

Министерство образования и науки, молодежи и спорта Украины

**Севастопольский национальный технический
университет**

МЕТОДИЧЕСКИЕ УКАЗАНИЯ

к лабораторным работам № 1 – 4

по дисциплине

**«Технология проектирования и разработки
программно-аппаратных систем»**

**для студентов дневной формы обучения
направления 6.050101 – «Компьютерные науки»**

Часть 1

**Севастополь
2012**

УДК 004.415.2

Методические указания к лабораторным работам по дисциплине **«Технология проектирования и разработки программно-аппаратных систем»** для студентов дневной формы обучения направления 6.050101 – «Компьютерные науки»/Сост. В.А.Строганов. – Севастополь: Изд-во СевНТУ, 2012. – Ч. 1 – 48 с.

Методические указания предназначены для оказания помощи студентам при выполнении лабораторных работ по дисциплине «Технология проектирования и разработки программно-аппаратных систем».

Методические указания составлены в соответствии с требованиями программы дисциплины «Технология проектирования и разработки программно-аппаратных систем» для студентов направления 6.050101 и утверждены на заседании кафедры Информационных систем, протокол № от « » 20 г.

Допущено учебно-методическим центром СевНТУ в качестве методических указаний.

Рецензент: Сергеев Г. Г., канд. техн. наук, доцент кафедры КиВТ.

Содержание

Общие положения.....	4
Лабораторная работа №1.....	11
Лабораторная работа №2.....	22
Лабораторная работа №3.....	33
Лабораторная работа №4.....	41
Библиографический список.....	46

Общие положения

Целью лабораторных работ является получения практических навыков применения паттернов при объектно-ориентированном проектировании программных продуктов.

Данный раздел лабораторного практикума представляет собой цикл из четырех лабораторных работ. Лабораторная работа №1 предназначена для повторения основных принципов объектно-ориентированного проектирования, а также для изучения основ языка моделирования UML. Оставшиеся три работы посвящены практическому применению паттернов проектирования.

В качестве лабораторной установки используется персональный компьютер и программное обеспечение — UML-редактор с возможностью синтеза программного кода на основе UML-диаграмм. Рекомендуется использовать StarUML, ArgoUML. Для компиляции и отладки программного кода рекомендуется использовать среду разработки Microsoft Visual Studio.

На выполнение каждой лабораторной работы предусмотрено 2 академических часа (кроме лабораторно работы №1, рассчитанной на 4 часа). Вариант задания студент получает у преподавателя.

Объем работы распределяется следующим образом: в ходе домашней самостоятельной подготовки студенты изучают необходимый теоретический материал (для работы № 1: основные принципы объектно-ориентированного проектирования, понятие класса и объекта, основные типы отношений между классами; для работ № 2 – 4: назначение соответствующего паттерна проектирования, его структуру, порядок применения). В ходе аудиторных занятий студенты выполняют построение UML-диаграмм, генерацию кода программы, компиляцию и проверку работоспособности программ.

Результаты лабораторных работ оформляются студентом в виде отчета, включающего название работы, цель работы, постановку задачи, результаты работы в виде UML-диаграмм и программного кода, а также выводы по результатам работы. Защита результатов работы проходит в форме устного опроса студента преподавателем.

1.1. Порядок работы в UML-редакторе StarUML

StarUML представляет собой UML-редактор с возможностью синтеза программного кода на основе UML-диаграмм. Ниже рассматривается общий порядок работы в StarUML.

1.1.1. Создание нового проекта

Для создания нового проекта следует воспользоваться диалогом создания проекта (пункт меню File->New Project by Approach либо сочетание клавиш Ctrl + I). В диалоговом окне (рисунок 1) рекомендуется выбрать вариант «Empty project» (пустой проект).

