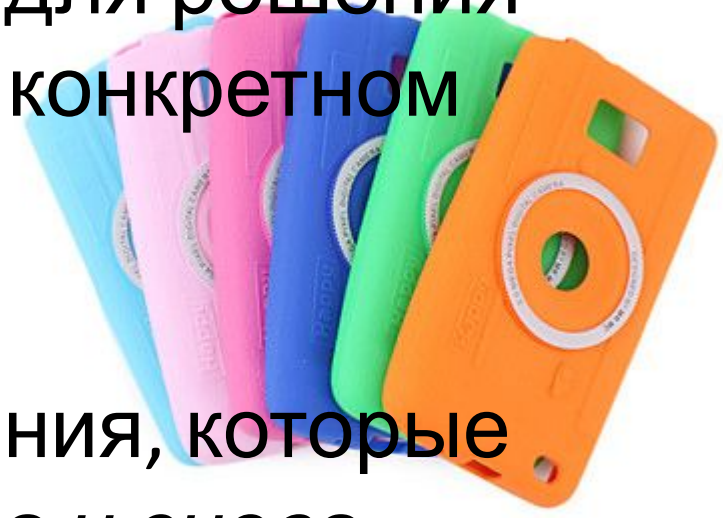
# Шаблоны проектирования
# Design Patterns

# Что такое «Шаблон проектирования»?

- Описание взаимодействия объектов и классов адаптированных для решения задачи проектирования в конкретном контексте.

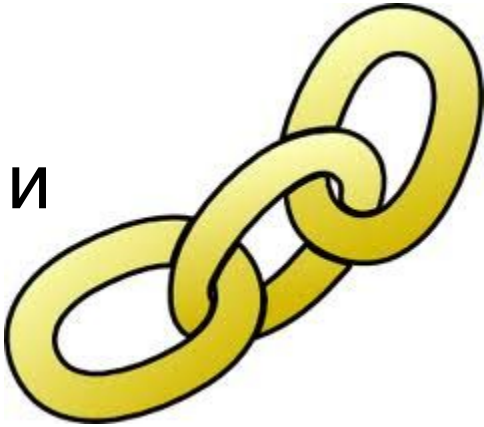


- Удачные проектные решения, которые можно использовать *снова и снова*.

# Какие задачи решает Шаблон проектирования?

- Именует, абстрагирует, идентифицирует ключевые аспекты структуры общего решения для *повторно* используемого дизайна.

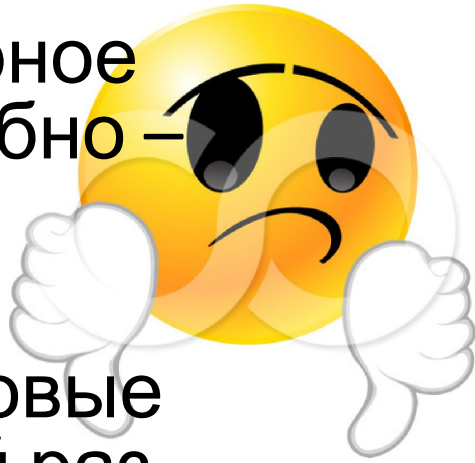
- Вычленяет участвующие классы и экземпляры, их роли и отношения и функции.



- Учитывает проектные ограничения и последствия.

# Чем Шаблон проектирования не является?

- Это не серебренная пуля, чрезмерное злоупотребление шаблонами пагубно – снижение качества анализа.



- Это не библиотеки классов, не готовые приложения или системы - каждый раз нужно реализовывать свою конкретную задачу.

- Не нечто очень сложное – в основе лежат простые принципы.

# Процесс Объектно-ориентированного проектирования

- Создание словаря предметной области и вспомогательных сущностей;
- Декомпозиция на *объекты;*
- Декомпозиция на *классы;*
- Создание модели предметной области;
- Анализ и следующая *итерация.*

# Классификация шаблонов проектирования

- Паттерны проектирования классов/обьектов (Классификация GoF)
- Архитектурные системные паттерны
- Паттерны интеграции корпоративных информационных систем

# Gang of Four

**Design Patterns: Elements of Reusable Object-Oriented Software** — книга 1994 года об инженерии программного обеспечения, описывающая решения некоторых частых проблем в проектировании ПО. Авторы книги: *Эрих Гамма (Erich Gamma), Ричард Хелм (Richard Helm), Ральф Джонсон (Ralph Johnson), Джон Влиссидс (John Vlissides)*. Коллектив авторов также известен как **«Банда четырёх»**

# Типы паттернов проектирования по GoF

Порождающие паттерны, предназначенные для создания новых объектов в системе.


Структурные паттерны, решающие задачи компоновки системы на основе классов и объектов.

Паттерны поведения, предназначенные для распределения обязанностей между объектами в системе.

# Структурные паттерны проектирования классов/объектов

- Адаптер (Adapter)
- Оболочка (Wrapper)
- Заместитель (Proxy)
- Компоновщик (Composite)
- Мост (Bridge)
- Приспособленец (Flyweight)
- Фасад (Facade)

- Адаптер (Adapter)

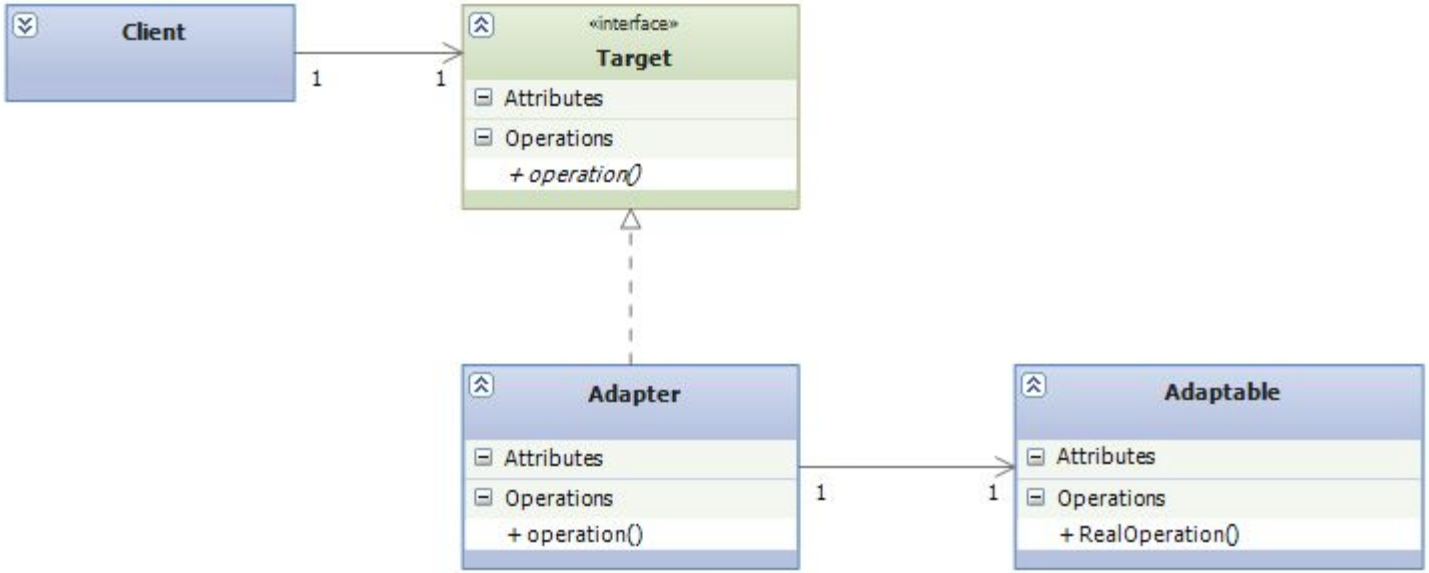
# Адаптер - Проблема

Необходимо обеспечить взаимодействие несовместимых интерфейсов или как создать единый устойчивый интерфейс для нескольких компонентов с разными интерфейсами.

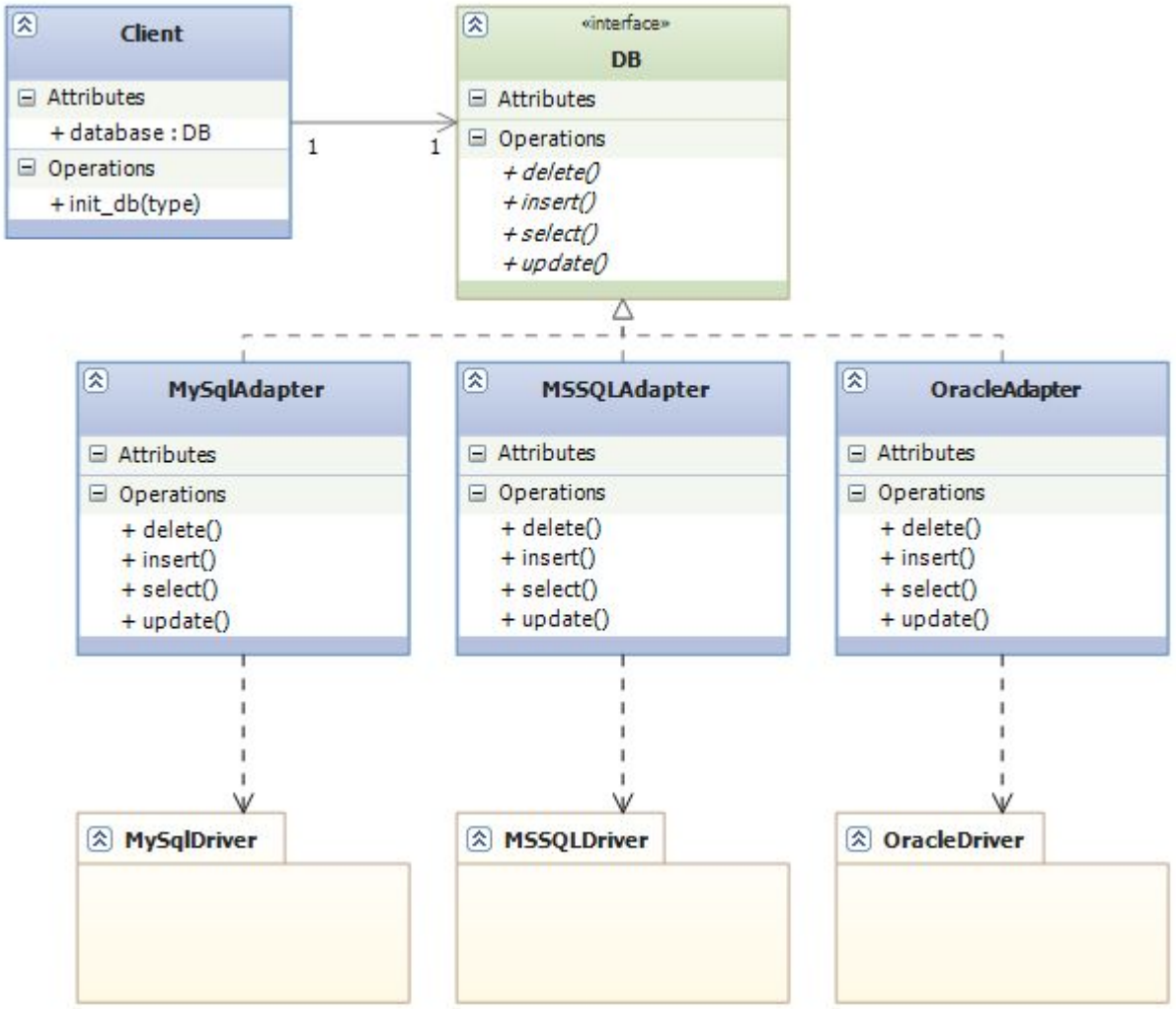
## Адаптер - Решение

Конвертировать исходный интерфейс компонента к другому виду с помощью промежуточного объекта - адаптера, то есть, добавить специальный объект с общим интерфейсом в рамках данного приложения и перенаправить связи от внешних обьектов к этому объекту - адаптеру.

# Адаптер – Структура



# Адаптер - Пример



## Адаптер - Применимость

- Необходимо использовать существующий класс, но его интерфейс не соответствует заданным требованиям
- Создание повторно используемого класса, который должен взаимодействовать с заранее неизвестными или не связанными с ним классами, имеющими несовместимые интерфейсы
- Использование нескольких существующих подклассов, приспосабливая интерфейс их общего родительского класса (только для адаптера объектов)

- Оболочка (Wrapper)

## Оболочка - Проблема

Возложить дополнительные обязанности (прозрачные для клиентов) на отдельный объект, а не на класс в целом.

