**Лабораторная работа № 1. Задание 1.1.**

[Задание 1.1. Паттерны объектно-ориентированного проектирования 1](#_Toc98778534)

# Задание 1.1. Паттерны объектно-ориентированного проектирования

**Цель**

Главной задачей данной лабораторной работы является ознакомление и практическое применение паттернов проектирования при разработке программного обеспечения.

**Описание**

Любая хорошо структурированная объектно-ориентированная архитектура изобилует паттернами. Проектирование объектно-ориентированных программ – нетривиальная задача, а если их нужно использовать повторно, то все становится еще сложнее. Необходимо подобрать подходящие объекты, отнести их к различным классам, соблюдая разумную степень детализации, определить интерфейсы классов и иерархию наследования и установить существенные отношения между классами. Дизайн должен, с одной стороны, соответствовать решаемой задаче, с другой – быть общим, что бы удалось учесть все требования, которые могут возникнуть в будущем. Хотелось бы также избежать вовсе или, по крайней мере, свести к минимуму необходимость перепроектирования.

Паттерны проектирования – это не то же самое, что связанные списки или хэш-таблицы, которые можно реализовать в виде класса и повторно использовать без каких бы то ни было модификаций. Но это и не сложные, предметно-ориентированные решения для целого приложения или подсистемы. Под паттернами проектирования понимается описание взаимодействия объектов и классов, адаптированных для решения общей задачи проектирования в конкретном контексте. Паттерн проектирования именует, абстрагирует и идентифицирует ключевые аспекты структуры общего решения, которые и позволяют применить его для создания повторно используемого дизайна. Он вычленяет участвующие классы и экземпляры, их роль и отношения, а также функции.

Паттерны по своей цели (назначению) подразделяются на:

* порождающие паттерны;
* структурные паттерны;
* паттерны поведения.

Порождающие паттерны классов частично делегируют ответственность за создание объектов своим подклассам, тогда как порождающие паттерны объектов передают ответственность другому объекту. Структурные паттерны классов используют наследование для составления классов, в то время как структурные паттерны объектов описывают способы сборки объектов из частей. Поведенческие паттерны классов используют наследование для описания алгоритмов и потока управления, а поведенческие паттерны объектов описывают, как объекты, принадлежащие некоторой группе, совместно функционируют и выполняют задачу, которая ни одному отдельному объекту не под силу.

В данной лабораторной работе будут рассмотрены следующие паттерны:

* «Одиночка» (Singleton);
* «Абстрактная фабрика» (Abstract Factory);
* «Фасад» (Facade);
* «Команда» (Command);
* «Прототип» (Prototype);
* «Стратегия» (Strategy);
* «Приспособленец» (Flyweight);
* «Шаблонный метод» (Template Method)

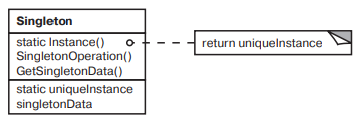
**Singleton**

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

Используется, когда:

* должен быть ровно один экземпляр некоторого класса, легко доступный всем клиентам;
* единственный экземпляр должен расширяться путем порождения подклассов, и клиентам нужно иметь возможность работать с расширенным экземпляром без модификации своего кода.

На рис. 1 представлена его структура.



*Рис.1. Структура паттерна Singleton*

Сторона **Singleton**:

* определяет операцию *Instance*, которая позволяет клиентам получать доступ к единственному экземпляру. *Instance* – это операция класса, статическая функция-член в C++;
* может нести ответственность за создание собственного уникального экземпляра.

Сторона **клиента**:

* получают доступ к экземпляру класса *Singleton* только через его операцию *Instance*.

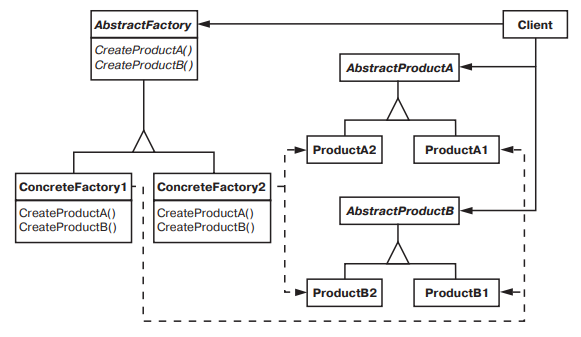
**Abstract Factory**

Является порождающим паттерном. Предоставляет интерфейс для создания семейств взаимосвязанных или взаимозависимых объектов, не специфицируя их конкретных классов.

