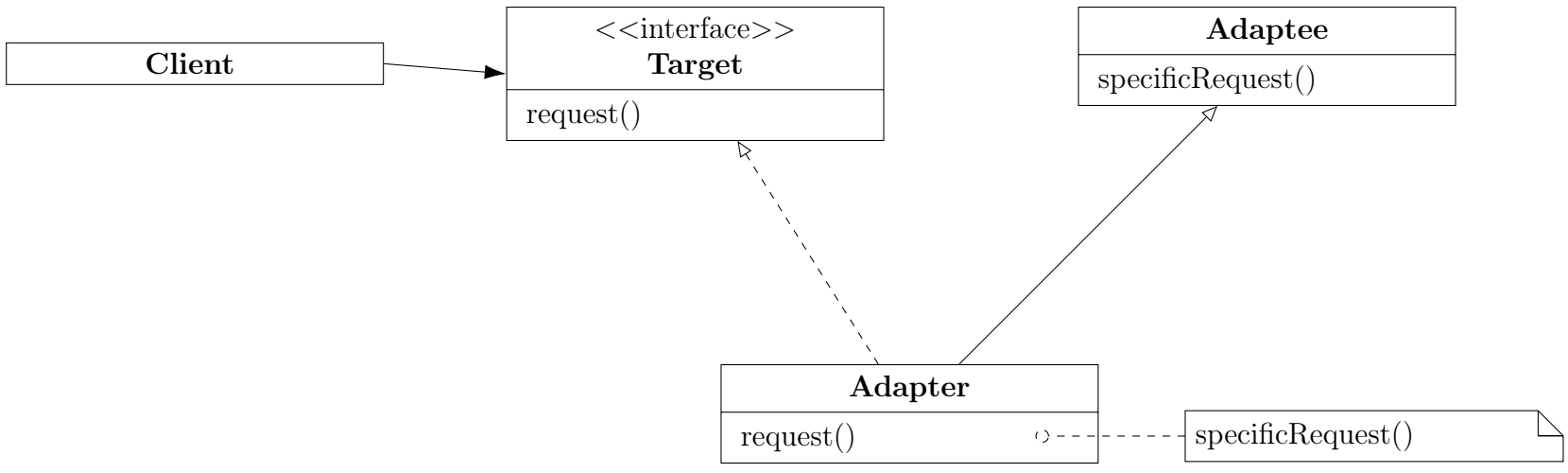
Отношения
- Клиенты получают доступ к экземпляру класса Singleton только через его операцию Instance.

Плюсы/Минусы
+ контролируемый доступ к единственному экземпляру;
+ уменьшение числа имен;
+ допускает уточнение операций и представления;
+ допускает переменное число экземпляров;
+ большая гибкость, чем у операций класса.

1.2 Structural patterns

1.2.1 Adapter

Adapter — паттерн, структурирующий классы и объекты.
Назначение: преобразует интерфейс одного класса в интерфейс другого, который ожидают клиенты. Адаптер обеспечивает совместную работу классов с несовместимыми интерфейсами, которая без него была бы невозможна.



Описание структуры:
- Target — определяет зависящий от предметной области интерфейс, которым пользуется Client;
- Client — вступает во взаимоотношения с объектами, удовлетворяющими интерфейсу Target;
- Adaptee — определяет существующий интерфейс, который нуждается в адаптации;
- Adapter — адаптирует интерфейс Adaptee к интерфейсу Target.

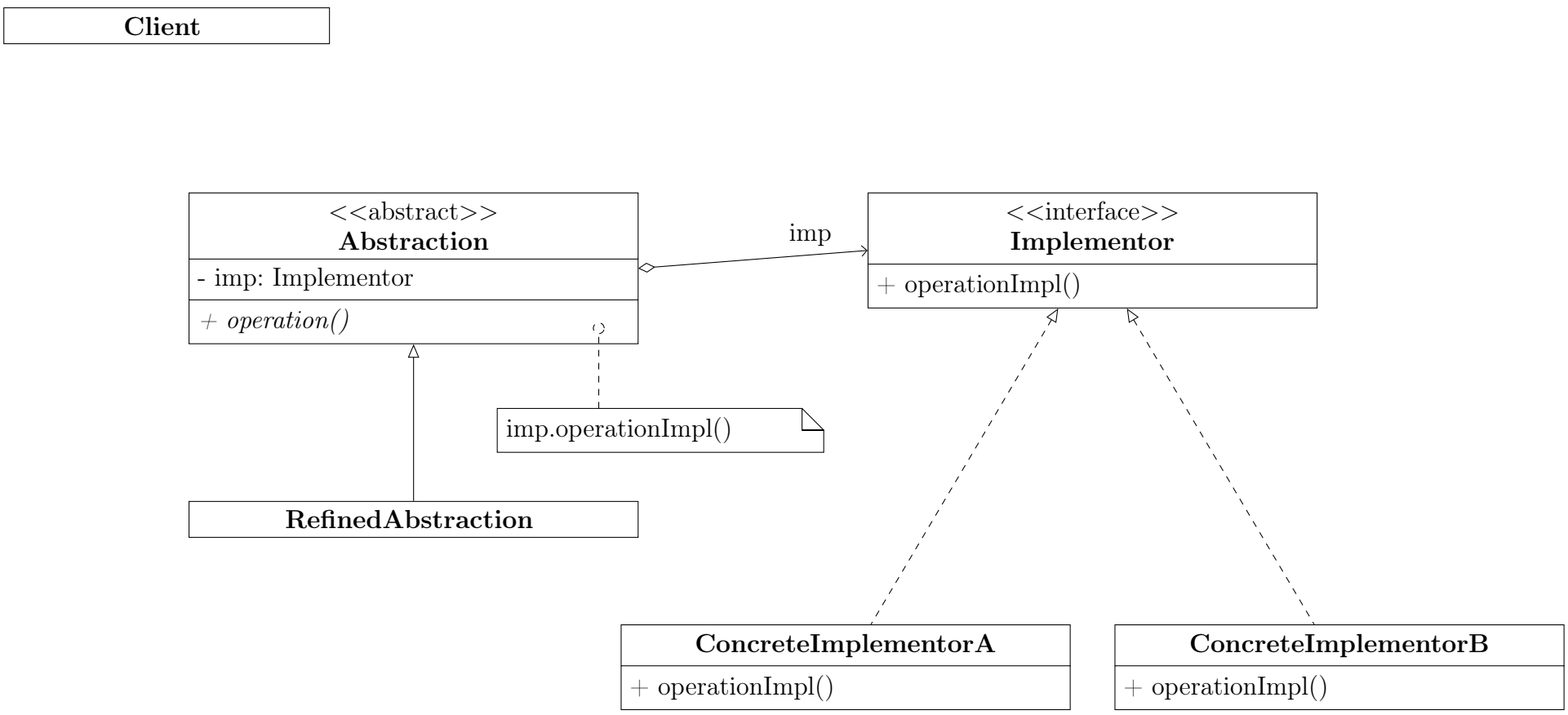
Варианты использования, когда:
- хотите использовать существующий класс, но его интерфейс не соответствует вашим потребностям;
- собираетесь создать повторно используемый класс, который должен взаимодействовать с заранее неизвестными или не связаными с ними классами, имеющими несовместимые интерфейсы;
- (только для адаптера объектова) нужно использовать несколько существующих подклассов, но непрактично адаптировать их интерфейсы путем порождения новых подклассов от каждого. В этом случае адаптер объектов может приспосабливать интерфейс их общего родительского класса.

Отношения
- Клиенты вызывают операции экземпляра адаптера Adapter. В свою очередь адаптер вызывает операции адаптируемого объекта или класса Adaptee, который и выполняет запрос.

Плюсы/Минусы
SHIIIT BOISHHHH

1.2.2 Bridge

Bridge — паттерн, структурирующий объекты.
Назначение: Отделить абстракцию от ее реализации так, чтобы то и другое можно было изменять независимо.



Описание структуры:
- Abstraction —
- RefinedAbstraction —
- Implementor —
- ConcreteImplementor —

