Учреждения образования

«Белорусский государственный университет информатики и радиоэлектроники»

Кафедра экономической информатики

**Лабораторная работа № 8**

«Использование паттернов проектирования и SVN

при разработке ПО»

по курсу

«Современные информационные технологии»

Выполнил ст. гр.172304

Котковец А.А.

Проверила:

Волошко Е.А.

Минск 2013

**Цель работы:** изучить понятие паттерна, виды паттернов, реализовать паттерн «Команда» для своего приложения.

**Краткие теоретические сведения**

Шаблон проектирования или паттерн (англ. design pattern) в разработке программного обеспечения — повторимая архитектурная конструкция, представляющая собой решение проблемы проектирования в рамках некоторого часто возникающего контекста.

Классификация паттернов:

– порождающие паттерны

– структурные паттерны

– паттерны поведения

Рассмотрим подробнее порождающие паттерны.

**1. Абстрактная фабрика(англ. Abstract factory).** Паттерн предоставляет интерфейс для создания семейств взаимосвязанных или взаимозависимых объектов, не специфицируя их конкретных классов.

Абстрактная фабрика – порождающий шаблон проектирования, позволяющий изменять поведение системы, варьируя создаваемыми объектами, при этом сохраняя интерфейсы. Он позволяет создавать целые группы взаимосвязанных объектов, которые, будучи созданными одной фабрикой, реализуют общее поведение. Шаблон реализуется созданием абстрактного класса Factory, который представляет собой интерфейс для создания компонентов системы (например, для оконного интерфейса он может создавать окна и кнопки). Затем пишутся наследники от него классы, реализующие этот интерфейс.

Плюсы: изолирует конкретные классы, упрощает замену семейств продуктов, гарантирует сочетаемость продуктов.

Минусы: сложно добавить поддержку нового вида продуктов.

**2. Строитель** – отделяет конструирование сложного объекта от его представления, так что в результате одного и того же процесса конструирования могут получаться разные представления.

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

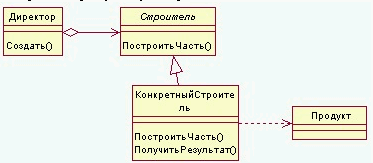


Рисунок 1 – Схема паттерна «Строитель»

3. **Отложенная (ленивая) инициализация** **(англ. Lazy initialization).** Приём в программировании, когда некоторая ресурсоёмкая операция (создание объекта, вычисление значения) выполняется непосредственно перед тем, как будет использован её результат. Таким образом, инициализация выполняется «по требованию», а не заблаговременно. Аналогичная идея находит применение в самых разных областях: например, компиляция «на лету» и логистическая концепция «Точно в срок».

Частный случай ленивой инициализации — создание объекта в момент обращения к нему — является одним из порождающих шаблонов проектирования. Как правило, он используется в сочетании с такими шаблонами как Фабричный метод, Одиночка и Заместитель.

**4. Прототип.** Задаёт виды создаваемых объектов с помощью экземпляра-прототипа и создаёт новые объекты путём копирования этого прототипа. Проще говоря, это паттерн создания объекта через клонирование другого объекта вместо создания через конструктор.

Паттерн используется чтобы избежать дополнительных усилий по созданию объекта стандартным путем (имеется в виду использование конструктора, т. к. в этом случае также будут вызваны конструкторы всей иерархии предков объекта), когда это непозволительно дорого для приложения; избежать наследования создателя объекта (object creator) в клиентском приложении, как это делает паттерн abstract factory.

Данный паттерн используется, когда система не должна зависеть от того, как в ней создаются, компонуются и представляются продукты.

**5. Одиночка.** Гарантирует, что у класса есть только один экземпляр, и предоставляет к нему глобальную точку доступа. Существенно то, что можно пользоваться именно экземпляром класса, так как при этом во многих случаях становится доступной более широкая функциональность. Например, к описанным компонентам класса можно обращаться через интерфейс, если такая возможность поддерживается языком.

Плюсом паттерна является контролируемый доступ к единственному экземпляру. Минусами – глобальные объекты могут быть вредны для объектного программирования, в некоторых случаях приводят к созданию немасштабируемого проекта; усложненность написания модульных тестов и следование TDD.

Рассмотрим подробнее структурные паттерны.