Используется, когда:

* система не должна зависеть от того, как создаются, компонуются и пред­ставляются входящие в нее объекты;
* входящие в семейство взаимосвязанные объекты должны использоваться вместе и необходимо обеспечить выполнение этого ограничения;
* система должна конфигурироваться одним из семейств составляющих ее объектов;
* необходимо предоставить библиотеку объектов, раскрывая только их интерфейсы, но не реализацию.

На рис. 2 представлена его структура.



*Рис. 2 Структура паттерна Abstract Factory*

Сторона **AbstractFactory**:

* объявляет интерфейс для операций, создающих абстрактные объекты­ продукты;

Сторона **ConcreteFactory** (конкретная фабрика):

* реализует операции, создающие конкретные объекты-продукты;

Сторона **AbstractProduct** (абстрактный продукт):

* объявляет интерфейс для типа объекта-продукта;

Сторона **ConcreteProduct** (конкретный продукт):

* определяет объект-продукт, создаваемый соответствующей конкретной фабрикой;
* реализует интерфейс *AbstractProduct*;

Сторона **клиента**:

* пользуется исключительно интерфейсами, которые объявлены в классах *AbstractFactory* и *AbstractProduct*.

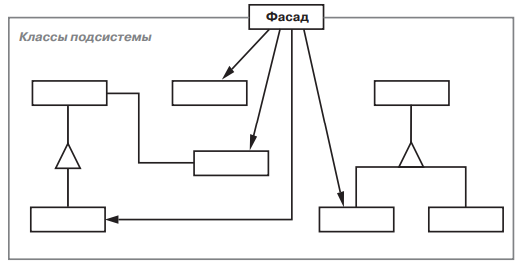
**Facade**

Является структурным паттерном. Предоставляет унифицированный интерфейс вместо набора интерфейсов не­которой подсистемы. *Facade* определяет интерфейс более высокого уровня, кото­рый упрощает использование подсистемы

Используется, когда:

* необходимо предоставить простой интерфейс к сложной подсистеме. Часто подсистемы усложняются по мере развития. Применение большинства паттернов приводит к появлению меньших классов, но в большем количестве. Такую подсистему проще повторно использовать и настраивать под конкретные нужды, но вместе с тем применять подсистему без настройки становится труднее. *Facade* предлагает некоторый вид системы по умолчанию, устраивающий большинство клиентов. И лишь те объекты, которым нужны более широкие возможности настройки, могут обратиться напрямую к тому, что находится за фасадом;
* между клиентами и классами реализации абстракции существует много зависимостей. *Facade* позволит отделить подсистему как от клиентов, так и от других подсистем, что, в свою очередь, способствует повышению степе­ни независимости и переносимости;
* необходимо разложить подсистему на отдельные слои. *Facade* будет использоваться для определения точки входа на каждый уровень подсистемы. Если подсистемы зависят друг от друга, то зависимость можно упростить, разрешив подсистемам обмениваться информацией только через фасады.

На рис. 3 представлена его структура.



*Рис. 3 Структура паттерна Facade*

Сторона **Facade**:

* «знает», каким классам подсистемы адресовать запрос;
* делегирует запросы клиентов подходящим объектам внутри подсистемы;

Сторона **классов** **подсистемы**:

* реализуют функциональность подсистемы;
* выполняют работу, порученную объектом *Facade*;
* ничего не «знают» о существовании фасада, то есть не хранят ссылок на него.

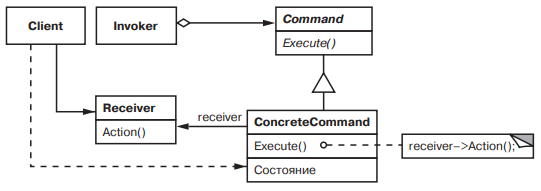
**Command**

Является паттерном поведения. Инкапсулирует запрос как объект, позволяя тем самым задавать параметры клиентов для обработки соответствующих запросов, ставить запросы в очередь или протоколировать их, а также поддерживать отмену операций.