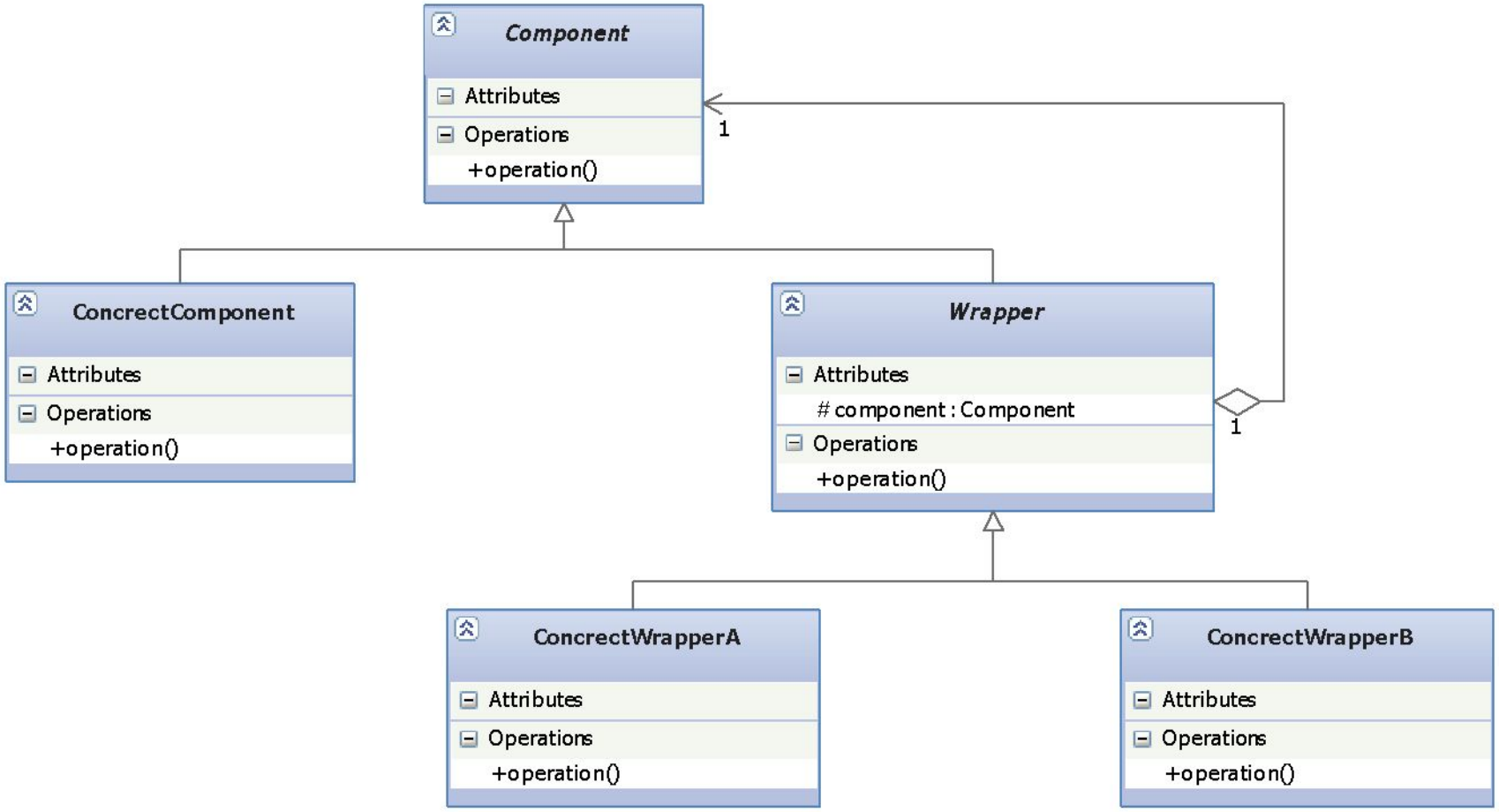
## Оболочка - Рекомендации

Применение нескольких «Оболочек" к одному "Компоненту" позволяет произвольным образом сочетать обязанности, например, одно свойство можно добавить дважды.

## Оболочка - Решение

Динамически добавить объекту новые обязанности не прибегая при этом к порождению подклассов (наследованию). «Компонент» определяет интерфейс для объектов, на которые могут быть динамически возложены дополнительные обязанности, «Конкретный Компонент» определяет объект, на который возлагаются дополнительные обязанности, «Оболочка» - хранит ссылку на объект "Компонент" и определяет интерфейс, соответствующий интерфейсу "Компонента". "Конкретная Оболочка" возлагает дополнительные обязанности на компонент. «Оболочка" переадресует запросы объекту "Компонент".

# Оболочка - Структура



## Оболочка - Преимущество

Большая гибкость, чем у статического наследования: можно добавлять и удалять обязанности во время выполнения программы в то время как при использовании наследования надо было бы создавать новый класс для каждой дополнительной обязанности. Данный паттерн позволяет избежать перегруженных методами классов на верхних уровнях иерархии - новые обязанности можно добавлять по мере необходимости.

## Оболочка - Недостатки

«Оболочка» и его "Компонент" не идентичны, и, кроме того, получается что система состоит из большого числа мелких объектов, которые похожи друг на друга и различаются только способом взаимосвязи, а не классом и не значениями своих внутренних переменных - такая система сложна в изучении и отладке.

- Заместитель (Proxy)

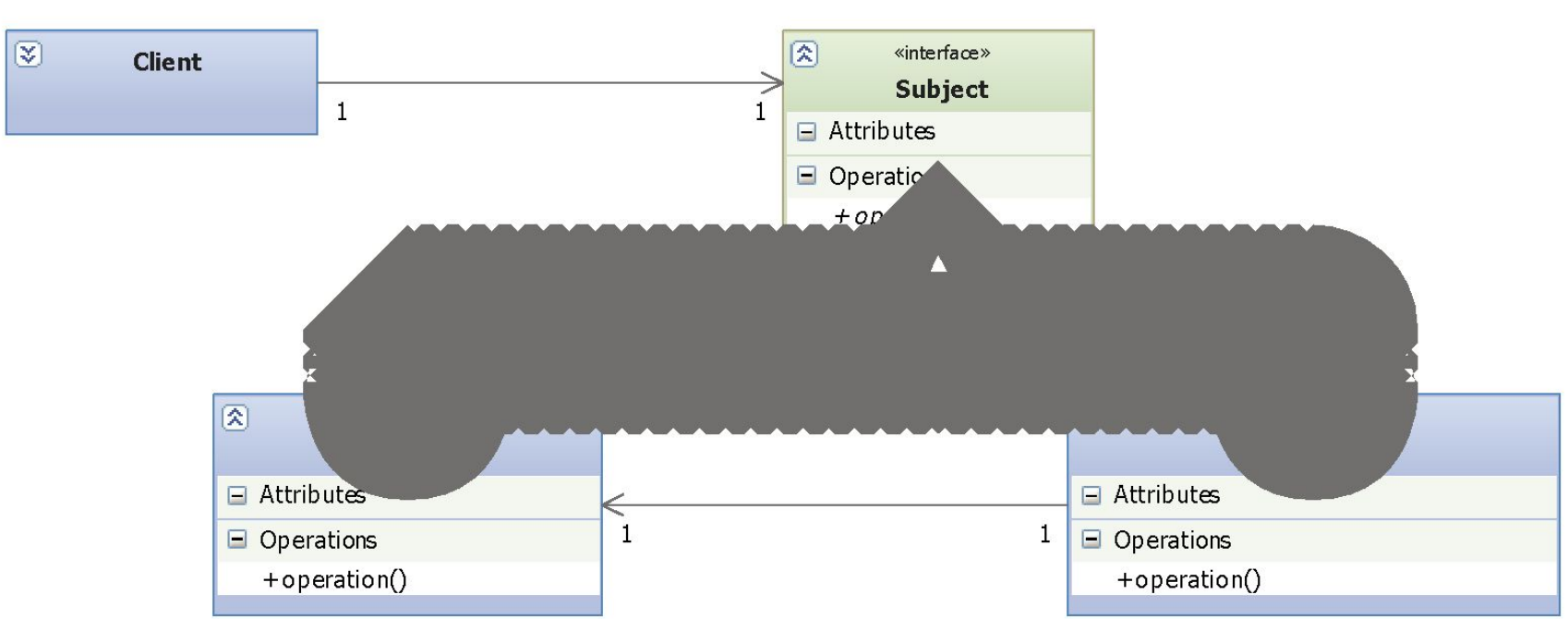
## Заместитель - Проблема

Необходимо управлять доступом к объекту, так чтобы создавать громоздкие объекты "по требованию".

# Заместитель - Решение

Создать суррогат громоздкого объекта. "Заместитель" хранит ссылку, которая позволяет заместителю обратиться к реальному субъекту (объект класса "Заместитель" может обращаться к объекту класса "Субъект", если интерфейсы "РеальногоСубъекта" и "Субъекта" одинаковы). Поскольку интерфейс "РеальногоСубъекта" идентичен интерфейсу "Субъекта", так, что "Заместителя" можно подставить вместо "РеальногоСубъекта", контролирует доступ к "РеальномуСубъекту", может отвечать за создание или удаление "РеальногоСубъекта". "Субъект" определяет общий для "РеальногоСубъекта" и "Заместителя" интерфейс, так, что "Заместитель" может быть использован везде, где ожидается "РеальныйСубъект". При необходимости запросы могут быть переадресованы "Заместителем" "РеальномуСубъекту".

# Заместитель - Структура



# Заместитель – Виды

- Удаленный заместитель : обеспечивает связь с «Субъектом», который находится в другом адресном пространстве или на удалённой машине. Так же может отвечать за кодирование запроса и его аргументов и отправку закодированного запроса реальному «Субъекту».
- Виртуальный заместитель обеспечивает создание реального «Субъекта» только тогда, когда он действительно понадобится. Так же может кэшировать часть информации о реальном «Субъекте», чтобы отложить его создание.
- Копировать-при-записи: обеспечивает копирование «субъекта» при выполнении клиентом определённых действий.
- Защищающий заместитель: может проверять, имеет ли вызывающий объект необходимые для выполнения запроса права.
- Кэширующий прокси: обеспечивает временное хранение результатов расчёта до отдачи их множественным клиентам, которые могут разделить эти результаты.
- Экранирующий прокси: защищает «Субъект» от опасных клиентов (или наоборот).
- Синхронизирующий прокси: производит синхронизированный контроль доступа к «Субъекту» в асинхронной многопоточной среде.
- Smart reference proxy: производит дополнительные действия, когда на «Субъект» создается ссылка, например, рассчитывает количество активных ссылок на «Субъект».

- Компоновщик (Composite)

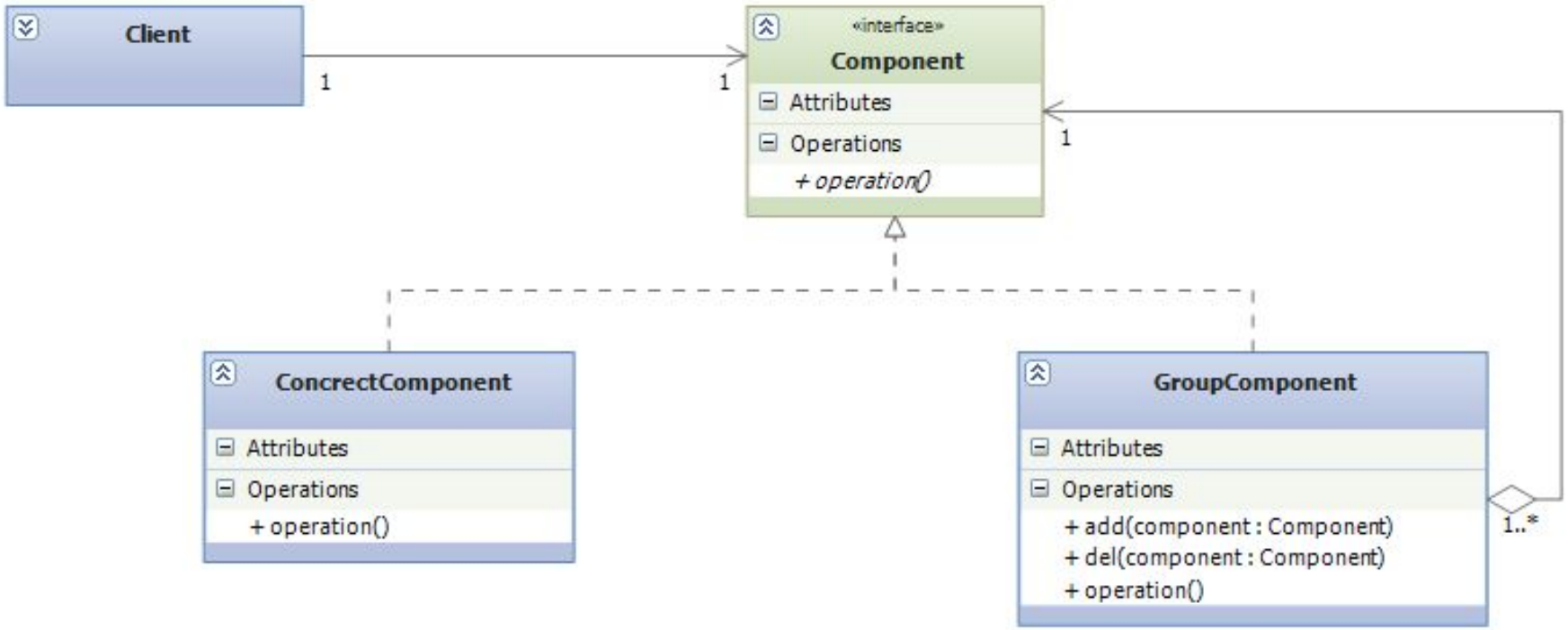
# Компоновщик - Проблема

Как обрабатывать группу или композицию структур объектов одновременно?

## Компоновщик - Решение

Определить классы для композитных и атомарных объектов таким образом, чтобы они реализовывали один и тот же интерфейс.

# Компоновщик - Структура



## Компоновщик - Использование

- Необходимо объединять группы схожих объектов и управлять ими.
- Объекты могут быть как примитивными (элементарными), так и составными (сложными). Составной объект может включать в себя коллекции других объектов, образуя сложные древовидные структуры. Пример: директория файловой системы состоит из элементов, каждый их которых также может быть директорией.
- Код клиента работает с примитивными и составными объектами единообразно.

- Мост (Bridg)

## Мост - Проблема

Требуется отделить абстракцию от реализации так, чтобы и то и другое можно было изменять независимо. При использовании наследования реализация жестко привязывается к абстракции, что затрудняет независимую модификацию.

## Мост - Решение

Поместить абстракцию и реализацию в отдельные иерархии классов.