**1. Заместитель (англ. Proxy) –** структурный шаблон проектирования, который предоставляет объект, который контролирует доступ к другому объекту, перехватывая все вызовы (выполняет функцию контейнера).

Шаблон Proxy может применяться в случаях работы с сетевым соединением, с огромным объектом в памяти (или на диске) или с любым другим ресурсом, который сложно или тяжело копировать. Хорошо известный пример применения – объект, подсчитывающий число ссылок.

**2. Приспособленец (Flyweight)** используется для уменьшения затрат при работе с большим количеством мелких объектов. При проектировании приспособленца необходимо разделить его свойства на внешние и внутренние. Внутренние свойства всегда неизменны, тогда как внешние могут отличаться в зависимости от места и контекста применения и должны быть вынесены за пределы приспособленца.

Flyweight дополняет паттерн Factory таким образом, что Factory при обращении к ней клиента для создания нового объекта ищет уже созданный объект с такими же параметрами, что и у требуемого, и возвращает его клиенту. Если такого объекта нет, то фабрика создаст новый.

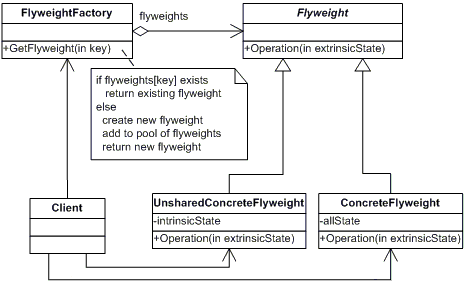


Рисунок 2 – Схема паттерна «Приспособленец»

3. **Мост.** Если требуется отделить абстракцию от реализации так, чтобы и то, и другое можно было изменять независимо, использование наследования недопустимо, так как реализация жестко привязывается к абстракции, что затрудняет независимую модификацию. Решением является помещение абстракции и реализации в отдельные иерархии классов.

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

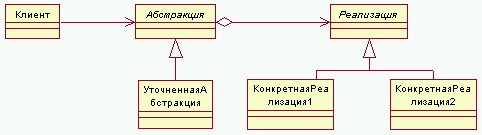


Рисунок 3 – Схема паттерна «Мост»

Рассмотрим подробнее поведенческие паттерны.

**1. Команда (англ. Command)** – поведенческий шаблон проектирования, используемый при объектно-ориентированном программировании, представляющий действие. Объект команды заключает в себе само действие и его параметры.

Паттерн создается c целью создания структуры, в которой класс-отправитель и класс-получатель не зависят друг от друга напрямую, а также для организации обратного вызова к классу, который включает в себя класс-отправитель.

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

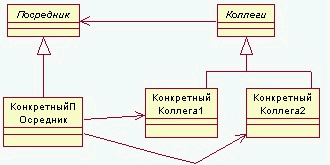


Рисунок 4 – Схема паттерна «Интерпретатор»

**3. Iterator (англ. Cursor)** — поведенческий шаблон проектирования, относится к паттернам поведения. Представляет собой объект, позволяющий получить последовательный доступ к элементам объекта-агрегата без использования описаний каждого из объектов, входящий в состав агрегации.

Например, такие элементы как дерево, связанный список, хэш-таблица и массив могут быть пролистаны (и модифицированы) с помощью паттерна (объекта) Итератор.

Перебор элементов выполняется объектом итератора, а не самой коллекцией. Это упрощает интерфейс и реализацию коллекции, а также способствует более логичному распределению обязанностей.

Особенностью полноценно реализованного итератора является то, что код, использующий итератор, может ничего не знать о типе итерируемого агрегата.

Конечно же, почти любой агрегат можно итерировать указателем void\*, но при этом не ясно, что является значением «конец агрегата»( так как для двусвязного списка это &ListHead, для массива это &array[size], для односвязного списка это NULL) и операция Next сильно зависит от типа агрегата.

Итераторы абстрагируют именно эти 2 проблемы, используя полиморфный Next (часто реализованный как operator++ в С++) и полиморфный aggregate.end(), возвращающий значение «конец агрегата».

Таким образом, появляется возможность работы с диапазонами итераторов, при отсутствии знания о типе итерируемого агрегата.

**Решение задачи**

****

Рисунок 5 – Схема паттерна «Команда»



Рисунок 5 – Диаграмма последовательности