Шаблоны Gang-of-Four

Design Patterns: Elements of Reusable Object-Oriented Software" (Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides

(Gang of Four, GoF))

Шаблоны GoF (Gang-of-Fou)

- Структурные паттерны
  - Применяются при компоновке системы на основе классов и объектов (Adapter, Facade, Decorator, Proxy)
- Порождающие паттерны
  - Предназначены для создания объектов, позволяя системе оставаться независимой как от самого процесса порождения, так и от типов порождаемых объектов (Factory Method, Abstract Factory,
- Паттерны поведения
  - Описывают правильные способы организации взаимодействия между используемыми объектами (Template Method, Strategy)

Структурные паттерны

Структурные паттерны применяются при компоновке системы на основе классов и объектов

- Структурные паттерны уровня класса используют наследование для составления композиций
- Структурные паттерны уровня объекта используют композицию объектов для получения новой функциональности

Адаптер

Структурный паттерн проектирования

Проблема

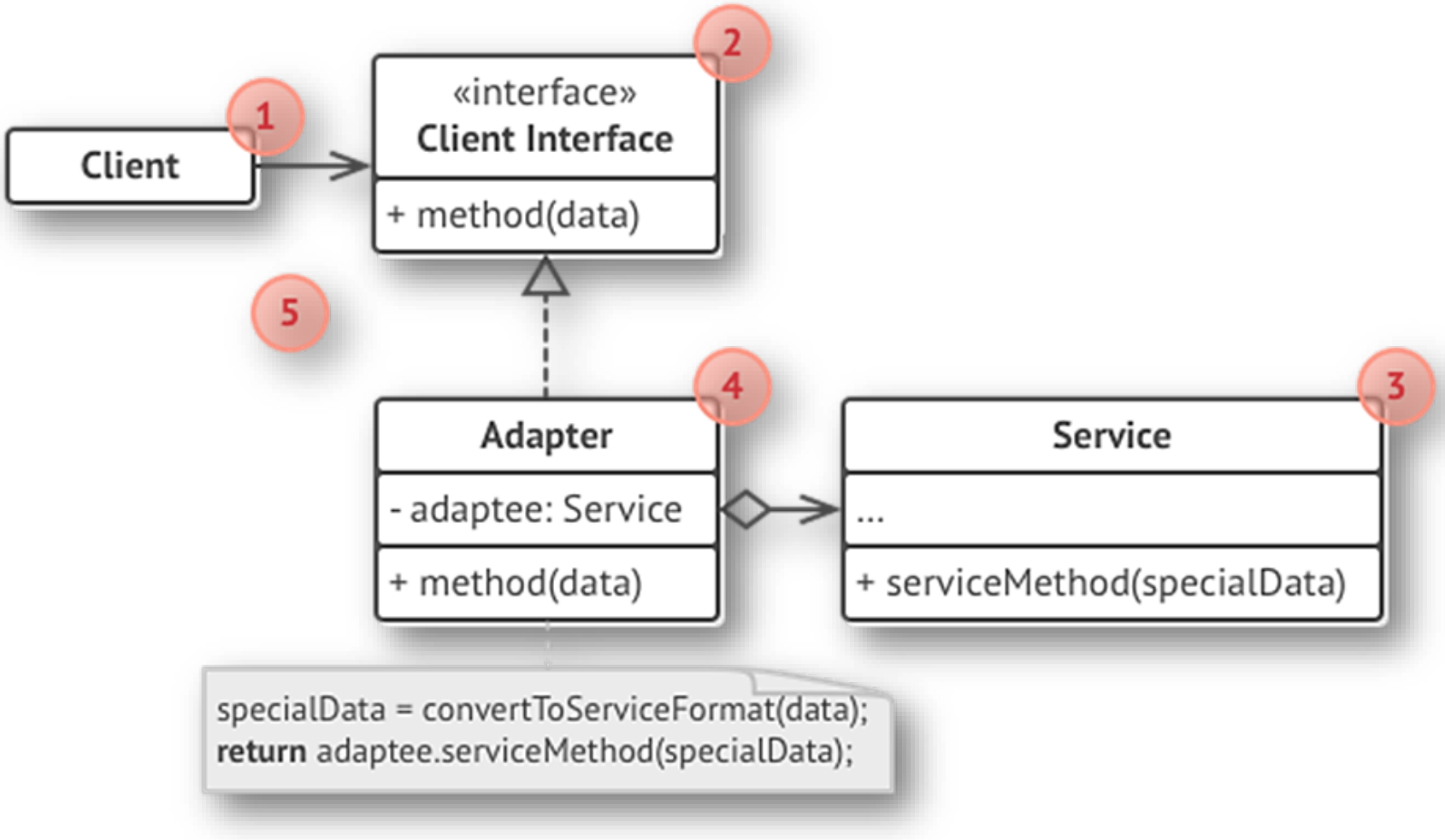
- Классы обладают нужной функциональностью, но имеют при этом несовместимые интерфейсы (набор методов)
- В программном проекте не удается повторно использовать уже существующий код

Решение

- Использовать паттерн Adapter (адаптер) – представляющий собой программную обертку (объект-адаптер) над существующими классами, преобразующую их интерфейсы к виду, пригодному для последующего использования

