ТЕХНОСФЕРА

Лекция № 8

Углубленное программирование на языке C / C++

Алексей Петров

Рекомендуемая литература: модуль №4 (1 / 2)

Гамма Э., Хелм Р., Джонсон Р., Влиссидес Дж. Приемы объектно-ориентированного проектирования. Паттерны проектирования. — Питер, 2007. — 366 с.

Дьюхэрст С. Скользкие места C++. Как избежать проблем при проектировании и компиляции ваших программ. — ДМК Пресс, 2012. — 264 с.

Кериевски Дж. Рефакторинг с использованием шаблонов. — Вильямс, 2006. — 400 с.

Коплиен Дж. Программирование на C++. — Питер, 2005. — 480 с.

Макконнелл С. Совершенный код. Мастер-класс. — Русская редакция, 2012. — 896 с.

Мейерс С. Наиболее эффективное использование C++. 35 новых рекомендаций по улучшению ваших программ и проектов. — ДМК Пресс, 2012. — 298 с.

Мейерс С. Эффективное использование C++. 55 верных советов улучшить структуру и код ваших программ. — ДМК Пресс, 2006. — 300 с.

Рекомендуемая литература: модуль №4 (2 / 2)

Саттер Г. Новые сложные задачи на C++. — Вильямс, 2005. — 272 с.

Саттер Г. Решение сложных задач на C++. — Вильямс, 2008. — 400 с.

Саттер Г., Александреску А. Стандарты программирования на C++. — Вильямс, 2008. — 224 с.

Фаулер М. Рефакторинг. Улучшение существующего кода. — Символ-Плюс, 2008. — 432 с.

Лекция №8. Принципы и шаблоны ОО-проектирования. Базовые шаблоны, шаблоны GoF

1. Ключевые проблемы и принципы ОО-проектирования. Типология шаблонов.
2. Базовые шаблоны: наследование и композиция, интерфейс, делегирование, неизменяемые объекты.
3. Каталог GoF.
4. Основные конфликты. CAP-теорема. Закон Деметра.
5. **Постановка задач к практикуму №6.**

Объектно-ориентированное проектирование: общие сведения

Цель объектно-ориентированного проектирования — **разработка архитектуры** (дизайна) сложных программных систем в соответствии с заданными или общепринятыми критериями качества и с учетом **реализуемости** архитектуры на выбранном языке объектно-ориентированного программирования.

Критерии качества архитектуры, как правило, обеспечивают:

- возможность повторного использования;
- гибкость настройки;
- расширяемость и переносимость;
- структурированность и модульность;
- компактность и разумный уровень детализации;
- понятность и простоту (в том числе взаимодействия компонентов)

Проектирование как искусство компромисса. Проблемы проектирования

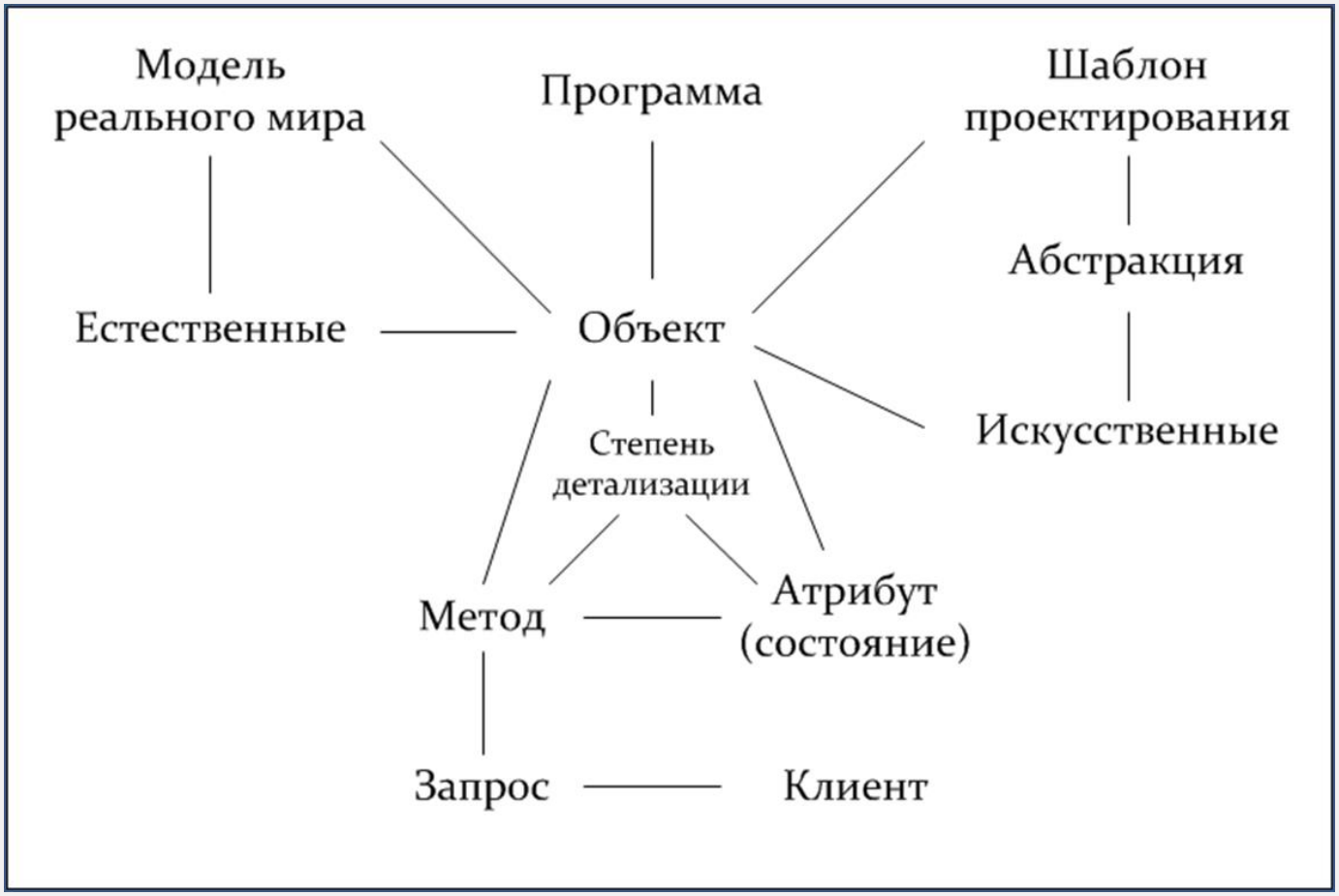
Решение задач объектно-ориентированного проектирования в большинстве случаев предполагает достижения множества компромиссов, например:

- соответствие дизайна задаче ←→ общность дизайна;
- доступность элементов системы ←→ безопасность;
- удобство вызова ←→ возможность тонкой настройки.

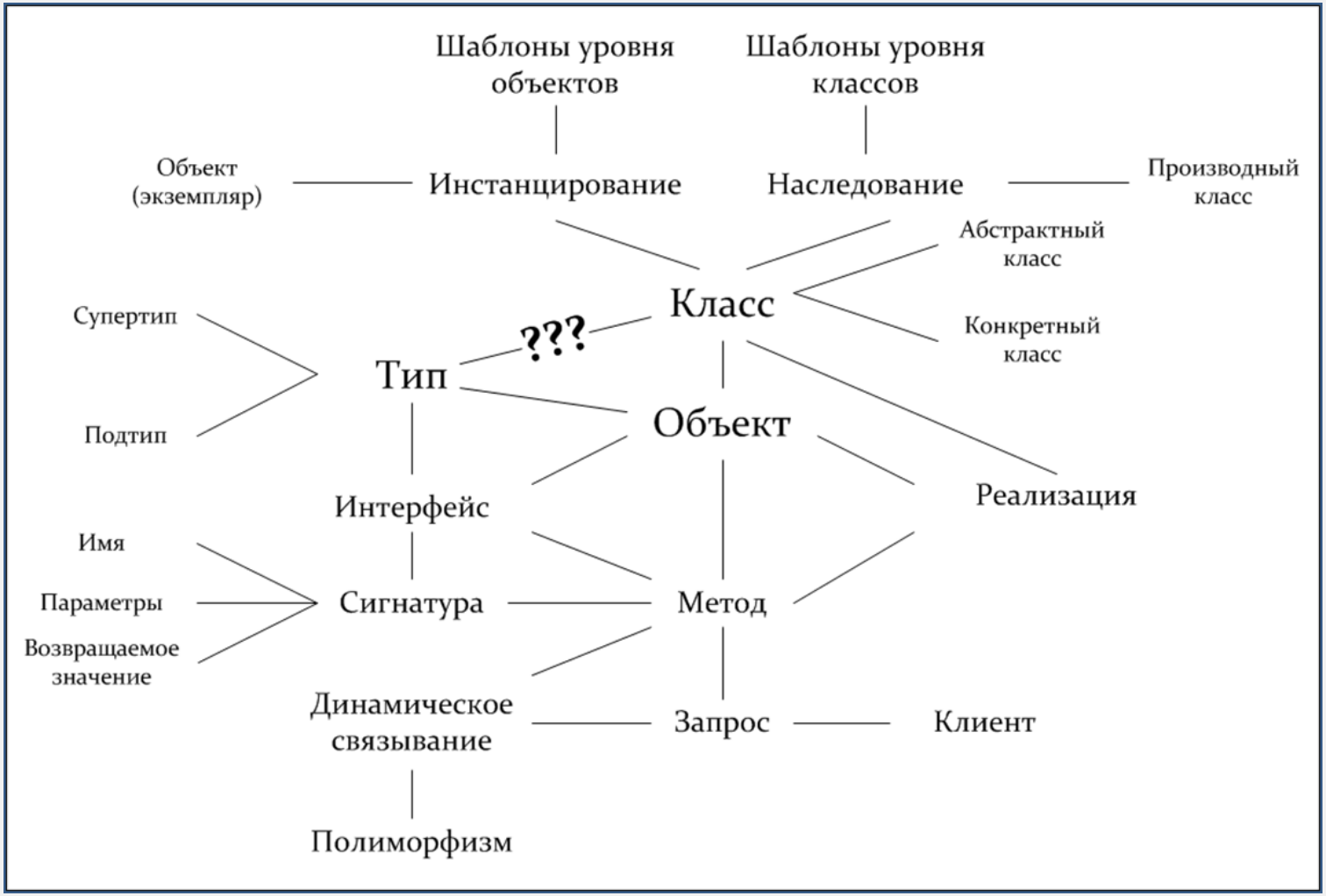
При этом основные проблемы объектно-ориентированного проектирования сводятся, главным образом, к следующим:

- Проблема №1. Идентификация объектов.
- Проблема №2. Определение степени детализации.
- Проблема №3. Определение интерфейса объекта.
- Проблема №4. Определение реализации объекта.

Проблемы №1–2. Определение состава и степени детализации объектов



Проблемы №3–4. Определение интерфейса и реализации объектов



Причины перепроектирования

Явное задание классов при создании объектов:

- привязка к реализации (классу), а не к интерфейсу (типу).

Явное задание способа выполнения операций:

- сужение множества вариантов обслуживания запроса до единственного возможного.

Зависимость от программной и (или) аппаратной **платформы**.

Зависимость клиента **от представления или реализации** объекта:

- потенциальная необходимость модификации клиента при изменении способа представления, хранения или реализации объекта.

Зависимость от алгоритмов.

Сильная связанность классов:

- формирование монолитных систем без слоевой структуры.

Чрезмерное использование наследования.

Максимы проектирования

Не решать каждую задачу «с нуля». Инкапсулировать допускающие изменения элементы дизайна и поведения (напр., алгоритмы, состояния, процессы создания объектов).

Программировать в соответствии с интерфейсом, а не с реализацией (англ. Programming to interfaces):

- клиент не должен обладать информацией о типах используемых объектов, если они имеют ожидаемый интерфейс;
- клиент не должен знать о классах, посредством которых используемые объекты реализованы.

Проектировать системы с учетом будущих изменений.

Предпочитать композицию объектов, а не наследование классов.

Использовать возможности языка программирования.

Шаблоны (паттерны) проектирования

Обобщенные типовые архитектурные решения задач объектно-ориентированного проектирования известны как **шаблоны (паттерны) проектирования**.

Примечание: использованный ранее термин «шаблон класса (функции)» известен в английском языке как **class (function) template**. Для обозначения архитектурных шаблонов применяется англоязычное **design pattern**.

Один из первых авторитетных каталогов шаблонов проектирования составлен Э. Гаммой (Erich Gamma), Р. Хелмом (Richard Helm),
Р. Джонсоном (Ralph Johnson) и Дж. Влиссидесом (John Vlissides), известными как **«банда четырех»** (англ. GoF — Gang of Four) и опубликовавшими книгу *Design Patterns: Elements of Reusable Object-Oriented Software*.

Шаблоны: определение и преимущества. Типология шаблонов

По определению членов GoF, шаблон — это **описание взаимодействия объектов и классов**, адаптированных для решения общей задачи проектирования в конкретном контексте.

Активное **использование** шаблонов проектирования **позволяет**:

- повысить качество кода;
- улучшить техническую документацию;
- обеспечить качество сопровождения ПО.

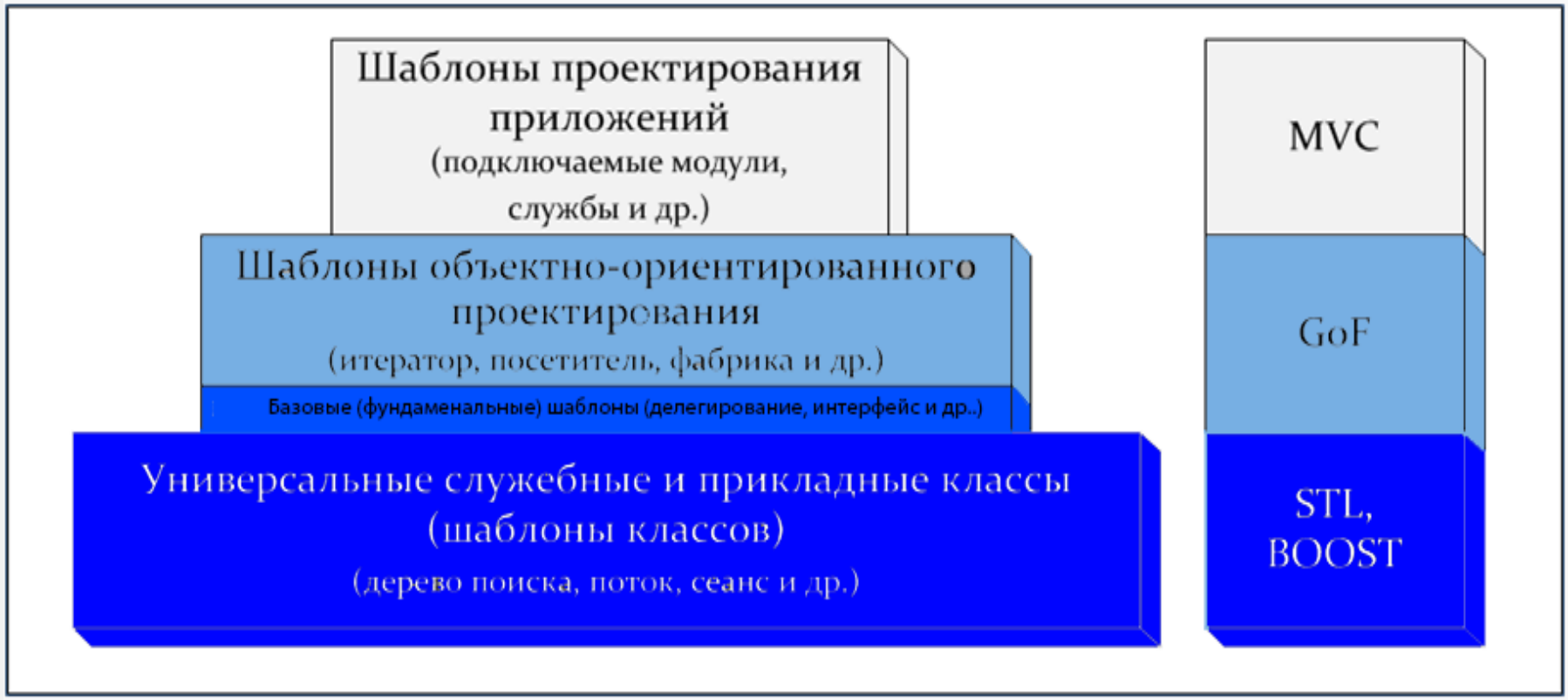
В зависимости от цели использования шаблоны делятся на три категории:

- **порождающие** — описывают процессы создания объектов;
- **структурные** — описывают способы композиции классов (объектов);
- **поведенческие** — описывают взаимодействие классов (объектов) между собой.