Используется, когда:

* необходимо параметризовать объекты выполняемым действием, как в случае с пункта­ми меню. В процедурном языке такую параметризацию можно выразить с помощью функции обратного вызова, то есть такой функции, которая регистрируется, чтобы быть вызванной позднее. Команды представ­ляют собой объектно-ориентированную альтернативу функциям обратного вызова;
* определять, ставить в очередь и выполнять запросы в разное время. Время жизни объекта *Command* необязательно должно зависеть от времени жизни исходного запроса. Если получателя запроса удается реализовать так, чтобы он не зависел от адресного пространства, то объект-команду можно передать другому процессу, который займется его выполнением;
* поддержать отмену операций. Операция *Execute* объекта *Command* может сохранить состояние, необходимое для отката действий, выполненных командой. В этом случае в интерфейсе класса *Command* должна быть дополнительная операция *Unexecute*, которая отменяет действия, выполненные предшествующим обращением к *Execute*. Выполненные команды хранятся в списке истории. Для реализации произвольного числа уровней отмены и повтора команд нужно обходить этот список соответственно в обратном и прямом направлениях, вызывая при посещении каждого элемента команду *Unexecute* или *Execute*;
* поддержать протоколирование изменений, чтобы их можно было выполнить повторно после аварийной остановки системы. Дополнив интерфейс класса *Command* операциями сохранения и загрузки, возможно вести протокол изменений во внешней памяти. Для восстановления после сбоя нужно будет загрузить сохраненные команды с диска и повторно выполнить их с помощью операции *Execute*;
* структурировать систему на основе высокоуровневых операций, построенных из примитивных. Такая структура типична для информационных систем, поддерживающих транзакции. Транзакция инкапсулирует набор изменений данных. Паттерн *Command* позволяет моделировать транзакции. У всех команд есть общий интерфейс, что дает возможность работать одинаково с любыми транзакциями. С помощью этого паттерна можно легко добавлять в систему новые виды транзакций.

На рис. 4 представлена его структура.



*Рис. 4 Структура паттерна Command*

Сторона **Command**:

* объявляет интерфейс для выполнения операции;

Сторона **ConcreteCommand** (конкретная команда):

* определяет связь между объектом-получателем *Receiver* и действием;
* реализует операцию *Execute* путем вызова соответствующих операций объекта *Receiver*;

Сторона **клиента**:

* создает объект класса *ConcreteCommand* и устанавливает его получателя;

Сторона **Invoker** (инициатор):

* обращается к команде для выполнения запроса;

Сторона **Receiver** (получатель):

* располагает информацией о способах выполнения операций, необходи­мых для удовлетворения запроса. В роли получателя может выступать любой класс.

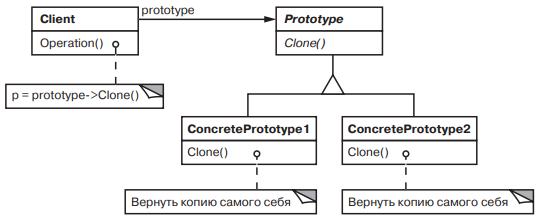
**Prototype**

Является порождающим паттерном. Задает виды создаваемых объектов с помощью экземпляра-прототипа и создает новые объекты путем копирования этого прототипа.

Используется, когда:

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

На рис. 5 представлена его структура.



*Рис. 5 Структура паттерна Prototype*

Сторона **Prototype**:

* объявляет интерфейс для клонирования самого себя;

Сторона **ConcretePrototype** (конкретный прототип):

* реализует операцию клонирования себя;

Сторона **клиента**:

* создает новый объект, обращаясь к прототипу с запросом клонировать себя.

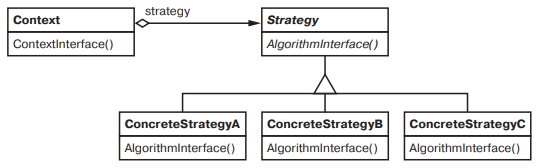
**Strategy**

Является паттерном поведения. Определяет семейство алгоритмов, инкапсулирует каждый из них и делает их взаимозаменяемыми. *Strategy* позволяет изменять алгоритмы независимо от клиентов, которые ими пользуются.