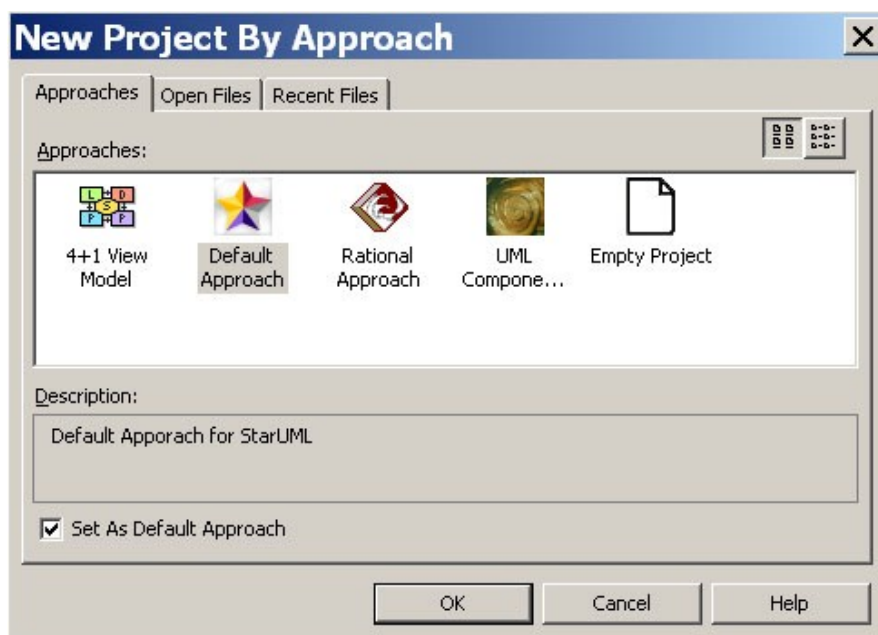


Рисунок 1 – Создание нового проекта

В результате будет создан новый проект, не содержащий никаких UML-диаграмм. Вид главного окна StarUML после создание нового проекта показан на рисунке 2.

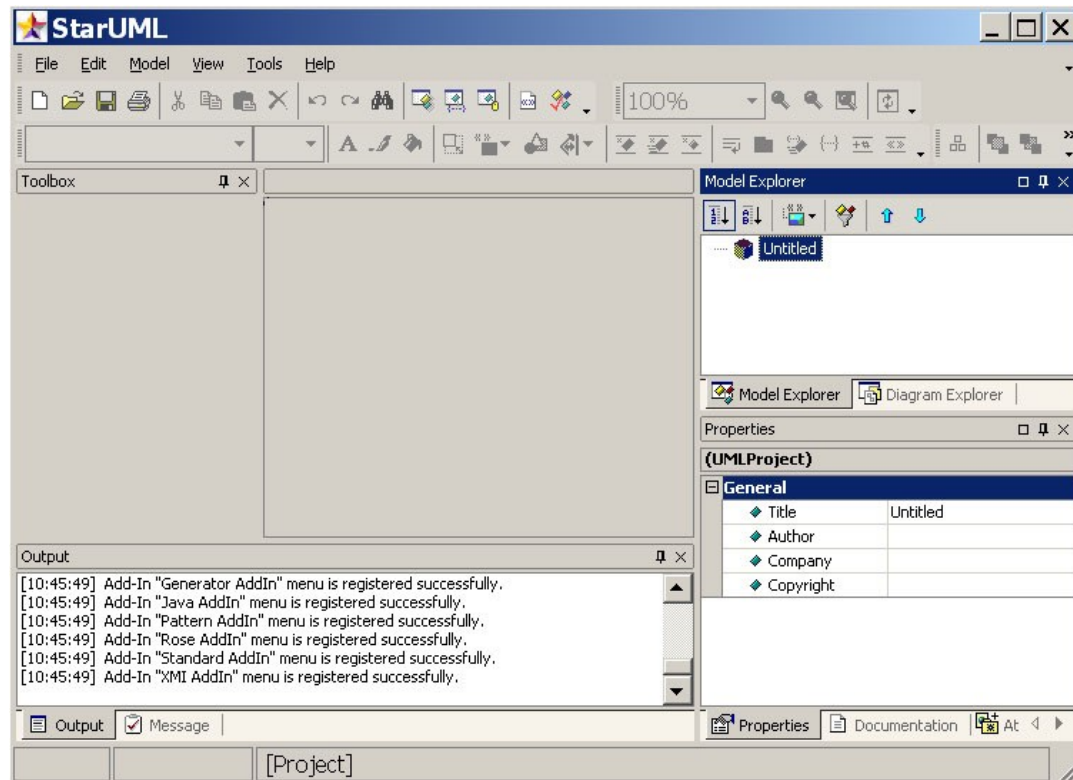


Рисунок 2 – Главное окно StarUML

В центральной части окна располагается основное поле (в данный момент – пустое), в котором будут отображаться UML-диаграммы.