Пространство шаблонов GoF

Уровень применения	Порождающие шаблоны (5)	Структурные шаблоны (8)	Поведенческие шаблоны (11)
Класс	Фабричный метод	Адаптер класса	Интерпретатор Шаблонный метод
Объект	Абстрактная фабрика Класс с единственным экземпляром Прототип Строитель	Адаптер объекта Декоратор Заместитель Компоновщик Мост Приспособленец Фасад	Итератор Команда Наблюдатель Посетитель Посредник Состояние Стратегия Хранитель Цепочка ответственности

Стек шаблонов: от потоков до служб



Базовые шаблоны: наследование и композиция

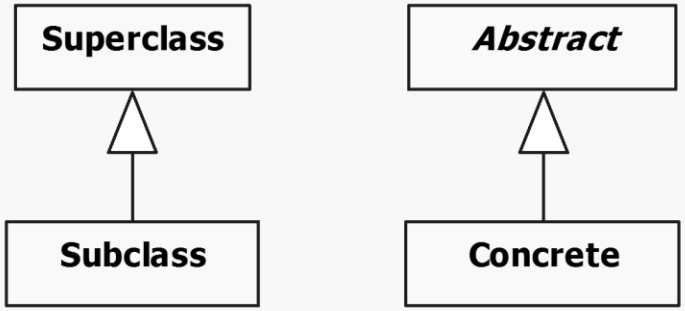
Наследование классов (англ. inheritance) и **композиция объектов** (англ. composition) являются распространенными **приемами повторного использования** функциональных возможностей объектно-ориентированных систем, а потому могут рассматриваться как базовые шаблоны.

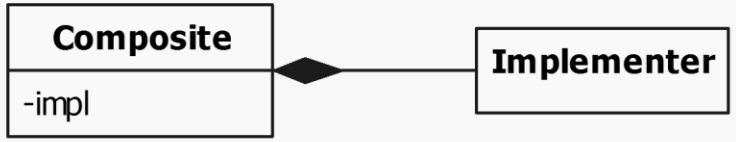
	Наследование	Композиция
Что?	Определяет реализацию одного класса в терминах другого (white-box reuse)	Определяет реализацию одного объекта в терминах другого (black-box reuse)
Как?	Определение подкласса	Определение атрибута (объединение объектов)
Когда?	При компиляции (статически)	При компиляции (статически) или при выполнении (динамически)

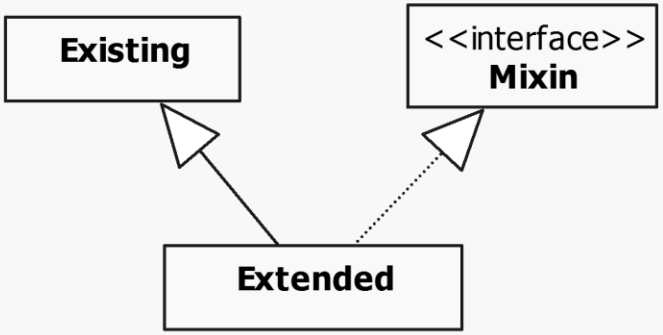
Наследование и композиция: «за» и «против»

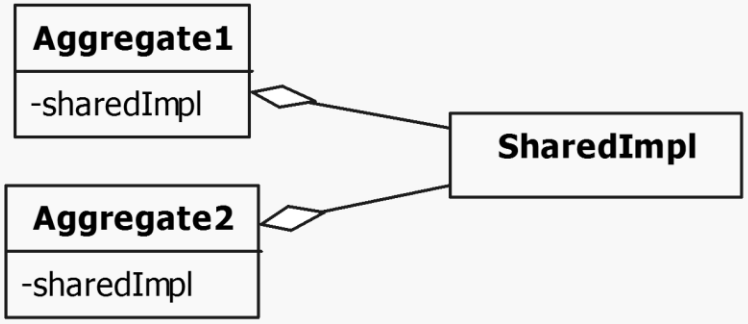
Критерий качества	Наследование	Композиция
Реализация и модификация	**Простая**	Сложная
Зависимость от реализации	Высокая	**Низкая**
Соблюдение инкапсуляции	Нет	**Есть**
Замена реализации при выполнении	Невозможна	**Возможна**
Необходимость тщательного проектирования интерфейса	**Нет**	Есть
Гибкость и возможность повторного применения	Низкая	**Высокая**
Размер класса	Большой	**Небольшой**
Количество классов	Зависит от дизайна	
Количество используемых объектов	**Мало**	Много

Наследование и композиция: реализация









Три способа композиции

Характеристика участника `Implementer`	Инкапсуляция экземпляра	Инкапсуляция ссылки	Инкапсуляция указателя
Обязательность	Обязателен	Обязателен	Необязателен
Количество (кратность)	1 .. *	1 .. *	0 .. *
Зависимость времени жизни (ВЖ) от ВЖ агрегата	Зависит (совпадает)	Не зависит (совпадает или превышает)	Не зависит
Возможность совместного использования	Нет	Есть	Есть
Определение атрибута	`Implementer impl;`	`Implementer &impl;`	`Implementer *impl;`

Агрегирование или осведомленность?

Агрегирование (англ. aggregation) и **осведомленность** (англ. acquaintance) — две стороны композиции. Различия между ними весьма существенны, хотя и определяются предполагаемым использованием объектов, а не механизмами языка.

Характеристика	Агрегирование	Осведомленность
Семантика	«Содержит», «владеет», «несет ответственность»	«Знает», «использует», «ассоциирован с...»
Сила	Сильное	Слабое
Постоянство	Высокое	Низкое
Частота применения	Низкая	Высокая
Время жизни	Одинаково для агрегата и составляющих	Любое
Инкапсулируемый элемент	Экземпляр, ссылка, указатель	Ссылка, указатель

Базовые шаблоны: делегирование

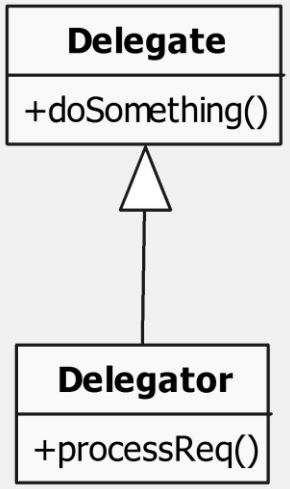
Делегирование (англ. delegation) — передача ответственности за выполнение запроса клиента от непосредственного **получателя** (делегирующей стороны, англ. delegator) **уполномоченному** (делегату, англ. delegate). Различают делегирование **при наследовании** классов и **при композиции** объектов.

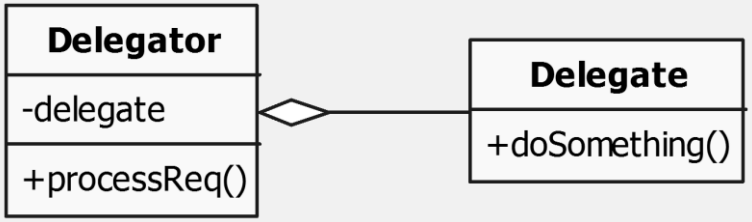
Назначение делегирования — **абстрагирование** (при композиции — также инкапсуляция) **поведения** (реакции на клиентский запрос). Шаблон делегирования используется в целом ряде шаблонов GoF («посетитель», «стратегия» и др.).

Ключевое **достоинство** делегирования — упрощение композиции поведений на стадии выполнения. Ключевой **недостаток** делегирования — трудность статического анализа и понимания сильно параметризованного исходного кода.

Делегирование: реализация

	Наследование	Композиция
Получатель	Производный класс	Объект на стороне «целое»
Уполномоченный	Базовый класс	Объект на стороне «часть»
Доступ к получателю	Указатель `this`	Указатель на получателя (должен быть передан)
Абстрактная семантика	«Является»	«Содержит»





Базовые шаблоны: неизменяемые объекты

Неизменяемый объект (англ. immutable object) — шаблон, позволяющий создать программный объект, **структурно не допускающий модификации** после (окончательного) создания.

Основное предназначение шаблона — **устранение дорогостоящих операций** копирования и сравнения объектов путем использования семантики ссылок.

В дизайне систем различают неизменность объекта:

- с точки зрения самой **системы** и ее **пользователей**;
- на **битовом** (внутримашинном) и **абстрактном** (логическом) уровне.

Достоинства неизменяемого объекта:

- константность, гарантируемая компилятором;
- потоковая безопасность, структурная надежность;
- простота анализа и понимания кода.

Неизменяемые объекты: реализация

Реализация неизменяемого объекта на языке C++ **предполагает**:

- полное построение объекта конструктором класса (кроме отложенной инициализации подмножества атрибутов);
- полное отсутствие неконстантных открытых статических и нестатических атрибутов (без спецификатора const или со спецификатором mutable);
- отсутствие мутаторов, изменяющих состояние всего объекта (для побитовой неизменности) или части, видимой извне пользователю (для логической неизменности).

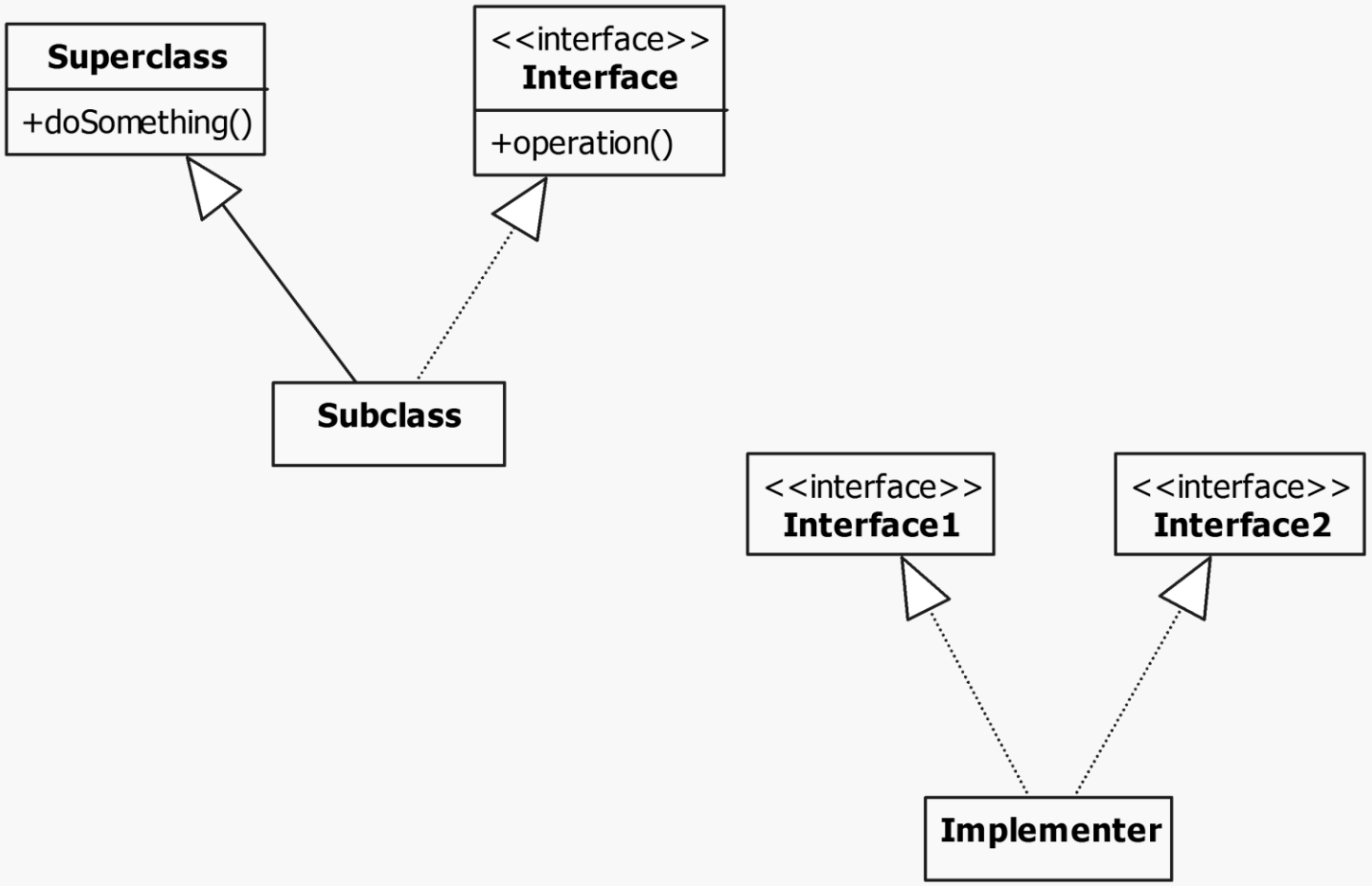
Базовые шаблоны: интерфейс

Интерфейсный класс, или **интерфейс** (англ. interface) — шаблон, структурирующий способы доступа к одному или нескольким (другим) классам.

Классическое назначение интерфейса — **определение** нового **абстрактного типа данных** (ADT) в целях его дальнейшего повторного использования. Такой абстрактный тип обычно не содержит никаких данных, но демонстрирует необходимое поведение.

Достоинство интерфейса состоит в обеспечении возможности статической или динамической замены конкретных классов, реализующих указанный интерфейс.

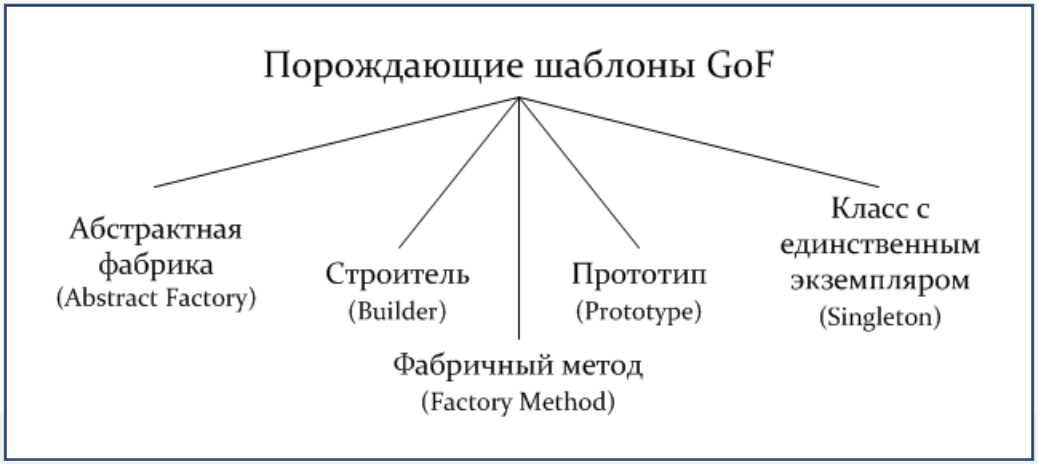
Интерфейс: реализация



Порождающие шаблоны: общие сведения

Порождающие шаблоны:

- абстрагируют процессы создания объектов и инкапсулируют сведения об инстанцируемых классах;
- обеспечивают независимость системы от способа создания, композиции и представления объектов;
- наиболее важны для систем, основанных на композиции больше, чем на наследовании.