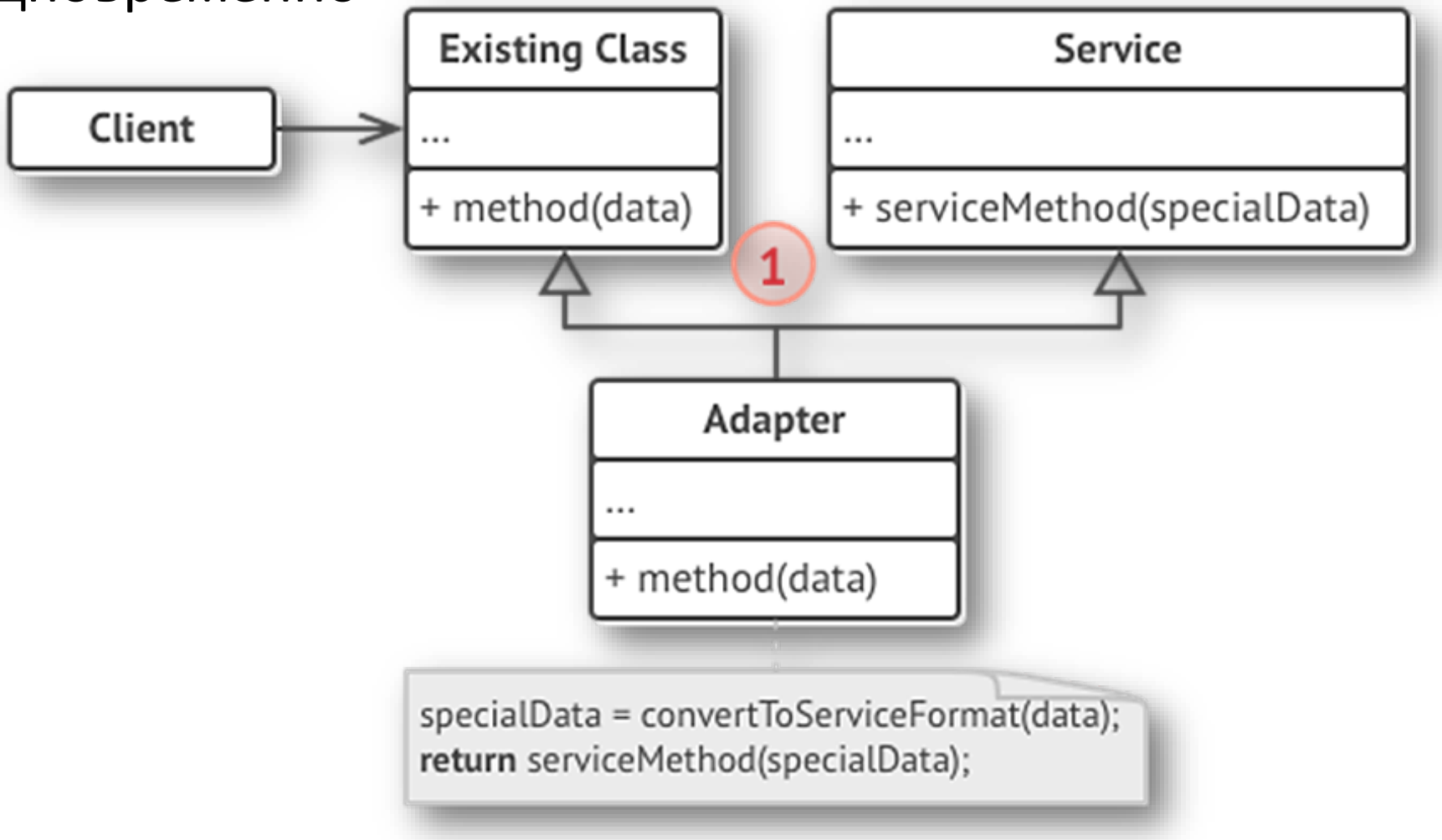
Адаптер объектов

- Эта реализация использует принцип композиции: объект адаптера «оборачивает», то есть содержит ссылку на служебный объект



Адаптер классов

- Эта реализация базируется на наследовании: адаптер наследует оба интерфейса одновременно



Применение адаптера

Есть два класса с неудобными интерфейсами:

- полезный сервис — служебный класс, который вы не можете изменять (он либо сторонний, либо от него зависит другой код);
- один или несколько клиентов — классов приложения, несовместимых с сервисом из-за неудобного или несовпадающего интерфейса.

- Опишите клиентский интерфейс, через который классы приложения смогли бы использовать сторонний класс.
- Создайте класс адаптера, реализовав этот интерфейс.

Применение адаптера

- Поместите в адаптер поле-ссылку на объект-сервис
  - В большинстве случаев, это поле заполняется объектом, переданным в конструктор адаптера.
  - В случае простой адаптации этот объект можно передавать через параметры методов адаптера.
- Реализуйте все методы клиентского интерфейса в адаптере. Адаптер должен делегировать основную работу сервису.
- Приложение должно использовать адаптер только через клиентский интерфейс

Сравнение с другими паттернами

- **Мост** проектируют заранее, чтобы развивать большие части приложения отдельно друг от друга. **Адаптер** применяется постфактум, чтобы заставить несовместимые классы работать вместе.
- **Адаптер** меняет интерфейс существующего объекта. **Декоратор** улучшает другой объект без изменения его интерфейса.
- **Адаптер** предоставляет классу альтернативный интерфейс. **Декоратор** предоставляет расширенный интерфейс. **Заместитель** предоставляет тот же интерфейс.
- **Фасад** задаёт новый интерфейс, **Адаптер** повторно использует старый. **Адаптер** оборачивает только один класс, а **Фасад** оборачивает целую подсистему. **Адаптер** позволяет двум существующим интерфейсам работать вместе, вместо того, чтобы задать полностью новый.
- **Мост**, **Стратегия** и **Состояние** и **Адаптер** имеют схожие структуры классов — построены на принципе «композиции», то есть делегирования работы другим объектам. Но они решают разные проблемы.