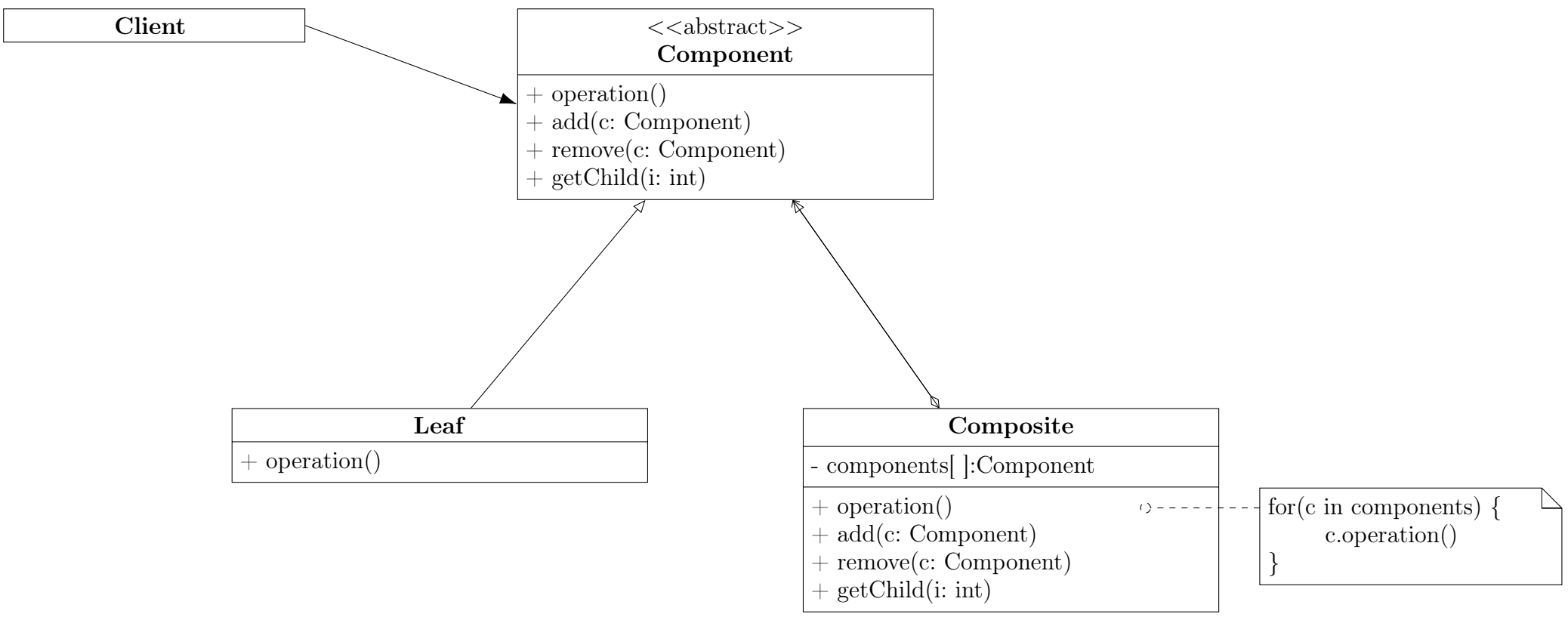
Варианты использования, когда:
- хотите избежать постоянной привязки абстракции к реализации. Так, например, бывает, когда реализацию необходимо выбрать во время выполнения программы;
- и абстракции, и реализации должны расширяться новыми подклассами. В таком случае паттерн позволяет комбинировать разные абстракции и реализации и изменять их независимо;
- изменения в реализации абстракиции не должны сказываться на клиентах, то есть клиентский код не должен перекомпилироваться;
- (только C++!) вы хотите полностью скрыть от клиентов реализацию абстракции. В C++ представление класса видимо через его интерфейс;
- число классов начинает быстро расти. Это признак того, что иерархию следует разделить на две части. Для таких иерархий классов Рамбо() использует термин «вложенные обобщения» ;
- вы хотите разделить одну реализацию между несколькими объектами(быть может, применяя подсчет ссылок), и этот факт необходимо скрыть от клиента.

Отношения
- Объект Abstraction перенаправляет своему объекту Implementor запросы клиента.

Плюсы/Минусы
- \+ отделение реализации от интерфейса;
- \+ повышение степени расширяемости;
- \+ сокрытие деталей реализации от клиентов.

1.2.3 Composite

Composite — паттерн, структурирующий объекты.
Назначение: Компонует объекты в древовидные структуры для представления иерархий часть целое. Позволяет клиентам единообразно трактовать индивидуальные и составные объекты.



Описание структуры:
- Component —
- Leaf —
- Composite —
- Client — манипулирует объектами композиции через интерфейс Component.

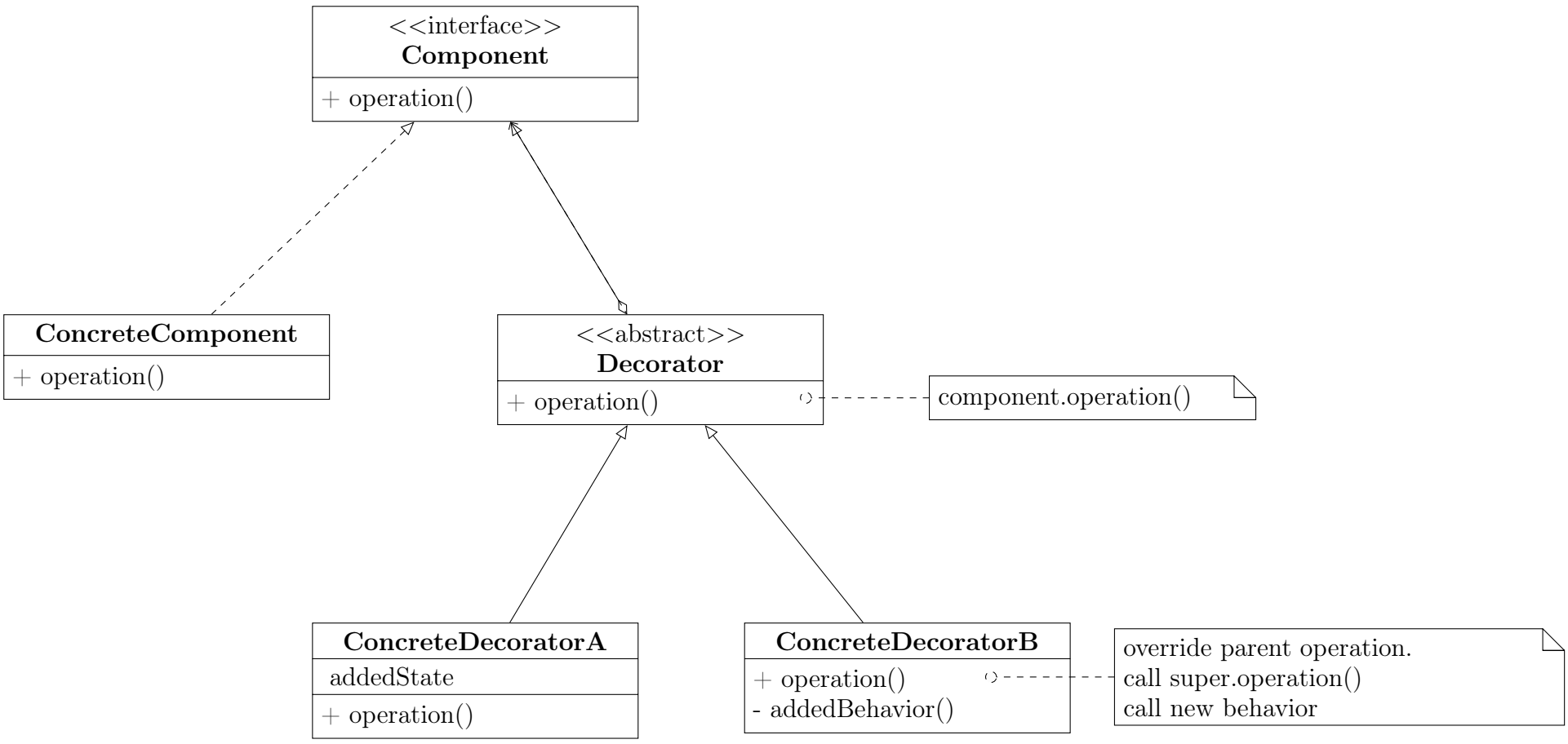
Варианты использования, когда:
- нужно представить иерархию объектов вида часть-целое;
- хотите, чтобы клиенты единообразно трактовали составные и индивидуальные объекты.

Отношения
- Клиенты используют интерфейс класса Component для взаимодействия с объектами в составной структуре. Если получателем запроса является листовый объект Leaf, то он и обрабатывает запрос. Когда же получателем является составной объект Composite, то обычно он перенаправляет запрос своим потомкам, возможно, выполняя некоторые дополнительные операции до или после перенаправления.

Плюсы/Минусы
+ определяет иерархии классов, состоящие из примитивных и составных объектов;
+ упрощает архитектуру клиента;
+ облегчает добавление новых видов компонентов;
- способствует созданию общего дизайна.

1.2.4 Decorator

Decorator — паттерн, структурирующий объекты.
Назначение: Динамически добавляет объекту новые обязанности. Является гибкой альтернативой порождению подклассов с целью расширения функциональности.



Описание структуры:
- Component — определяет интерфейс для объектов, на которые могут быть динамически возложены дополнительные обязанности;
- ConcreteComponent — определяет объект, на который возлагаются дополнительные обязанности;
- Decorator — хранит ссылку на объект Component и определяет интерфейс, соответствующий интерфейсу Component;
- ConcreteDecorator — возлагает дополнительные обязанности на компонент.

Варианты использования:
- для динамического, прозрачного для клиентов добавления обязанностей объектам;
- для реализации обязаностей, которые могут быть сняты с объекта;
- когда расширение путем порождения подклассов по каким-то причинами неудобно или невозможно. Иногда приходится реализовывать много независимых расширений, так что порождение подклассов для поддержки всех возможных комбинаций приведет к комбинаторному росту их числа. В других случаях определение класса может быть скрыто или почему-либо еще недоступно, так что породить от него подкласс нельзя.