В левой части окна располагается панель «Toolbox», которая будет содержать инструменты для создания элементов диаграмм (классы, интерфейсы, связи и т. п.). Содержимое панели «Toolbox» определяется типом редактируемой в данный момент UML-диаграммы.

В правой части главного окна располагаются панели «Model Explorer» (Обозреватель моделей) и «Properties» (Свойства). Панель «Model Explorer» содержит древовидную структуру текущего проекта, включающую модели, подсистемы, пакеты и входящие в их состав диаграммы различных типов (диаграммы классов, прецедентов и проч.).

1.1.2. Создание модели

Для добавления новой модели к текущему проекту необходимо воспользоваться контекстным меню панели «Model Explorer», как показано на рисунке 3.



Рисунок 3 – Создание новой модели

В результате новая модель будет создана и добавлена в дерево проекта (рисунок 4).

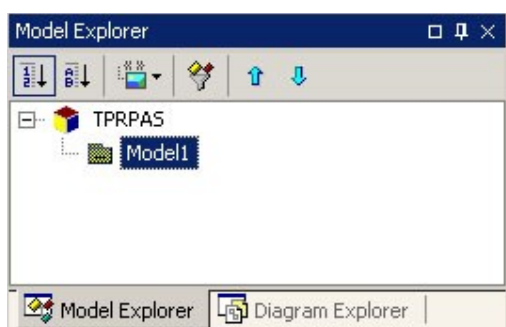


Рисунок 4 – Дерево проекта

1.1.3. Создание и редактирование диаграммы классов

После того, как создана модель, можно приступать к созданию диаграмм классов. Для добавления диаграммы классов следует воспользоваться

контекстным меню для модели в панели «Model Explorer», как показано на рисунке 5.

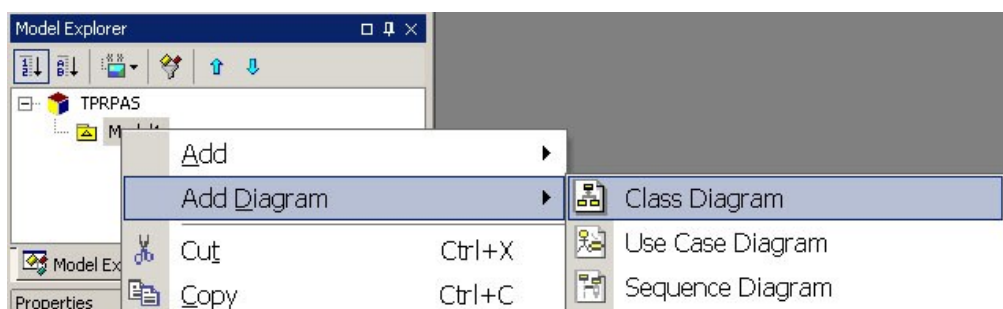


Рисунок 5 – Создание диаграммы классов

В результате вновь созданная диаграмма классов будет открыта в центральной части окна StarUML (рисунок 6).