Фасад

Структурный паттерн проектирования

Проблема проектирования больших систем

Проблема

- Большие программные системы достаточно сложно проектировать и сопровождать

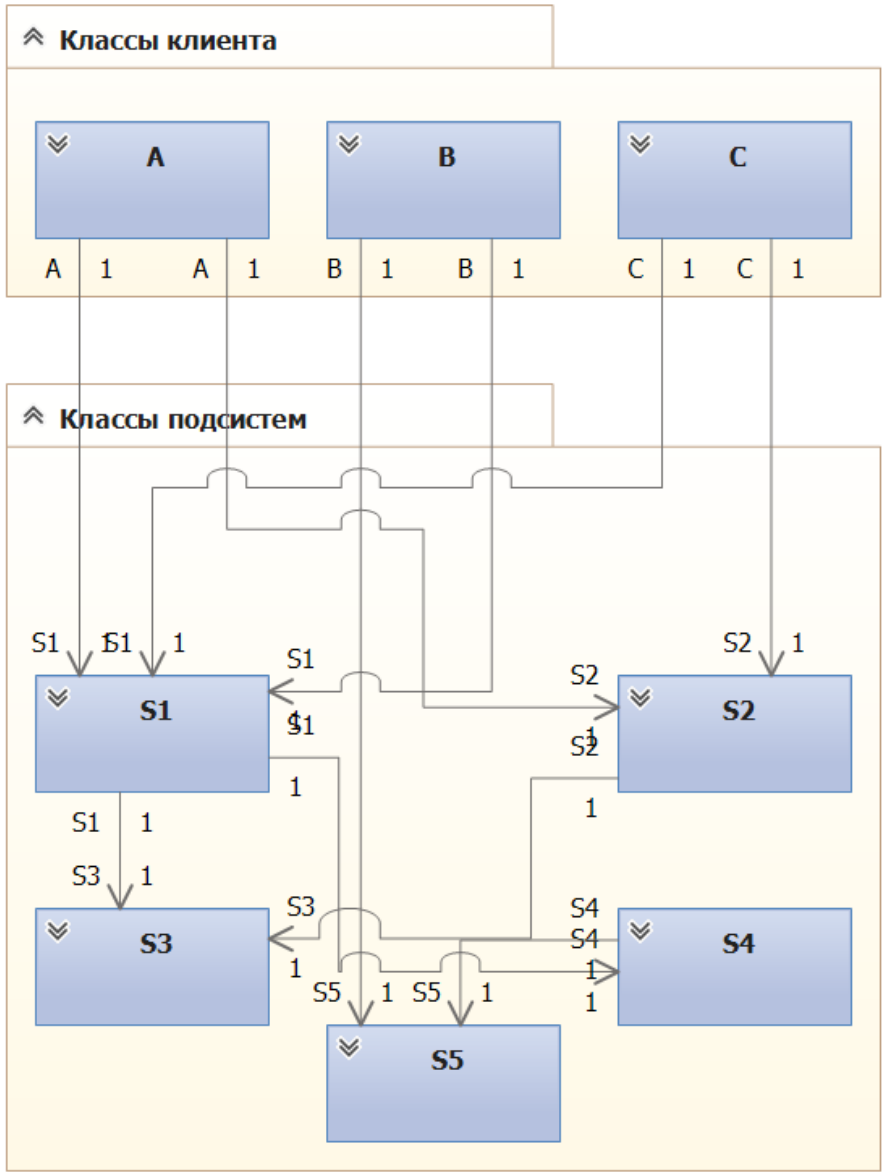
Решение

- Декомпозиция – большие системы принято разбивать на подсистемы

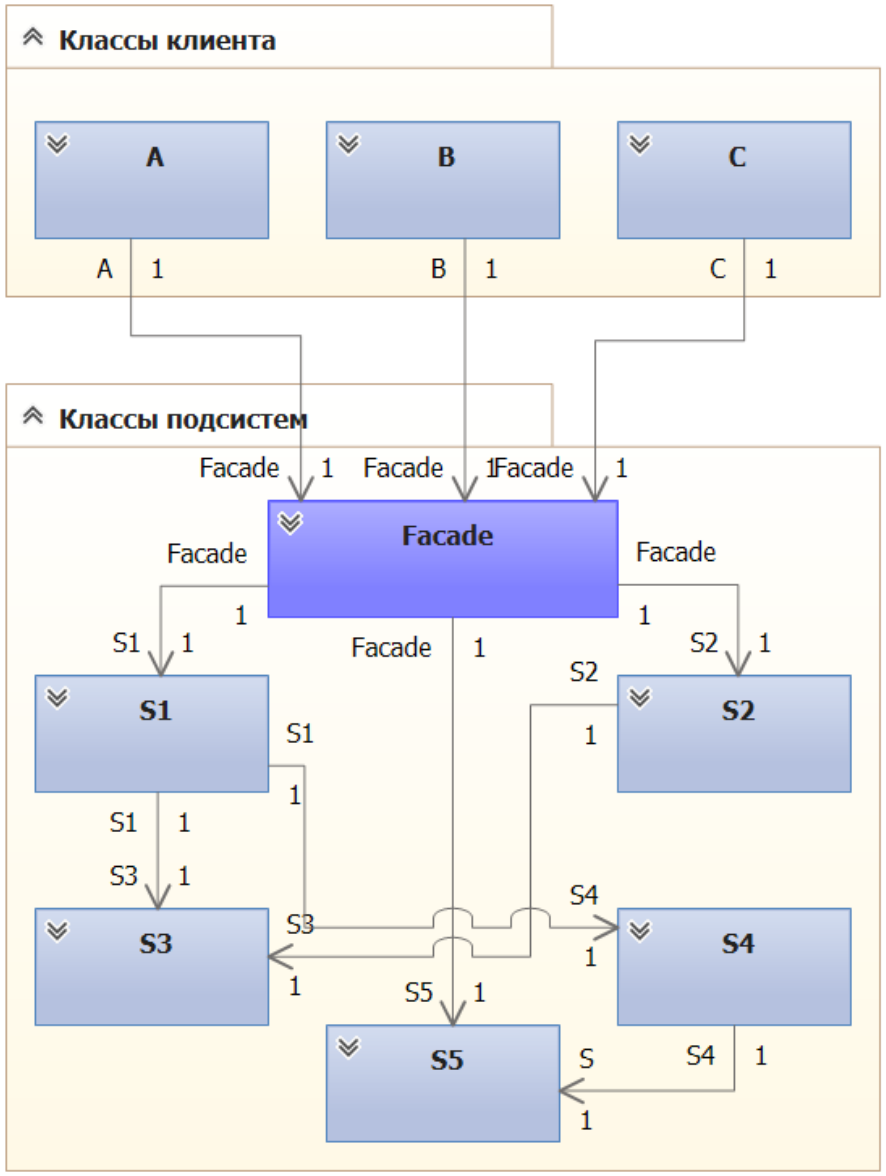
Главная задача, решаемая при проектировании

- сокращение количества связей отношений между классами, т.е. уменьшение зависимостей классов друг от друга

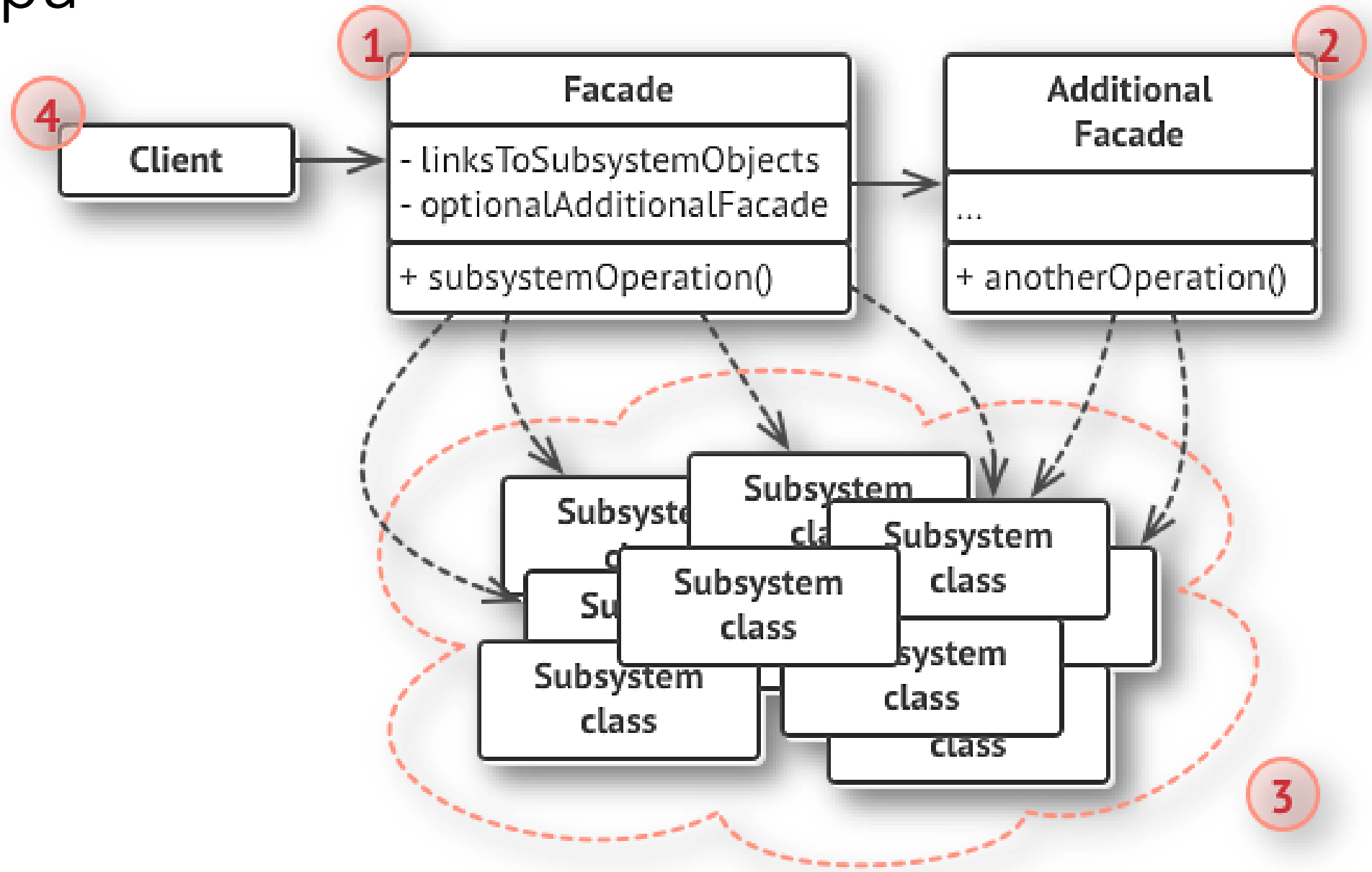
- из клиентского кода происходит много обращений к классам подсистем.



- вводится дополнительный ассоциативный класс Facade, через интерфейс которого происходит взаимодействие с классами подсистем