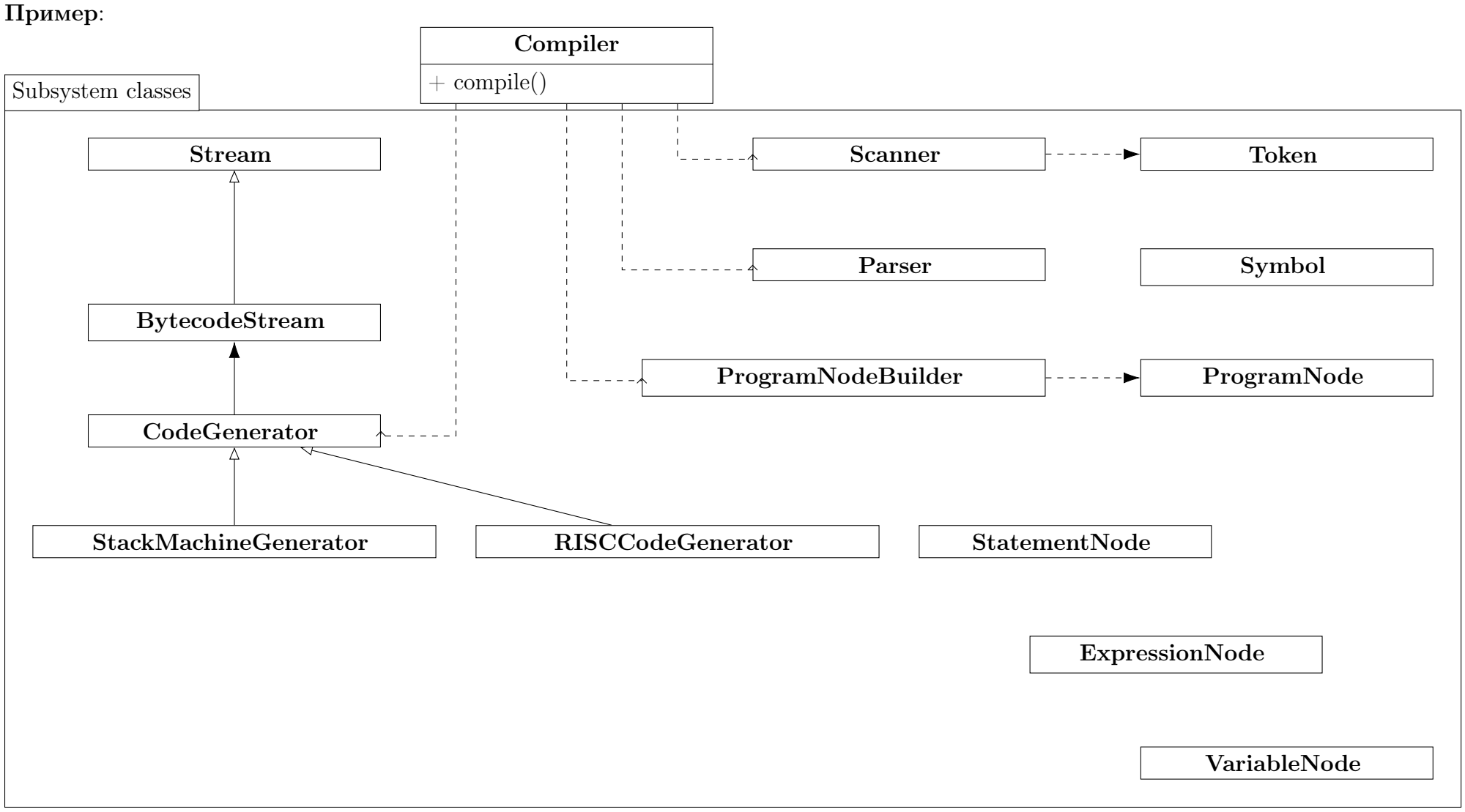
Отношения
- Decorator переадресует запросы объекту Component. Может выполнятьи дополнительные операции до и после переадресации.

Плюсы/Минусы
+ большая гибкость, нежели у статического наследования
+ позволяет избежать перегруженных функциями классов на верхних уровнях иерархии
  декоратор и его компонент не идентичны
- множество мелких объектов

1.2.5 Facade

Facade — паттерн, структурирующий объекты.
Назначение: Предоставляет унифицированный интерфейс вместо набора интерфейсов некоторой подсистемы. Фасад определяет интерфейс более высокого уровня, который упрощает использование подсистемы.

Пример:



Описание структуры:
- Facade —
- Subsystem classes —

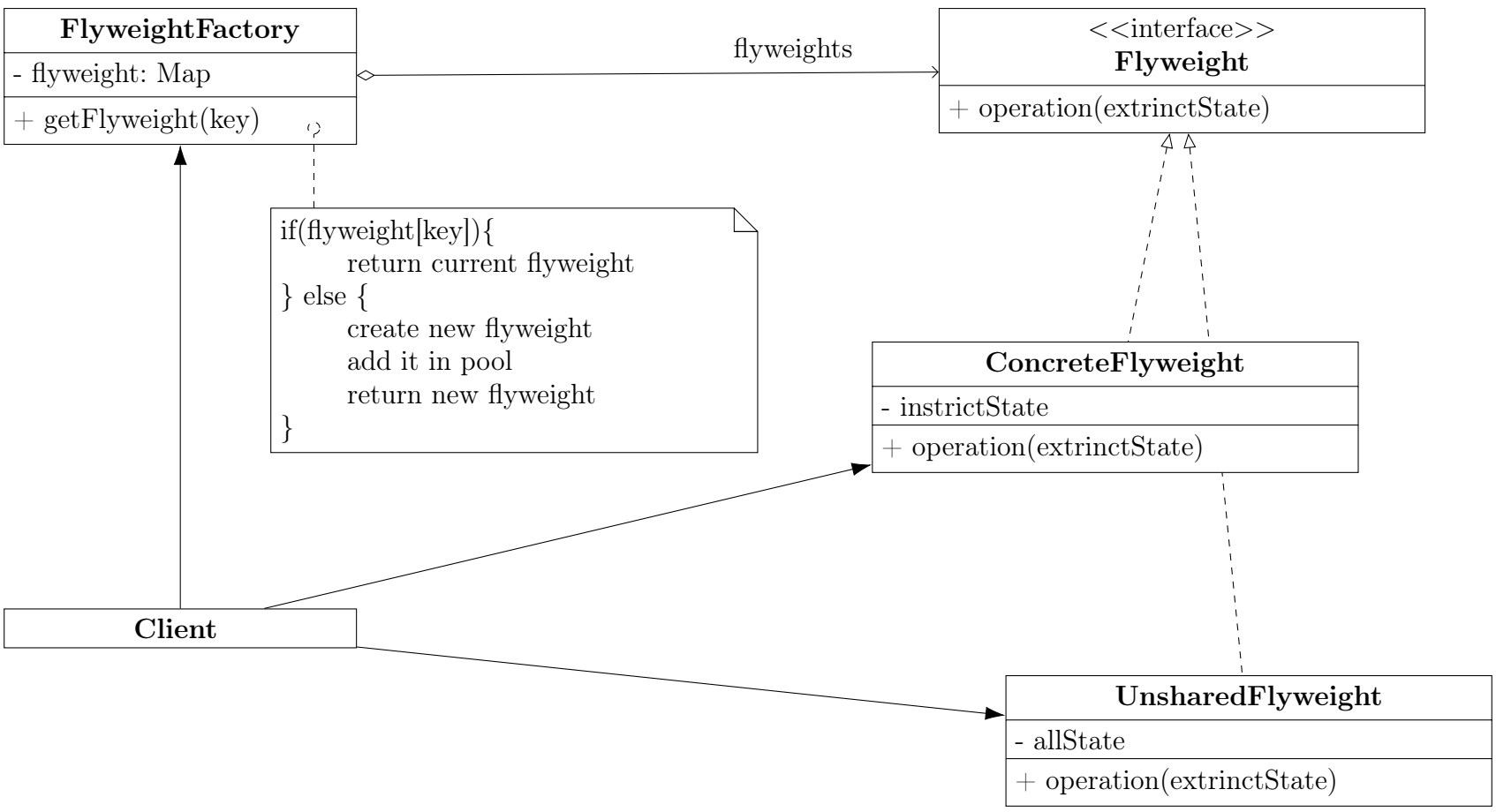
Варианты использования, когда:
- хотите предоставить простой интерфейс к сложной подсистеме. Часто подсистемы усложняются по мере развития. Применение большинства паттернов приводит к появлению меньших классов, но в большем количестве. Такую подсистему проще повторно использовать и настраивать под конкретные нужды, но вместе с тем применять подсистему настройки становится труднее. предлагает некоторый вид системы по умолчанию, устраивающий большинство клиентов. И лишь те объекты, которым нужны более широкие возможности настройки, могут обратиться напрямую к тому, что находится за фасадом;
- между клиентами и классами реализации абстракции существует много зависимостей. позволит отделить подсистему как от клиентов, так и от других подсистем, что, в свою очередь, способствует повышению степени независимости и переносимости;
- вы хотите разложить подсистему на отдельные слои. Используйте 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.

Отношения
- Клиенты общаются с подсистемой, посылая запросы фасаду. Он переадресует их подходящим объектам внутри подсистемы. Хотя основную работу выполняют именно объекты подсистемы, фасаду, возможно, придется преобразовать свой интерфейс в интерфейсы подсистемы.
- Клиенты, пользующиеся фасадом, не имеют прямого доступа к объектам под системы.

Плюсы/Минусы
- \+ изолирует клиентов от компонентов подсистемы
- \+ позволяет ослабить связанность между подсистемой и ее клиентами
- \+ фасад не препятствует приложениям напрямую обращаться к классам подсистемы, если это необходимо

1.2.6 Flyweight

Flyweight — паттерн, структурирующий объекты.
Назначение: Использует разделение для эффективной поддержки множества мелких объектов.