Порождающие шаблоны: абстрактная фабрика

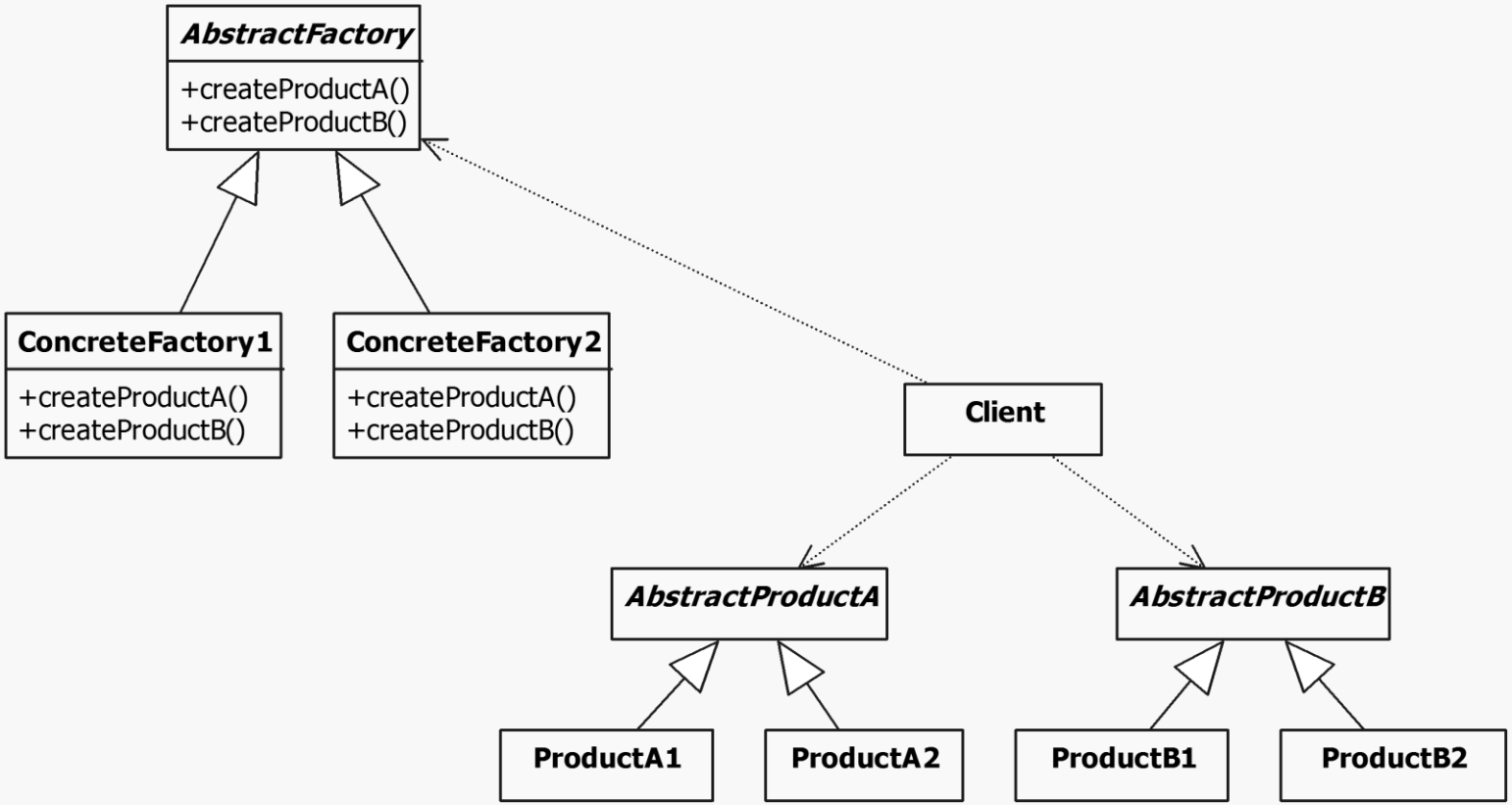
Абстрактная фабрика (инструментарий, англ. kit) — шаблон уровня объекта, предоставляющий интерфейс для создания семейств взаимосвязанных (взаимозависимых) объектов.

Используется, когда:

- система не должна зависеть от способов создания, компоновки и представления объектов;
- объекты разных семейств гарантированно не должны использоваться совместно;
- используемое семейство объектов должно являться параметром конфигурации системы.

Участники: абстрактная фабрика, конкретная фабрика, абстрактный продукт, конкретный продукт, клиент.

Абстрактная фабрика: реализация



Порождающие шаблоны: строитель

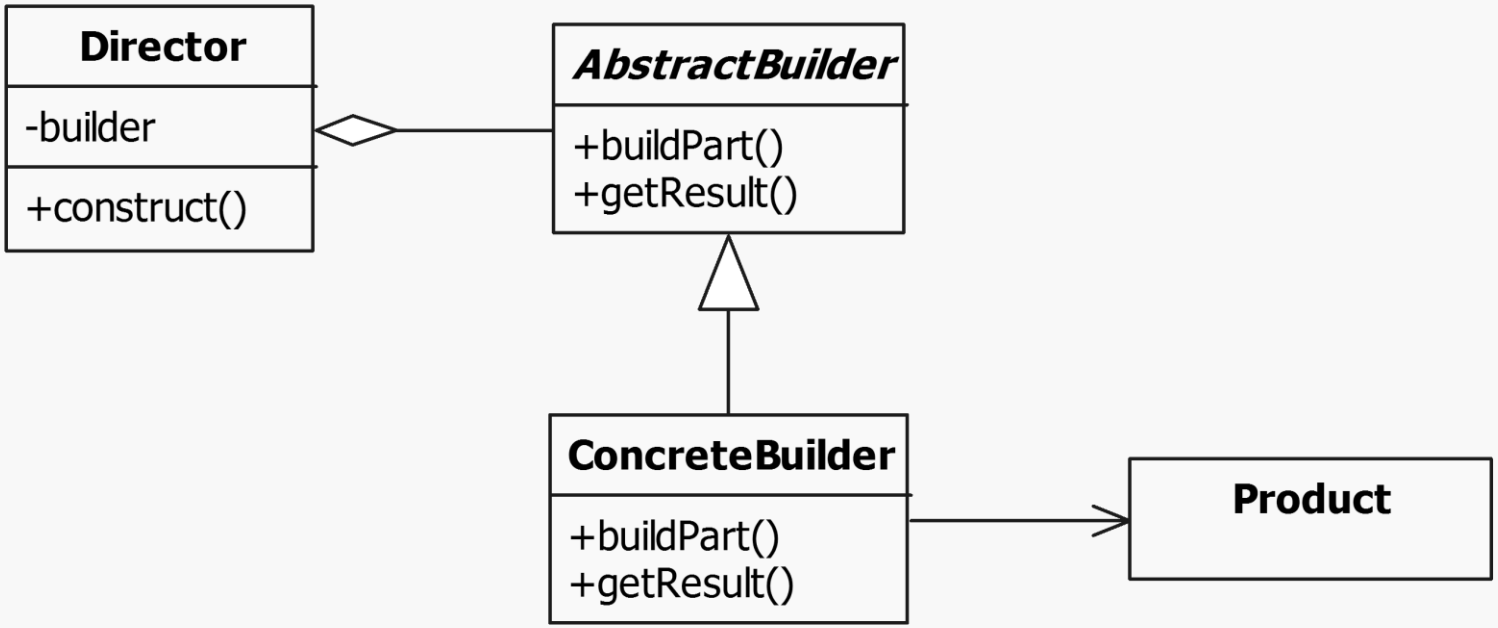
Строитель — шаблон уровня объекта, отделяющий конструирование сложного объекта от его представления.

Используется, когда:

- алгоритм создания сложного объекта не должен зависеть от состава объекта и способов компоновки его частей;
- процесс конструирования объекта должен обеспечивать его различные представления.

Участники: абстрактный строитель, конкретный строитель, распорядитель, продукт, клиент.

Строитель: реализация



Порождающие шаблоны: фабричный метод

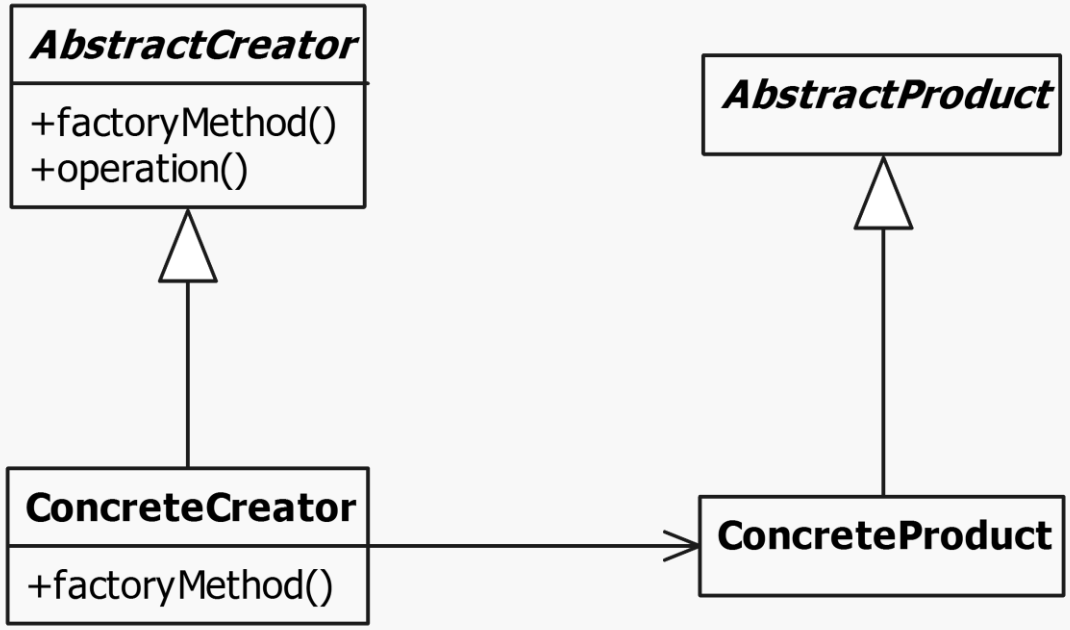
Фабричный метод (виртуальный конструктор, англ. virtual constructor) — шаблон уровня класса, определяющий абстрактный интерфейс создания объектов и делегирующий выбор типов инстанцируемых объектов конкретным подклассам.

Используется, когда:

- заранее неизвестен тип инстанцируемых объектов;
- дизайн класса с фабричным методом предполагает, что создаваемые объекты специфицируются подклассами;
- класс делегирует свои обязанности по инстанцированию объектов одному из своих подклассов.

Участники: абстрактный создатель, конкретный создатель, абстрактный продукт, конкретный продукт.

Фабричный метод: реализация



Порождающие шаблоны: прототип

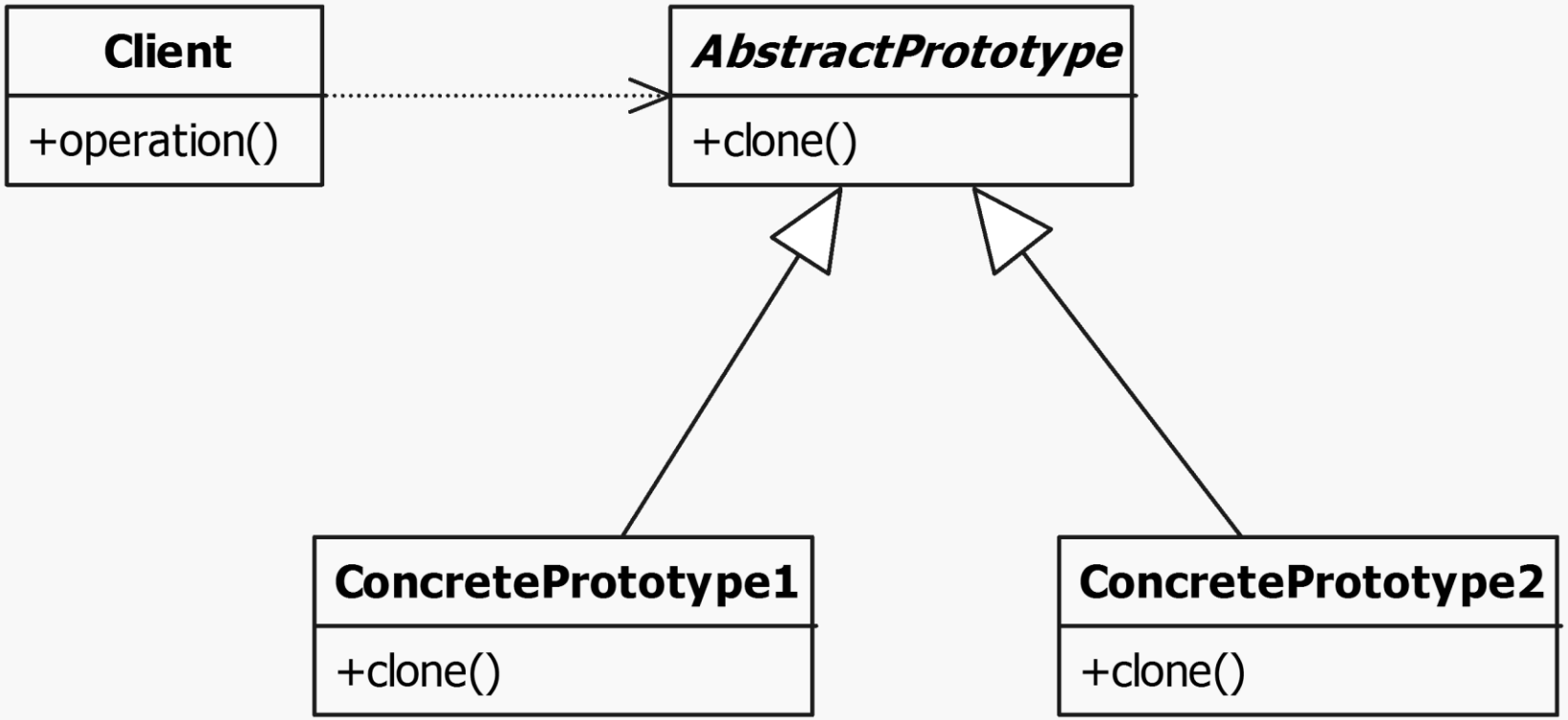
Прототип — шаблон уровня объекта, определяющий вид создаваемого объекта при помощи экземпляра-прототипа, который явно копируется при создании объекта.

Используется, когда:

- инстанцируемые классы определяются динамически;
- необходимо избежать возникновения иерархий классов или фабрик, параллельных иерархиям классов продуктов;
- количество прототипов невелико.

Участники: абстрактный прототип, конкретный прототип, клиент.

Прототип: реализация



Порождающие шаблоны: класс с единственным экземпляром

Класс с единственным экземпляром (одиночка, жарг. синглтон, англ. singleton) — шаблон уровня объекта, гарантирующий, что класс располагает единственным экземпляром, и предоставляющий к нему глобальную точку доступа.