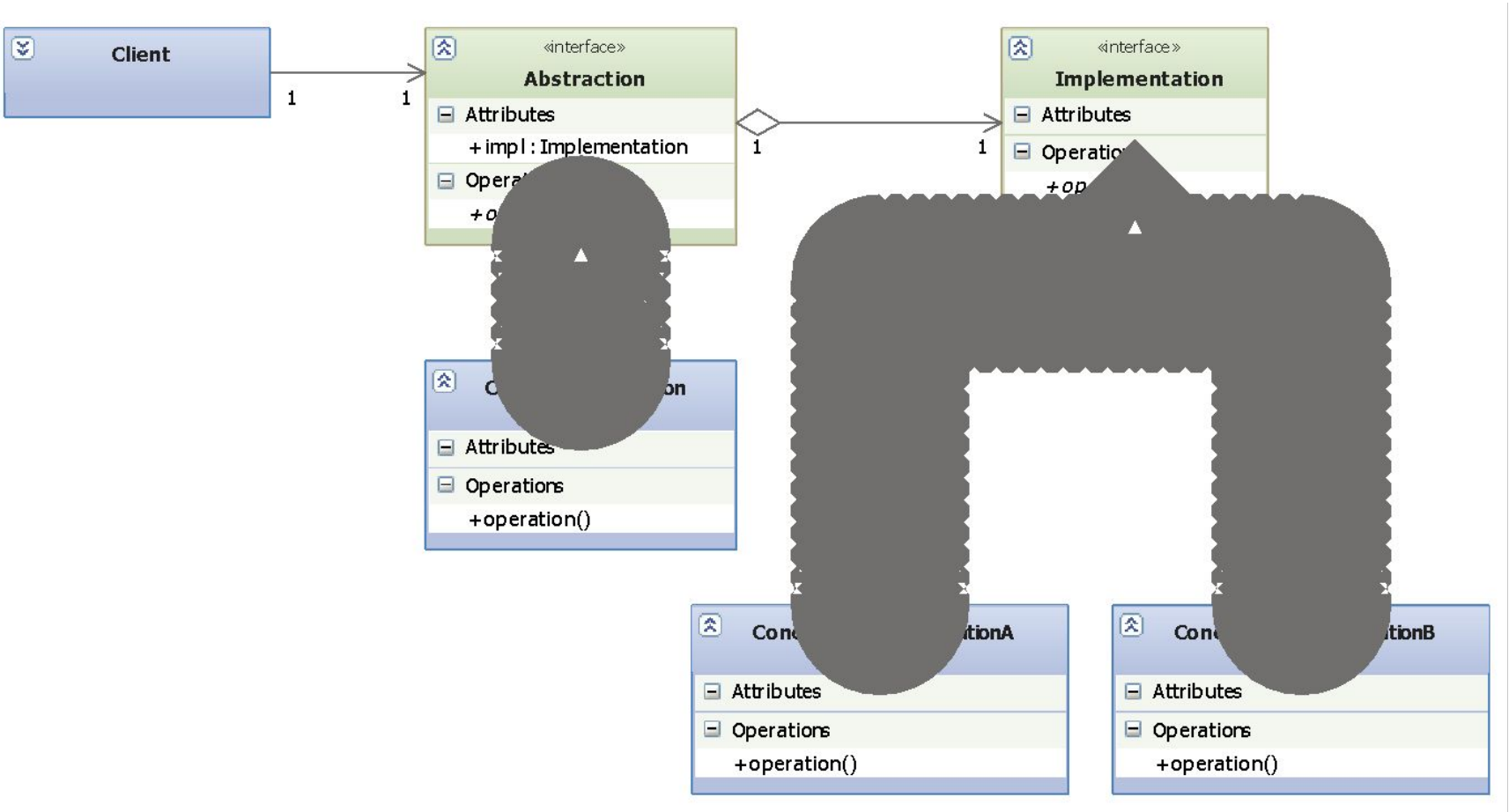
# Мост - Рекомендации

Можно использовать если, например, реализацию необходимо выполнять во время реализации программы.

## Мост - Решение

"Абстракция" определяет интерфейс "Абстракции" и хранит ссылку на объект "Реализация", "УточненнаяАбстракция" расширяет интерфейс, определенный "Абстракцией". "Реализация" определяет интерфейс для классов реализации, он не обязан точно соответствовать интерфейсу класса "Абстракция" - оба интерфейса могут быть совершенно различны. Обычно интерфецйс класса "Реализация" предоставляет только примитивные операции, а класс "Абстракция" определяет операции более высокого уровня, базирующиеся на этих примитивных. "КонкретнаяРеализация" содержит конкретную реализацию класса "Реализация". Объект "Абстракция" перенаправляет своему объекту "Реализация" запросы "Клиента".

# Мост - Структура



## Мост – Преимущество

Отделение реализации от интерфейса, то есть, "Реализацию" "Абстракции" можно конфигурировать во время выполнения. Кроме того, следует упомянуть, что разделение классов "Абстракция" и "Реализация" устраняет зависимости от реализации, устанавливаемые на этапе компиляции: чтобы изменить класс "Реализация" вовсе не обязательно перекомпилировать класс "Абстракция".

- Приспособленец (Flyweight)

## Приспособленец - Проблема

Необходимо обеспечить поддержку множества мелких объектов.

# Приспособленец - Рекомендации

Приспособленцы моделируют сущности, число которых слишком велико для представления объектами. Имеет смысл использовать данный паттерн если одновременно выполняются следующие условия:

- в приложении используется большое число объектов, из-за этого расходы на хранение высоки;
- большую часть состояния объектов можно вынести вовне;
- многие группы объектов можно заменить относительно небольшим количеством объектов, поскольку состояния объектов вынесены вовне.

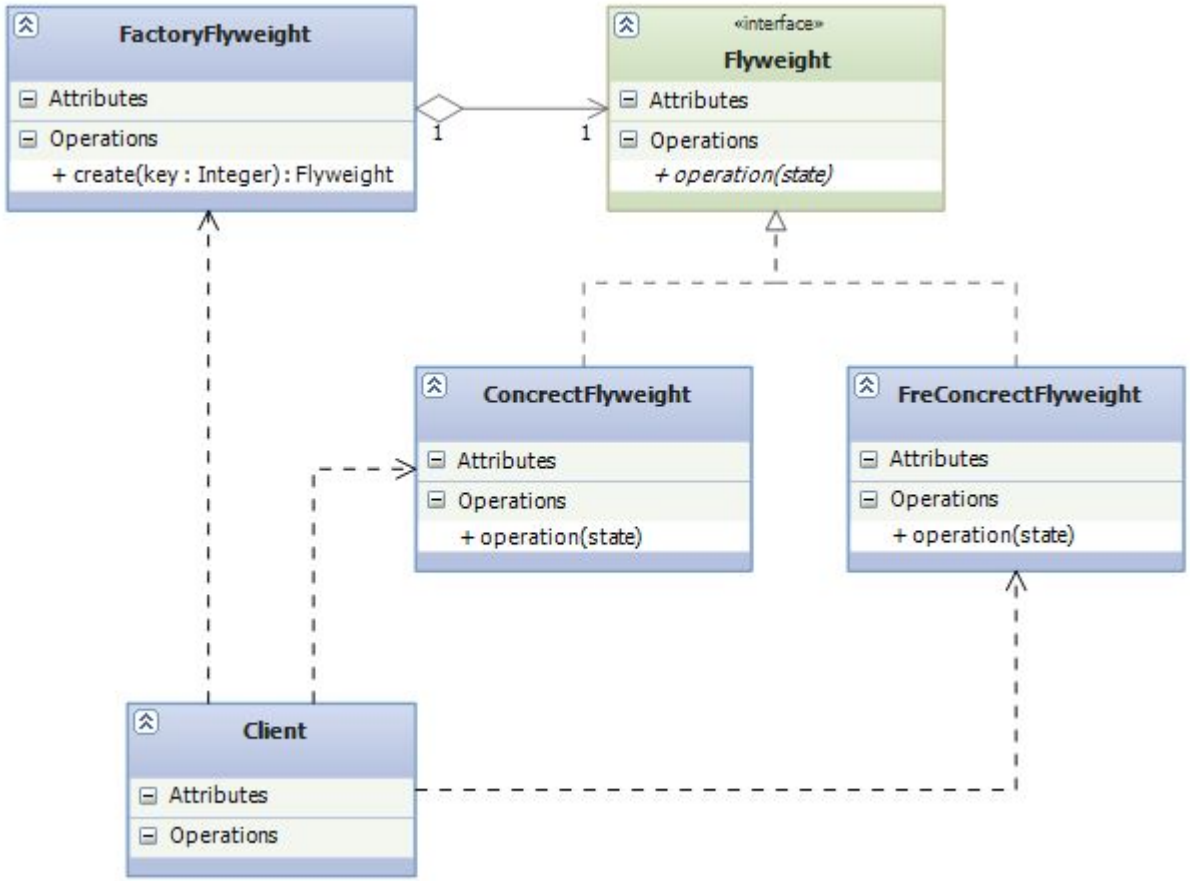
## Приспособленец – Решение(1)

Создать разделяемый объект, который можно использовать одновременно в нескольких контекстах, причем, в каждом контексте он выглядит как независимый объект (неотличим от экземпляра, который не разделяется). «Приспособленец» объявляет интерфейс, с помощью которого приспособленцы могут получить внешнее состояние или как-то воздействовать на него, "КонкретныйПриспособленец" реализует интерфейс класса "Приспособленец" и добавляет при необходимости внутреннее состояние. Внутреннее состояние хранится в объекте "КонкретныйПриспособленец", в то время как внешнее состояние хранится или вычисляется "Клиентами" ("Клиент" передает его "Приспособленцу" при вызове операций).

## Приспособленец – Решение(2)

Объект класса "КонкретныйПриспособленец" должен быть разделяемым. Любое сохраняемое им состояние должно быть внутренним, то есть независимым от контекста, "ПриспособленецФабрика" - создает объекты - "Приспособленцы" (или предоставляет существующий экземпляр) и управляет ими. "НеразделяемыйКонкретныйПриспособленец" - не все подклассы "Приспособленца" обязательно должны быть разделяемыми. "Клиент" - хранит ссылки на одного или нескольких "Приспособленцев", вычисляет и хранит внешнее состояние "Приспособленцев".

# Приспособленец - Структура



## Приспособленец - Преимущества

Вследствие уменьшения общего числа экземпляров и вынесения состояния экономится память.

- Фасад (Facade)

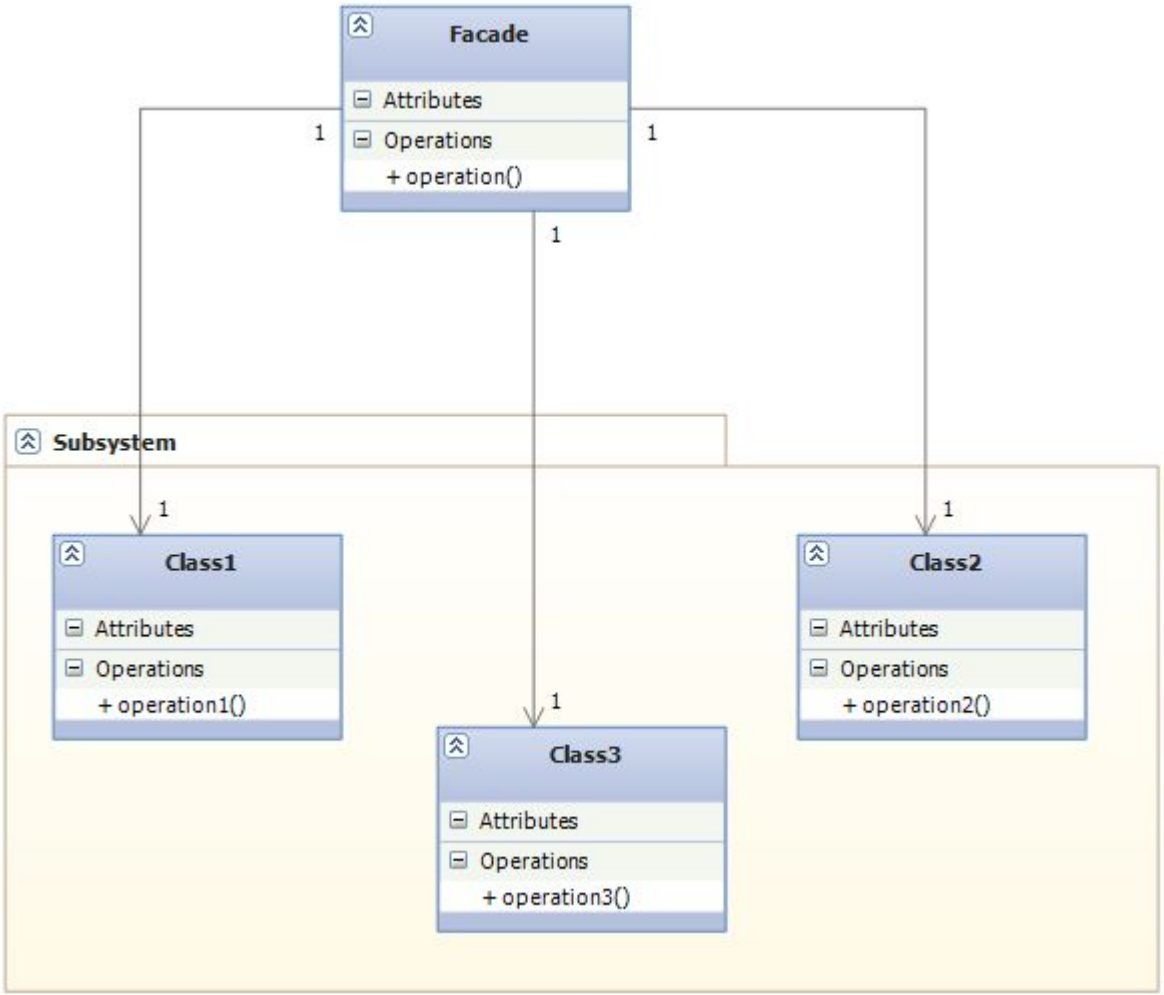
## Фасад - Проблема

Как обеспечить унифицированный интерфейс с набором разрозненных реализаций или интерфейсов, например, с подсистемой, если нежелательно высокое связывание с этой подсистемой или реализация подсистемы может измениться?