Описание структуры:
- Flyweight —
- ConcreteFlyweight —
- UnsharedConcreteFlyweight —
- FlyweightFactory —
- Client —

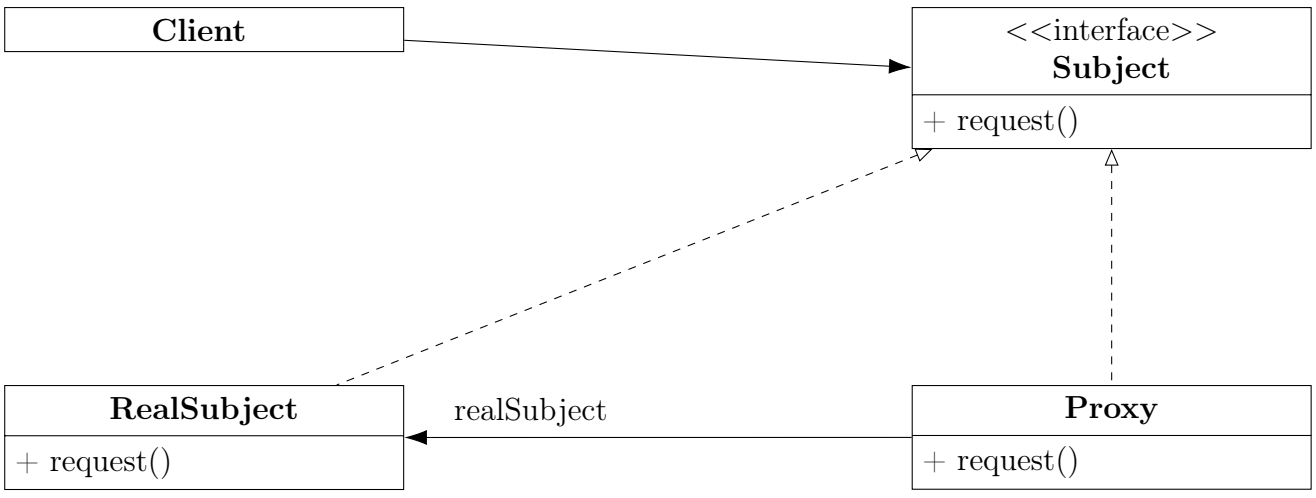
Варианты использования, когда все из ниже перечисленого присутствует:
- в приложении используется большое число объектов;
- из-за этого накладные расходы на хранение высоки;
- большую часть состояния объектов можно вынести вовне;
- многие группы объектов можно заменить относительно небольшим количеством разделяемых объектов, поскольку внешнее состояние вынесено;
- приложение не зависит от идентичности объекта. Поскольку объекты-приспособленцы могут разделяться, то проверка на идентичность возвратит «истину» для концептуально различных объектов.

Отношения
- состояние, необходимое приспособленцу для нормальной работы, можно охарактеризовать как внутреннее или внешнее. Первое хранится в самом объекте ConcreteFlyweight. Внешнее состояние хранится или вычисляется клиентами. Клиент передает его приспособленцу при вызове операций;
- клиенты не должны создавать экземпляры класса ConcreteFlyweight напрямую, а могут получать их только от объекта FlyweightFactory. Это позволит гарантировать корректное разделение.

Плюсы/Минусы

1.2.7 Proxy

Proxy — паттерн, структурирующий объекты.
Назначение: является суррогатом другого объекта и контролирует доступ к нему.



Описание структуры:

- Proxy —
- Subject —
- RealSubject —

Варианты использования:

- *виртуальный* proxy создает 'тяжелые' объекты по требованию;
- *защитный* proxy контролирует доступ к исходному объекту. Такие заместители полезны, когда для разных объектов определены различные права доступа;
- *удаленный* proxy предоставляет локального представителя вместо объекта, находящегося в другому адресном пространстве(Джеймс Коплиен называет такой proxy «послом»);
- *умный/интеллектуальный* proxy — это замена обычного указателя. Она позволяет выполнить дополнительные действие при доступе к объекту.

Отношения

- Proxy при необходимости переадресует запросы объекту RealSubject. Детали зависят от вида заместителя

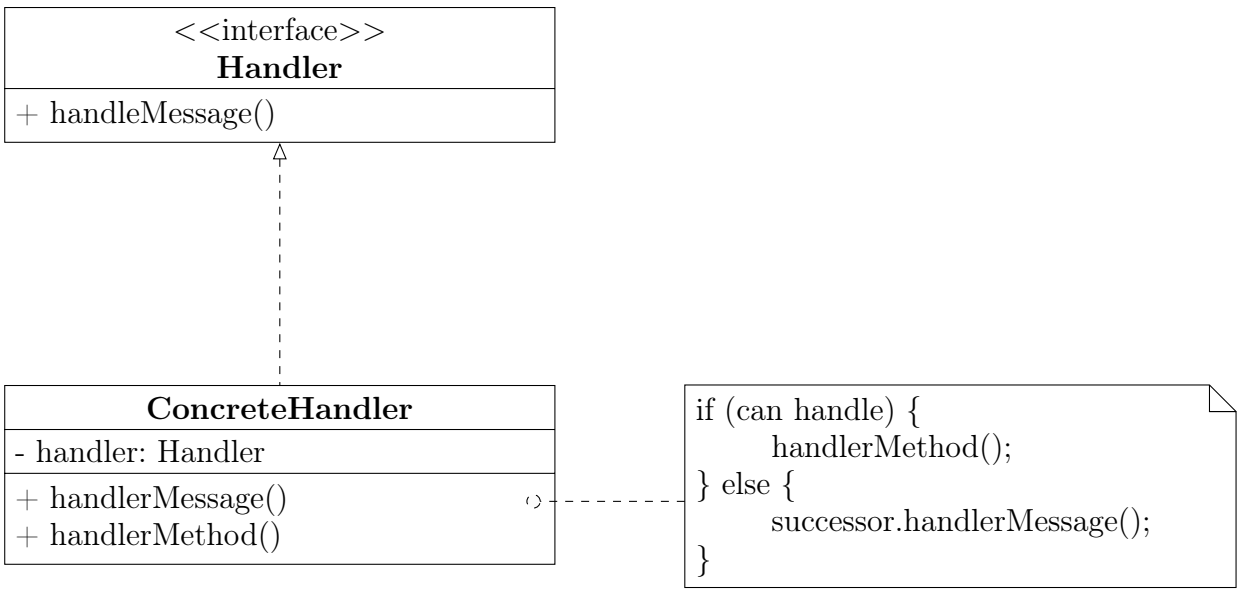
Плюсы/Минусы

SHIIIT BOIIISHHH

1.3 Behavioral patterns

1.3.1 Chain of responsibility

Chain of responsibility — паттерн поведения объектов.
Назначение: Позволяет избежать привязки отправителя запроса к его получателю, давая шанс обработать запрос нескольким объектам. Связывает объекты получатели в цепочку и передает запрос вдоль этой цепочки, пока его не обработают.



Описание структуры:
- Handler —
- ConcreteHandler —
- Client —

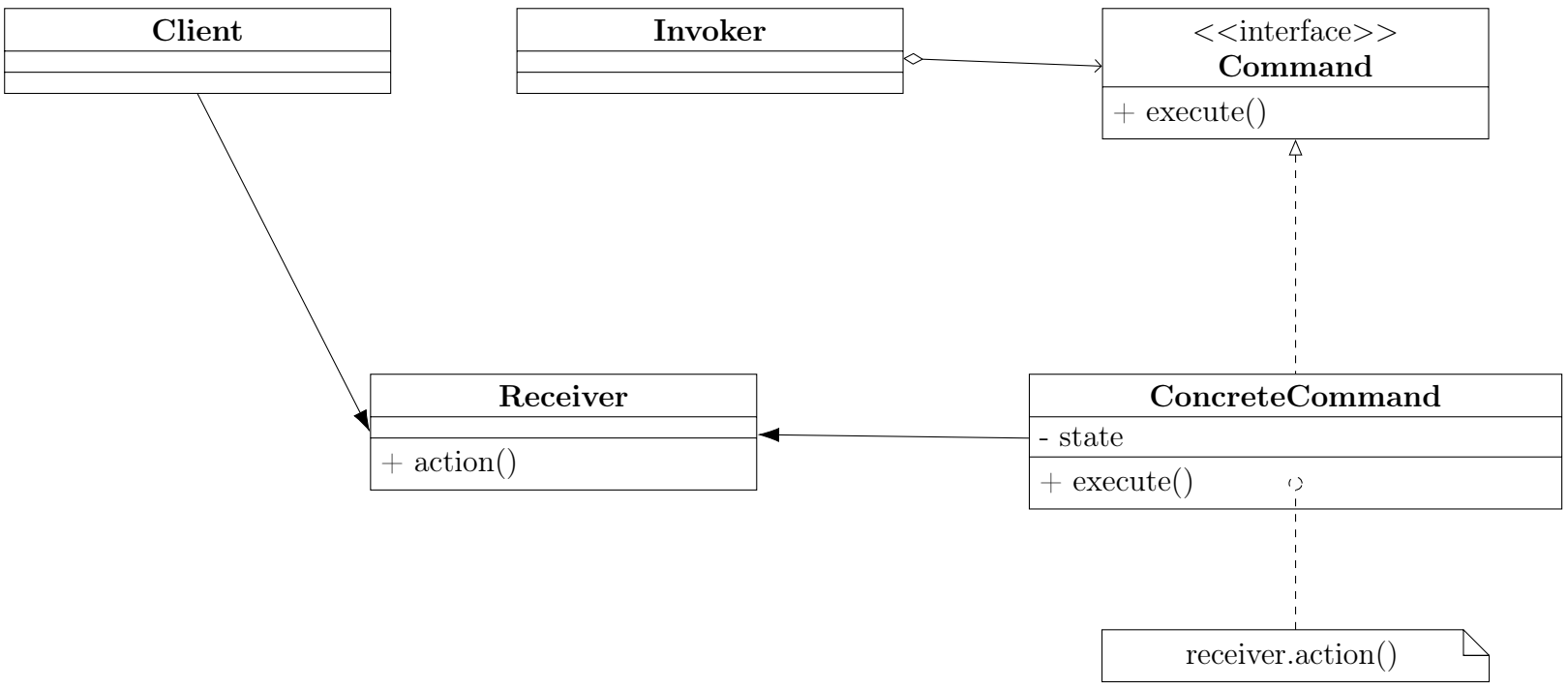
Варианты использования:
- есть более одного объекта, способного обработать запрос, причем настоящий обработчик заранее неизвестен и должен быть найден автоматически;
- вы хотите отправить запрос одному из нескольких объектов, не указывая явно, какому именно;
- набор объектов, способных обработать запрос, должен задаваться динамически.