Используется, когда:

- экземпляр класса должен быть один и только один;
- экземпляр класса должен быть доступен для любого клиента;
- класс должен допускать порождение подклассов с возможностью работать с ними без модификации клиентской части системы.

Участник: одиночка.

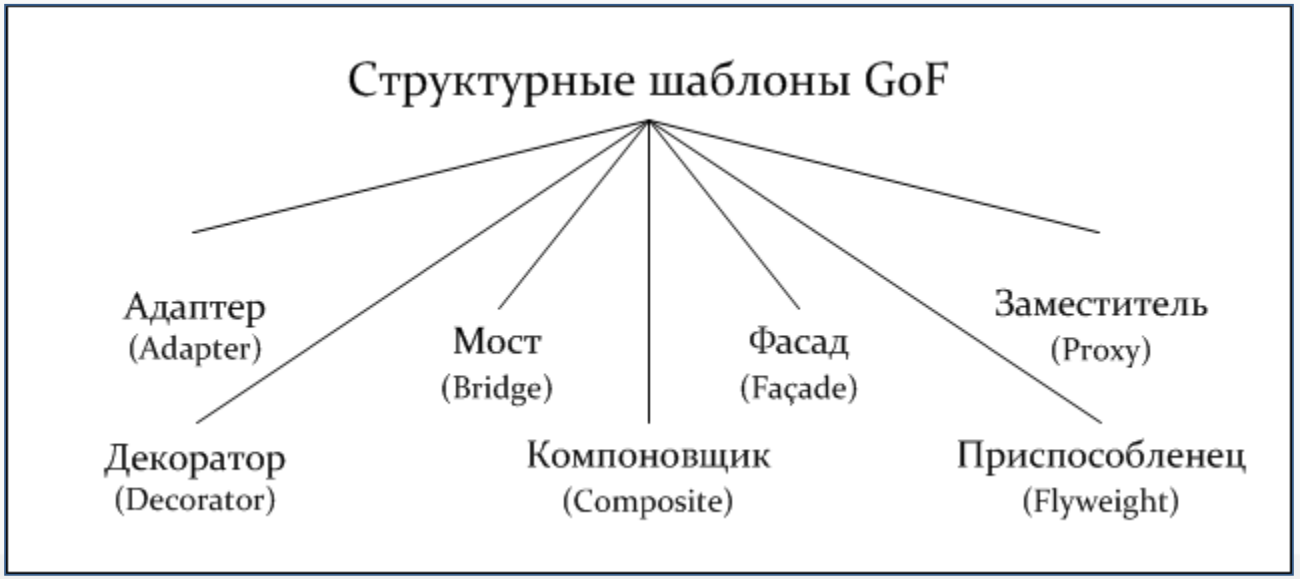
Класс с единственным экземпляром: минимальная реализация

```
class Singleton {
public:         static Singleton* Instance();
protected:      Singleton();
private:        static Singleton* _instance;
};

Singleton* Singleton::_instance = NULL;

Singleton* Singleton::Instance() {
    if(_instance == NULL)
        _instance = new Singleton;
    return _instance;
}
```

Структурные шаблоны: общие сведения

Структурные шаблоны:

- определяют порядок композиции (взаимосвязи) объектов, интерфейсов и реализаций;
- абстрагируют процессы построения программных структур, более крупных, чем отдельные классы и экземпляры.



Структурные шаблоны: адаптер

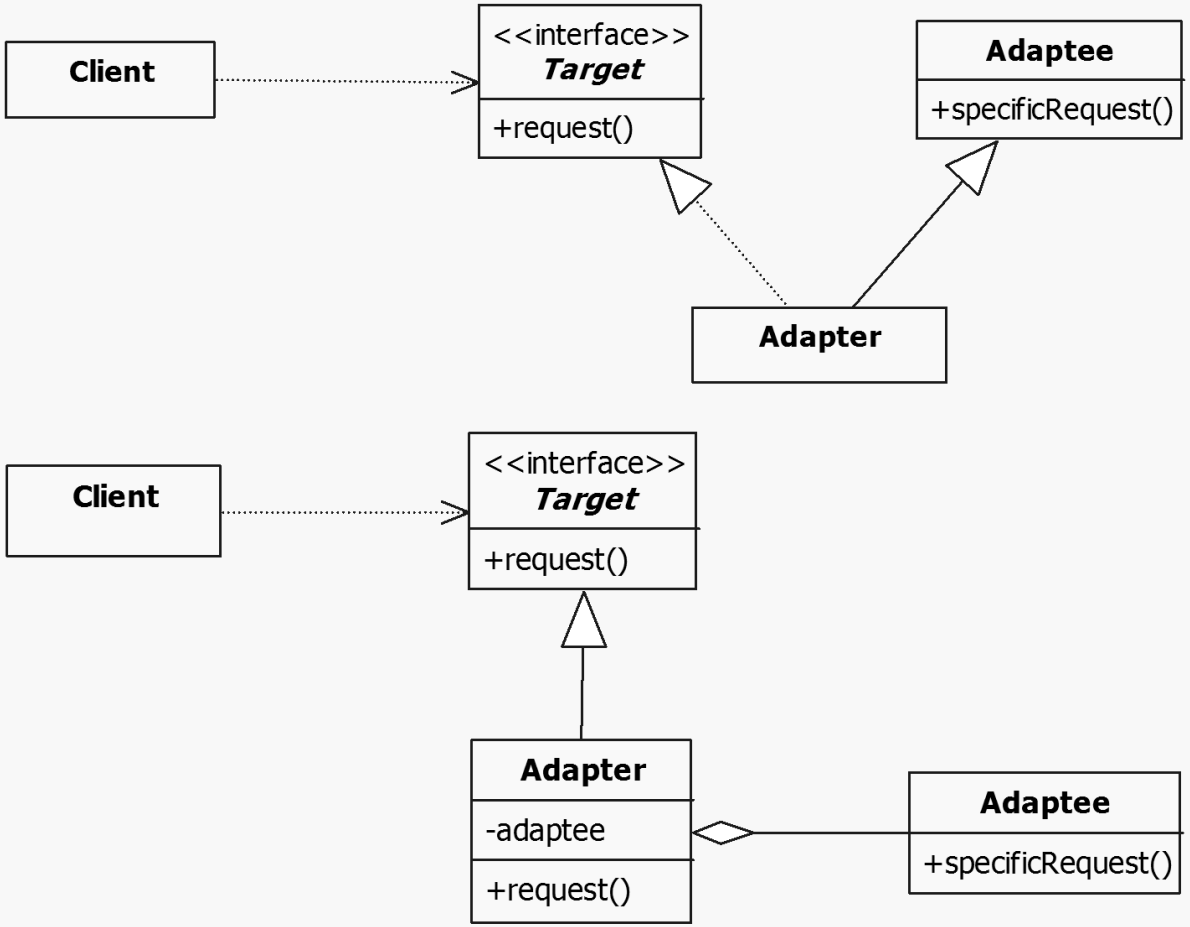
Адаптер (обертка, англ. wrapper) — шаблон уровня класса или объекта, преобразующий интерфейс одного класса в интерфейс другого и обеспечивающий совместную работу классов с несовместимыми интерфейсами.

Используется, когда:

- необходимо использовать класс, интерфейс которого не отвечает предъявляемым требованиям;
- повторно используемый класс должен взаимодействовать с заранее неизвестными или не связанными с ним классами с несовместимыми интерфейсами;
- необходимо использовать подклассы, (а) от которых для адаптации нежелательно порождать новые подклассы, но (б) их общий предок может быть адаптирован.

Участники: адаптер, целевой и адаптируемый классы, клиент.

Адаптер класса и адаптер объекта: реализация



Структурные шаблоны: мост

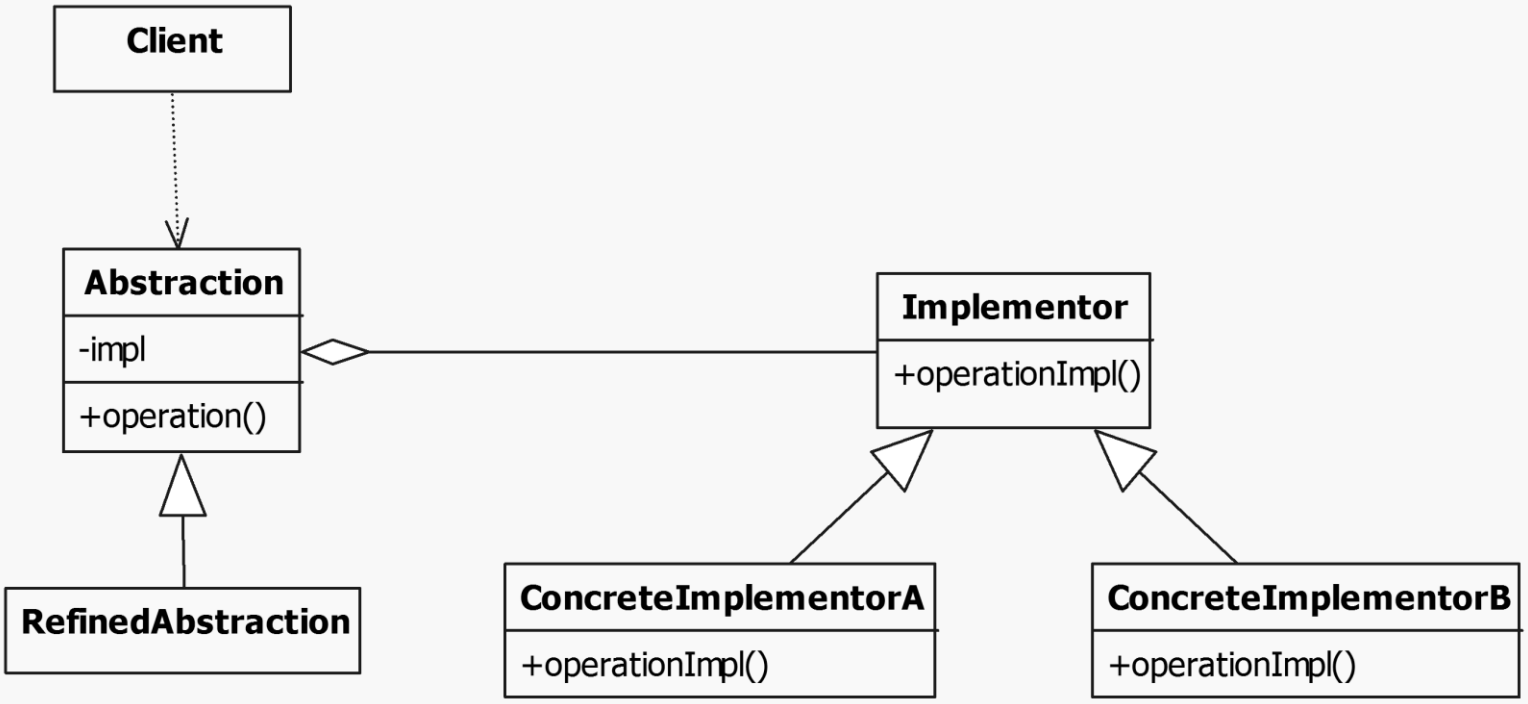
Мост (описатель/тело, англ. handle/body) — шаблон уровня объекта, отделяющий абстракцию от ее реализации с тем, чтобы каждая из них могла независимо изменяться, расширяться и использоваться повторно.

Используется, когда:

- необходимо избежать неразрывной связи абстракции с реализацией;
- абстракция и реализация должны расширяться путем наследования;
- изменения в реализации не должны сказываться на клиентах абстракции;
- необходимо скрыть реализацию абстракции от клиентов;
- реализация требует обобществления между несколькими объектами, и этот факт должен быть скрыт.

Участники: абстракция, уточненная абстракция, реализация, конкретная реализация.

Мост: реализация



Структурные шаблоны: компоновщик

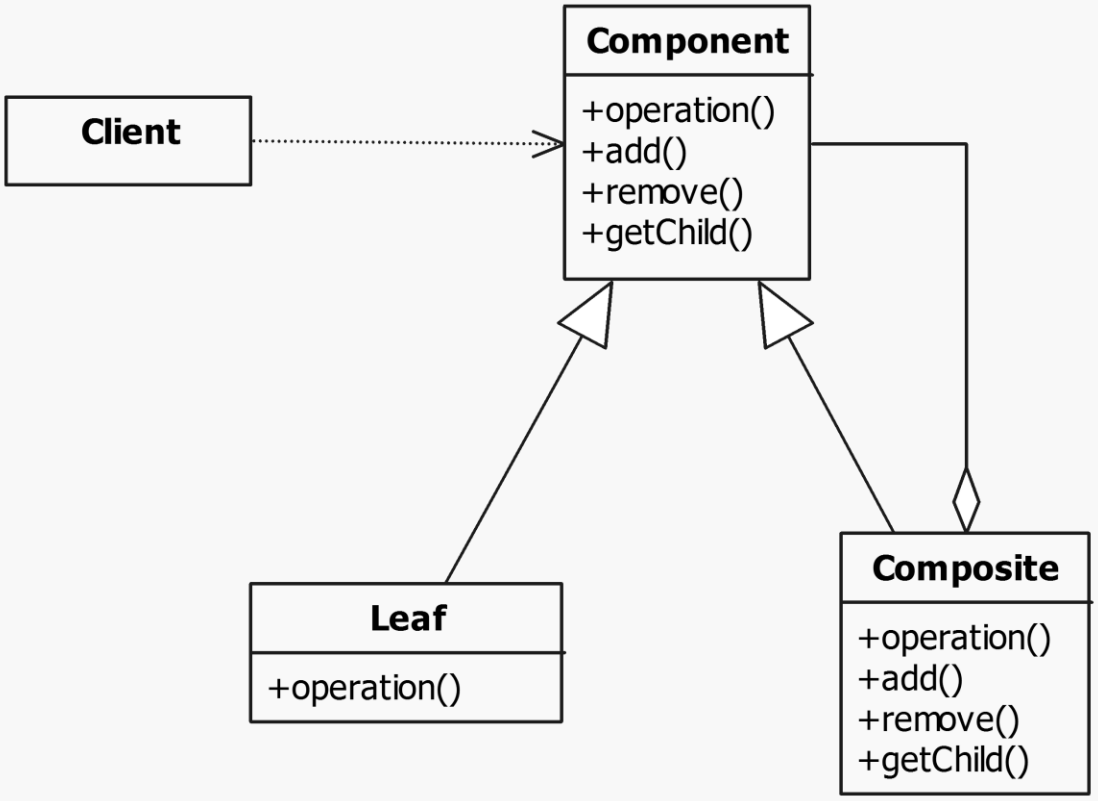
Компоновщик — шаблон уровня объектов, объединяющий их в древовидные структуры вида «часть – целое» и предоставляющий пользователям иерархий унифицированный интерфейс для работы с примитивными и составными объектами.

Используется, когда:

- необходимо сформировать иерархию объектов вида «часть – целое»;
- необходимо обеспечить одинаковую трактовку примитивных и составных объектов.

Участники: компонент, составной объект, лист, клиент.