## Фасад - Решение

Определить одну точку взаимодействия с подсистемой - фасадный объект, обеспечивающий общий интерфейс с подсистемой и возложить на него обязанность по взаимодействию с ее компонентами. Фасад - это внешний объект, обеспечивающий единственную точку входа для служб подсистемы. Реализация других компонентов подсистемы закрыта и не видна внешним компонентам.

# Фасад - Структура



# Паттерны проектирования поведения классов/объектов

- Интерпретатор
- Итератор
- Команда
- Наблюдатель
- Посетитель
- Посредник
- Состояние
- Стратегия
- Хранитель
- Цепочка обязанностей
- Шаблонный метод

- Интерпретатор

## Интерпретатор - Проблема

Имеется часто встречающаяся, подверженная изменениям задача.

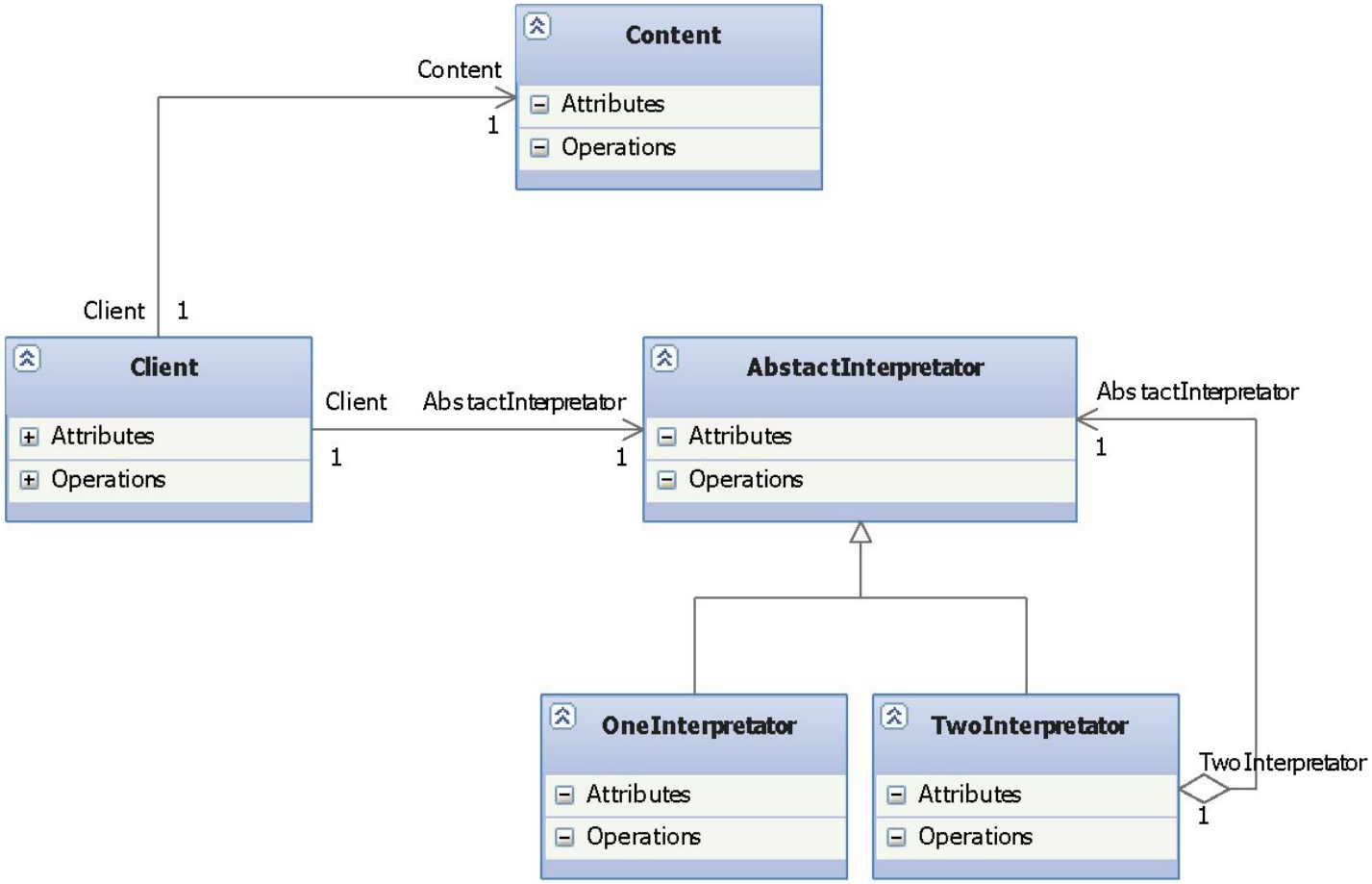
# Интерпретатор - Решение

Создать интерпретатор, который решает данную задачу.

## Интерпретатор - Пример

Задача поиска строк по образцу может быть решена посредством создания интерпретатора, определяющего грамматику языка.

# Интерпретатор - Структура



## Интерпретатор - Преимущества

Грамматику становится легко расширять и изменять, реализации классов, описывающих узлы абстрактного синтаксического дерева похожи (легко кодируются). Можно легко изменять способ вычисления выражений.

## Интерпретатор - Недостатки

Сопровождение грамматики с большим числом правил затруднительно.

- Итератор (Iterator)

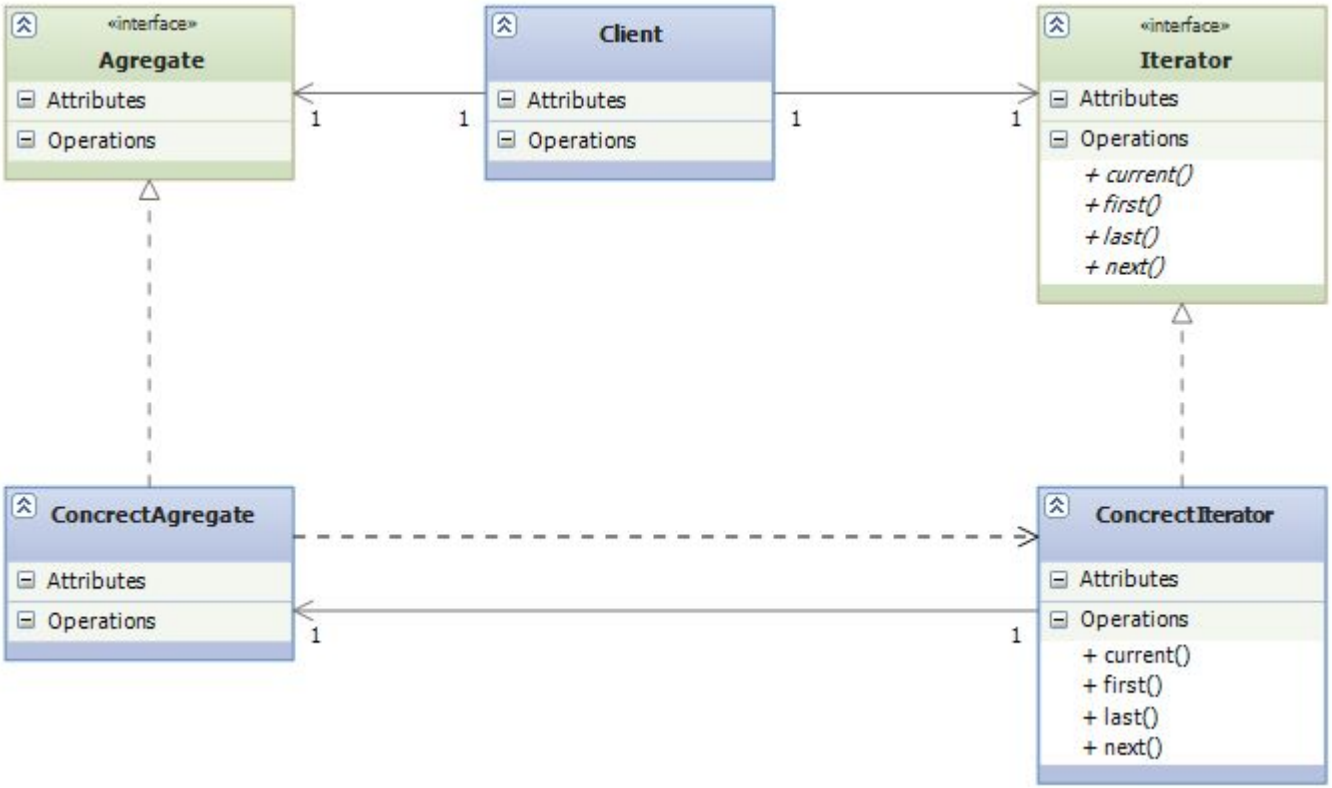
# Итератор - Проблема

Составной объект, например, список, должен предоставлять доступ к своим элементам (объектам), не раскрывая их внутреннюю структуру, причем перебирать список требуется по-разному в зависимости от задачи.

## Итератор - Решение

Создается класс "Итератор", коорый определяет интерфейс для доступа и перебора элементов, "КонкретныйИтератор" реализует интерфейсласса "Итератор" и следит за текущей позицией при обходе "Агрегата". "Агрегат" определяет интерфейс для создания объекта - итератора. "КонкретныйАгрегат" реализует интерфейс создания итератора и возвращает экземпляр класса "КонкретныйИтератор", "КонкретныйИтератор" отслеживает текущий объект в агрегате и может вычислить следующий объект при переборе.

# Итератор - Структура



## Итератор - Преимущества

Поддерживает различные способы перебора агрегата, одновременно могут быть активны несколько переборов.

- Команда (Command)

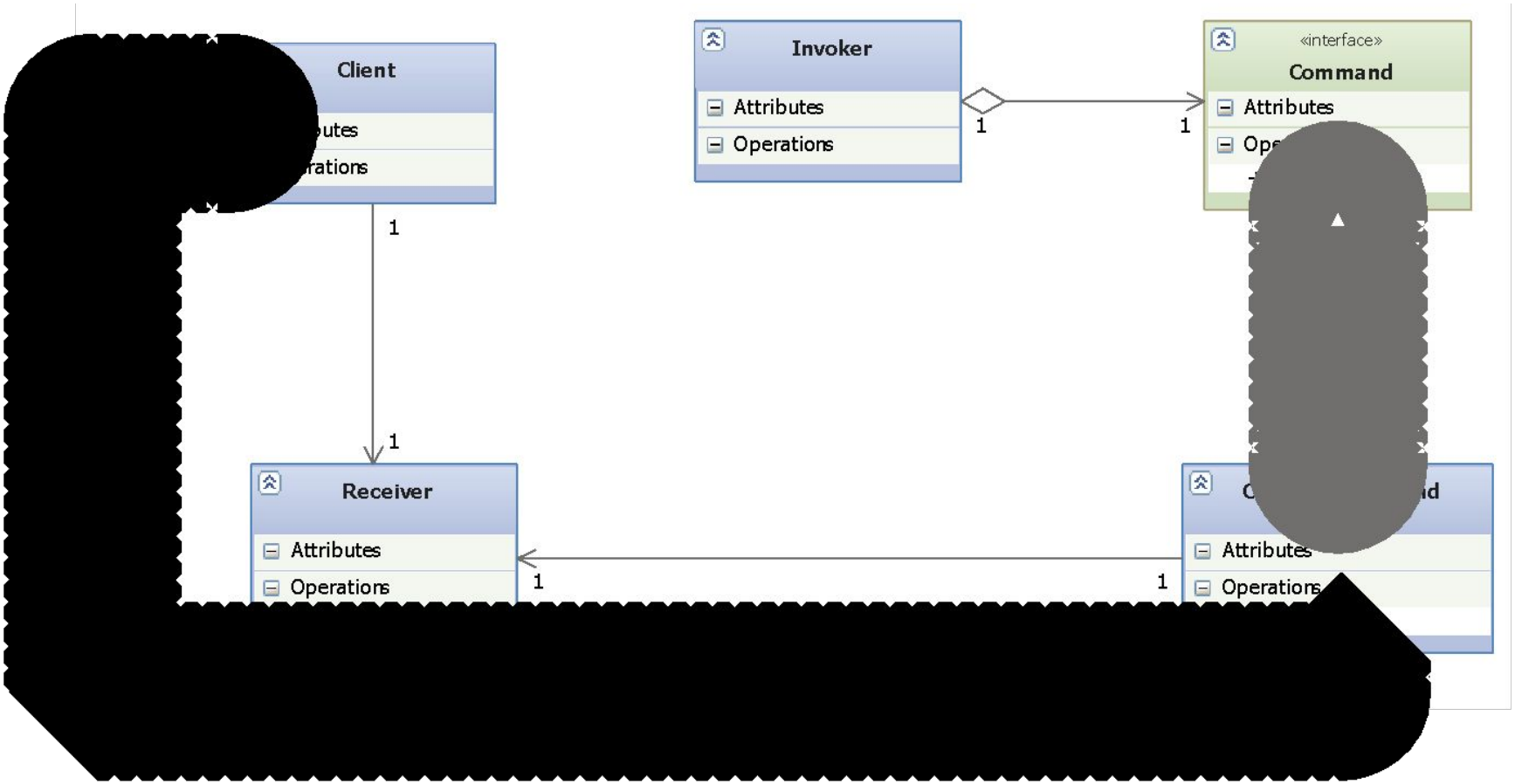
## Команда- Проблема

Необходимо послать объекту запрос, не зная о том, выполнение какой операции запрошено и кто будет получателем.