Отношения
- Когда клиент инициирует запрос, он продвигается по цепочке, пока некоторый объект ConcreteHandler не возьмет на себя ответственность за его обработку.

Плюсы/Минусы
+ ослабление связанности
+ дополнительная гибкость при распределении обязанностей между объектами
- получение не гарантировано

1.3.2 Command

Command — паттерн поведения объектов.
Назначение: 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.



Описание структуры:
- Command
- ConcreteCommand
- Client
- Invoker
- Receiver

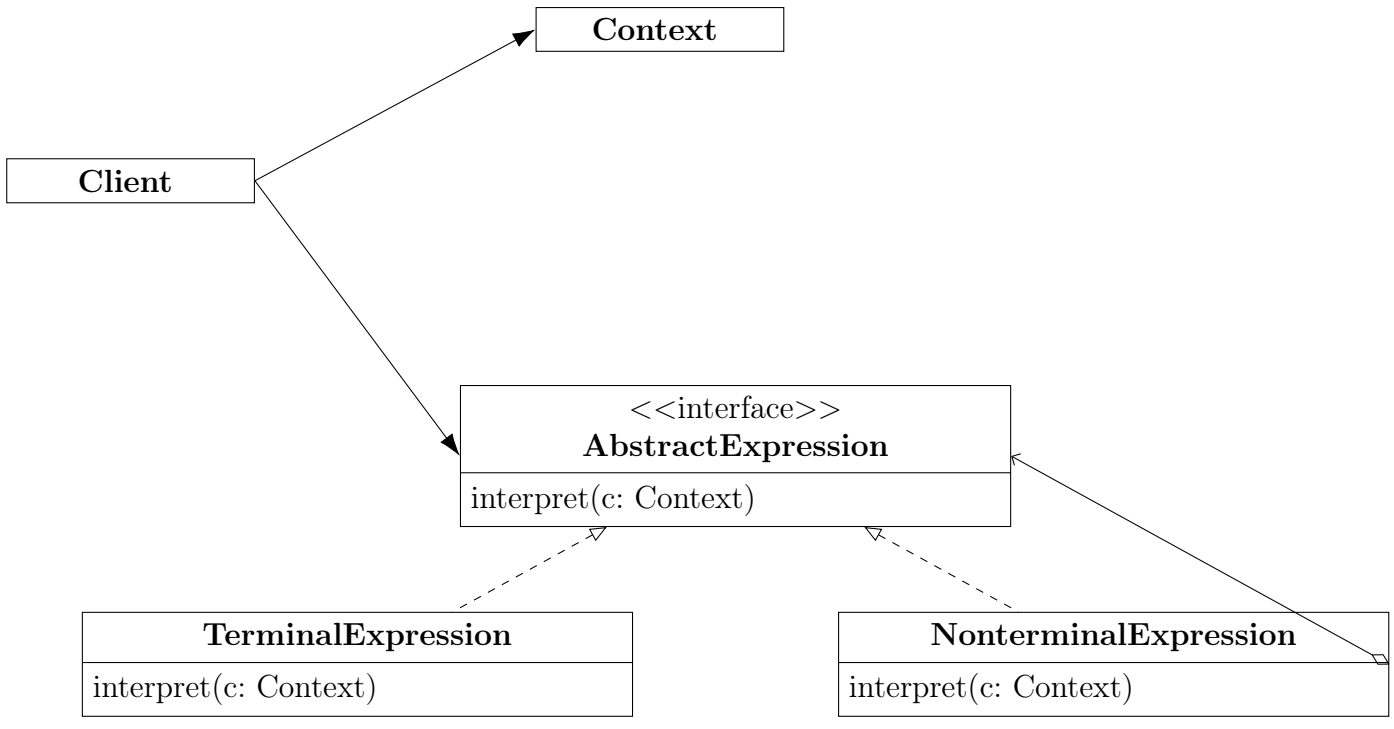
Варианты использования, когда хотите:
- параметризовать объекты выполняемым действием, как в случае с пунктами меню **MenuItem**. В процедурном языке такую параметризацию можно выразить с помощью функции обратного вызова, то есть такой функции, которая регистрируется, чтобы быть вызванной позднее. Команды представляют собой объектно-ориентированную альтернативу функциям обратного вызова;
- 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 удается реализовать так, чтобы он не зависел от адресного пространства, то объект-команду можно передать другому процессу, который займется его выполнением;
- поддержать отмену операций. Операция **Execute** объекта **Command** может сохранить состояние, необходимиое 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 нужно обходить этот список соответственно в обратном и прямом направлениях, вызывая при посещении каждого элемента команду **Unexecute** или **Execute**;
- поддержать протоколирование изменений, чтобы их можно было выполнить повторно после аварийной остановки системы. Дополнив интерфейс класса **Command** операциями сохранения и загрузки, вы сможете вести протокол изменений во внешней памяти. Для восстановления после сбоя нужно будет загрузить сохраненные команды с диска и повторно выполнить их с помощью операции **Execute**;
- структурировать систему на основе высокоуровневых операций, построеных из примитивных. Такая структура типична для информационных систем, поддерживающих транзакции. Транзакция инкапсулирует набор изменений данных. Паттерн Command позволяте моделировать транзакции. У всех команд есть общий интрефейс, что дает возможность работать одинаково с любыми транзакциями. С помощью этого паттерна можно легко добавлять в систему новые виды транзакций.

Отношения
- клиент создает объект ConcreteCommand и устанавливает для него получателя;
- инициатор Invoker сохраняет объект ConcreteCommand;
- инициатор отправляет запрос, вызывая операцию команды Execute. Если поддерживается отмена выполненных действий, то ConcreteCommand перед вызовом Execute сохраняет информацию о состоянии, достаточную для выполнения отката;
- объект ConcreteCommand вызывает операции получателя для выполнения запроса.

Плюсы/Минусы
- \+ команда разрывает связь между объектом, инициирующим операцию, и объектом, имеющим информацию о том, как ее выполнить
- \+ команды – это самые настоящие объекты
- \+ из простых команд можно собирать составные
- \+ добавлять новые команды легко

1.3.3 Interpreter

Interpreter — паттерн поведения классов.
Назначение: Для заданного языка определяет представление его грамматики, а также интерпретатор предложений этого языка.



Описание структуры:
- AbstractExpression —
- TerminalExpression —
- NonterminalExpression —
- Context —
- Client —

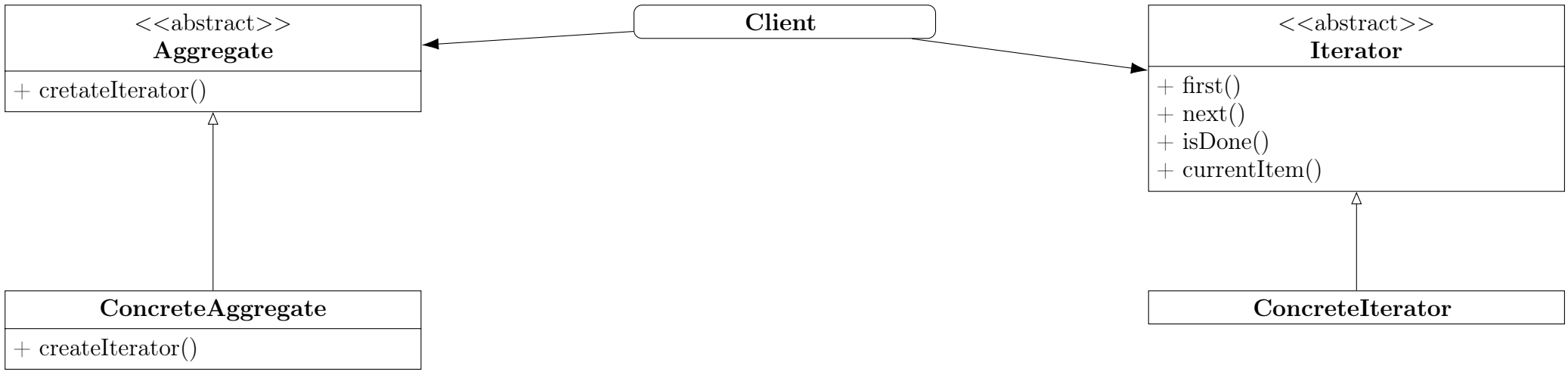
Варианты использования:
- используйте паттерн **Interpreter**, когда есть язык для интерпретации, предложения которого можно представить в виде абстрактных синтаксических деревьев;
- грамматика проста. Для сложных гамматик иерархия классов становится слишком громоздкой и неуправляемой. В таких случаях лучше применять генераторы синтаксических анализаторов, поскольку они могут интерпретировать выражения, не строя абстрактных синтаксических деревьев, что экономит память, а возможно, и время;
- эффективность не является главным критерием. Наиболее эфективные интерпретаторы обычно не работают непосредственно с деревьями, а сначала транслируют их в другую форму. Так, регулярное выражение часто преобразуют в конечный автомат. Но даже в этом случае сам транслятор можно реализовать с помощью паттерна **Interpreter**.