Компоновщик: реализация



Структурные шаблоны: декоратор

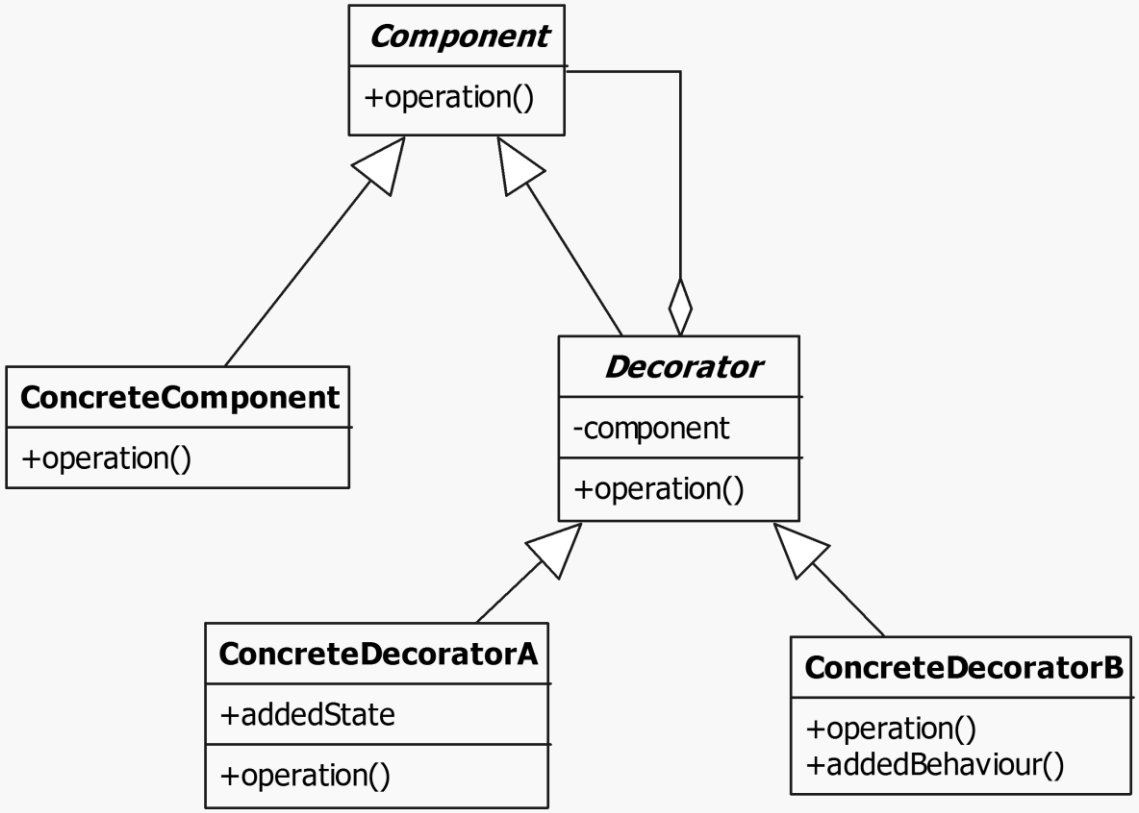
Декоратор (обертка, англ. wrapper) — шаблон уровня объекта, динамически назначающий объекту новый функционал.

Используется, когда:

- дополнительный функционал должен динамически и прозрачным для клиента образом назначаться и сниматься с объекта;
- невозможно или неудобно расширять функциональные возможности объекта путем порождения подклассов — например, по причине комбинаторного роста количества независимых расширений.

Участники: абстрактный и конкретный компонент, абстрактный и конкретный декоратор.

Декоратор: реализация



Структурные шаблоны: фасад

Фасад — шаблон уровня объекта, предоставляющий единый, упрощенный, высокоуровневый интерфейс к подсистеме и действующий наряду с интерфейсами отдельных ее объектов.

Используется, когда:

- требуется простой интерфейс к совокупности сложных, взаимодействующих и взаимозависимых объектов (подсистеме);
- стоит задача изолировать подсистему от клиента и других подсистем, повысить степень переносимости, независимости, избавиться от лишних связей между объектами;
- необходимо декомпозировать систему (подсистему) на слои (уровни).

Участники: фасад, классы подсистемы.

Структурные шаблоны: заместитель (1 / 2)

Заместитель (суррогат, англ. surrogate) — шаблон уровня объекта, реализующий суррогатный объект и контролирующий доступ к представляемому объекту.

Частные случаи:

- удаленный заместитель — локально представляет объект из другого адресного пространства;
- виртуальный заместитель — создает «тяжеловесные» объекты по требованию;
- защищающий заместитель — контролирует доступ к представляемому объекту;
- «умная» ссылка («умный» указатель) — дополняет традиционную семантику ссылок (указателей) техниками отложенной инициализации, подсчета ссылок, управления блокировками и др.

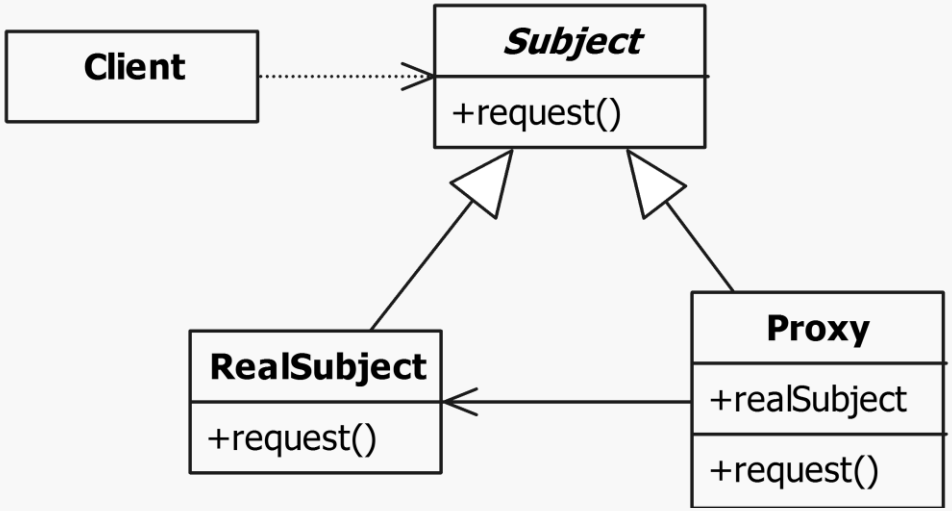
Структурные шаблоны: заместитель (2 / 2)

Используется, когда:

- необходимо ссылаться на объект более тонко, чем при помощи «обычного» указателя.

Участники: заместитель, абстрактный и реальный субъект.

Заместитель: реализация



Поведенческие шаблоны: общие сведения

Поведенческие шаблоны:

- систематизируют распределение обязанностей между объектами;
- характеризуют поток управления программы;
- реализуют типичные способы взаимодействия объектов.

Поведенческие шаблоны: команда

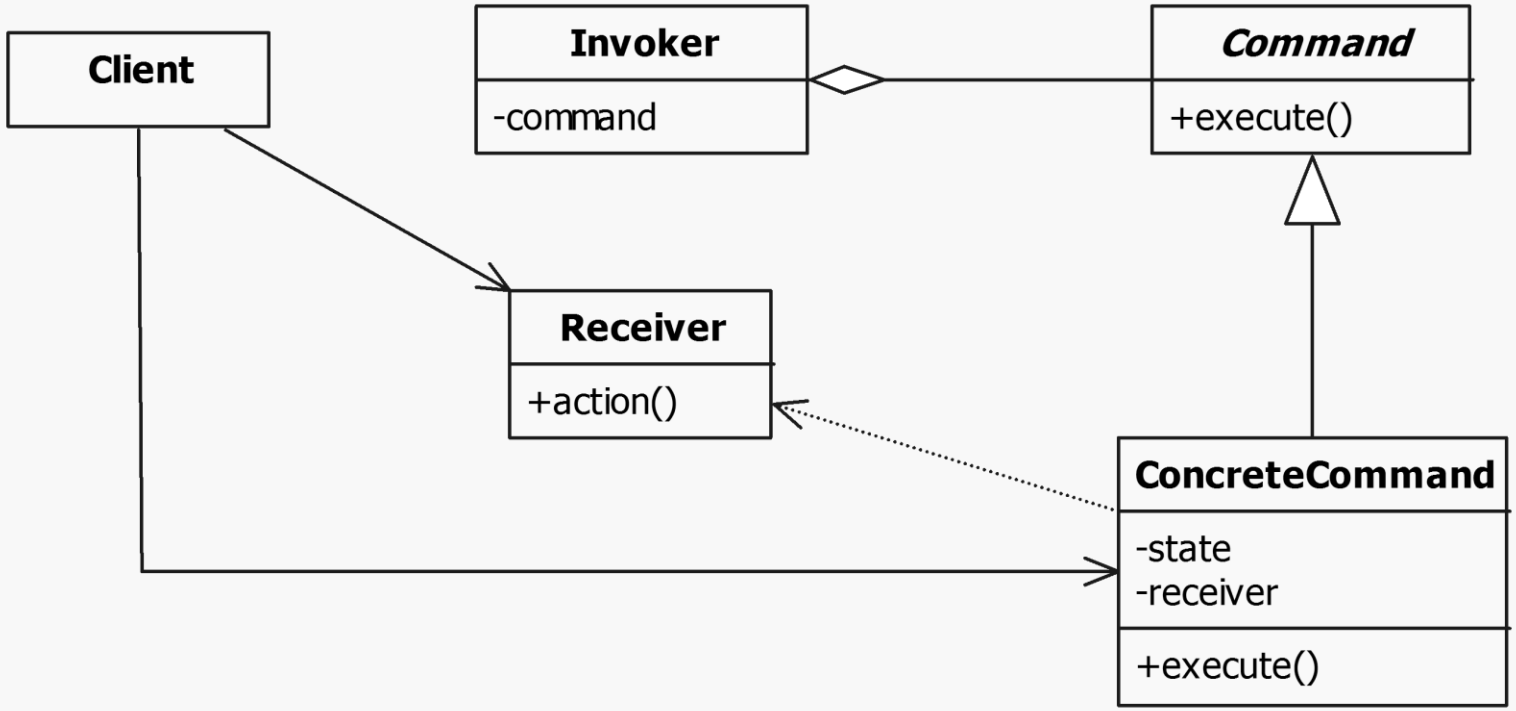
Команда (действие, англ. action; транзакция, англ. transaction) — шаблон уровня объекта, инкапсулирующий запрос как объект.

Используется, когда:

- необходимо делать запросы параметрами объектов — аналогично функциям обратного вызова;
- необходимо отправлять запросы неизвестным объектам-получателям;
- необходимо ставить запросы в очередь, хранить и протоколировать их;
- необходимо отменять результаты запросов и повторно их исполнять.

Участники: абстрактная и конкретная команда, инициатор, получатель, клиент.

Команда: реализация



Команда «без памяти»: шаблон класса

```
template <class Receiver>
class SimpleCommand : public Command {
public:
    typedef void (Receiver::* Action)();
    SimpleCommand(Receiver r, Action a) : _receiver(r),
                                          _action(a) { }
    virtual void Execute();

private:
    Action _action;
    Receiver* _receiver;
};
```

Команда «без памяти»: реализация и использование

```
template <class Receiver>
void SimpleCommand<Receiver>::Execute() {
    (_receiver->*action)();
}

// в точке использования
SomeClass* receiver = new SomeClass();

// ...
Command* command = new SimpleCommand<SomeClass>
                        (receiver, &SomeClass::Action);
// ...
command->Execute();
```

Поведенческие шаблоны: итератор

Итератор (курсор, англ. cursor) — шаблон уровня объекта, предоставляющий механизм последовательного доступа к элементам составного объекта без нарушения инкапсуляции его программного представления.

Используется, когда:

- необходимо обеспечение доступа к содержимому агрегата без раскрытия его представления;
- требуется поддерживать несколько активных точек и (или) способов обхода одного агрегата;
- должен существовать (быть создан) унифицированный интерфейс обхода различных агрегированных структур — полиморфная итерация.

Участники: абстрактный и конкретный агрегат, абстрактный и конкретный итератор.

Итератор: частные случаи (1 / 2)

Частные случаи:

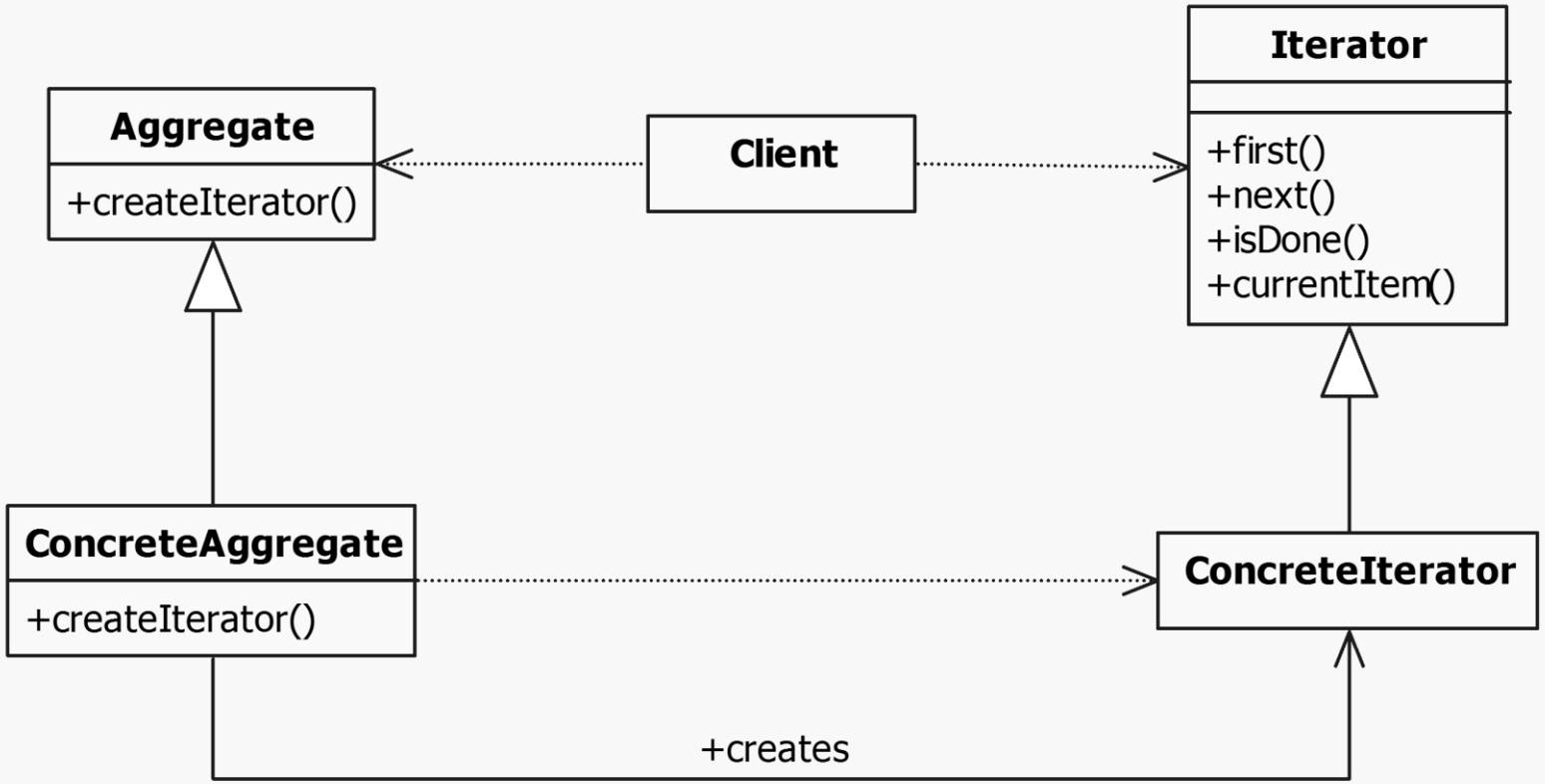
- внешний итератор — обходом агрегата управляет «активный» клиент;
- внутренний итератор — обход агрегата осуществляется итератором (при «пассивном» клиенте);
- курсор — использует алгоритм обхода, определяемый агрегатом, и лишь указывает на текущую позицию в нем;
- устойчивый итератор — гарантирует, что операции вставки и удаления не препятствуют обходу по агрегату без копирования последнего;
- итератор с расширенным интерфейсом — реализует операции, дополнительные к first(), next(), isDone() и currentItem();
- полиморфный итератор — как правило, создается фабричным методом и удаляется клиентом;
- пустой итератор — обслуживает обработку граничных условий, является вырожденным (isDone() возвращает true).