## Команда- Решение

Инкапсулировать запрос как объект. "Клиент" создает объект "КонкретнаяКоманда", который вызывает операции получателя для выполнения запроса, "Инициатор" отправляет запрос, выполоняя операцию "Команды" Выполнить(). "Команда" объявляет интерфейс для выполнения операции, "КонкретнаяКоманда" определяет связь между объектом "Получатель" и операцией Действие (), и, кроме того, реализует операцию Выполнить() путем вызова соответствующих операций объекта "Получатель". "Клиент" создает экземпляр класса "КонкретнаяКоманда" и устанавливает его получателя, "Инициатор" обращается к команде для выполнения запроса, "Получатель" (любой класс) располагает информацией о способах выполнения операций, необходимых для выполнения запроса.

# Команда - Структура



## Команда- Преимущество

Паттерн "Команда" разрывает связь между объектом, инициирующим операции, и объектом, имеющим информацию о том, как ее выполнить, кроме того создается объект "Команда", который можно расширять и манипулировать им как объектом.

- Наблюдатель (Observer)

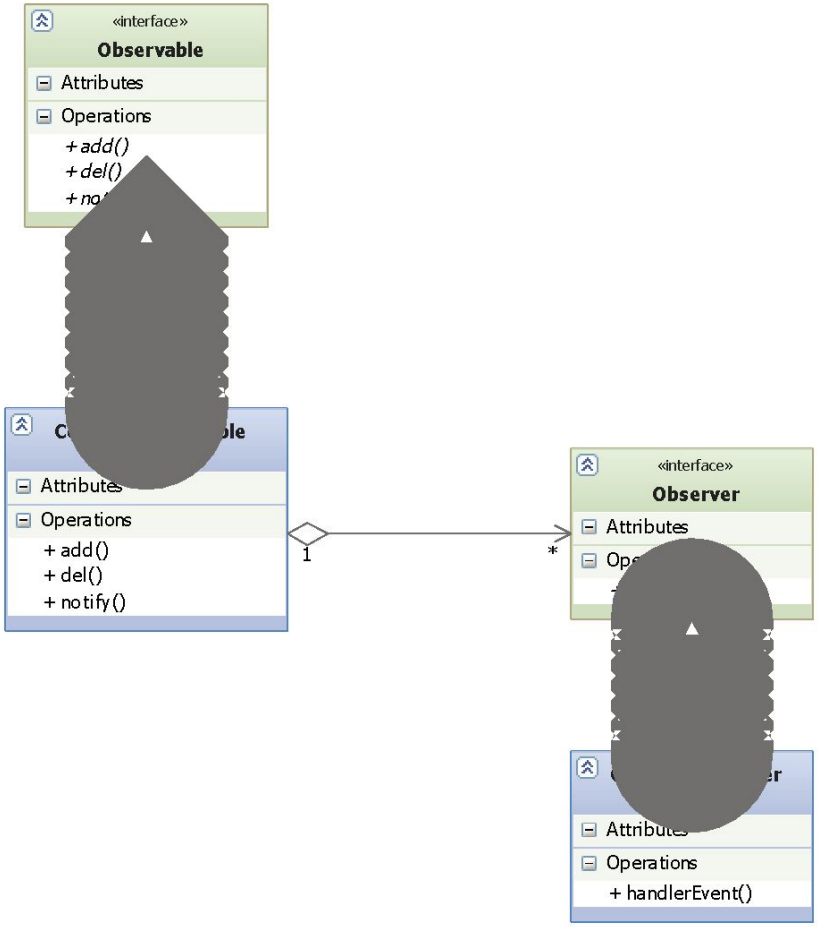
## Наблюдатель - Проблема

Один объект ("Подписчик") должен знать об изменении состояний или некоторых событиях другого объекта. При этом необходимо поддерживать низкий уровень связывания с объектом - "Подписчиком".

## Наблюдатель - Решение

Определить интерфейс "Подписки". Объекты - подписчики реализуют этот интерфейс и динамически регистрируются для получении информации о некотором событии. Затем при реализации условленного события оповещаются все объекты - подписчики.

# Наблюдатель - Структура



- Посетитель (Visitor)

## Посетитель- Проблема

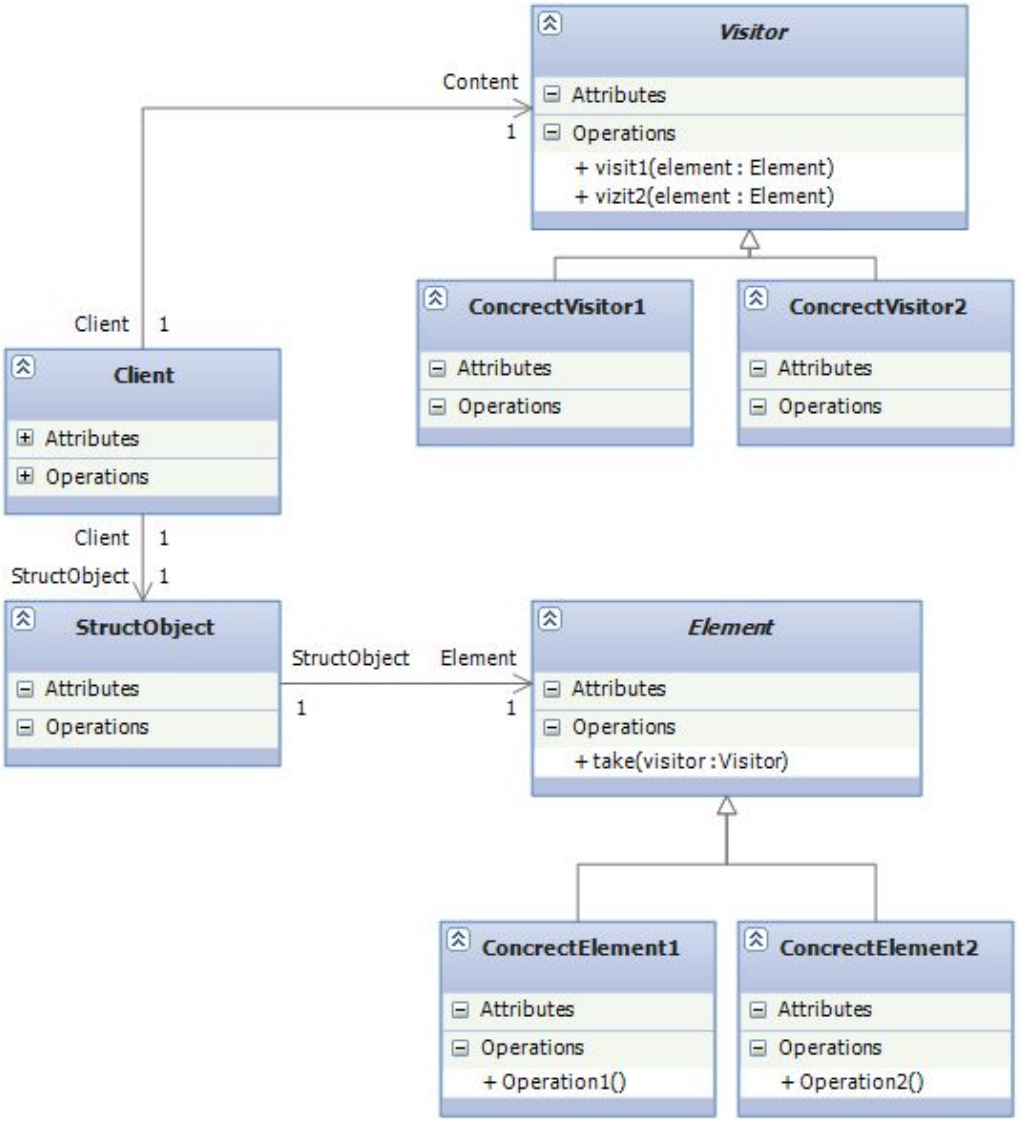
Над каждым объектом некоторой структуры выполняется операция. Определить новую операцию, не изменяя классы объектов.

# Посетитель- Решение

Клиент, использующий данный паттерн, должен создать объект класса "КонкретныйПосетитель", а затем посетить каждый элемент структуры. "Посетитель" объявляет операцию "Посетить" для каждого класса "КонкретныйЭлемент" (имя и сигнатура данной операции идентифицируют класс, элемент которого посещает "Посетитель" - то есть, посетитель может обращаться к элементу напрямую). "КонкретныйПосетитель" реализует все операции, обьявленные в классе "Посетитель". Каждая операция реализует фрагмент алгоритма, определенного для класса соответствующего объекта в структуре.

Класс "КонкретныйПосетитель"предоставляет контекст для этого алгоритма и сохраняет его локальное состояние. "Элемент" определяет операцию "Принять", которая принимает "Посетителя" в качестве аргумента, "КонкретныйЭлемент" реализует операцию "Принять", которая принимает "Посетителя" в качестве аргумента. "СтруктураОбьекта" может перечислить свои аргументы и предоставить посетителю высокоуровневый интерфейс для посещения своих элементов.

# Посетитель- Структура



## Посетитель- Рекомендации

Логично использовать, если в структуре присутствуют объекты многих классов с различными интерфейсами, и необходимо выполнить над ними операции, зависящие от конкретных классов, или если классы, устанавливающие структуру обьектов изменяются редко, но новые операции над этой структурой добавляются часто.

## Посетитель- Преимущества

Упрощается добавление новых операций, объединяет родственные операции в классе "Посетитель".

## Посетитель- Недостатки

Затруднено добавление новых классов "КонкретныйЭлемент", поскольку требуется объявление новой абстрактной операции в классе "Посетитель".

- Посредник (Mediator)

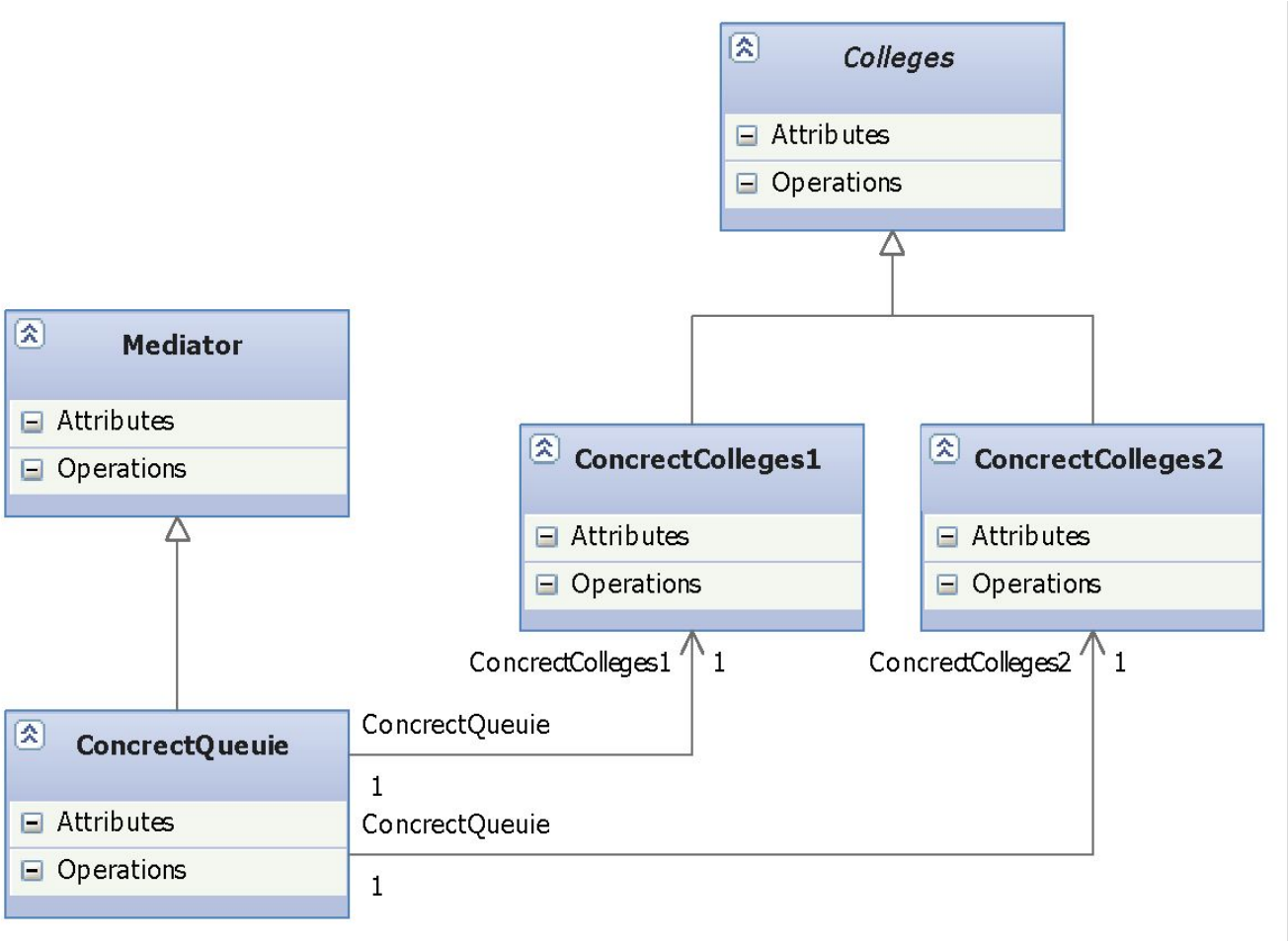
## Посредник- Проблема

Затруднено добавление новых классов "КонкретныйЭлемент", поскольку требуется объявление новой абстрактной операции в классе "Посетитель".

