Министерство образования Республики Беларусь

БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

Факультет прикладной математики и информатики

Кафедра информационных систем управления

Анищенко Арсений Игоревич

**Паттерны проектирования**

**Команда, Фасад, Адаптер, Итератор, Компоновщик**

Реферат

Студента 1 курса 2 группы

**Преподаватель**

Баранов Геннадий Аркадьевич

Минск, 2017

РЕФЕРАТ

Реферат, с. 16, 2 источника.

**Ключевые слова:** ПАТТЕРНЫ ПРОЕКТИРОВАНИЯ, ШАБЛОНЫ ПРОЕКТИРОВАНИЯ, КОМАНДА, ФАСАД, АДАПТЕР, ИТЕРАТОР, КОМПОНОВЩИК

**Объект исследования** – Шаблоны (паттерны) проектирования архитектуры компьютерных программ

**Цель работы** – изучить шаблоны проектирования и их применение

**Методы исследования** – изучение каталога шаблонов, применение их при написании приложений

**Область применения –** разработка программного обеспечения, приложений с использованием объектно-ориентированных языков программирования.

Содержание

[ВВЕДЕНИЕ 3](#_Toc484474825)

[КОМАНДА 4](#_Toc484474826)

[ИТЕРАТОР 6](#_Toc484474827)

[ФАСАД 8](#_Toc484474828)

[АДАПТЕР 10](#_Toc484474829)

[КОМПОНОВЩИК 12](#_Toc484474830)

[ЗАКЛЮЧЕНИЕ 14](#_Toc484474831)

[СПИСОК ЛИТЕРАТУРЫ 15](#_Toc484474832)

ВВЕДЕНИЕ

В сегодняшнее время при проектировании архитектуры приложений разработчиками часто решаются схожие задачи по организации взаимосвязей между элементами в программе. Типы данных задач естественным образом имеют свойство повторяться, и нужды «изобретать велосипед» разработчикам больше нет, ибо существует множество уже устоявшихся моделей организации взаимосвязей между элементами в программе, которые применяются в зависимости от поставленной задачи. Данные модели оказались функциональными в прошлом, что может гарантировать функциональное использование их в проектировании новых приложений в будущем. Во многих объектно-ориентированных системах можно встретить повторяющиеся шаблоны, состоящие из классов и взаимодействующих объектов. Использование данных систем позволяет решить ряд конкретных задач при проектировании, а архитектура и дизайн приложений с их использованием становятся более аккуратными и удобными, что наталкивает на то, чтобы воспользоваться ими повторно, тем самым облегчив работу разработчику в проектировании нового приложения.

Такие подходы к организации систем и называются шаблонами проектирования или паттернами (от английского pattern – шаблон). Проектировщик, знакомый с шаблонами, может сразу же применять их к новой задачи, не придумывая в очередной раз то, что уже было создано другими разработчиками. Паттерное проектирования упрощает разработку нового приложения, повторно используя удачные проектные и архитектурные решений.

Всего существует большое количество шаблонов проектирования, каждый из которых направлен на решение определённых задач и обладает своими преимуществами и недостатками. Задача проектировщика является выбор наиболее подходящего паттерна при моделировании системы приложения. Для этого необходимо ясно представлять все объекты, взаимодействующие между собой в системе, характер их связей, а также учитывать, как возможные изменения, которые будут внесены в систему, могут повлиять на архитектуру. Для того, чтобы применять верные шаблоны, правильно их адаптировать к конкретной задаче, необходимо обладать хорошими знаниями относительно существующих паттернов и их всевозможных вариантов использования. В данной работе будут рассмотрены следующие паттерны (шаблоны проектирования): Команда, Фасад, Адаптер, Итератор, Компоновщик.

КОМАНДА

Команда (от английского command) – поведенческий шаблон проектирования, что означает, что данный шаблон определяет алгоритмы и способы реализации взаимодействия различных объектов и классов. Данный шаблон используется при объектно-ориентированном программировании и представляет собой действие, то есть объекты Команды заключают в себе само действие и его параметры.

Целью Команды является создание структуры, в которой класс-отправитель и класс-получатель не зависят друг от друга напрямую. Другими словами – организация обратного вызова к классу, который включает в себя класс-отправитель.