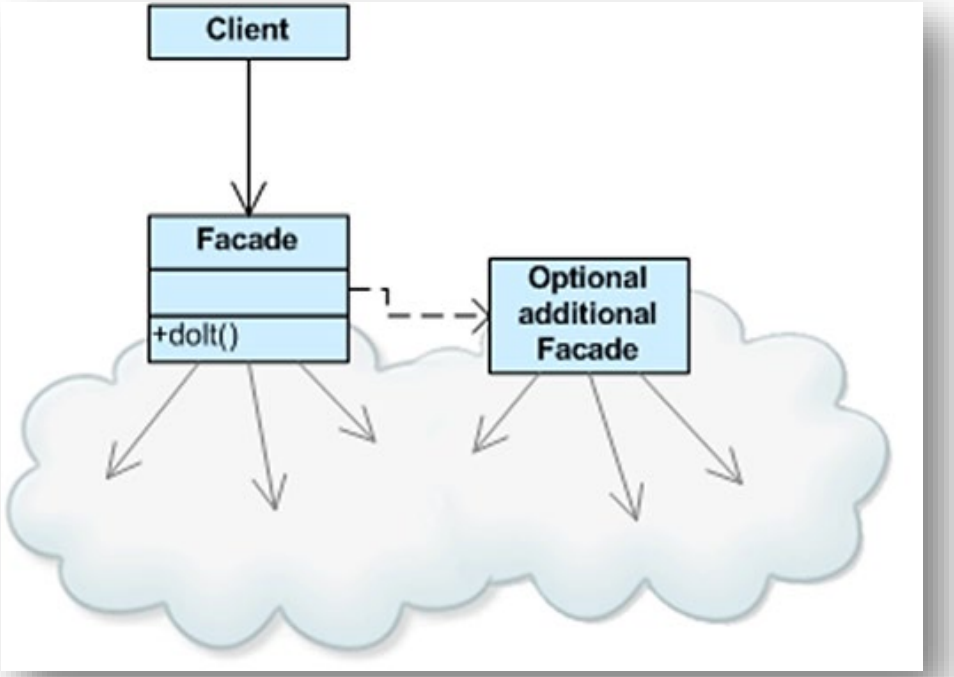


Структура

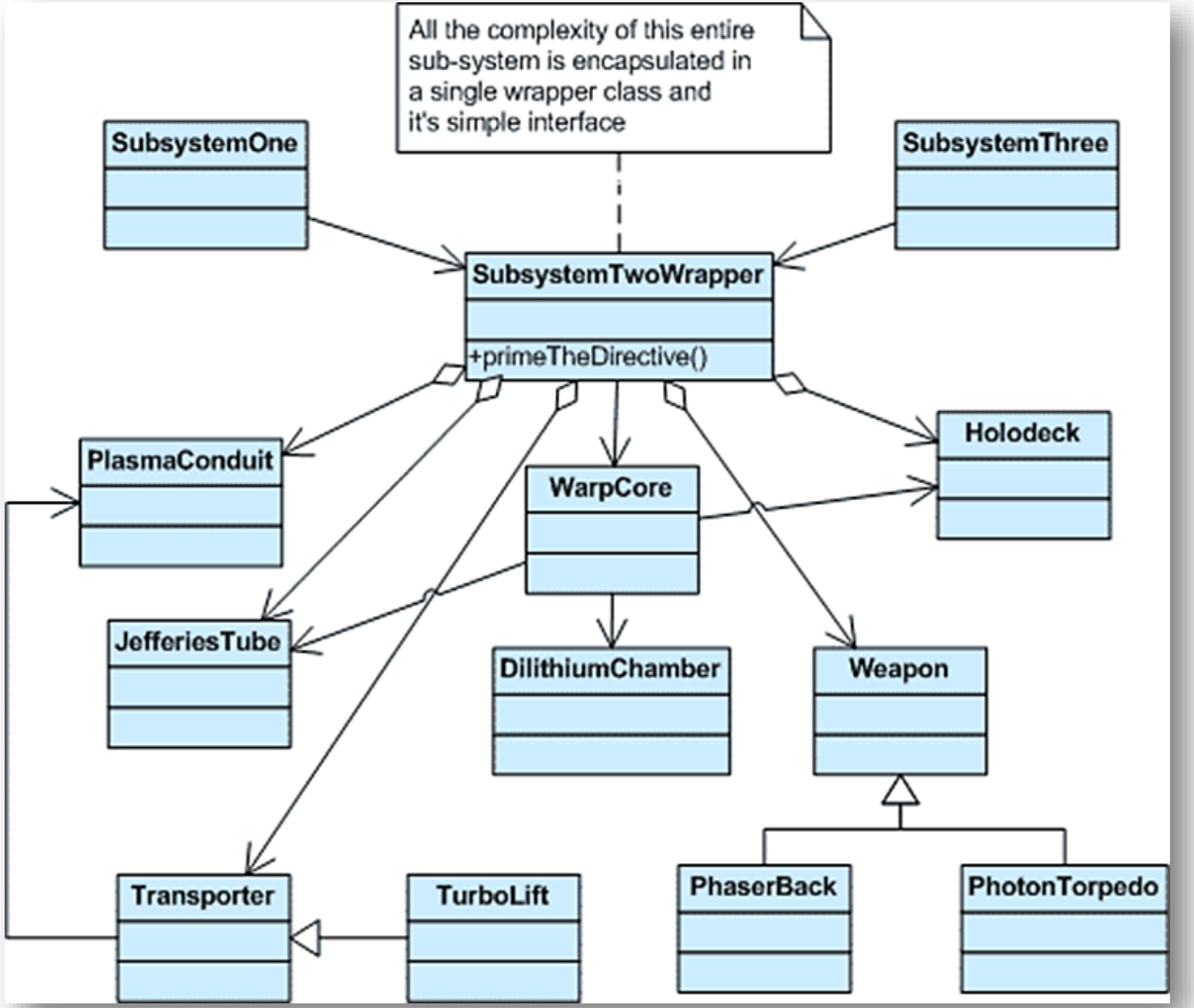

1
Facade
- linksToSubsystemObjects
- optionalAdditionalFacade
+ subsystemOperation()
2
Additional Facade
...
+ anotherOperation()
4
Client
Subsystem class
Subsystem class
Subsystem class
Subsystem class
Subsystem class
3

Фасад

- предоставляет унифицированный интерфейс вместо набора интерфейсов некоторой подсистемы
- определяет интерфейс более высокого уровня, который упрощает использование подсистемы.
- объединяет группу объектов в рамках одного специализированного интерфейса и переадресует вызовы его методов к этим объектам



UML-диаграмма классов паттерна Facade



Использование паттерна Facade

- Определите для подсистемы простой, унифицированный интерфейс.
  - Спроектируйте класс, инкапсулирующий подсистему.
- Создайте класс фасада, реализующий этот интерфейс
  - Вся сложность подсистемы и взаимодействие ее компонентов должны быть скрыты от клиентов
  - Фасад должен правильно инициализировать объекты подсистемы
  - Фасад переадресует пользовательские запросы подходящим методам подсистемы
- Клиент использует только Фасад
  - Изменения в подсистеме будут затрагивать только код фасада, а клиентский код останется рабочим
- Рассмотрите вопрос о целесообразности создания дополнительных "фасадов".

Декоратор

Структурный паттерн проектирования

Проблема

- Требуется добавить новые обязанности в поведении или состоянии отдельных объектов во время выполнения программы.
- Использование наследования не представляется возможным, поскольку это решение статическое и распространяется целиком на весь класс

Решение

- Паттерн Decorator динамически добавляет новые обязанности объекту.