## Посредник- Решение

Посредник" определяет интерфейс для обмена информацией с объектами "Коллеги", "КонкретныйПосредник" координирует действия обьектов "Коллеги". Каждый класс "Коллеги" знает о своем объекте "Посредник", все "Коллеги" обмениваются информацией только с посредником, при его отсутствии им пришлось бы обмениваться информацией напрямую. "Коллеги" посылают запросы посреднику и получают запросы от него. "Посредник" реализует кооперативное поведения, пересылая каждый запрос одному или нескольким "Коллегам".

# Посредник- Структура



# Посредник - Преимущества

Устраняется связанность между "Коллегами", централизуется управление.

- Состояние (State)

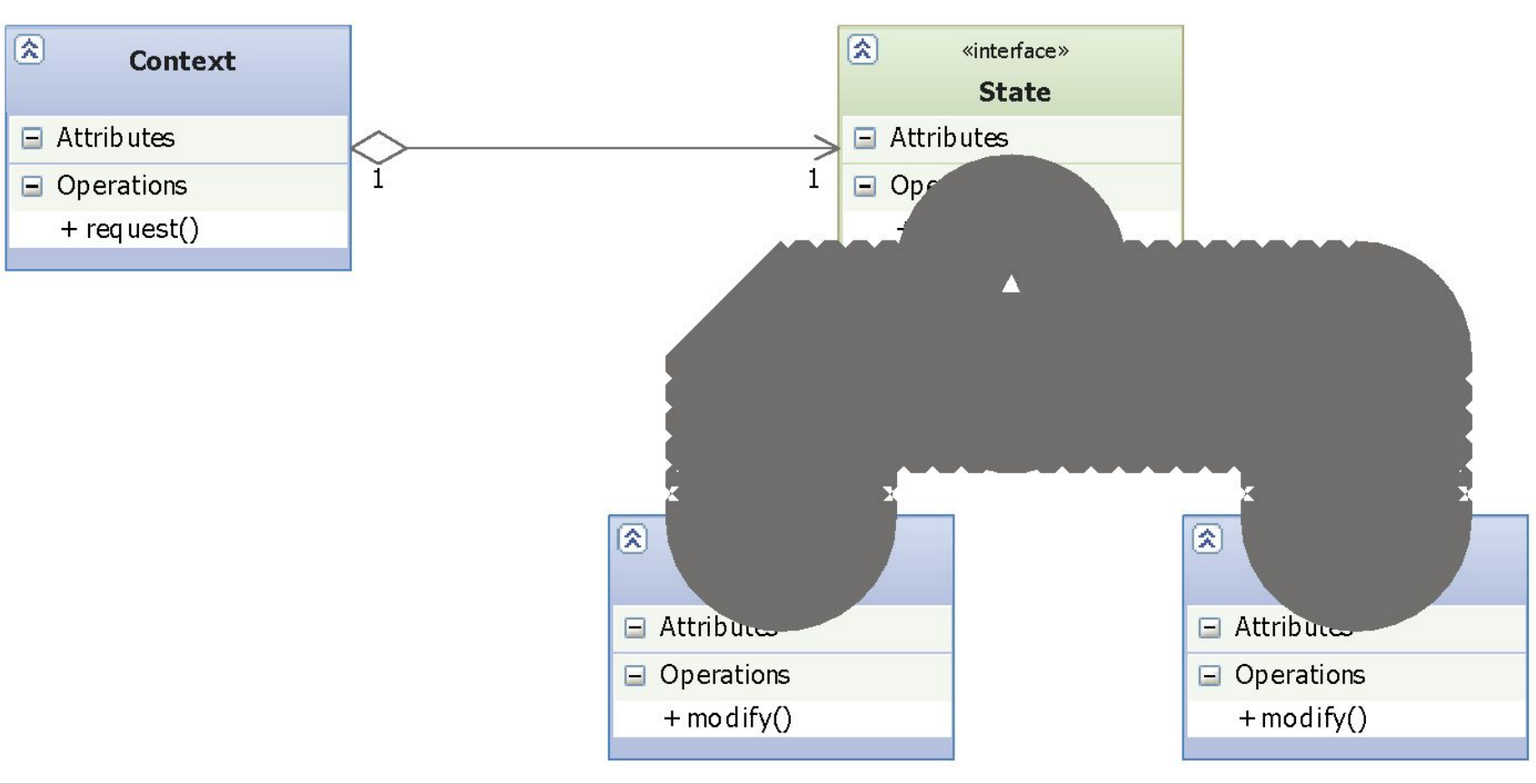
## Состояние - Проблема

Варьировать поведение объекта в зависимости от его внутреннего состояния

## Состояние - Решение

Класс "Контекст" делегирует зависящие от состояния запросы текущему объекту "КонкретноеСостояние" (хранит экземпляр подкласса "КонкретноеСостояние", которым определяется текущее состояние), и определяет интерфейс, представляющий интерес для клиентов.
"КонкретноеСостояние" реализует поведение, ассоциированное с неким состоянием объекта "Контекст". "Состояние" определяет интерфейс для инкапсуляции поведения, ассоциированного с конкретным экземпляром "Контекста".

# Состояние - Структура



## Состояние - Преимущества

Локализует зависящее от состояния поведение и делит его на части, соответствующие состояниям, переходы между состояниями становятся явными.

- Стратегия (Strategy)

## Стратегия - Проблема

Спроектировать изменяемые, но надежные алгоритмы или стратегии

## Стратегия - Решение

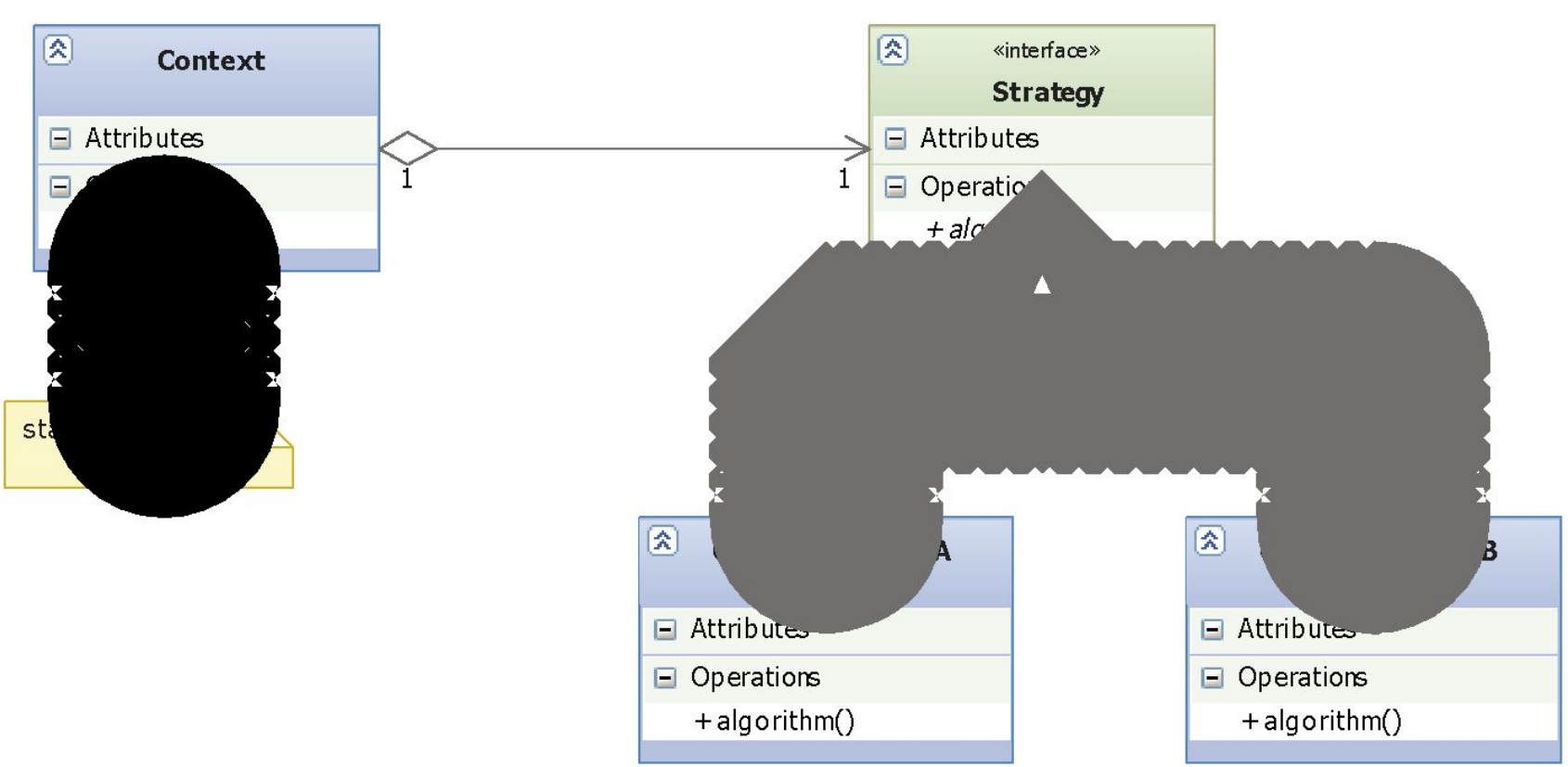
Определить для каждого алгоритма или стратегии отдельный класс со стандартным интерфейсом.

# Стратегия - Пример

Обеспечение сложной логики вычисления стоимости товаров с учетом сезонных скидок, скидок постоянным клиентам и т. п. Данная стратегия может изменяться.

Создается несколько классов "Стратегия", каждый из которых содержит один и тот же полиморфный метод "ЦенаРассчитать". В качестве параметров в этот метод передаются данные о продаже. Объект стратегии связывается с контекстным объектом (тем объектом, к которому применяется алгоритм).

# Стратегия - Структура



- Хранитель (Memento)

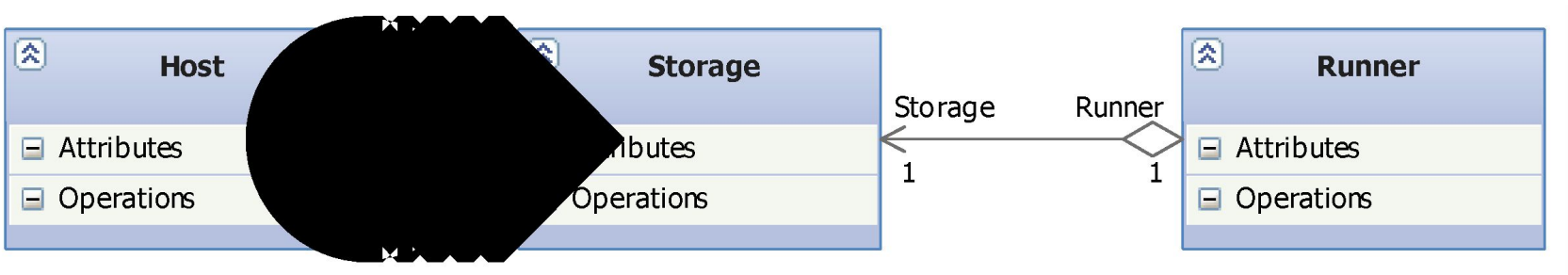
## Хранитель - Проблема

Необходимо зафиксировать поведение объекта для реализации, например, механизма отката.

# Хранитель - Решение

Зафиксировать и вынести (не нарушая инкапсуляции) за пределы объекта его внутреннее состояние так, чтобы впоследствии можно было восстановить в нем объект. "Хранитель" сохраняет внутреннее состояние объекта "Хозяин" и запрещает доступ к себе всем другим объектам кроме "Хозяина", который имеет доступ ко всем данным для восстановления в прежнем состоянии. "Посыльный" может лишь передавать "Хранителя" другим объектам. "Хозяин" создает "Хранителя", содержащего снимок текущего внутреннего состояния и использует "Хранитель" для восстановления внутреннего состояния. "Посыльный" отвечает за сохранение "Хранителя", при этом не производит никаких операций над "Хранителем" и не исследует его внутреннее содержимое. "Посыльный" запрашивает "Хранитель" у "Хозяина", некоторое время держит его у себя, а затем возвращает "Хозяину".

# Хранитель - Структура



## Хранитель - Преимущества

Не раскрывается информация, которая доступна только "Хозяину", упрощается структура "Хозяина".

## Хранитель - Недостатки

С использованием "Хранителей" могут быть связаны значительные издержки, если "Хозяин" должен копировать большой объём информации, или если копирование должно проводиться часто.

- Цепочка обязанностей (Chain of Responsibility)

# Цепочка обязанностей - Проблема

Запрос должен быть обработан несколькими объектами.