Итератор: частные случаи (2 / 2)

Внешний или внутренний итератор?

	Внешний итератор	Внутренний итератор
Сторона управления	Клиент	Итератор
Место определения алгоритма	Клиент	Итератор Агрегат (для курсоров)
Роль клиента	Активный	Пассивный
Функция клиента	Запрашивает следующий элемент	Передает операцию для применения к каждому посещенному элементу
Гибкость обхода	Высокая	Низкая
Простота обхода	Средняя	Высокая
Обход рекурсивных агрегатов	Затруднен	Не вызывает проблем (используется стек вызовов)

Итератор: реализация



Поведенческие шаблоны: посредник

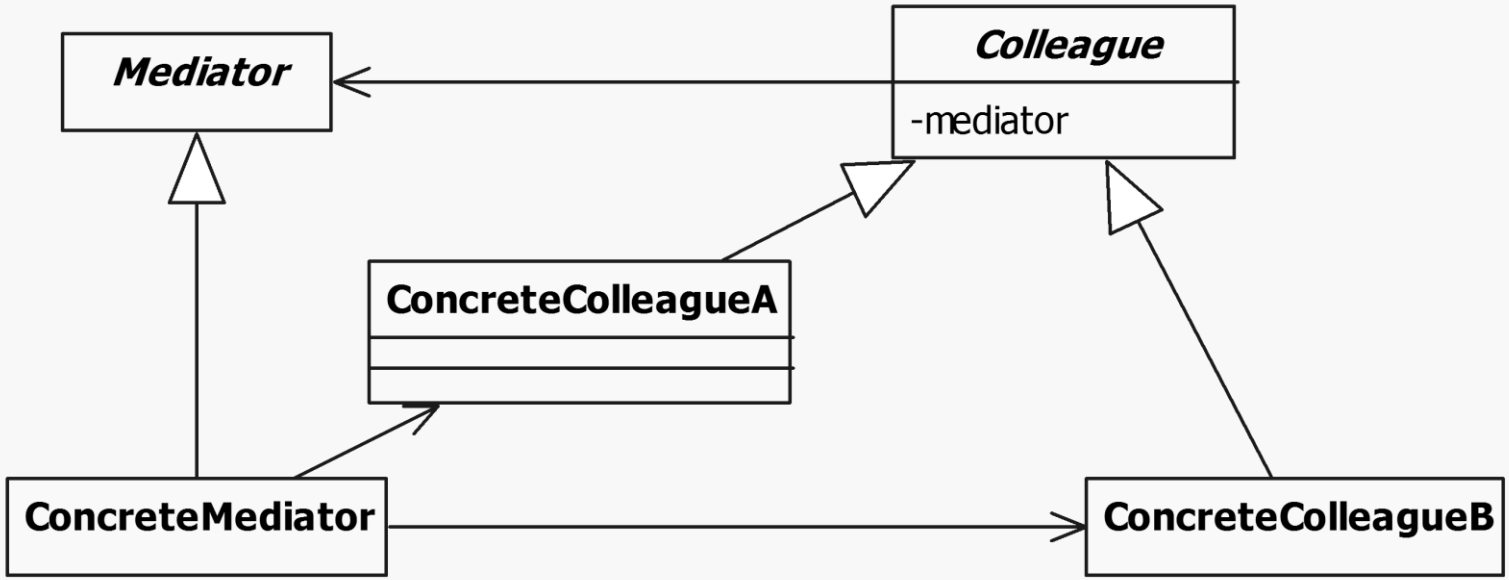
Посредник — шаблон уровня объекта, абстрагирующий способ кооперирования большого числа объектов.

Используется, когда:

- необходимо обеспечить слабую связанность системы;
- избавить от явных множественных перекрестных ссылок друг на друга объекты со сложными, неструктурированными и трудными для понимания связями;
- затруднено повторное использование объекта, активно участвующего в информационном обмене с другими его участниками;
- поведение, распределенное между классами, должно настраиваться без создания большого числа подклассов.

Участники: абстрактный и конкретный посредник, объект–коллега.

Посредник: реализация



Поведенческие шаблоны: наблюдатель

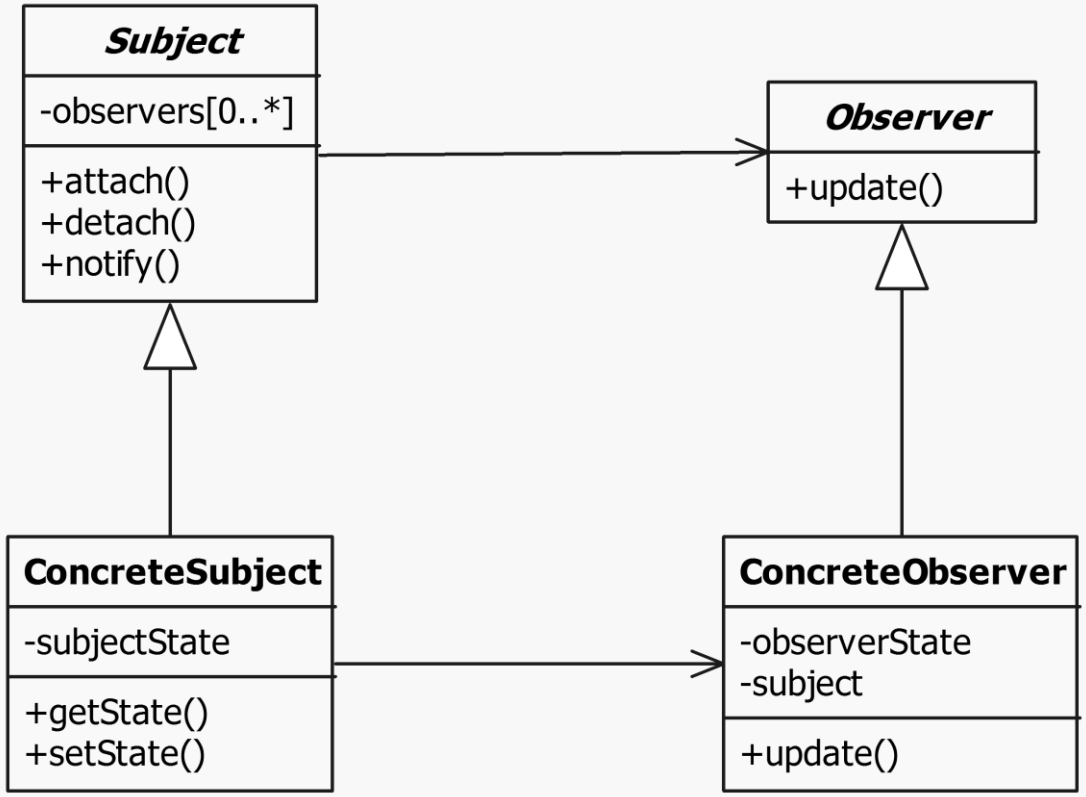
Наблюдатель (подчиненные, англ. dependents; издатель/подписчик, англ. publish/subscribe) — шаблон уровня объекта, определяющий связь «один ко многим» для обновления всех объектов, зависимых от изменившегося объекта.

Используется, когда:

- существуют (выявлены) два аспекта абстракции, один из которых является зависимым от другого — например, «данные — представление»;
- предполагается, что изменение состояния одного объекта повлечет за собой изменение других объектов, которые заранее неизвестны;
- необходимо обеспечить возможность извещения одним объектом других, предположения о которых отсутствуют у инициатора извещений.

Участники: абстрактный и конкретный субъект, абстрактный и конкретный наблюдатель.

Наблюдатель: реализация



Вытягивание или проталкивание?

	Модель проталкивания (push)	Модель вытягивания (pull)
Активная сторона	Субъект	Наблюдатель
Содержимое посылки	Детальная информация об изменениях	Минимальное уведомление
Информированность субъекта о наблюдателях	Высокая	Низкая (нулевая)
Степень повторного использования субъектов	Низкая	Высокая
Общая эффективность модели	Выше	Ниже

Поведенческие шаблоны: состояние

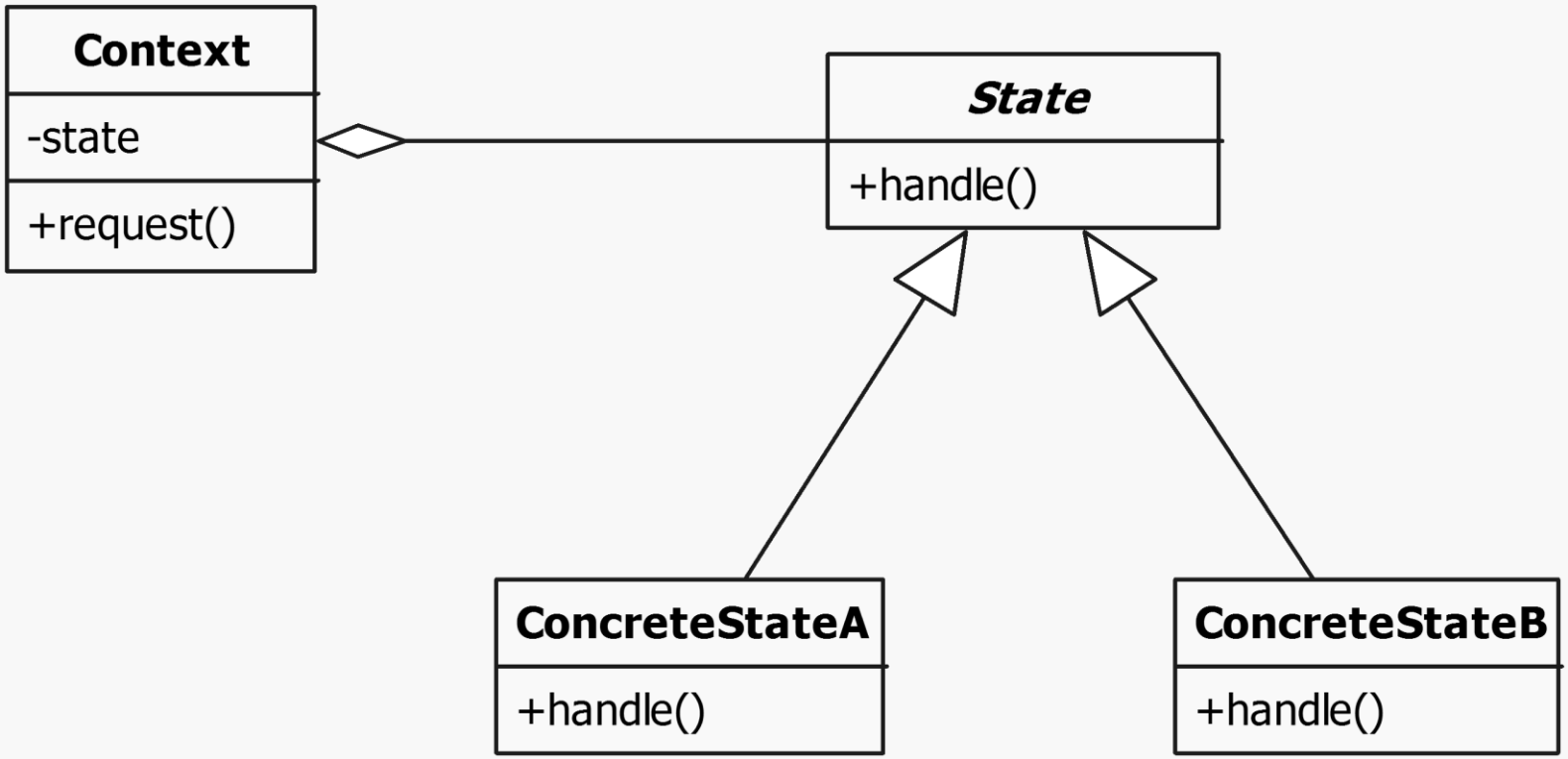
Состояние — шаблон уровня объекта, предоставляющий таковому возможность изменять свое поведение с учетом своего текущего состояния.

Используется, когда:

- поведение объекта зависит от его состояния и должно динамически изменяться;
- в системе присутствуют многочисленные, структурно идентичные условные операторы, в которых выбор ветви зависит от состояния объекта.

Участники: контекст, абстрактное и конкретное состояние.

Состояние: реализация



Поведенческие шаблоны: стратегия

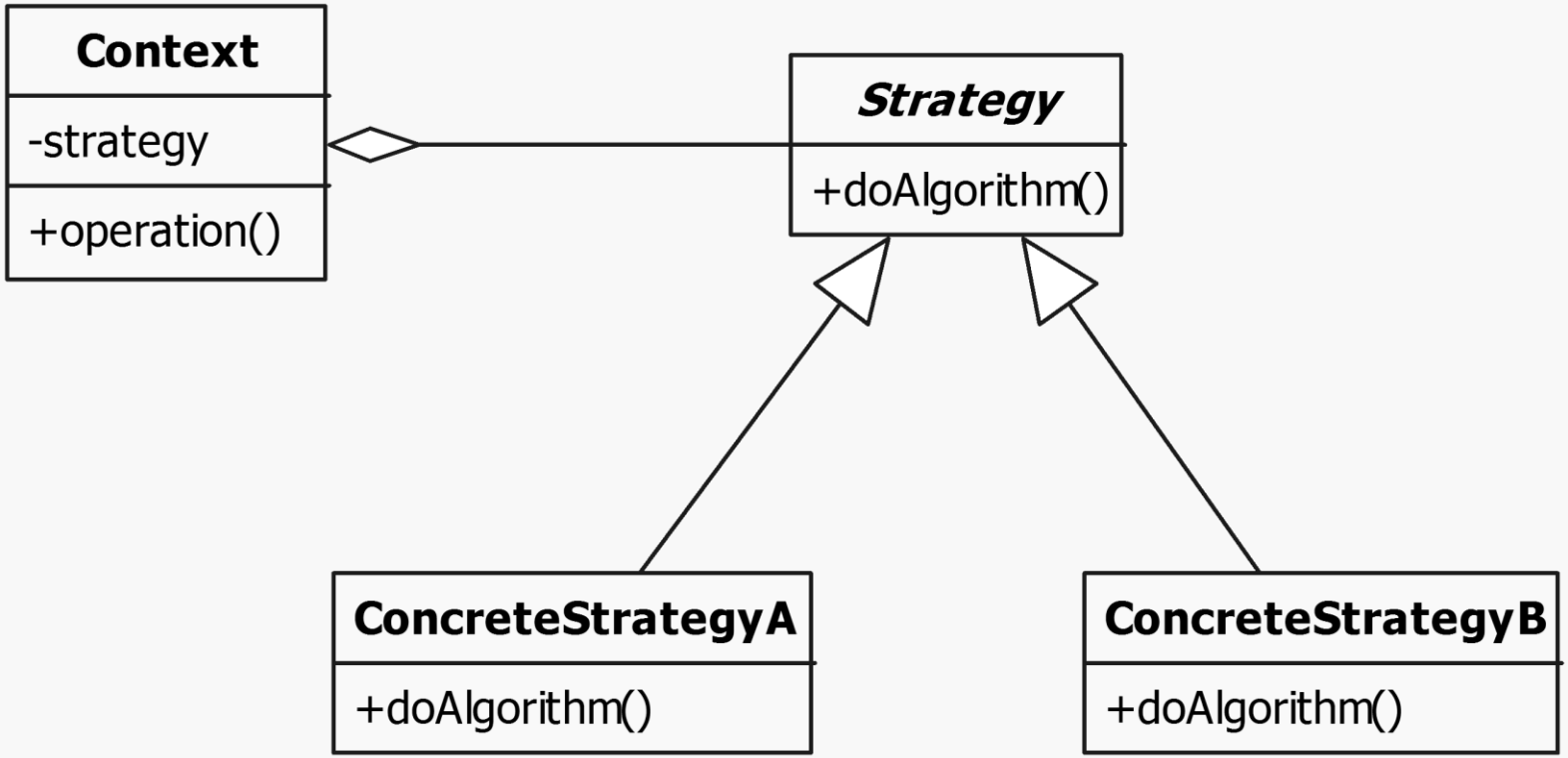
Стратегия (политика, англ. policy) — шаблон уровня объекта, определяющий семейство алгоритмов, инкапсулирующий каждый из них и делающий их взаимозаменяемыми.