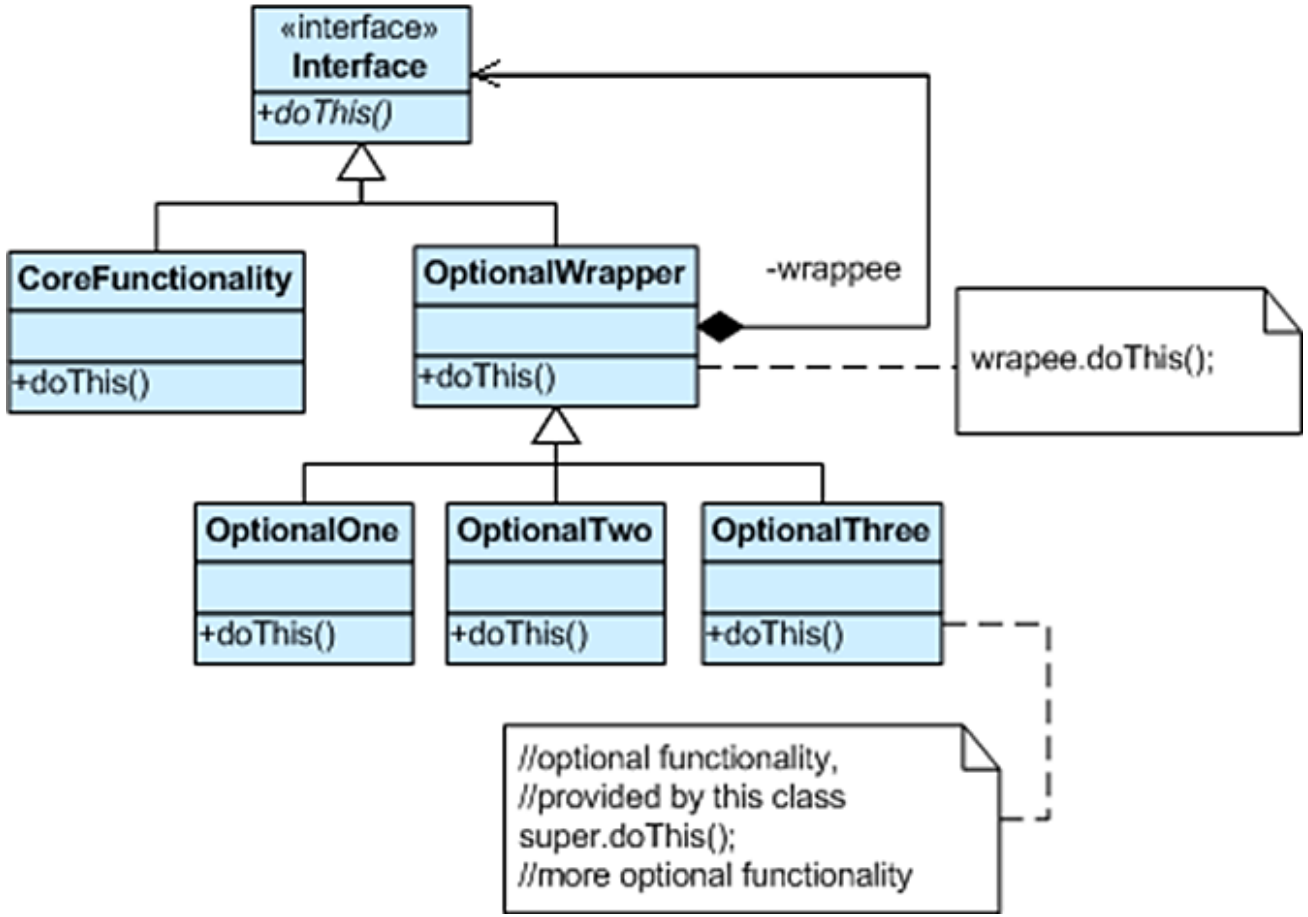
Пример

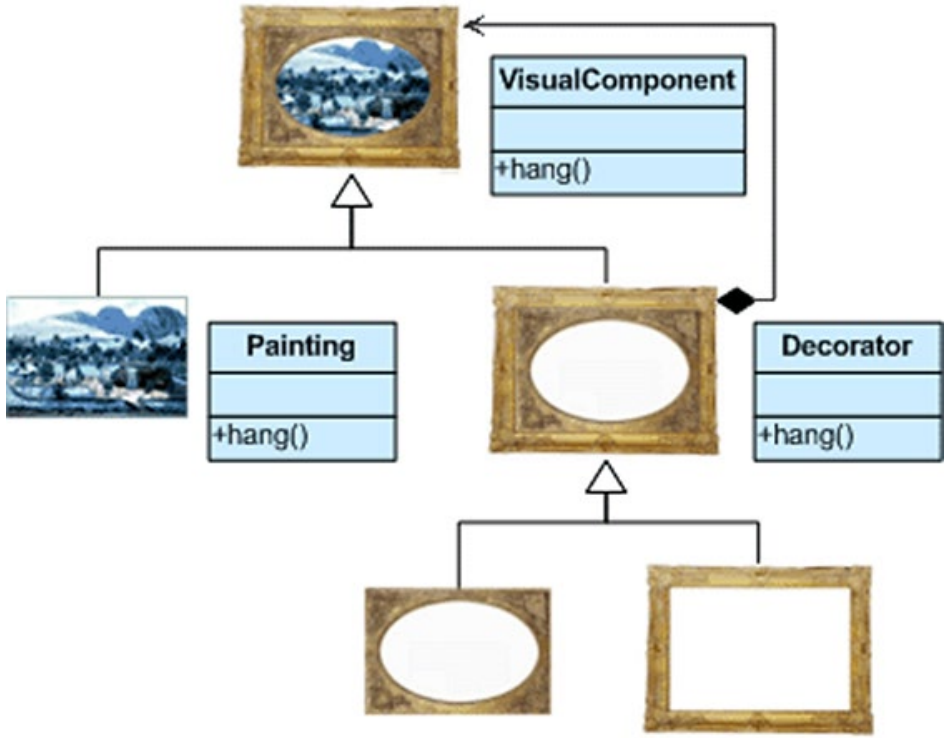
- Каскадное соединение свойств для придания объекту нужных характеристик

```
Stream*  aStream =
new CompressingStream(new ASCII7Stream(new FileStream(
"fileName.dat" )));

AutoBase newmyReno = new SystemSecurity (new MediaNAV(new
Renault));
```

Структура





- Клиент всегда заинтересован в функциональности CoreFunctionality.doThis().
- Клиент может или не может быть заинтересован в методах OptionalOne.doThis() и OptionalTwo.doThis().
- Каждый из этих классов переадресует запрос базовому классу Decorator, а тот направляет его в декорируемый объект

Использование паттерна Decorator

- Выделите один основной компонент и несколько дополнительных (необязательных) "оберток".
- Создайте общий для всех классов интерфейс по принципу "наименьшего общего знаменателя НОЗ" (lowest common denominator LCD). Этот интерфейс должен делать все классы взаимозаменяемыми.
- Создайте базовый класс второго уровня (Decorator) для поддержки дополнительных декорирующих классов.
- Основной класс и класс Decorator наследуют общий НОЗ-интерфейс.
- Класс Decorator использует отношение композиции. Указатель на НОЗ-объект инициализируется в конструкторе.
- Класс Decorator делегирует выполнение операции НОЗ-объекту.
- Для реализации каждой дополнительной функциональности создайте класс, производный от Decorator.
- Подкласс Decorator реализует дополнительную функциональность и делегирует выполнение операции базовому классу Decorator.
- Клиент несет ответственность за конфигурирование системы: устанавливает типы и последовательность использования основного объекта и декораторов.

Proxy (заместитель)

Структурный паттерн проектирования

Проблема

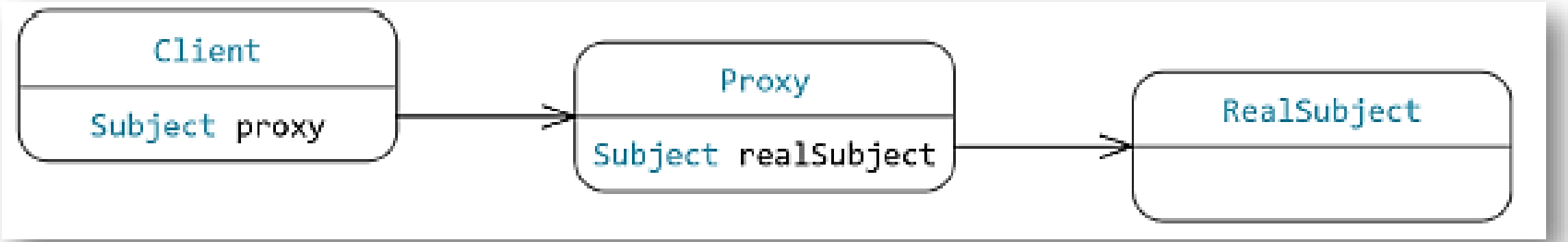
- Требуется управлять ресурсоемкими объектами.
- Нет возможности создавать экземпляры таких объектов до момента их реального использования

Решение.

- Предоставить дополнительный уровень косвенности при доступе к объекту.
- Использовать "обертку" реального компонента, защитив его от излишней сложности

Использование паттерна Proxy

- Заместитель это объект, интерфейс которого идентичен интерфейсу реального объекта.
- При первом запросе клиента заместитель создает реальный объект, сохраняет его адрес и затем отправляет запрос этому реальному объекту.
- Все последующие запросы просто переадресуются инкапсулированному реальному объекту



Использование паттерна Proxy

- Определите ту часть системы, которая лучше всего реализуется через заместителя.
- Определите интерфейс, который сделает заместителя и оригинальный компонент взаимозаменяемыми.
- Класс заместителя содержит указатель на реальный объект и реализует общий интерфейс.
- Указатель на реальный объект может инициализироваться в конструкторе или при первом использовании.
- Методы заместителя выполняют дополнительные действия и вызывают методы реального объекта.