Используется, когда:

* имеется много родственных классов, отличающихся только поведением. Стратегия позволяет сконфигурировать класс, задав одно из возможных поведений;
* необходимо иметь несколько разных вариантов алгоритма. Например, можно определить два варианта алгоритма, один из которых требует больше времени, а другой – больше памяти. Стратегии разрешается применять, когда варианты алгоритмов реализованы в виде иерархии классов;
* в алгоритме содержатся данные, о которых клиент не должен «знать». Используйте паттерн стратегия, чтобы не раскрывать сложные, специфичные для алгоритма структуры данных;
* в классе определено много поведений, что представлено разветвленными условными операторами. В этом случае проще перенести код из ветвей в отдельные классы стратегий.

На рис. 6 представлена его структура.



*Рис. 6 Структура паттерна Strategy*

Сторона **Strategy**:

* объявляет общий для всех поддерживаемых алгоритмов интерфейс. Класс *Context* пользуется этим интерфейсом для вызова конкретного *алгоритма*, определенного в классе *ConcreteStrategy*;

Сторона **ConcreteStrategy** (конкретная стратегия):

* реализует алгоритм, использующий интерфейс, объявленный в классе *Strategy*;

Сторона **Context** (контекст):

* конфигурируется объектом класса *ConcreteStrategy*;
* хранит ссылку на объект класса *Strategy*;
* может определять интерфейс, который позволяет объекту *Strategy* получить доступ к данным контекста.

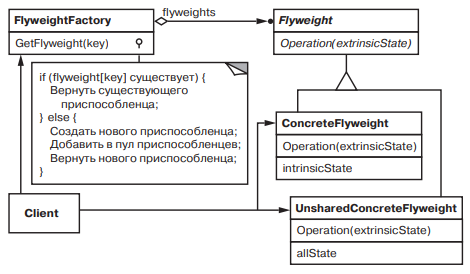
**Flyweight**

Является порождающим паттерном. Использует разделение для эффективной поддержки множества мелких объектов.

Используется, когда выполнены все ниже перечисленные условия (максимальная эффективность применения данного паттерна):

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

На рис. 7 представлена его структура.



*Рис. 7 Структура паттерна Flyweight*

Сторона **Flyweight**:

* объявляет интерфейс, с помощью которого приспособленцы могут получать внешнее состояние или как-то воздействовать на него;

Сторона **ConcreteFlyweight** (конкретный приспособленец):

* реализует интерфейс класса *Flyweight* и добавляет при необходимости внутреннее состояние. Объект класса *ConcreteFlyweight* должен быть разделяемым. Любое сохраняемое им состояние должно быть внутрен­ним, то есть не зависящим от контекста;

Сторона **UnsharedConcreteFlyweight** (неразделяемый конкретный приспособленец):

* не все подклассы *Flyweight* обязательно должны быть разделяемыми. Интерфейс *Flyweight* допускает разделение, но не навязывает его. Часто у объектов *UnsharedConcreteFlyweight* на некотором уровне структуры приспособленца есть потомки в виде объектов класса *ConcreteFlyweight*;

Сторона **FlyweightFactory** (фабрика приспособленцев):

* создает объекты-приспособленцы и управляет ими;
* обеспечивает должное разделение приспособленцев. Когда клиент запра­шивает приспособленца, объект *FlyweightFactory* предоставляет существующий экземпляр или создает новый, если готового еще нет;

Сторона **клиента**:

* хранит ссылки на одного или нескольких приспособленцев;
* вычисляет или хранит внешнее состояние приспособленцев.

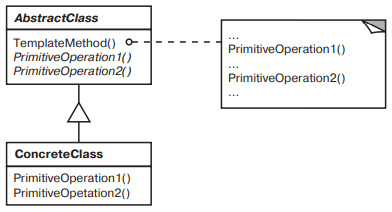
**Template Method**

Является паттерном поведения. Определяет основу алгоритма и позволяет подклассам пе­реопределить некоторые шаги алгоритма, не изменяя его структуру в целом.