Отношения
- клиент строит (или получает в готовом виде) предложение в виде абстрактного синтаксического дерева, в узлах которого находятся объекты классов NonterminalExpression и TerminalExpression. Затем клиент инициализирует контекст и вызывает операцию Interpret;
- в каждом узле вида NonterminalExpression через операции Interpret определяется операция Interpret для каждого подвыражения. Для класса TerminalExpression операция Interpret определяет базу рекурсии;
- операции Interpret в каждом узле используют контекст для сохранения и доступа к состоянию интерпретатора.

Плюсы/Минусы
- \+ грамматику легко изменять и расширять;
- \+ простая реализация грамматики;
- \- сложные грамматики трудно сопровождать;
- \+ добавление новых способов интерпретации выражений.

1.3.4 Iterator

Iterator — паттерн поведения объектов.
Назначение: Предоставляет способ последовательного доступа ко всем элементам составного объекта, не раскрывая его внутреннего представления.



Описание структуры:
- Iterator —
- ConcreteIterator —
- Aggregate —
- ConcreteAggregate —

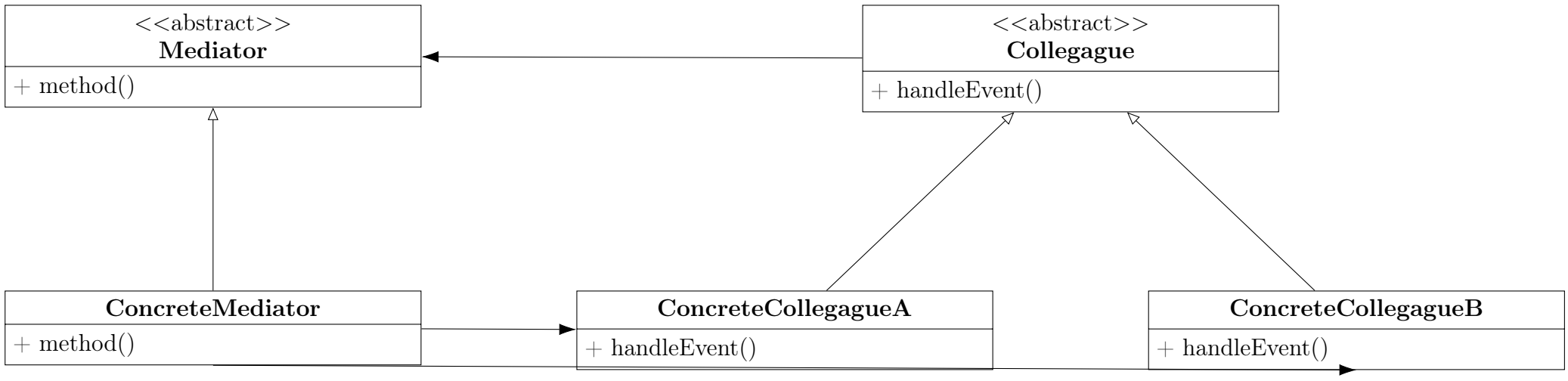
Варианты использования:
- для доступа к содержимому агрегированных объектов без раскрытия их внутреннего представления;
- для поддержки нескольких активных обходов одного и того же агрегированного объекта;
- для предоставления единообразного интерфейса с целью обхода различных агрегированных структур(то есть для поддержки полморфной итерации).

Отношения
- ConcreteIterator отслеживает текущий объект в агрегате и может вычислить идущий за ним.

Плюсы/Минусы
+ поддерживает различные виды обхода агрегата
+ итераторы упрощают интерфейс класса Aggregate
+ одновременно для данного агрегата может быть активно несколько обходов

1.3.5 Mediator

Mediator — паттерн поведения объектов.
Назначение: 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.



Описание структуры:
- Mediator —
- ConcreteMediator —
- Colleagues —

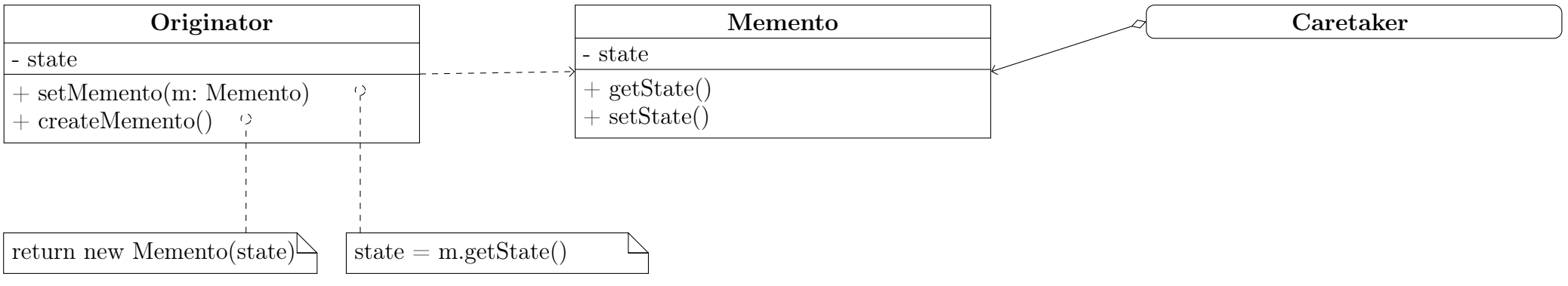
Варианты использования:
- имеются объекты, связи между которыми сложны и четко определены. Получающиеся при этом взаимозависимости не структурированы и трудны для понимания;
- нельзя повторно использовать объект, поскольку он обменивается информацией со многими другими объектами;
- поведение, распределенное между несколькими классами, должно поддаваться настройке без порождения множества подклассов.

Отношения
- Коллеги посылают запросы посреднику и получают запросы от него. Посредник реализует кооперативное поведение путем переадресации каждого запроса подходящему коллеге (или нескольким коллегам).

Плюсы/Минусы
+ снижает число порождаемых подклассов;
+ устраняет связанность между коллегами;
+ упрощает протоколы взаимодействия объектов;
+ абстрагирует способ кооперирования объектов;
+ централизует управление.

1.3.6 Memento

Memento — паттерн поведения объектов.
Назначение: Не нарушая инкапсуляции, фиксирует и выносит за пределы объекта его внутреннее состояние так, чтобы позднее можно было восстановить в нем объект.



Описание структуры:
- Memento —
- Originator —
- Caretaker —

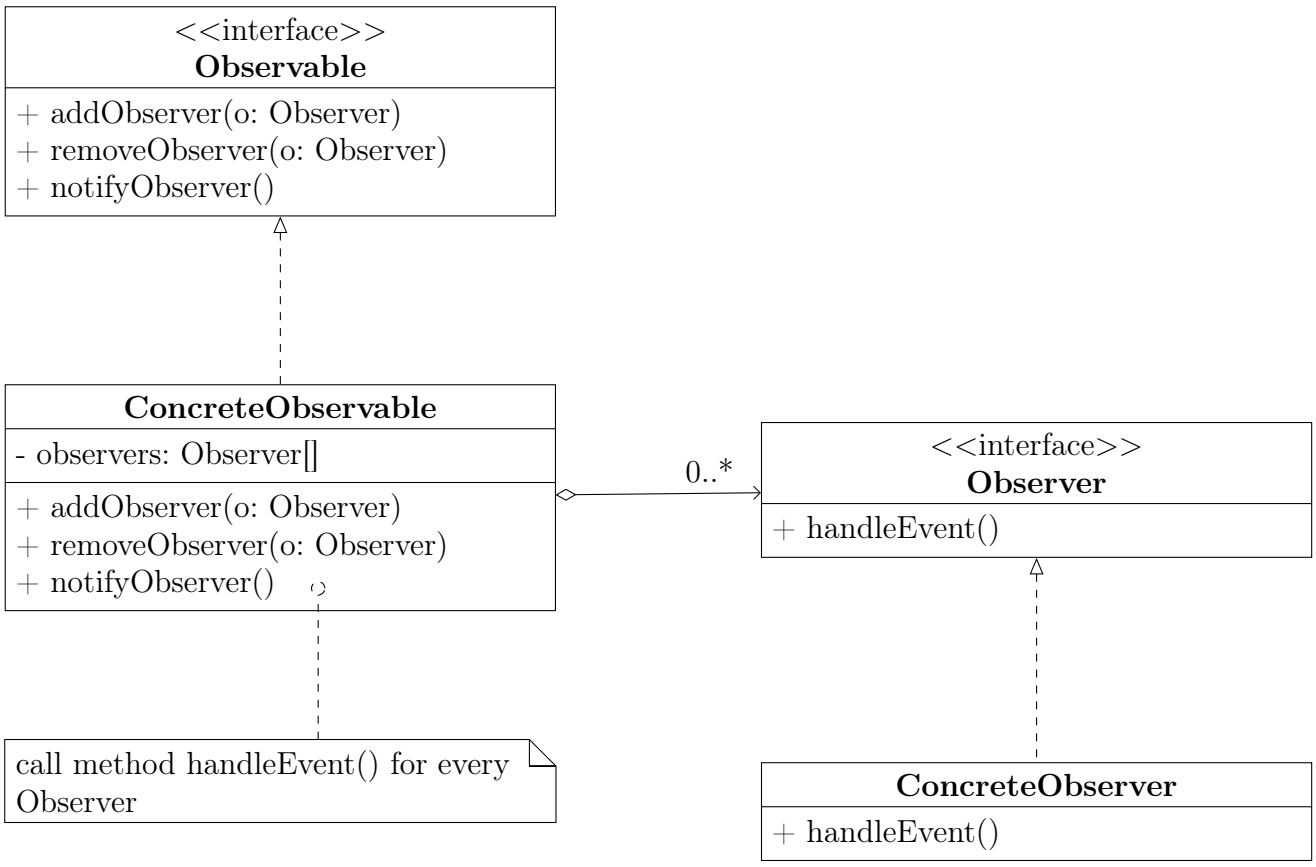
Варианты использования, когда:
- необходимо сохранить мгновенный снимок состояния объекта(или его части), чтобы впоследствии объект можно было восстановить в том же состоянии;
- прямое получение этого состояния раскрывает детали реализации и нарушает инкапсуляцию объекта.