Связь с другими паттернами

- Adapter предоставляет своему объекту другой интерфейс.
- Proxy предоставляет тот же интерфейс.
- Decorator предоставляет расширенный интерфейс.
- Decorator и Proxy имеют разные цели, но схожие структуры.
- Оба вводят дополнительный уровень косвенности: их реализации хранят ссылку на объект, на который они отправляют запросы.

Порождающие паттерны

Порождающие паттерны проектирования предназначены для создания объектов, позволяя системе оставаться независимой как от самого процесса порождения, так и от типов порождаемых объектов

Factory Method (фабричный метод)

- Порождающий шаблон проектирования, предоставляющий подклассам абстрактный интерфейс (набор методов) для создания экземпляров некоторого класса (объекта-продукта)

Проблема

- Программа управления грузовыми перевозками – изначально планировалось перевозить товары только на автомобилях, поэтому весь код работает с объектами класса Грузовик.
- Возникло желание расширить сферу влияния – морские перевозки и требуется добавить поддержку морской логистики в программу, но добавить новый класс не просто, весь код уже завязан на конкретные классы

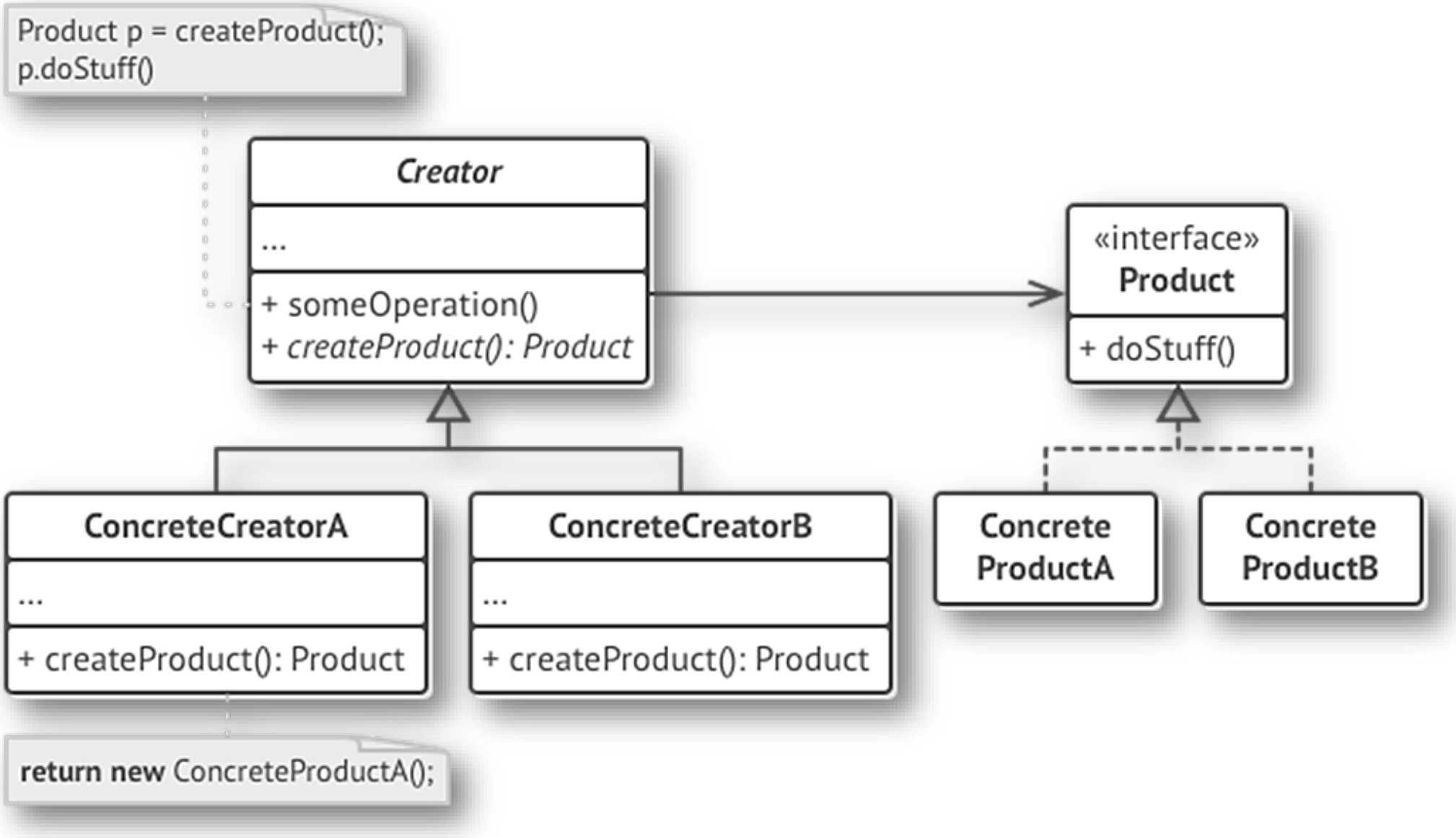
Решение

- Система должна оставаться расширяемой путем добавления объектов новых типов
- При проектировании известно, что нужно будет создавать объект, но тип его неизвестен

Описание паттерна Factory Method

- Паттерн Factory Method использует механизм полиморфизма –
  - классы всех конечных типов наследуют от одного абстрактного базового класса, предназначенного для полиморфного использования
  - в этом базовом классе определяется единый интерфейс, через который пользователь будет оперировать объектами конечных типов.
- Для обеспечения простого добавления в систему новых типов паттерн Factory Method локализует создание объектов конкретных типов в специальном классе-фабрике
  - Методы этого класса, посредством которых создаются объекты конкретных классов, называются фабричными.

Структура паттерна



Abstract Factory (абстрактная фабрика)

- Порождающий шаблон проектирования предоставляет интерфейс для создания семейств взаимосвязанных объектов с определенными интерфейсами без указания конкретных типов данных объектов.
- С помощью такой фабрики удастся создавать группы объектов, реализующих общее поведение

Проблема

- Система должна оставаться независимой как от процесса создания новых объектов, так и от типов порождаемых объектов
- Необходимо создавать группы или семейства взаимосвязанных объектов, исключая возможность одновременного использования объектов из разных семейств в одном контексте