Используется, когда:

- множество родственных классов отличается исключительно поведением;
- необходимо наличие нескольких вариантов одного алгоритма — например, оптимизированного по использованию ресурсов памяти (один) и процессора (другой);
- нежелательно раскрывать клиенту сложные, специфичные для алгоритма структуры данных;
- класс содержит множество вариантов поведения, представленных разветвленными условными операторами.

Участники: абстрактная и конкретная стратегия, контекст.

Стратегия: реализация



Параметризация стратегией шаблона класса-контекста

```
template <class Strategy> class Context {
public: // ...
    void operation() { _strategy.doAlgorithm(); }
private:
    Strategy _strategy;
};

class SomeStrategy {
public:
  void doAlgorithm();
};
// в точке использования
Context<SomeStrategy> context;
```

Поведенческие шаблоны: шаблонный метод

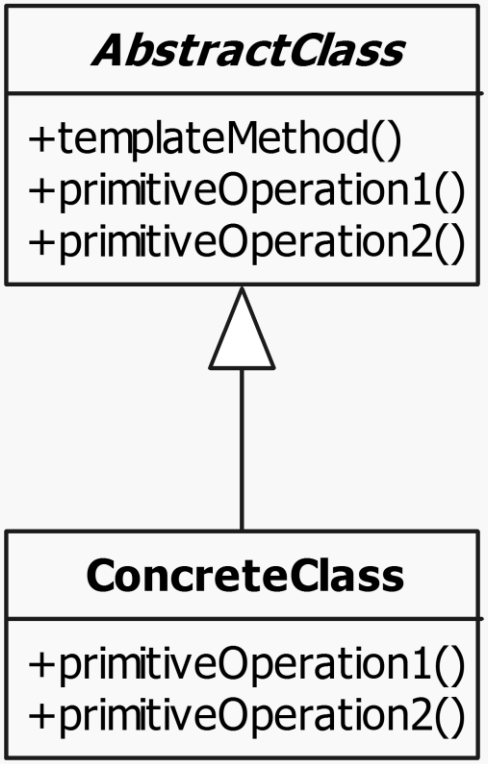
Шаблонный метод — шаблон уровня класса, один из фундаментальных приемов повторного использования кода. Определяет логическую структуру алгоритма и позволяет подклассам определить (переопределить) его некоторые шаги.

Используется, когда:

- имеется возможность однократно определить инвариантные части алгоритма и делегировать реализацию изменяющегося поведения производным классам;
- необходимо локализовать в одном классе (методе) поведение, общее для подклассов, — пример техники «вынесения за скобки для обобщения»;
- актуален вопрос управляемости расширений в подклассах — в точках вызова операций-зацепок (англ. hook operations).

Участники: абстрактный и конкретный класс.

Шаблонный метод: реализация



«Зацепки» или примитивные операции?

	Операция-зацепка	Примитивная операция
Роль в шаблонном методе	Метод с реализацией по умолчанию (часто пустой)	Абстрактный метод (чисто виртуальная функция)
Подлежит перекрытию	Если необходимо	Да

«Родительский контроль»

```
// без шаблонного метода
void DerivedClass::operation() {
    ParentClass::operation();
    // ...
}

// с шаблонным методом
void ParentClass::operation() {
    // ...
    hookOperation();
}
void ParentClass::hookOperation() { }
void DerivedClass::hookOperation() { /* ... */ }
```

Поведенческие шаблоны: посетитель

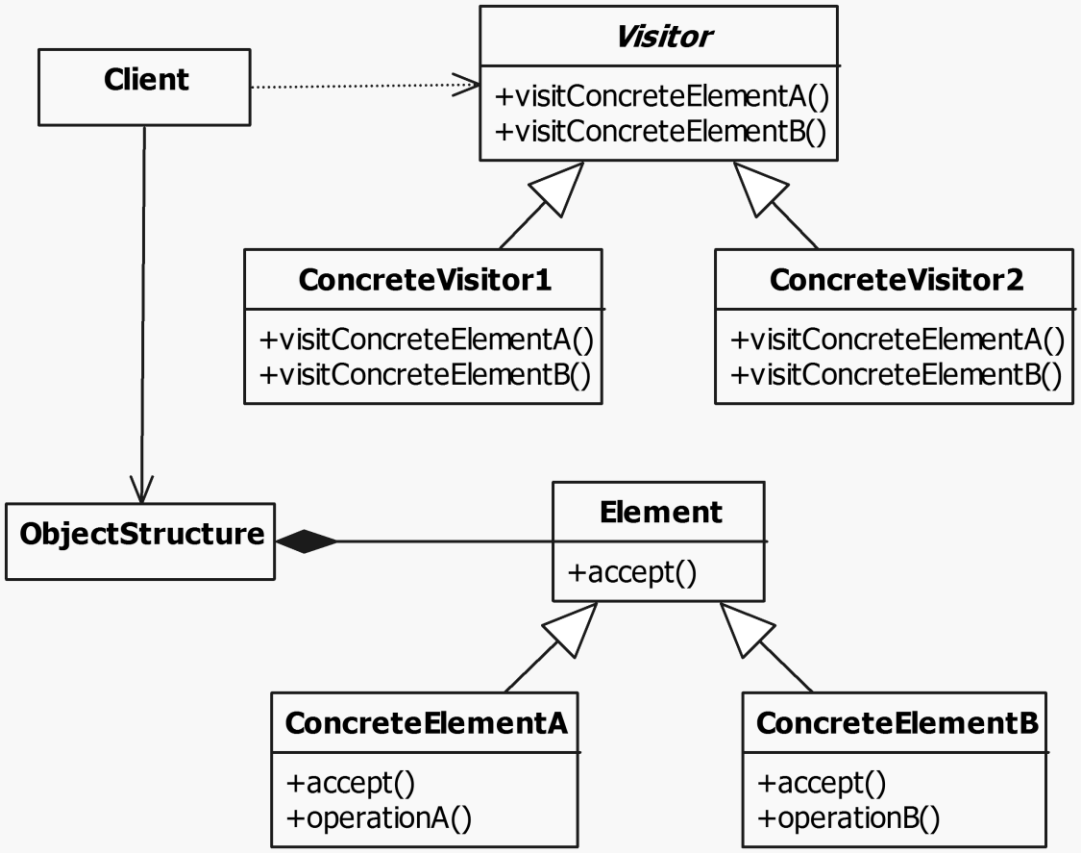
Посетитель — шаблон уровня объекта, средство описания операции, выполняемой с каждым объектов из некоторой структуры.

Используется, когда:

- необходимо определить новую операцию над объектами класса, не модифицируя класс;
- в структуре присутствуют объекты различных классов (и типов!), над которыми требуется выполнять операции, зависящие от класса;
- над объектами в составе структуры необходимо выполнять разнообразные операции, которые нецелесообразно включать в состав классов.
- номенклатура классов структуры стабильна, но часто пополняется новыми операциями.

Участники: абстрактный и конкретный посетитель, абстрактный и конкретный элемент, структура объектов.

Посетитель: реализация



Двойная диспетчеризация

Двойная диспетчеризация — известная в языках программирования техника выбора вызываемой операции в зависимости от двух типов.

Примечание. В контексте шаблона «посетитель» операция выбирается в зависимости от типа конкретного посетителя и конкретного элемента.

Двойная диспетчеризация является частным случаем множественной диспетчеризации и **не поддерживается в C++ напрямую**.

Примечание. Поддерживаемая C++ одинарная диспетчеризация предполагает, что операция выбирается исходя из имени запроса (идентификатора операции) и класса объекта-получателя. Поддержка двойной диспетчеризации в C++ свела бы ценность данного шаблона к нулю.

Структура абстрактного посетителя

```
class Visitor {
public:
    virtual void visitElementA(ElementA*);
    virtual void visitElementB(ElementB*);
    // ...

protected:
    Visitor();
};
```

Структура абстрактного и конкретного элементов

```
class Element {
public:
    virtual ~Element();
    virtual void Accept(Visitor&) = 0;
protected:
    Element();
};
class ElementA : public Element {
public:
    ElementA();
    virtual void Accept(Visitor &visitor) {
        visitor.visitElementA(this);
    }
};
```

Неустранимые конфликты в архитектуре

Достижение экстремальных значений одних показателей качества системы нередко приводит к недопустимому «провалу» по другим показателям. Подобные ситуации составляют суть **конфликтов** в архитектуре, многие из которых **неустранимы**. Например:

- скорость работы — объем требуемой памяти;
- гибкость — простота;
- CAP-теорема (Э. Брюер, 2000). В любой реализации распределенных вычислений возможно обеспечить **не более 2 из 3** следующих свойств:
  (1) **согласованность данных** — во всех вычислительных узлах в один момент времени данные не противоречат друг другу;
  (2) **доступность** — любой запрос к распределённой системе завершается корректным откликом;
  (3) **устойчивость к разделению** — расщепление распределённой системы на несколько изолированных секций не приводит к некорректности отклика от каждой из секций.

CAP-теорема

The CAP Theorem. □ You can have at most two of these properties for any shared-data system: Consistency, Availability, Tolerance to network Partitions. ■

- CA — системы с поддержкой ACID-транзакций (напр., LDAP):
  - ACID — атомарность, согласованность, изолированность, надежность;
- AP — «слабо целостные» системы с приемлемым уровнем практической значимости хранимых данных (напр., DNS);
- CP — распределенные системы, способные функционировать при распаде, но допускающие отсутствие отклика.

Закон Деметра

Закон Деметра [для функций и методов] (англ. Law of Demeter for Functions / Methods, LoD-F) — правило, относящееся к дизайну и стилю оформления объектно-ориентированного кода в соответствии с принципом «наименьшего знания» об экземплярах классов внутри (и более широко — организации) вызываемых методов. Формально, LoD-F гласит:

☐ Метод *M* объекта *O* вправе вызывать только методы:

- самого себя;
- своих параметров;
- любых объектов, экземпляры которых он создает;
- своих прямых подобъектов. ■

Соблюдение LoD-F позволяет достичь наименьшей возможной связанности объектов и стабильности кода при изменении глубинных структур объектов.

Шаблоны GRASP

Альтернативным каталогом шаблонов ОО-проектирования, имеющим несомненный авторитет, является каталог К. Лармана (англ. Craig Larman) **GRASP** (General Responsibility Assignment Software Patterns).

Девятку шаблонов GRASP образуют:

- **Информационный эксперт** (англ. Information Expert);
- **Создатель** (англ. Creator);
- **Контроллер** (англ. Controller);
- **Сильная связность** (англ. High Cohesion);
- **Слабое зацепление** (англ. Low Coupling);
- **Полиморфизм** (англ. Polymorphism);
- **Чистая выдумка** (англ. Pure Fabrication);
- **Посредник** (англ. Indirection);
- **Сокрытие реализации** (англ. Protected Variation).

Практикум №6

Постановка задачи

Дополнить учебный проект реализацией трех шаблонов объектно-ориентированного проектирования, как минимум два из которых должны быть заимствованы из каталога GoF.

Цель — выявить потенциальные проблемы дальнейшего сопровождения и развития иерархии классов, спланировать и осуществить системную трансформацию интерфейсов и реализации приложения.

ТЕХНОСФЕРА

Спасибо за внимание!

Алексей Петров

Приложение

Абстрактная фабрика: обсуждение

Результаты:

- сокрытие (изоляция) конкретных классов — имена изготавливаемых классов известны только абстрактной фабрике;
- упрощение замены семейств продуктов — класс фабрики упоминается в приложении только при ее инстанцировании;
- обеспечение гарантии сочетаемости продуктов.

Недостаток:

- трудность поддержки новых видов продуктов.

Аспекты реализации:

- фабрика как объект класса с единственным экземпляром;
- создание продуктов через фабричные методы или прототипы;
- расширяемость фабрик.

Строитель: обсуждение

Результаты:

- возможность изменения внутреннего представления продукта — процесс сборки, представление и внутренняя структура продукта могут быть скрыты за абстрактным интерфейсом строителя;
- изоляция кода создания и представления сложного объекта — классы, определяющие внутреннюю структуру продукта, в интерфейсе строителя не присутствуют;
- возможность пошаговой сборки — предоставляемый строителем контроль над сборкой продукта более тонок, чем у других шаблонов.

Аспекты реализации:

- интерфейс сборки и конструирования;
- отсутствие абстрактных продуктов;
- определение пустых методов в абстрактных строителях.

Фабричный метод: обсуждение (1 / 2)

Результаты:

- независимость системы от классов продуктов — имена классов продуктов инкапсулированы в фабричных методах конкретных создателей;
- возможность использования фабричных методов как операций-«зацепок» (англ. hook operations);
- соединение параллельных иерархий классов — фабричный метод инкапсулирует сведения о том, какие классы могут работать вместе.

Фабричный метод: обсуждение (2 / 2)

Недостаток:

- подкласс конкретного создателя должен определяться для создания даже одного экземпляра интересующего продукта.

Аспекты реализации:

- базовый класс–создатель с неабстрактным фабричным методом
- параметризация фабричного метода дискриминантом продукта
- использование шаблонов классов для автоматизации определения классов конкретных создателей.

Прототип: обсуждение (1 / 2)

Результаты:

- сокрытие (изоляция) конкретных классов — сокращает количество имен классов, известных клиенту;
- поддержка динамического изменения состава продуктов — клиент может удалять и устанавливать прототипы во время выполнения кода;
- сокращение количества необходимых системе классов — роль «класса»-продукта может выполнять должным образом настроенный и зарегистрированный на стороне клиента экземпляр-прототип, а роль класса-создателя — операция клонирования;
- динамическое конфигурирование приложений.

Прототип: обсуждение (2 / 2)

Недостаток:

- необходимость поддержки (в общем случае, нетривиальной) операции клонирования объектов.

Аспекты реализации:

- использование диспетчера прототипов (ассоциативного реестра, возвращающего искомый прототип по заданному ключу);
- клонирование путем «глубинного» (не имеет встроенной поддержки в C++) или «поверхностного» (выполняется почленно, по умолчанию) копирования;
- инициализация клонов.

Класс с единственным экземпляром: обсуждение (1 / 2)

Результаты:

- контролируемый доступ к единственному экземпляру;
- сокращение количества глобальных переменных;
- возможность уточнения операций и представления;
- возможность существования переменного количества экземпляров;
- большая гибкость в сравнении с классом, содержащим только статические атрибуты и методы (альтернативный подход к реализации одиночек).

Класс с единственным экземпляром: обсуждение (2 / 2)

Недостатки:

- искусственное ограничение масштабируемости проекта;
- усложнение процедур модульного тестирования системы.

Аспекты реализации:

- гарантия единственности;
- отложенная инициализация — статическая может быть невозможна;
- невозможность установления взаимозависимостей одиночек — порядок вызова конструкторов глобальных объектов через границы единиц трансляции в C++ не определен.