Отношения
- посыльный запрашивает хранитель у хозяина, некоторое время держит его у себя, а затем возвращает хозяину, как видно на представленной диаграмме взаимодействий.
- хранители пассивны. Только хозяин, создавший хранитель, имеет доступ к информации о состоянии.

Плюсы/Минусы
+ сохранение границ инкапсуляции;
+ упрощение структуры хозяина;
- значительные издержки при использовании хранителей;
- определение «узкого» и «широкого» интерфейсов;
- скрытая плата за содержание хранителя.

1.3.7 Observer

Observer — паттерн поведения объектов.
Назначение: 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.



Описание структуры:
- Subject —
- Observer —
- ConcreteSubject —
- ConcreteObserver —

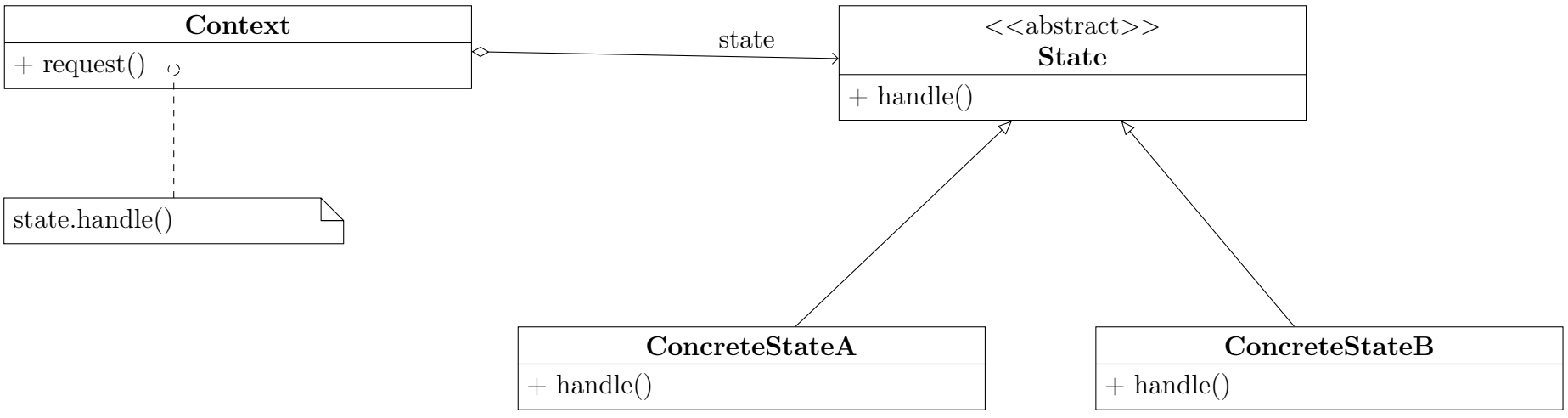
Варианты использования:
- когда у абстракции есть два аспекта, один из которых зависит от другого. Инкапсуляции этих аспектов в разные объекты позоляют изменять и повторно ипользовать их независимо;
- когда при модификации одного объекта требуется изменить другие и вы не знаете, сколько именно объектов нужно изменить;
- когда один объект должен оповещать других, не делая предположений об уведомляемых объектах. Другими словами, вы не хотите, чтобы объекты были тесно связаны между собой.

Отношения
- объект ConcreteSubject уведомляет своих наблюдателей о любом изменении, которое могло бы привести к рассогласованности состояний наблюдателя и субъекта;
- после получения от конкретного субъекта уведомления об изменении объект ConcreteObserver может запросить у субъекта дополнительнуюинформацию, которую использует для того, чтобы оказаться в состоянии, согласованном с состоянием субъекта.

Плюсы/Минусы
- \+ абстрактная связанность субъекта и наблюдателя;
- \+ поддержка широковещательных коммуникаций;
- \- неожиданные обновления.

1.3.8 State

State — паттерн поведения объектов.
Назначение: озволяет объекту варьировать свое поведение в зависимости от внутреннего состояния. Извне создается впечатление, что изменился класс объекта.



Описание структуры:
- Context —
- State —
- ConcreteState subclasses —

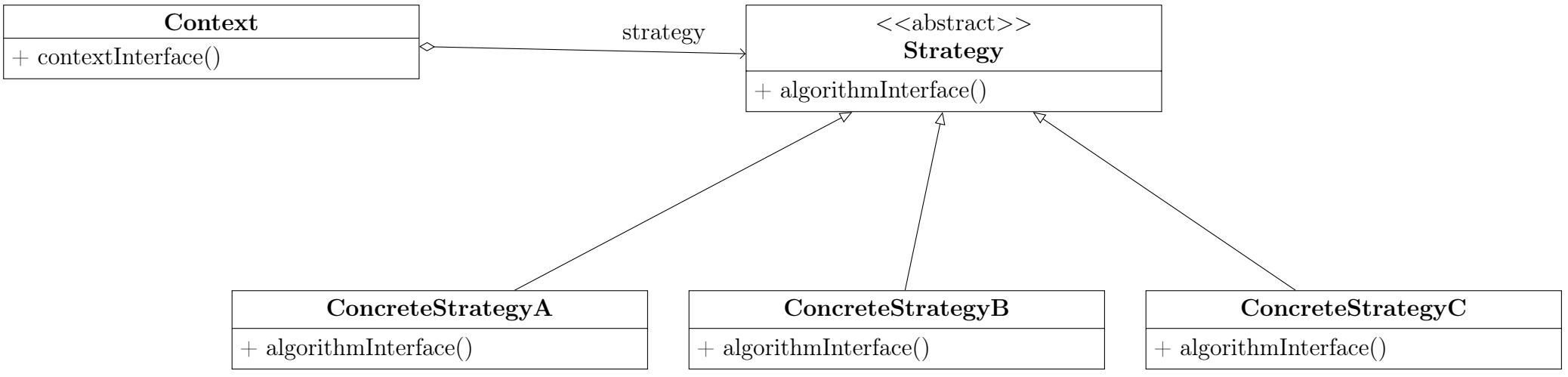
Варианты использования:
- когда поведение объекта зависит от его состояния и должно изменяться вовремя выполнения;
- когда в коде операций встречаются состоящие из многих ветвей условные операторы, в которых выбор ветви зависит от состояния. Обычно в таком случае состояние представлено перечисляемыми константами. Часто одна и та же структура условного оператора повторяется в нескольких операциях. Паттерн **State** предлагает поместить каждую ветвь в отдельный класс. Это позволяет трактовать состояние объекта как самостоятельный объект, который может изменяться независимо от других.

Отношения
- ласс Context делегирует зависящие от состояния запросы текущему объекту ConcreteState;
- онтекст может передать себя в качестве аргумента объекту State, который будет обрабатывать запрос. Это дает возможность объекту состоянию при необходимости получить доступ к контексту;
- Context – это основной интерфейс для клиентов. Клиенты могут конфигурировать контекст объектами состояния State. Один раз сконфигурировав контекст, клиенты уже не должны напрямую связываться с объектами состояния;
- либо Context, либо подклассы ConcreteState могут решить, при каких условиях и в каком порядке происходит смена состояний.

Плюсы/Минусы
- \+ локализует зависящее от состояния поведение и делит его на части, соответствующие состояниям;
- \+ делает явными переходы между состояниями;
- \+ объекты состояния можно разделять.

1.3.9 Strategy

Strategy — паттерн поведения объектов.
Назначение: Определяет семейство алгоритмов, инкапсулирует каждый из них и делает их взаимозаменяемыми. Стратегия позволяет изменять алгоритмы независимо от клиентов, которые ими пользуются.