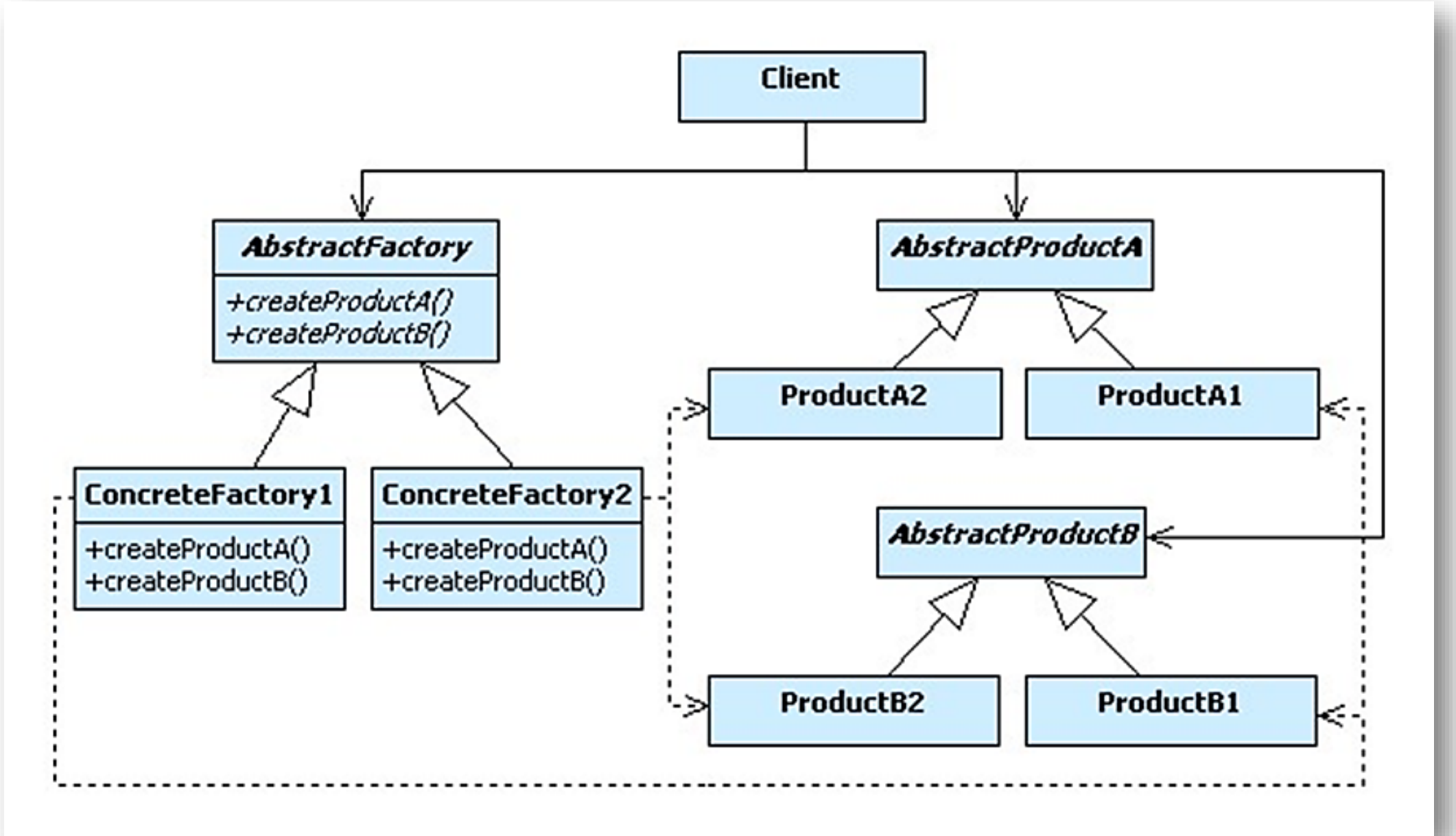
Решение

- Для решения задачи по созданию семейств взаимосвязанных объектов паттерн Abstract Factory вводит понятие абстрактной фабрики

Описание паттерна Abstract Factory

- Абстрактная фабрика представляет собой некоторый полиморфный базовый класс, назначением которого является объявление интерфейсов фабричных методов, служащих для создания продуктов всех основных типов (один фабричный метод на каждый тип продукта).
- Производные от него классы, реализующие эти интерфейсы, предназначены для создания продуктов всех типов внутри семейства или группы

Структура


Client
AbstractFactory
+createProductA()
+createProductB()
AbstractProductA
ProductA2
ProductA1
ConcreteFactory1
+createProductA()
+createProductB()
ConcreteFactory2
+createProductA()
+createProductB()
AbstractProductB
ProductB2
ProductB1

Особенности паттерна

Достоинства паттерна Abstract Factory

- Скрывает сам процесс порождения объектов, а также делает систему независимой от типов создаваемых объектов, специфичных для различных семейств или групп
- Позволяет быстро настраивать систему на нужное семейство создаваемых объектов.
  - Например, в случае многоплатформенного графического приложения для перехода на новую платформу, то есть для замены графических элементов (кнопок, меню, полос прокрутки) одного стиля другим достаточно создать нужный подкласс абстрактной фабрики. При этом условие невозможности одновременного использования элементов разных стилей для некоторой платформы будет выполнено автоматически.

Недостатки паттерна Abstract Factory

- Трудно добавлять новые типы создаваемых продуктов или заменять существующие, так как интерфейс базового класса абстрактной фабрики фиксирован

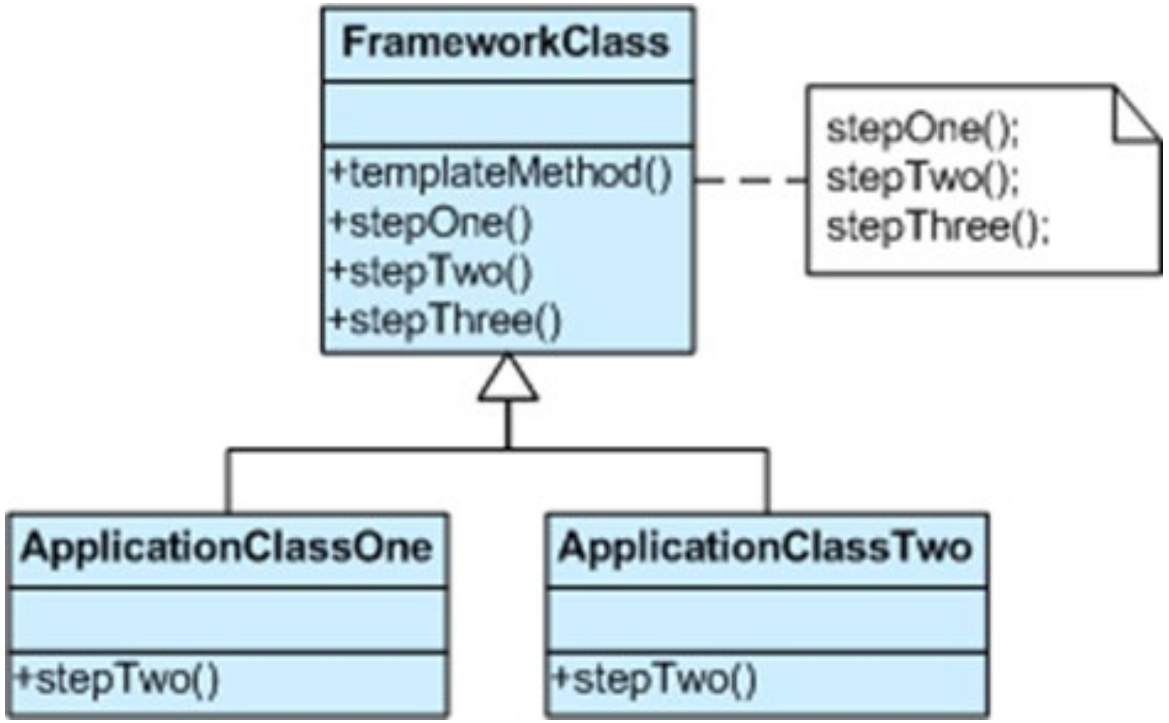
Паттерны поведения

Паттерны поведения описывают правильные способы организации взаимодействия между используемыми объектами

Паттерн Template Method (шаблонный метод)

- Паттерн Template Method определяет основу алгоритма и позволяет подклассам изменить некоторые шаги этого алгоритма без изменения его общей структуры.
- Базовый класс определяет шаги алгоритма с помощью абстрактных операций, а производные классы их реализуют.

Структура паттерна Template Method



- Реализация метода templateMethod() вызывает методы stepOne(), stepTwo() и stepThree().
- Метод stepTwo() является "замещающим" методом. Он объявлен в базовом классе, а определяется в производных классах

Использование паттерна Template Method

- Исследуйте алгоритм и решите, какие шаги являются стандартными, а какие должны определяться подклассами.
- Создайте новый абстрактный базовый класс и поместите в него основу алгоритма (шаблонный метод) и определения стандартных шагов.
- Для каждого шага, требующего различные реализации, определите "замещающий" виртуальный метод. Этот метод может иметь реализацию по умолчанию или быть чисто виртуальным.
- Вызовите "замещающий" метод из шаблонного метода.
- Создайте подклассы от нового абстрактного базового класса и реализуйте в них "замещающие" методы.