Адаптер: результаты

Для адаптера класса:

- делегирование адаптируемому классу операций единственного целевого класса — шаблон не позволяет одновременно адаптировать класс и его подклассы;
- возможность полного замещения отдельных операций адаптируемого класса;
- отсутствие дополнительного уровня косвенности — обращение по указателю или ссылке не производится.

Для адаптера объекта:

- возможность одновременной адаптации иерархии классов — адаптер способен расширить функциональность самого адаптируемого класса и всех его имеющихся подклассов;
- сложность замещения операций адаптируемого класса — требует порождения его подкласса и композиции с ним адаптера.

Адаптер: обсуждение

Аспекты реализации:

- использование открытого наследования интерфейса (от целевого класса) и закрытого наследования реализации (от адаптируемого класса);
- трудоемкость выполнения адаптации — зависит от степени различия интерфейсов;
- двухсторонняя адаптация — соответствие адаптера интерфейсам двух адаптируемых классов и обеспечение его прозрачности для каждого из участников.

Мост: обсуждение (1 / 2)

Результаты:

- отделение реализации от интерфейса;
- улучшение показателей расширяемости — возможность независимого расширения иерархий классов реализации и абстракции;
- сокрытие деталей реализации от клиентов.

Мост: обсуждение (2 / 2)

Аспекты реализации:

- отсутствие абстрактного класса реализации — взаимно-однозначное соответствие абстракции и реализации (вырожденный мост);
- наличие реализации по умолчанию;
- выбор реализации — в конструкторе класса-абстракции, в процессе работы (динамическая замена), принятие решения о замене объектом-уполномоченным;
- создание объекта-реализации абстрактной фабрикой;
- обобществление объектов-реализаций.

Компоновщик: обсуждение (1 / 2)

Результаты:

- интерфейсное единообразие иерархии примитивных и составных объектов — рекурсивная композиция;
- упрощение архитектуры клиента — устранение ветвлений в операциях обработки объектов;
- облеченное добавление компонентов.

Недостаток:

- нереализуемость статического ограничения компонентного состава объектов — проверки возможны только при выполнении кода.

Аспекты реализации (начало):

- применение явных ссылок на родительские объекты — упрощает обход по дереву и поддержку шаблона «цепочка ответственности»;
- обобществление дочерних объектов — возможное применение «приспособленцев».

Компоновщик: обсуждение (2 / 2)

Аспекты реализации (окончание):

- максимизация интерфейса участника «компонент» — включение в интерфейс операций составных и примитивных (вырожденных составных!) объектов;
- реализация операций управления дочерними объектами — выбор между прозрачным (в корне иерархии, участник «компонент») и безопасным (в классе составного объекта) решением;
- преобразование к типу составного объекта;
- реализация перечня дочерних объектов — выбор оптимальной структуры данных;
- реализация операций вставки и удаления;
- упорядочивание дочерних объектов (например, как Z-порядок);
- кэширование результатов обхода и поиска — выбор объекта кэширования, инвалидация результатов.

Декоратор: обсуждение (1 / 2)

Результаты:

- достижение большей гибкости, чем при наследовании классов — в том числе за счет возможности многократного применения декоратора;
- избежание перегруженности функциями классов на вершине иерархии.

Недостатки:

- неполнота сходства компонента и декоратора — декорированный компонент близок, но не идентичен исходному;
- наличие множества мелких, похожих друг на друга объектов — сложность отладки и изучения системы.

Декоратор: обсуждение (2 / 2)

Аспекты реализации:

- отсутствие абстрактного декоратора — опциональное назначение (снятие) единственной обязанности позволяет делегировать переадресацию запросов конкретному декоратору;
- соответствие интерфейсов декорируемого компонента и декоратора — наследование общему классу, определяющему интерфейс, но не хранящему данные;
- рекурсивное вложение декораторов — декоратор прозрачен для компонента.

Фасад: обсуждение

Результаты:

- изоляция клиентов от компонентов подсистемы — уменьшение количества используемых клиентами имен, упрощение работы;
- ослабление связей клиентов и подсистемы — облегчение декомпозиции системы на слои (уровни), структуризации зависимостей между ее компонентами;
- упрощение повторной компиляции и переноса системы.

Аспекты реализации:

- уменьшение степени связанности клиентов и подсистемы — использование абстрактных фасадов или фасадов, параметризованных одним или несколькими объектами подсистемы;
- открытые и закрытые классы подсистем.

Структурные шаблоны: приспособленец

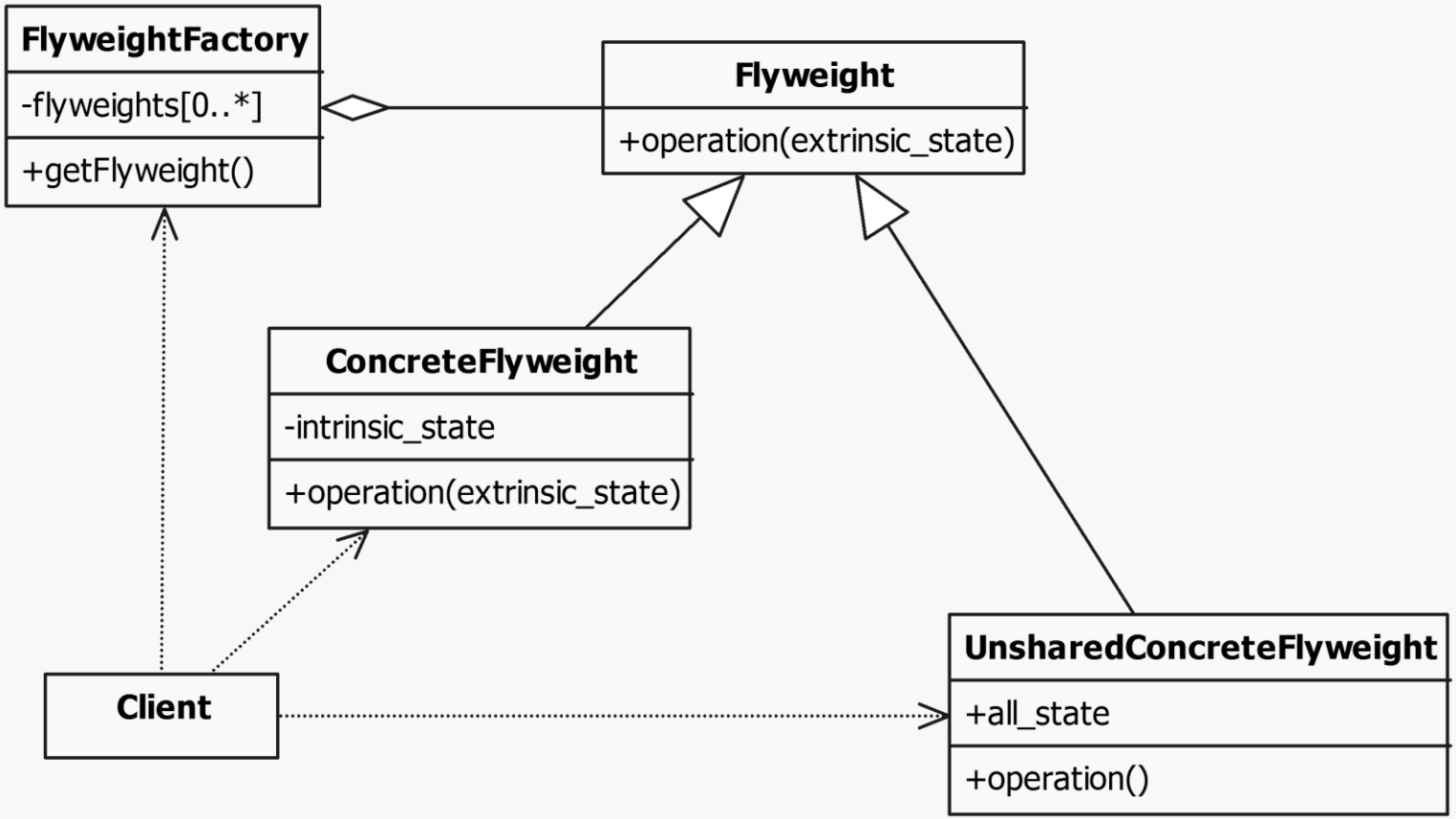
Приспособленец — шаблон уровня объекта, обобществляющий множественные мелкие объекты для их эффективного применения.

Используется, когда (Sic!) **одновременно:**

- число объектов в приложении велико, и накладные расходы на их хранение высоки;
- состояние объектов может передаваться им извне как параметры операций;
- множество групп объектов можно заменить небольшим числом обобществленных объектов;
- приложение не зависит от идентичности (не равенства!) объектов.

Участники: абстрактный и конкретный приспособленец, неразделяемый конкретный приспособленец, фабрика, клиент.

Приспособленец: реализация



Приспособленец: обсуждение

Результаты:

- уменьшение количества экземпляров — экономия памяти;
- сокращения объема памяти, отводимой под внутреннее состояние, — экономия памяти;
- вычисление (не хранение!) внешнего состояния — экономия памяти.

Недостатки:

- неизбежные затраты на передачу, поиск, вычисление внешнего состояния.

Аспекты реализации:

- простота выявления и вынесения внешнего состояния — внешнее состояние предпочтительно вычислять по объектам с другой структурой, требующим меньше памяти;
- управление обобществленными объектами — использование фабрик приспособленцев для генерации и ассоциативных массивов для поиска.

Заместитель: обсуждение

Результаты:

- создание дополнительного уровня косвенности;
- **для удаленного заместителя:** сокрытие пребывания субъекта в ином адресном пространстве;
- **для виртуального заместителя:** оптимизация работы — например, инстанцирование субъекта по требованию, копирование при записи (англ. copy-on-write), подсчет ссылок и т.д.;
- **для защищающего заместителя:** решение специализированных задач доступа.

Аспекты реализации:

- перегрузка операций доступа к членам класса;
- использование (замещение) абстрактных субъектов — незнание заместителем типа реального субъекта.

Команда: обсуждение

Результаты:

- устранение связанности инициатора и получателя объекта-команды;
- возможность произвольного манипулирования командами, включая сборку составных команд (макрокоманд) из элементарных;
- возможность простого добавления новых команд.

Аспекты реализации:

- интеллектуальность команды — от примитивного носителя информации до самостоятельного агента, не зависящего от своего получателя;
- возможность обобщенного программирования команд (см. далее).

Итератор: обсуждение

Результаты:

- поддержка нескольких различных видов обхода — наиболее актуально для сложных агрегатов;
- упрощение интерфейса агрегатов — операции обхода реализуются исключительно итераторами;
- возможность существования нескольких активных обходов.

Посредник: обсуждение (1 / 2)

Результаты:

- абстрагирование способа кооперирования;
- устранение связанности коллег;
- сокращение количества порождаемых подклассов — для изменения поведения подклассы порождаются исключительно от посредника;
- упрощение протокола взаимодействия объектов — протокол «один ко многим» проще сопровождать, расширять и анализировать.

Посредник: обсуждение (2 / 2)

Недостаток:

- централизация управления — формирование монолитного объекта-посредника, более «тяжелого», чем объекты-коллеги.

Аспекты реализации:

- отсутствие абстрактного посредника — при наличии единственного способа кооперации и единственного посредника;
- порядок обмена информацией между коллегами п посредником — уведомления, извещения и пр.

Наблюдатель: обсуждение (1 / 2)

Результаты:

- возможность независимой модификации субъектов и наблюдателей, а также повторного их независимого использования;
- возможность добавления новых наблюдателей без изменения субъекта или других наблюдателей;
- абстрактный характер и минимализм связей;
- возможность пребывания субъектов и наблюдателей в разных слоях (на разных уровнях) системы — отсутствуют объекты, пересекающие границы и компрометирующие принципы формирования слоев (уровней) ИС;
- поддержка широковещательных коммуникаций — субъект не «знает» о количестве наблюдателей и их типах.

Наблюдатель: обсуждение (2 / 2)

Недостатки:

- возможность каскадных нежелательных обновлений;
- низкая информативность простого протокола уведомления — неизвестно, что именно изменилось в объекте.

Аспекты реализации:

- отображение субъектов на наблюдателей — возможность использования ассоциативных массивов и иных структур данных;
- наблюдение за несколькими субъектами;
- выбор уведомляющей стороны — клиенты или субъекты;
- недействительные ссылки на удаленные субъекты;
- непротиворечивость субъекта перед отправкой уведомления;
- (не)зависимость протокола обновления от наблюдателя;
- задание интересующих модификаций;
- инкапсуляция сложной семантики обновления (менеджер изменений).

Состояние: обсуждение (1 / 2)

Результаты:

- локализация зависимого от состояния поведения в отдельных классах;
- декомпозиция поведения на части, соответствующие состояниям;
- простота добавления новых состояний и переходов;
- устранение нежелательных громоздких ветвлений и процедур — код становится менее монолитным, что улучшает его структуру;
- приобретение переходами явно выраженного, атомарного характера — вызов методов взамен присваивания атрибутам новых значений предпочтительнее и усиливает защиту от рассогласования переменных;
- возможность обобществления состояний-приспособленцев.

Состояние: обсуждение (2 / 2)

Недостатки:

- распределение поведения контекста между несколькими подклассами;
- увеличение числа подклассов;

Аспекты реализации:

- децентрализация логики переходов — следующее состояние контекста и момент перехода определяются классами-состояниями, что требует включения соответствующих операций в их интерфейс и вносит реализационные зависимости;
- выбор момента создания и уничтожения объекта состояния.

Стратегия: обсуждение (1 / 2)

Результаты:

- определение семейства родственных алгоритмов, допускающих повторное использование в разных контекстах;
- устранение необходимости порождения подклассов или использования условных операторов.

Недостатки:

- необходимость предоставления клиенту сведений о стратегиях — шаблон целесообразен, когда различия в поведении значимы для клиента;
- увеличение числа объектов.

Стратегия: обсуждение (2 / 2)

Аспекты реализации:

- определение интерфейса между стратегией и контекстом — контекст должен передавать стратегиям список параметров в объеме, требуемом наименее тривиальной стратегией, или себя как единственный аргумент;
- параметризация стратегией шаблона класса-контекста — устраняется необходимость в абстрактной стратегии; статическое связывание стратегии и контекста повышает эффективность решения;
- отсутствие стратегий — наличие поведения по умолчанию.

Шаблонный метод: обсуждение

Результаты:

- создание кода с инвертированной структурой — родительский класс вызывает операции подкласса, а не наоборот.

Аспекты реализации:

- контроль доступа к операциям — предназначенные для вызова шаблонным методом примитивные операции могут определяться как защищенные;
- сокращение количества примитивных операций — простота программирования клиента.

Посетитель: обсуждение (1 / 2)

Результаты:

- упрощение добавления операций;
- объединение родственных операций;
- возможность (в отличие от итератора) посещения объектов, не имеющих общего родительского класса;
- возможность аккумулирования состояния обхода в атрибутах посетителя, — а не параметрах его операций или глобальных переменных.

Недостатки:

- сложность добавления новых конкретных элементов структуры;
- большое количество необходимых для посетителя открытых операций доступа к содержимому элементов ставит под угрозу инкапсуляцию элементов.

Посетитель: обсуждение (2 / 2)

Аспекты реализации:

- двойная диспетчеризация — вызываемая операция зависит от класса элемента и класса посетителя одновременно;
- сторона управления обходом — структура, посетитель, уполномоченный внешний или внутренний итератор.

Примечание: Вариант посетителя нежелателен, так как приводит к дублированию кода обхода структуры в каждом конкретном посетителе для каждого агрегата.