Описание структуры:
- Strategy —
- ConcreteStrategy —
- Context —

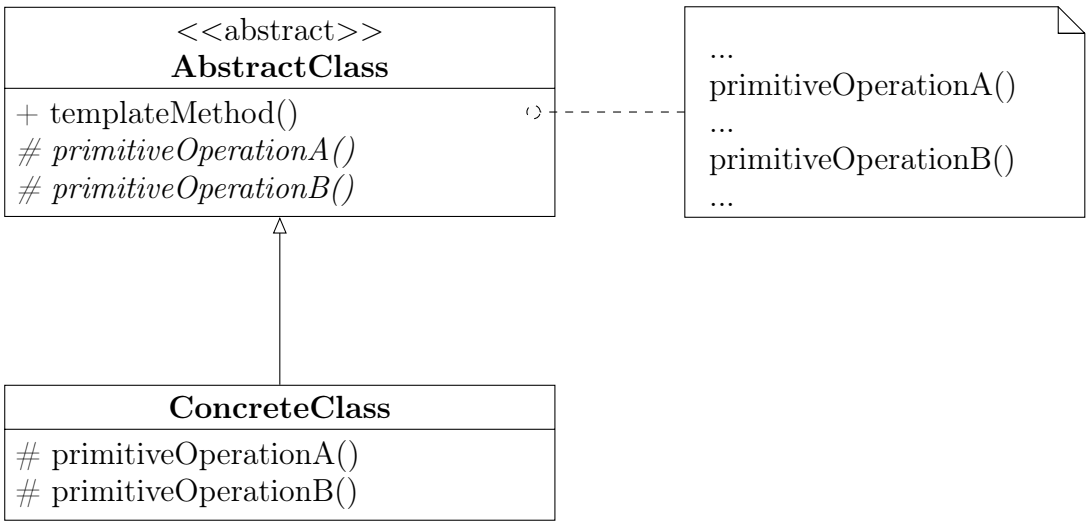
Варианты использования, когда:
- имеется много родственных классов, отличающихся только поведением. **Strategy** позволяет сконфигурировать класс, задав одно из возможных поведений;
- вам нужно иметь несколько разных вариантов алгоритма. Например, можно определить два варианта алгоритма, один из которых требует больше времени, а другой — больше памяти. **Strategy** разрешается применять, когда варианты алгоритмов реализованы в виде иерархии классов;
- в алгоритме содержатся данные, о которых клиент не должен «знать». Используйте паттерн **Strategy**, чтобы не раскрывать сложные, специфичные для алгоритма структуры данных;
- в классе определено много поведений, что предствалено развлетвленными условными операторвами. В этом случае проще перенести код из ветви в отдельные классы стратегий.

Отношения
- классы Strategy и Context взаимодействуют для реализации выбранного алгоритма. Контекст может передать стратегии все необходимые алгоритму данные в момент его вызова. Вместо этого контекст может позволить обращаться к своим операциям в нужные моменты, передав ссылку на самого себя операциям класса Strategy;
- контекст переадресует запросы своих клиентов объекту стратегии. Обычно клиент создает объект ConcreteStrategy и передает его контексту, после чего клиент «общается» исключительно с контекстом. Часто в распоряжении клиента находится несколько классов ConcreteStrategy, которые он может выбирать.

Плюсы/Минусы
+ семейства родственных алгоритмов
  альтернатива порождению подклассов.
+ с помощью стратегий можно избавиться от условных операторов
+ выбор реализации
- клиенты должны «знать» о различных стратегиях.
  обмен информацией между стратегией и контекстом
- увеличение числа объектов

1.3.10 Template method

Template method — паттерн поведения классов.
Назначение: Шаблонный метод определяет основу алгоритма и позволяет подклассам переопределить некоторые шаги алгоритма, не изменяя его структуру в целом.



Описание структуры:
- AbstractClass —
- ConcreteClass —

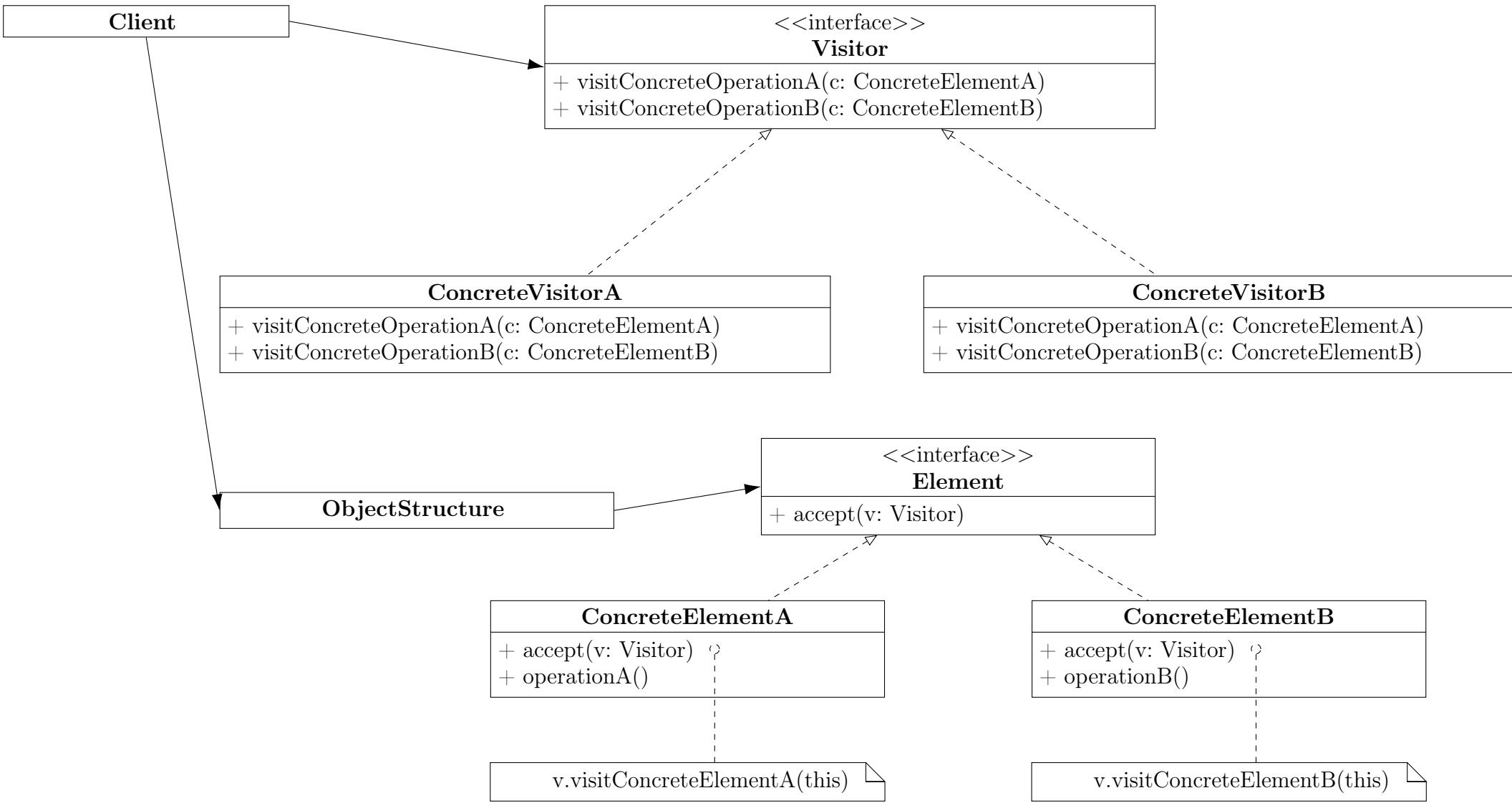
Варианты использования:
- чтобы однократно использовать инвариантные части алгоритма, оставляя реализацию изменяющегося поведения на усмотрение подклассов;
- когда нужно вычленить и локализовать в одном классе поведение, общее для всех подклассов, дабы избежать дублирования кода. Это хороший пример техники «вынесения за скобки с целью обобщения», описанной в работе Уильяма Опдайка и Ральфа Джонсона. Сначала индентифицируются различия в существующем коде, а затем они выносятся в отдельные операции. В конечном итоге различающиеся фрагменты кода заменяются шаблонным методом, из которого вызываются новые операции;
- для управления расширениями подклассов. Можно определить шаблонный метод так, что он будет вызывать операции-зацепки(hooks) — в определенных точках, разрешив тем самым расширение только в этих точках.

Отношения
- ConcreteClass предполагает, что инвариантные шаги алгоритма будут выполнены в AbstractClass.

Плюсы/Минусы
- SHIIIT BOISHHSH

1.3.11 Visitor

Visitor — паттерн поведения объектов.
Назначение: Описывает операцию, выполняемую с каждым объектом из некоторой структуры. Паттерн посетитель позволяет определить новую операцию, не изменяя классы этих объектов.



Описание структуры:
- Visitor —
- ConcreteVisitor —
- Element —
- ConcreteElement —
- ObjectStructure —

Варианты использования, когда:
- в структуре присутствуют объекты многих классов с различными интерфейсами и вы хотите выполнять над ними операции, зависящие от конкретных классов;
- над объектами, входящими в состав структуры, надо выполнять разнообразные, не связанные между собой операции и вы не хотите «засорять» классы такими операциями; **Visitor** позволяет объединить родственные операции, поместив их в один класс. Если структура объектов является общей для нескольких приложений, то паттерн **Visitor** позволит в каждое приложение включить только относящиеся к нему операции;
- классы, устанавливающие структуру объектов, изменяются редко, но новые операции над этой структурой добавляются часто. При изменении классов, представленных в структуре, нужно будет переопределить интерфейсы всех посетителей, а это может вызвать затруднения. Поэтому если классы меняются достаточно часто, то, вероятно, лучше определить операции прямо в них.

Отношения
- клиент, использующий паттерн посетитель, должен создать объект класса ConcreteVisitor, а затем обойти всю структуру, посетив каждый ее элемент.
- при посещении элемента последний вызывает операцию посетителя, соответствующую своему классу. Элемент передает этой операции себя в качестве аргумента, чтобы посетитель мог при необходимости получить доступ к его состоянию.

Плюсы/Минусы
- \+ упрощает добавление новых операций
- \+ объединяет родственные операции и отсекает те, которые не имеют к ним отношения
- \- добавление новых классов ConcreteElement затруднено
- \+ посещение различных иерархий классов
- \+ аккумулирование состояния
- \- нарушение инкапсуляции