Рассмотрим механизм работы Команды подробнее. Как сказано выше, в объектно-ориентированном программировании шаблон проектирования Команда является поведенческим шаблоном, в котором объект используется для инкапсуляции (упаковки данных и функций в единый компонент) всей информации, необходимой для выполнения действия или вызова события в более позднее время. Эта информация включает в себя имя метода, объект, который является владельцем метода и значения параметров метода.

Четыре термина всегда связанны с шаблоном Команда – это команды (command), приёмник команд (receiver), вызывающий команды (invoker) и клиент (client).



1 Схема. Команда.

Объект Command знает о приёмнике и вызывает метод приемника. Значения параметров приёмника сохраняются в команде. Вызывающий объект (invoker) знает, как выполнить команду и, возможно, делает учёт и запись выполненных команд. Вызывающий объект (invoker) ничего не знает о конкретной команде, он знает только об интерфейсе. Оба объекта (вызывающий объект и несколько объектов команд) принадлежат объекту клиента (client). Клиент решает, какие команды выполнить и когда. Чтобы выполнить команду он передает объект команды вызывающему объекту (invoker). Подробная схема шаблона Команда приведена на *1 Схема. Команда.*

Использование командных объектов упрощает построение общих компонентов, которые необходимо делегировать или выполнять вызовы методов в любое время без необходимости знать методы класса или параметров метода. Использование вызывающего объекта (invoker) позволяет ввести учёт выполненных команд без необходимости знать клиенту об этой модели учёта (такой учёт может пригодиться, например, для реализации отмены и повтора команд).

Таким образом, основополагающая идея шаблона заключается в использовании единого интерфейса для описания всех типов операций, которые можно производить с системой. Тогда для добавления в систему поддержки новой операции достаточно реализовать предлагаемый интерфейс. Каждая операция представляется самостоятельным объектом, инкапсулирующим некоторый набор дополнительных свойств. Система, в свою очередь, приобретает возможность выполнять дополнительный набор действий над запросами (объектами).

Рассмотрим самую очевидную мотивацию для использования шаблона Команда — логирование/undo-redo. Для начала каждая команда в оболочке инкапсулируется в отдельный класс — потомок класса Command. Затем необходимо реализовать поддержку механизмов протоколирования/логирования и что самое главное — отмены и повторения (undo/redo) действий пользователя (команд) для командной оболочки. Для достижения поставленной цели достаточно ввести понятие очереди выполненных команд, куда попадает всякая команда (объект, наследник Command) после выполнения и очереди отмененных команд, куда поступает любая отмененная команда. Помимо этого, необходимо иметь разделенные механизмы выполнения и отмены действий команды. Очевидно, что эти механизмы индивидуальны для каждой команды. Тогда для реализации целостной системы нам достаточно, оперируя двумя очередями применять к объектам, находящимся в их верхушке соответствующие действия — выполнение или отмену.

ИТЕРАТОР

К поведенческим шаблонам проектирования также относится такой шаблон проектирования как Итератор. Итератор представляет собой объект, позволяющий получить последовательный доступ к элементам объекта-агрегата без использования описаний каждого из агрегированных объектов, то есть Итератор предоставляет способ последовательного доступа ко всем элементам составного объекта, не раскрывая его внутреннего представления.

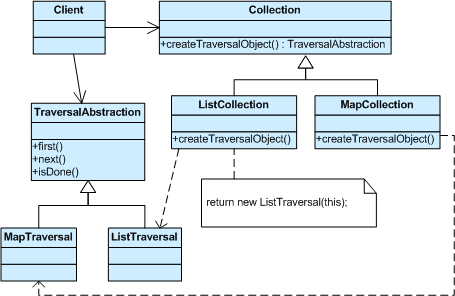
При помощи объекта Итератор можно легко перебрать и при необходимости модифицировать такие элементы как дерево, связанный список, хэш-таблица и массив. Перебор элементов будет выполнятся объектом итератора, а не самой коллекцией, что упрощает интерфейс и реализацию коллекции. Итератор обеспечивает более логичное разделение обязанностей.