Паттерн Strategy (стратегия)

- Паттерн проектирования, который определяет семейство схожих алгоритмов и помещает каждый из них в собственный класс
- Алгоритмы можно взаимозаменять во время исполнения программы

Проблема

- При использовании полиморфизма получаем набор родственных классов с общим интерфейсом и различными реализациями алгоритмов.

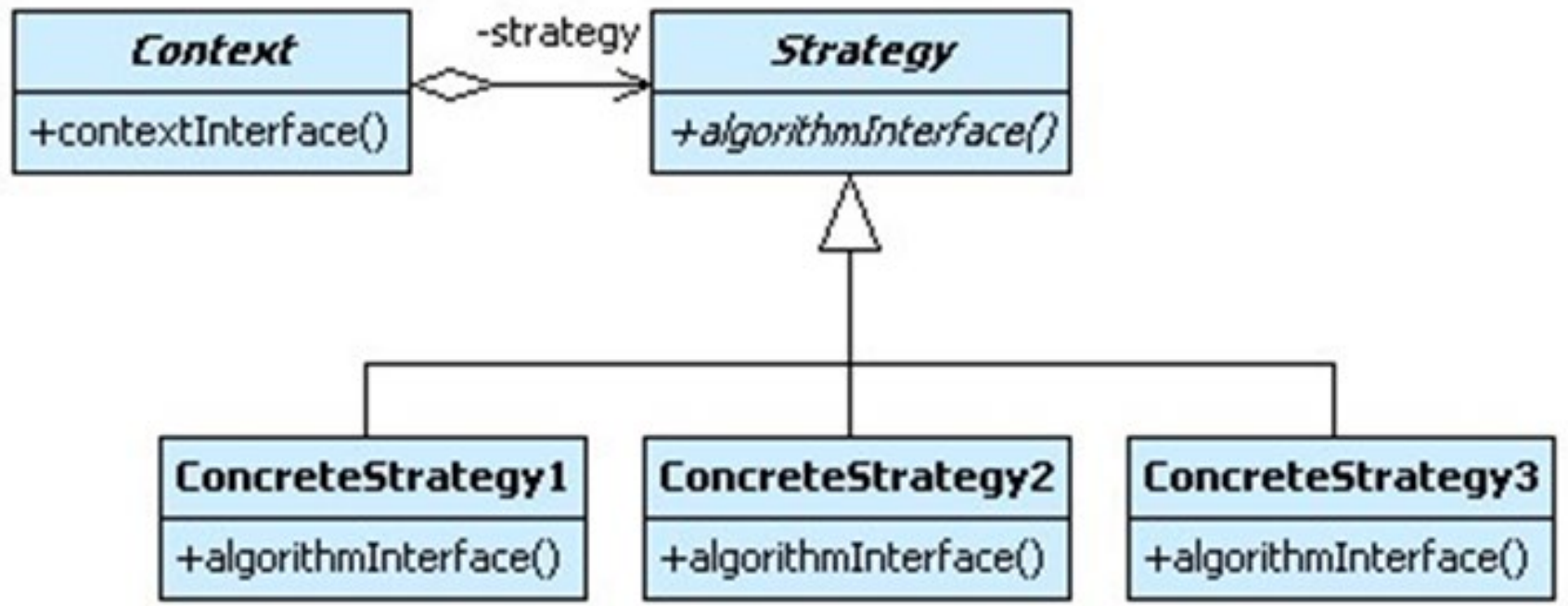
Недостатки:

- Реализация алгоритма жестко привязана к его подклассу, что затрудняет поддержку и расширение такой системы.
- Система, построенная на основе наследования, является статичной. Заменить один алгоритм на другой в ходе выполнения программы уже невозможно.

Решение

- Применение паттерна Strategy

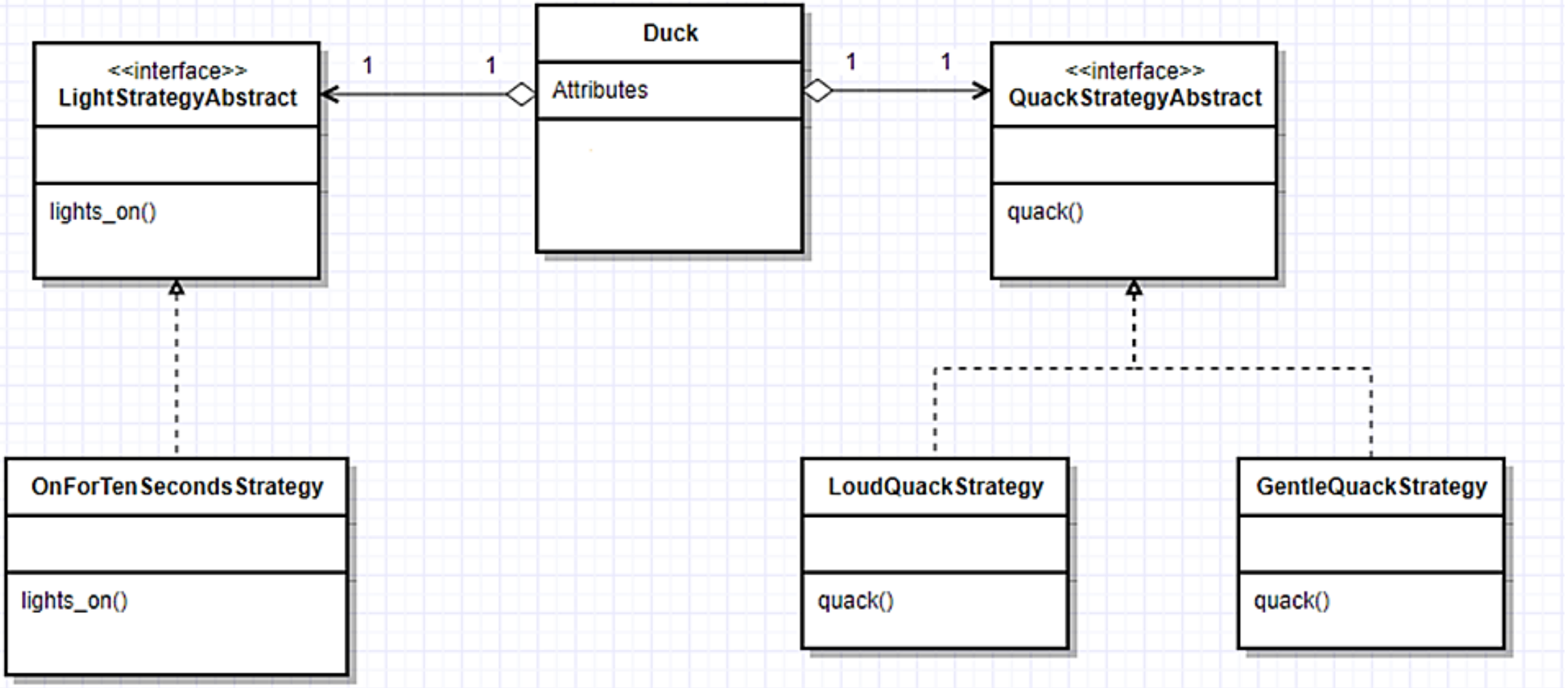
Структура паттерна Strategy



- Паттерн Strategy переносит в отдельную иерархию классов все детали, связанные с реализацией алгоритмов

Пример паттерна Strategy

strategy.py
strategyUse.py



Применение паттерна Strategy

Достоинства паттерна Strategy

- Систему проще поддерживать и модифицировать, так как семейство алгоритмов перенесено в отдельную иерархию классов.
- Паттерн Strategy предоставляет возможность замены одного алгоритма другим в процессе выполнения программы.
- Паттерн Strategy позволяет скрыть детали реализации алгоритмов от клиента.

Недостатки паттерна Strategy

- •Для правильной настройки системы пользователь должен знать об особенностях всех алгоритмов.
- Число классов в системе, построенной с применением паттерна Strategy, возрастает.