Используется, когда:

* необходимо однократно использовать инвариантные части алгоритма, оставляя реализацию изменяющегося поведения на усмотрение подклассов;
* нужно вычленить и локализовать в одном классе поведение, общее для всех подклассов, дабы избежать дублирования кода. Это хороший пример техники «вынесения за скобки с целью обобщения»;
* для управления расширениями подклассов. Можно определить шаблонный метод так, что он будет вызывать операции-зацепки.

На рис. 8 представлена его структура.



*Рис. 8 Структура паттерна Template Method*

Сторона **AbstractClass**:

* определяет абстрактные примитивные операции, замещаемые в конкретных подклассах для реализации шагов алгоритма;
* реализует шаблонный метод, определяющий скелет алгоритма. *Template* method вызывает примитивные операции, а также операции, определенные в классе *AbstractClass* или в других объектах;

Сторона **ConcreteClass** (конкретный класс):

* реализует примитивные операции, выполняющие шаги алгоритма способом, который зависит от подкласса.

**Задание**

Выполнение лабораторной работы предполагает использование языка программирования C++/C#. Конченая программа может быть реализована в виде консольного приложения или как приложение с графическим интерфейсом (возможно также использовать Qt).

Заданием к лабораторной работе является:

* выбор предметной области, для которой можно применить паттерн, соответствующий номеру по журналу;
* выбор любого другого второго паттерна из рассмотренных выше, который бы также ложился в идеологию выбранной предметной области и сочетался бы с первым паттерном (можно использовать и *Singleton*, но для логирования действий должна быть его отдельная реализация);
* аргументация выбора предметной области и паттерна;
* реализация программы, которая бы могла наглядно продемонстрировать выполнение лабораторной работы.

Каждый из вариантов предполагает, помимо задания по варианту, использование паттерна *Singleton* для класса *Log*, используемого для логирования всех происходящих действий в программе (например, создание объектов классов, их удаление, вызов методов и пр.) в текстовый файл.

В отчете к лабораторной работе **должны** присутствовать:

* цель;
* вариант задания;
* описание выбранной предметной области;
* аргументация выбора для данной области используемых паттернов (кроме *Singleton’а* для логирования);
* описание классов (полей, методов, назначения, взаимодействия);
* диаграмма классов;
* результат работы программы;
* вывод;
* листинг программы (если программа имеет графический интерфейс, то описание самого GUI вставлять в листинг не нужно, только все места, где идет непосредственная работа с разработанными классами).

**Варианты**

Если вариант по журналу больше 8, то вариантом будет остаток от деления на 9. Например, для 13 варианта необходимо выбрать 5 вариант.

|  |  |
| --- | --- |
| **Вариант** | **Паттерн №1** |
| 1 | Abstract Factory |
| 2 | Singleton |
| 3 | Command |
| 4 | Prototype |
| 5 | Flyweight |
| 6 | Facade |
| 7 | Template Method |
| 8 | Strategy |

*Пример области и паттернов:*

Допустим, первым паттерном является *Facade*. Выберем предметной областью работу ресторана, где в качестве Facade скроем работу кухни, официантам будут доступны следующие возможности:

* отдать заказ;
* узнать о его состоянии готовности;
* забрать заказ.

Само приготовление блюда, поиск ингредиентов, очередь на приготовление, распараллеливание каких-то этапов готовки, запросы на склад – все это будет скрыто, у официантов лишь будет некоторый интерфейс для работы с кухней.

Пусть вторым паттерном будет *Prototype*, допустим, что гипотетический ресторан специализируется только на определенных наборах блюд, к которым можно что-либо добавить или убрать (аналогия с бизнес ланчем). Возможно применить паттерн и к самим официантам, но этот случай в данном примере рассматривать не будем.

Стоит уточнить, что предметной областью могут быть также некоторые аспекты программирования. Например, разработка своей библиотеки, взаимодействие извне с некоторой формой GUI и пр.

**Литература**

Гамма Э., Хелм Р., Джонсон Р., Влиссидес Дж. Приемы объектно-ориентированного проектирования. Паттерны проектирования. – СПб.: Питер, 2020. – 368 с.: ил. – (Серия «Библиотека программиста»).