Составной объект, такой как список, должен предоставлять способ доступа к его элементам без раскрытия своей внутренней структуры. Более того, иногда нужно перебирать элементы списка различными способами, в зависимости от конкретной задачи. Но перегружать интерфейс списка операциями для различных обходов, даже если они необходимы, весьма трудоёмкий процесс. Кроме того, иногда нужно иметь несколько активных обходов одного списка одновременно. Было бы хорошо иметь единый интерфейс для обхода разных типов составных объектов, то есть полиморфная итерация. Паттерн Iterator позволяет все это делать. Ключевая идея состоит в том, чтобы ответственность за доступ и обход переместить из составного объекта на объект Iterator, который будет определять стандартный протокол обхода, что и обеспечивает более логичное разделение обязанностей.

Абстракция Iterator имеет основополагающее значение для технологии, называемой "обобщенное программирование". Эта технология четко разделяет такие понятия как "алгоритм" и "структура данных". Мотивирующие факторы: способствование компонентной разработке, повышение производительности и снижение расходов на управление.

В качестве примера, пусть требуется одновременно поддерживать четыре вида структур данных (массив, бинарное дерево, связанный список и хэш-таблица) и три алгоритма (сортировка, поиск и слияние). Традиционный подход потребует 12 вариантов конфигураций. А обобщенное программирование потребует лишь 7.

Что касается структуры шаблона Итератор – подробная схема приведена на *2 Схема. Итератор.*



2 Схема. Итератор.

Для манипулирования коллекцией клиент использует открытый интерфейс класса Collection. Однако доступ к элементам коллекции инкапсулируется дополнительным уровнем абстракции, называемым Iterator. Каждый производный от Collection класс знает, какой производный от Iterator класс нужно создавать и возвращать. После этого клиент использует интерфейс, определенный в базовом классе Iterator.

ФАСАД

Шаблон проектирование Фасад, в отличии от Команды и Итератора, является структурным шаблоном проектирования. Структурный шаблон проектирования – это шаблон проектирования, в котором рассматривается вопрос о том, как из классов и объектов образуются более крупные структуры. Фасад позволяет скрыть сложность системы путём сведения всех возможных внешних вызовов к одному объекту, делегирующему их соответствующим объектам системы.

При проектировании сложных систем, зачастую применяется принцип декомпозиции, при котором сложная система разбивается на более мелкие и простые подсистемы. Причем, уровень декомпозиции (ее глубину) определяет исключительно проектировщик. Благодаря такому подходу, отдельные компоненты системы могу быть разработаны изолированно, затем интегрированы вместе. Однако возникает, очевидная на первый взгляд, проблема — высокая связность модулей системы. Это проявляется, в первую очередь, в большом объеме информации, которой модули обмениваются друг с другом. К тому же, для подобной коммуникации одни модули должны обладать достаточной информацией о природе других модулей.  
  
Таким образом, минимизация зависимости подсистем, а также снижение объема передаваемой между ними информации — одна из основных задач проектирования. Решить данную задачу позволяет паттерн Фасад.  
  
*«Паттерн «Фасад» предоставляет унифицированный интерфейс вместо набора интерфейсов некоторой подсистемы. Фасад определяет интерфейс более высокого уровня, который упрощает использование подсистемы».*

Проще говоря, «Фасад» — есть ни что иное, как некоторый объект, аккумулирующий в себе высокоуровневый набор операций для работы с некоторой сложной подсистемой. Фасад агрегирует классы, реализующие функциональность этой подсистемы, но не скрывает их. Важно понимать, что клиент, при этом, не лишается более низкоуровневого доступа к классам подсистемы, если ему это, конечно, необходимо. Фасад упрощает выполнение некоторых операций с подсистемой, но не навязывает клиенту свое использование.

Фасад определить одну точку взаимодействия с подсистемой — фасадный объект, обеспечивающий общий интерфейс с подсистемой, и возлагает на него обязанность по взаимодействию с её компонентами. Фасад — это



3 Схема. Фасад.

внешний объект, обеспечивающий единственную точку входа для служб подсистемы. Схема *3 Схема. Фасад.* демонстрирует принцип работы Фасада.