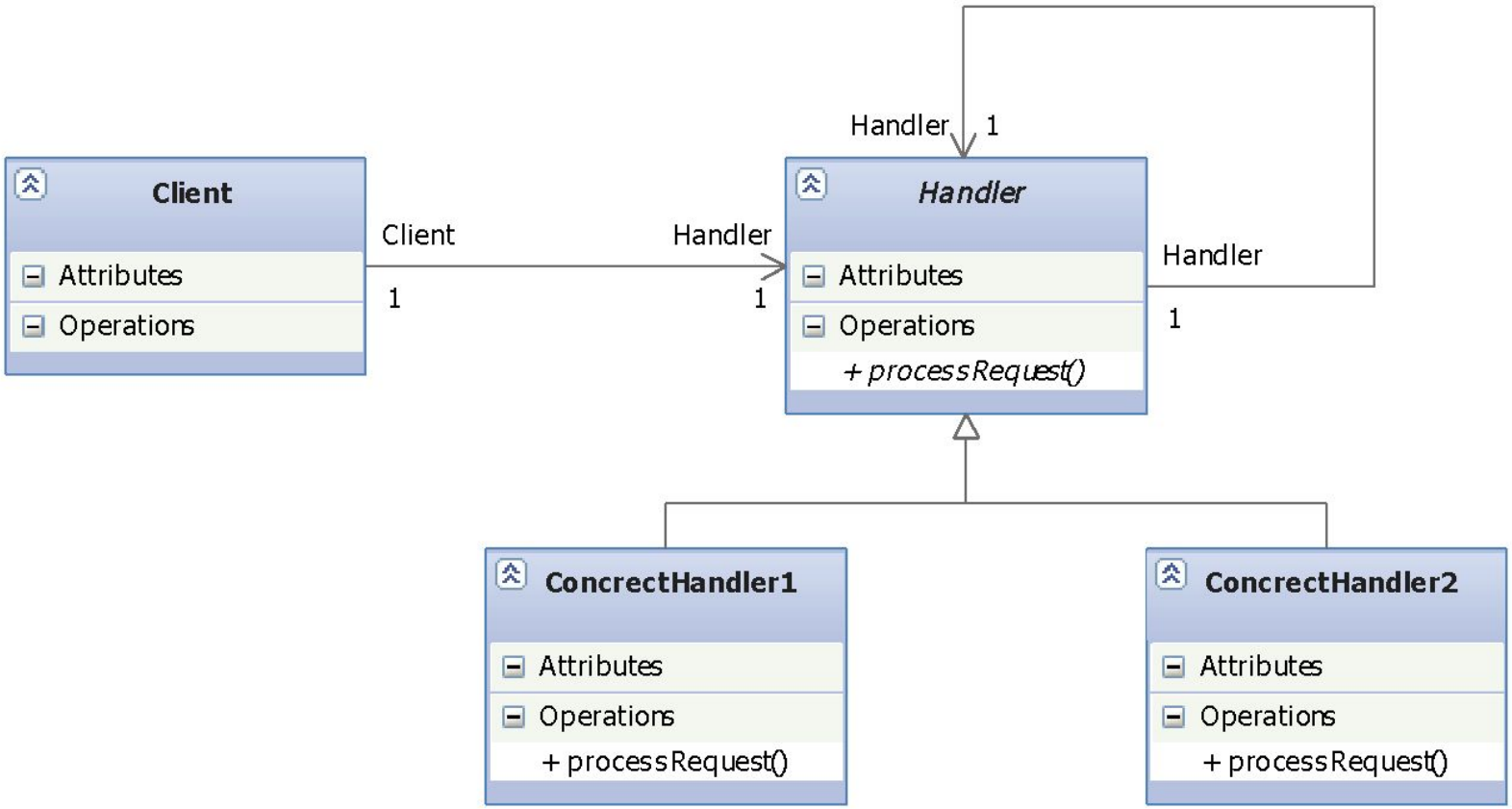
# Цепочка обязанностей - Рекомендации

Логично использовать данный паттерн, если имеется более одного объекта, способного обработать запрос и обработчик заранее неизвестен (и должен быть найден автоматически) или если весь набор обьектов, которые способны обработать запрос, должен задаваться автоматически.

# Цепочка обязанностей - Решение

Связать объекты - получатели запроса в цепочку и передать запрос вдоль этой цепочки, пока он не будет обработан. "Обработчик" определяет интерфейс для обработки запросов, и, возможно, реализует связь с преемником, "КонкретныйОбработчик" обрабатывает запрос, за который отвечает, имеет доступ к своему преемнику ("КонкретныйОбработчик" направляет запрос к своему преемнику, если не может обработать запрос сам.

# Цепочка обязанностей - Структура



# Цепочка обязанностей - Преимущества

Ослабляется связанность (объект не обязан "знать", кто именно обработает его запрос).

## Цепочка обязанностей - Недостатки

Нет гарантий, что запрос будет обработан, поскольку он не имеет явного получателя.

- Шаблонный метод (Template Method)

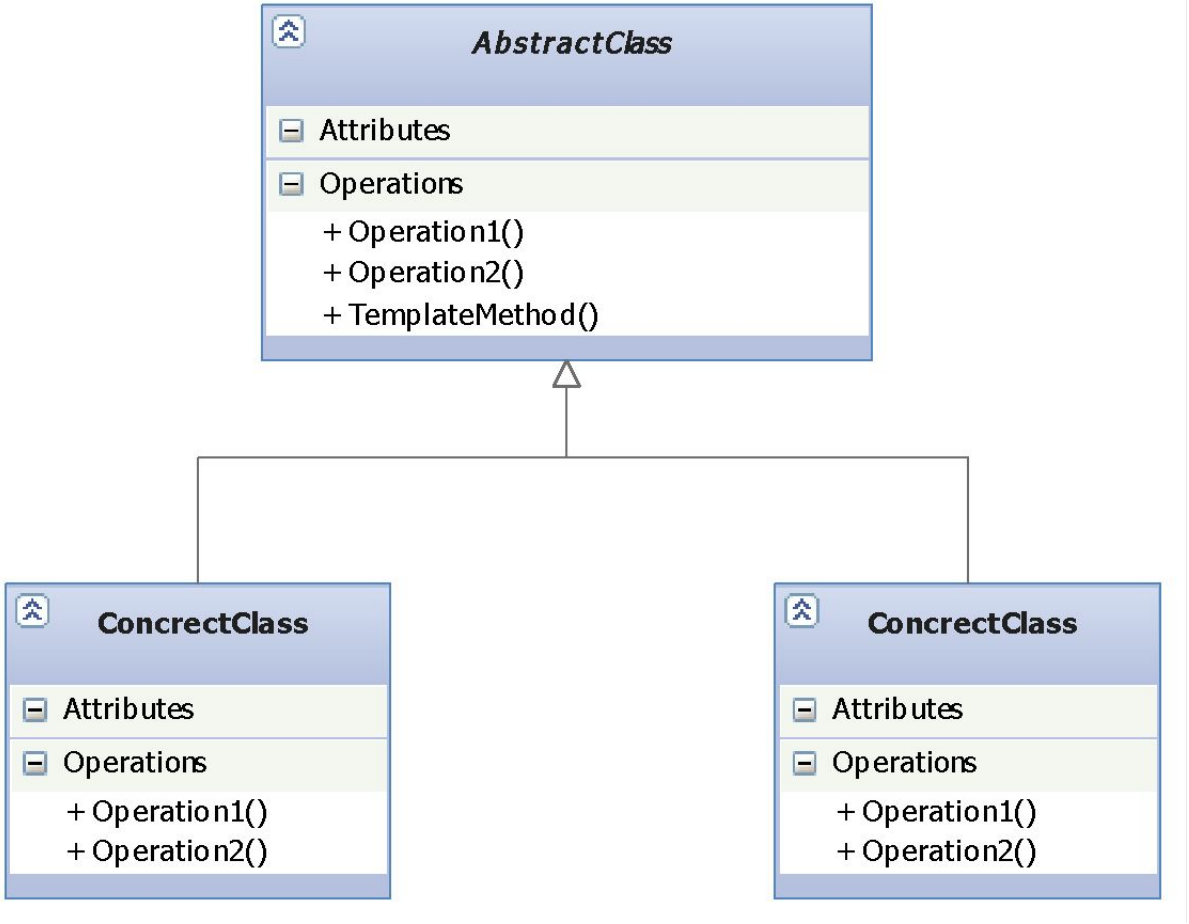
## Шаблонный метод - Проблема

Определить алгоритм и реализовать возможность переопределения некоторых шагов алгоритма для подклассов (без изменения общей структуры алгоритма.

## Шаблонный метод - Решение

АбстрактныйКласс" определяет абстрактные Операции(), замещаемые в конкретных подклассах для реализации шагов алгоритма, и реализует ШаблонныйМетод(), определяющий "скелет" алгоритма. "КонкретныйКласс" релизует Операции(), выполняющие шаги алгоритма способом, который зависит от подкласса. "КонкретныйКласс" предполагает, что инвариантные шаги алгоритма будут выполнены в "АбстрактномКлассе".

# Шаблонный метод - Структура



# Порождающие паттерны проектирования

- Абстрактная фабрика
- Одиночка
- Прототип
- Строитель
- Фабричный метод

- Абстрактная фабрика (Abstract Factory, Factory)

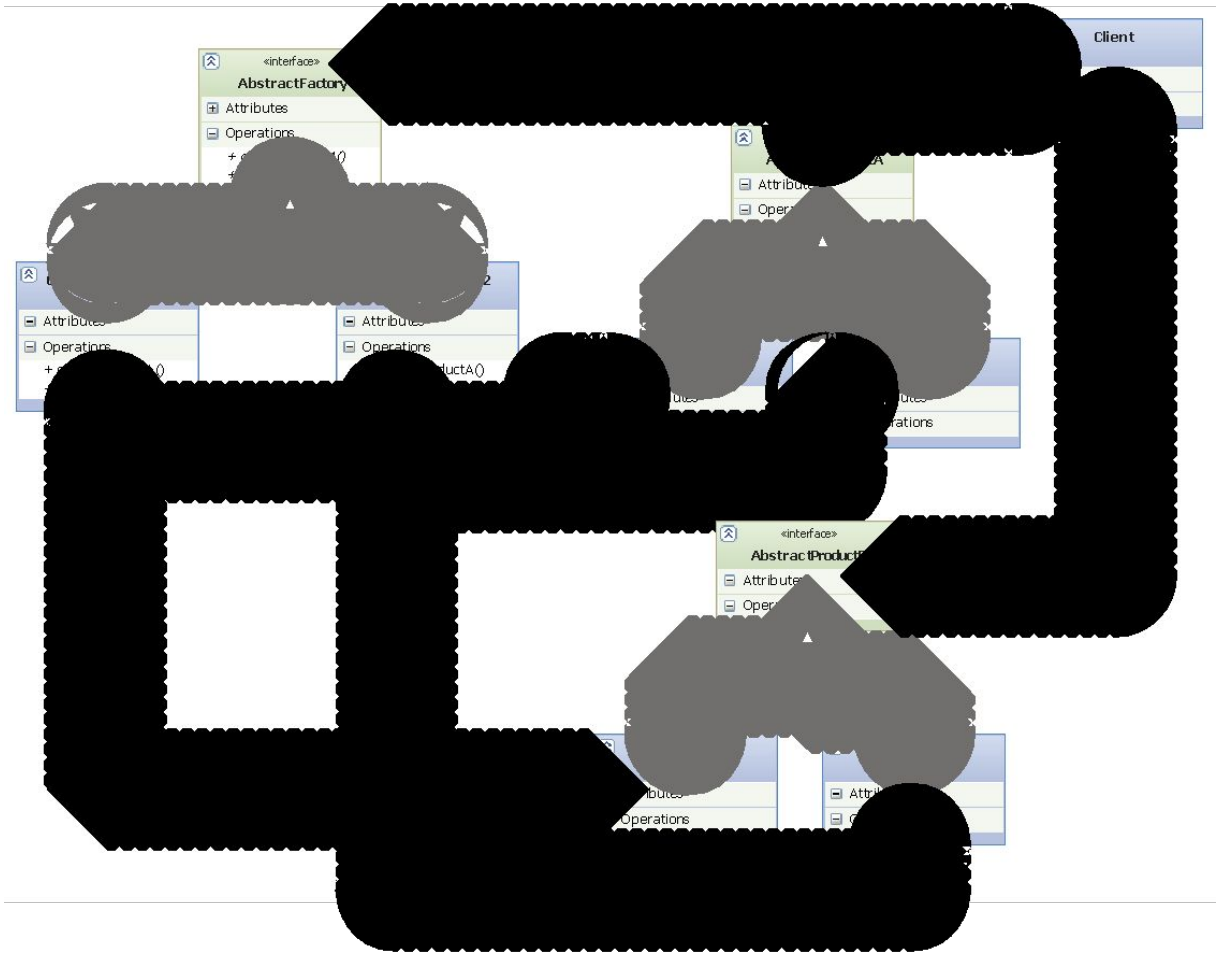
## Абстрактная фабрика – Проблема

Создать семейство взаимосвязанных или взаимозависимых обьектов (не специфицируя их конкретных классов).

## Абстрактная фабрика – Решение

Создать абстрактный класс, в котором объявлен интерфейс для создания конкретных классов.

# Абстрактная фабрика – Структура



## Абстрактная фабрика – Преимущества

Изолирует конкретные классы. Поскольку "Абстрактная фабрика" инкапсулирует ответственность за создание классов и сам процесс их создания, то она изолирует клиента от деталей реализации классов. Упрощена замена "Абстрактной фабрики", поскольку она используется в приложении только один раз при инстанцировании.

## Абстрактная фабрика – Недостатки

Интерфейс "Абстрактной фабрики" фиксирует набор обьектов, которые можно создать. Расширение "Абстрактной фабрики" для изготовления новых обьектов часто затруднительно.

- Одиночка (Singleton)

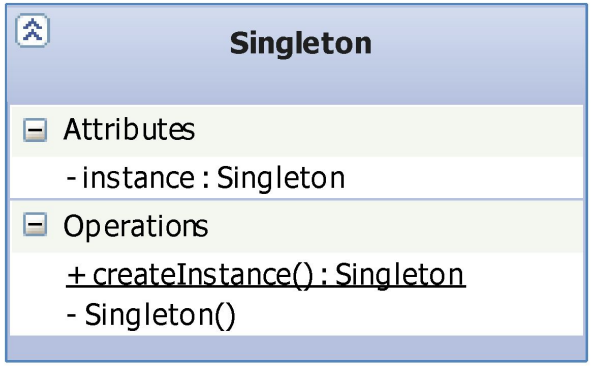
## Одиночка - Проблема

Создать класс и определить статический метод класса, возвращающий этот единственный объект.

## Одиночка - Решение

Интерфейс "Абстрактной фабрики" фиксирует набор обьектов, которые можно создать. Расширение "Абстрактной фабрики" для изготовления новых обьектов часто затруднительно.

# Одиночка - Струкутура



# Одиночка - Рекомендации

Разумнее создавать именно статический экземпляр специального класса, а не объявить требуемые методы статическими, поскольку при использовании методов экземпляра можно применить механизм наследования и создавать подклассы. Статические методы в языках программирования не полиморфны и не допускают перекрытия в производных классах.

Решение на основе создания экземпляра является более гибким, поскольку впоследствии может потребоваться уже не единственный экземпляр объекта, а несколько.



# Прототип (Prototype)

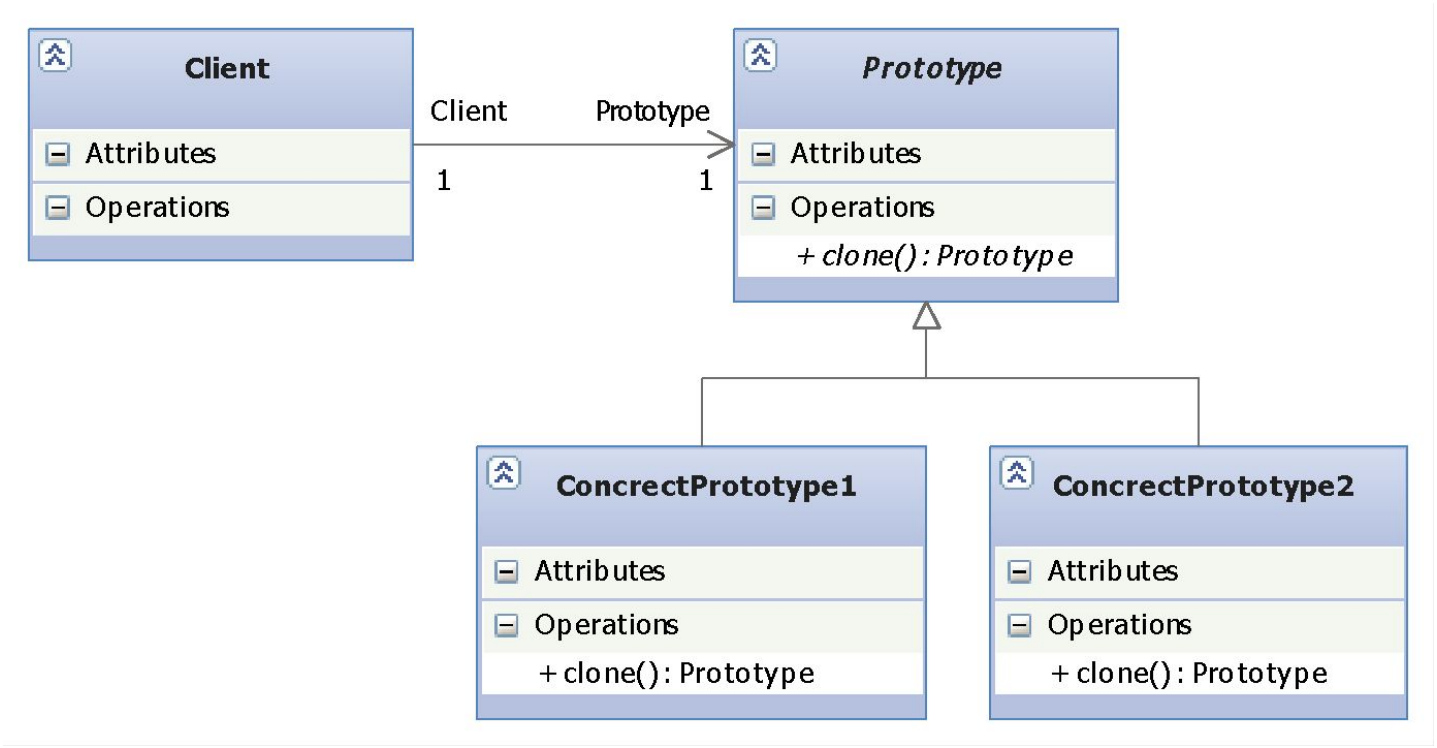
# Прототип - Проблема

Система не должна зависеть от того, как в ней создаются, компонуются и представляются объекты.

# Прототип - Решение

Создавать новые объекты с помощью паттерна - прототипа. "Прототип" объявляет интерфейс для клонирования самого себя. "Клиент" создает новый объект, обращаясь к "Прототипу" с запросом клонировать "Прототип".

# Прототип - Структура



# Строитель (Builder)

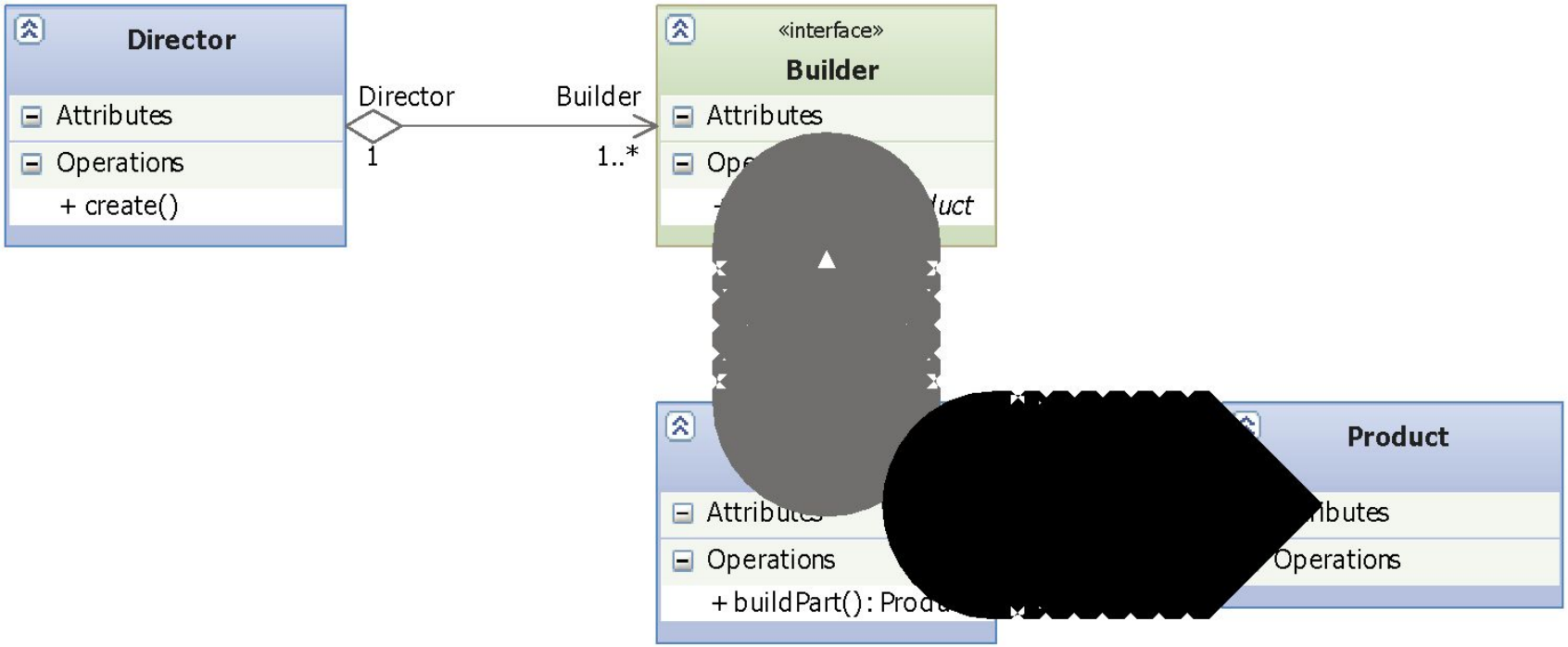
# Строитель - Проблема

Отделить конструирование сложного объекта от его представления, так чтобы в результате одного и того же конструирования могли получаться различные представления. Алгоритм создания сложного объекта не должен зависеть от того, из каких частей состоит объект и как они стыкуются между собой.

# Строитель - Решение

"Клиент" создает объект - распорядитель "Директор" и конфигурирует его объектом - "Строителем". "Директор" уведомляет "Строителя" о том, что нужно построить очередную часть "Продукта". "Строитель" обрабатывает запросы "Директора" и добавляет новые части к "Продукту", затем "Клиент" забирает "Продукт" у "Строителя".

# Строитель - Структура



# Строитель - Преимущество

Объект "Строитель" предоставляет объекту "Директор" абстрактный интерфейс для конструирования "Продукта", за которым может скрыть представление и внутреннюю структуру продукта, и , кроме того, процесс сборки "продукта". Для изменения внутреннего представления "Продукта" достаточно определить новый вид "Строителя". Данный паттерн изолирует код, реализующий создание объекта и его представление.

# Фабричный метод (Factory Method)

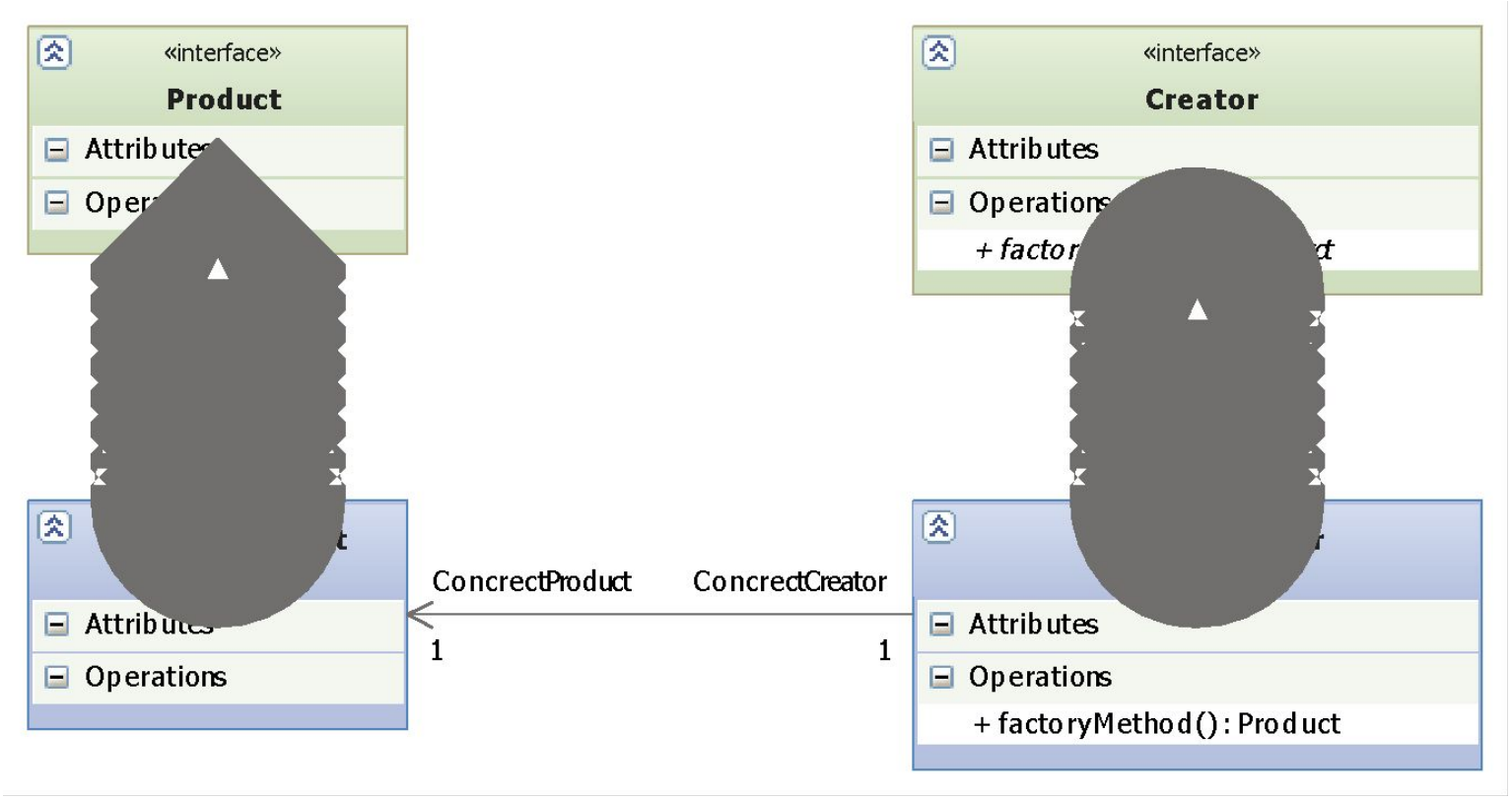
# Фабричный метод - Проблема

Определить интерфейс для создания объекта, но оставить подклассам решение о том, какой класс инстанцировать, то есть, делегировать инстанцирование подклассам.

# Фабричный метод - Решение

Абстрактный класс "Создатель" объявляет ФабричныйМетод, возвращающий объект типа "Продукт" (абстрактный класс, определяющий интерфейс обьектов, создаваемых фабричным методом). "Создатель также может определить реализацию по умолчанию ФабричногоМетода, который возвращает "КонкретныйПродукт". "КонкретныйСоздатель" замещает ФабричныйМетод, возвращающий объект "КонкретныйПродукт". "Создатель" "полагается" на свои подклассы в определении ФабричногоМетода, возвращающего объект "КонкретныйПродукт".

# Фабричный метод - Структура



# Фабричный метод - Преимущества

Избавляет проектировщика от необходимости встраивать в код зависящие от приложения классы.

# Фабричный метод - Недостатки

Возникает дополнительный уровень подклассов.

# The End…