*«Фасад не только упрощает интерфейс, но и обеспечивает логическую изоляцию клиента от подсистемы, состоящих из многих компонентов».*

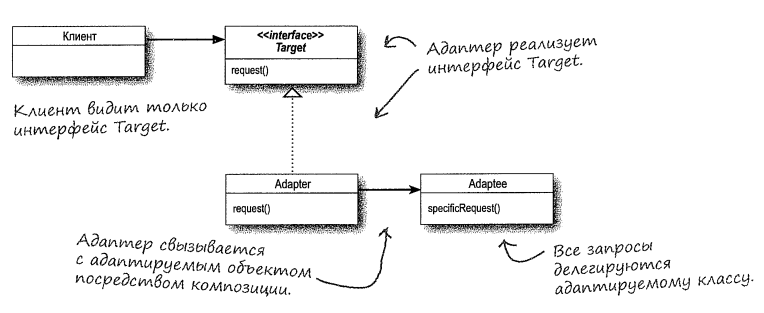
АДАПТЕР

Адаптер – это структурный шаблон проектирования, служащий для организации использования функций объекта, недоступного для модификации, через специально созданный интерфейс.

Паттерн Адаптер преобразует интерфейс класса к другому интерфейсу, на который рассчитан клиент. Адаптер обеспечивает совместную работу классов, невозможную в обычных условиях из-за несовместимости интерфейсов.

Иначе говоря, Адаптер адаптирует несовместимые интерфейсы.

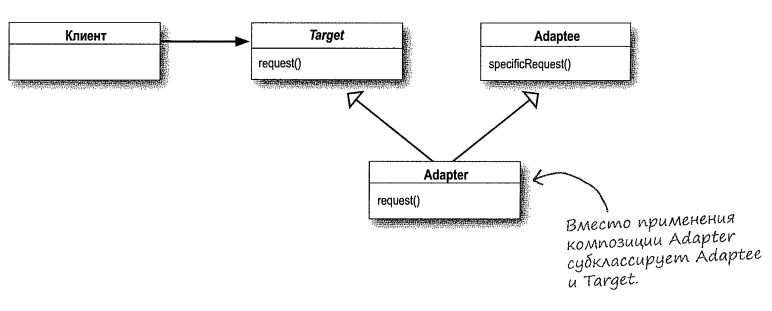
Для того, чтобы использовать клиента с несовместимым интерфейсом, можно создать адаптер, который выполнит преобразование. Таким образом клиент отделится от реализованного интерфейса, и, если ожидается, что интерфейс будет изменяться со временем, адаптер инкапсулирует эти изменения, чтобы клиента не приходилось изменять каждый раз, когда ему потребуется работать с новым интерфейсом.



4 Схема. Адаптер

На схеме *4 Схема. Адаптер.* приведена диаграмма классов, поясняющая принципы работы шаблона Адаптер. Стоит обратить внимание на то, что данный паттерн связывает клиента с интерфейсом, а не с реализацией. Можно использовать несколько адаптеров, каждый из которых будет выполнять преобразование для своего набора классов. Кроме того, новые реализации могут добавляться позднее, однако они должны соответствовать интерфейсу Target. Стоит обратить внимание и на использование композиции для упаковки адаптируемого объекта в измененый интерфейс. Дополнительным преимуществом такого решения является то, что адаптер будет работать с любым субклассом адаптируемого объекта.

Выше был приведён и рассмотрен Адаптер объектов. Однако, существует ещё один тип Адаптеров – Адаптер классов. Для реализации Адаптера классов необходимо множественное наследование, запрещённое во многих языках программирования. Но это не означает, что необходимости в Адаптерах классов не возникает.



5 Схема. Адаптер.

И так, на диаграмме классов *5 Схема. Адаптер.* приведён пример Адаптера классов. Не трудно видеть, что единственное различие между Адаптером объектов и классов является то, что с адаптером классов мы субклассируем Target и Adaptee, а с адаптером объектов для передачи запросов Adaptee используется механизм композиции.

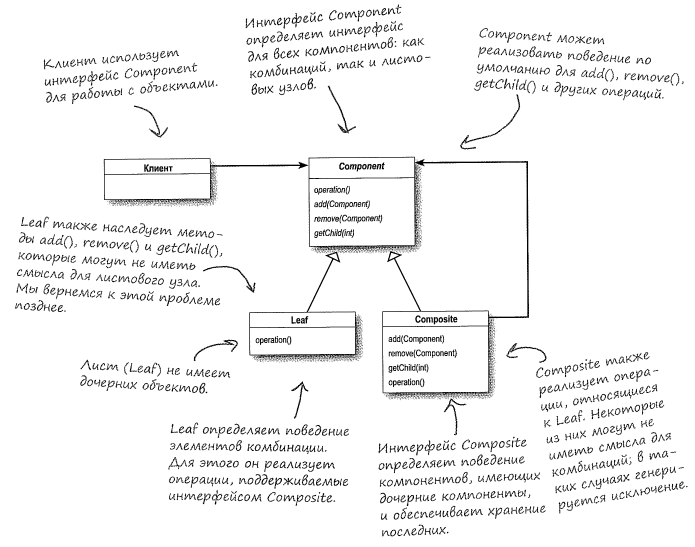
«Фасад применяется для упрощения, а Адаптер – для преобразования интерфейса к другой форме».

КОМПОНОВЩИК

Так же как Фасад и Адаптер, паттерн Компоновщик является структурным шаблоном проектирования. Паттерн Компоновщик объединяет объекты в древовидные структуры для представления иерархии. Компоновщик позволяет клиенту выполнять однородные операции с отдельными объектами и их совокупностями.

Существует большое количество программных систем, в которых так или иначе применяются древовидные структуры объектов. В большинстве случаев, это всевозможные конструкторы/редакторы, которые позволяют собрать что-то большое (составное) из чего-то более мелкого (листового). При этом, клиент трактует и большое, и мелкое как одно и тоже, а система должна различать составные и листовые объекты соответственно.

Не только программы-конструкторы используют данный паттерн. Ярким примером подобной древовидной структуры является пользовательский интерфейс.



6 Схема. Компоновщик.

Действительно, типичное окно пользовательского интерфейса — контейнер для более простых виджетов — панелей, кнопок, полей ввода, причем, панель, в свою очередь тоже являются контейнерными объектами и так вплоть до элементарных листовых объектов.

Итак, вернемся к поставленной задаче — предоставить клиенту единообразный интерфейс к листовым и составным элементам древовидной структуры. Очевидно, что для решения данной проблемы необходимо завести общий интерфейс, который будет описывать и элементарные и составные объекты. Более того, так как интерфейс описывает и составные объекты, он должен содержать контейнерные методы — add, remove, get для добавления, удаления и получения объекта из контейнера. Причем, данные методы должны быть параметризованны тем самым общим интерфейсом. Таким образом, автоматически появляется возможность добавлять в контейнер не только элементарные объекты, но и другие контейнеры.

Все объекты древовидной структуры (листовые и составные) должны реализовывать этот единообразный интерфейс, причем составные объекты переопределяют операции add, remove, get, а листовые их попросту игнорируют.

*«Паттерн Компоновщик позволяет создавать древовидные структуры, узлами которых являются как комбинации, так и отдельные объекты. В такой структуре одни и те же операции могут применяться и к комбинациям, и к отдельным объектам. Иначе говоря, во многих случаях различия между комбинациями и отдельными объектами игнорируются».*

ЗАКЛЮЧЕНИЕ

Безусловно, в сравнении с полностью самостоятельным проектированием, шаблоны обладают рядом преимуществ. Основная польза от использования шаблонов состоит в снижении сложности разработки за счёт готовых абстракций для решения целого класса проблем. Шаблон даёт решению своё имя, что облегчает коммуникацию между разработчиками, позволяя ссылаться на известные шаблоны. Таким образом, за счёт шаблонов производится унификация деталей решений: модулей, элементов проекта, — снижается количество ошибок. Однако с применением шаблонов могут быть связаны некоторые сложности. Во-первых, слепое следование некоторому выбранному шаблону может привести к усложнению программы. Во-вторых, у разработчика может возникнуть желание использовать некоторый шаблон в деле без особых оснований, что может привести к усложнению задачи. Таким образом знания шаблонов проектирования недостаточно для эффективного их использования. Эффективность паттерна в конкретной ситуации зависит от опыта и умений разработчика.

СПИСОК ЛИТЕРАТУРЫ

1. Эрик Фримен, Элиабет Фримен, «Паттерны Проектирования» - СПб.: Питер, 2011
2. Эрик Гамма, Ричард Хелм, Ральф Джонсон, «Приёмы объектно-ориентированного программирования» - СПб.: Питер, 2010