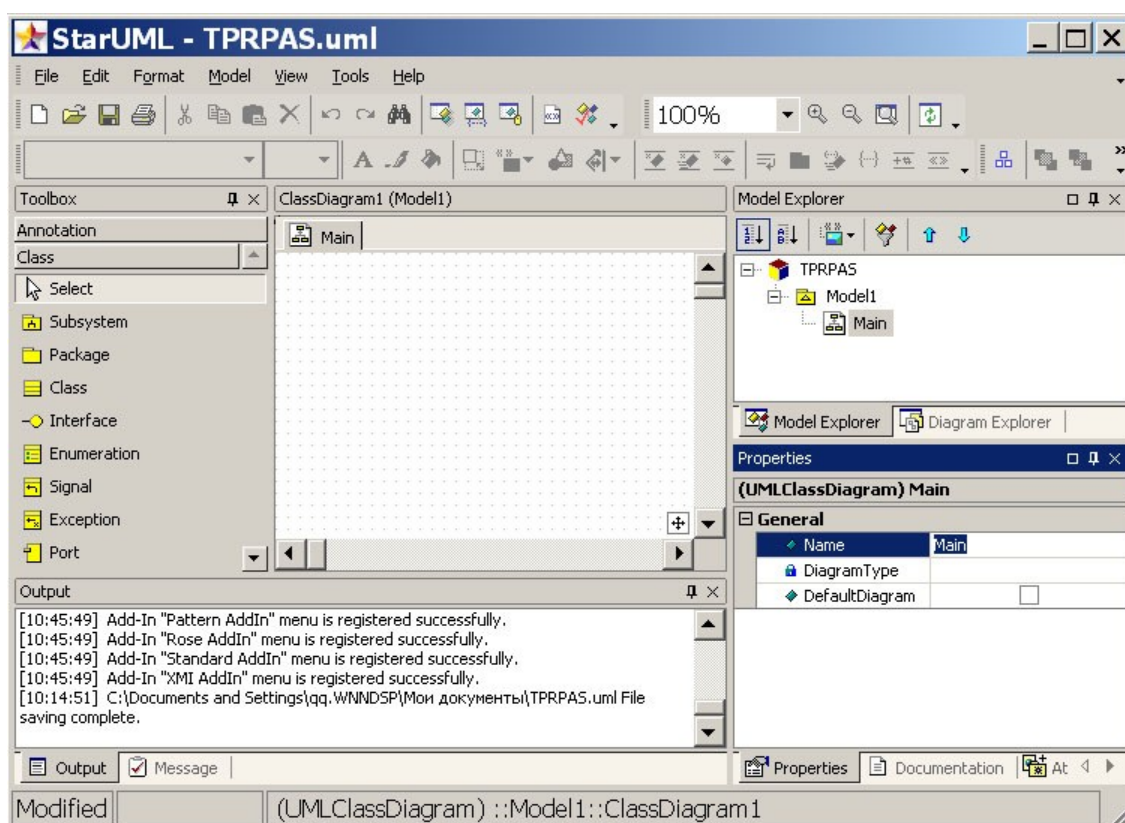


Рисунок 6 – Редактирование диаграммы классов

В панели инструментов («Toolbox») при этом появятся инструменты для создания элементов диаграммы классов (рисунок 7).

Для создания нового класса необходимо выбрать элемент «Class» на панели инструментов и нажать мышкой в центральном окне, в котором отображается диаграмма классов. В результате будет создан класс (рисунок 8). Для изменения имени класса можно воспользоваться панелью «Properties».

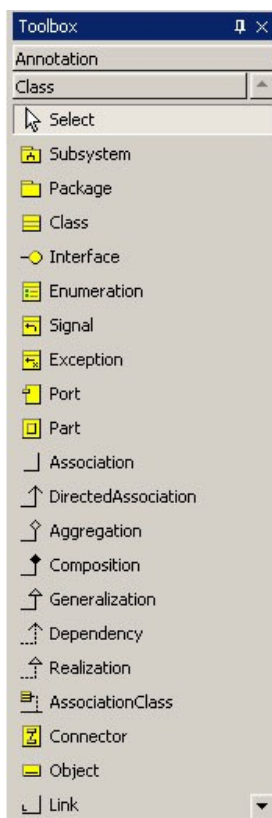


Рисунок 7 – Панель инструментов

Для добавления атрибутов и методов класса используется контекстное меню («Add / Attribute» и «Add / Operation» соответственно). Имена методов и атрибутов могут быть изменены с помощью панели «Properties».

Интерфейсы создаются аналогично классам.

Для добавления связей между классами и интерфейсами следует выбрать связь нужного типа (Association, Composition, Generalization) на панели инструментов «Toolbox», соединить нужные классы и при необходимости задать параметры связи на панели «Properties».

В качестве примера на рисунке 8 показана диаграмма классов, включающая интерфейс CanGo (может передвигаться) с методом go() и два класса (Car и Driver), реализующие этот интерфейс. Между классами Car и Driver установлено отношение ассоциации drives.

Для редактирования и удаления методов и атрибутов класса можно воспользоваться панелью «Properties» для класса (раздел General / Operations) или двойным нажатием левой кнопки мышки на нужном атрибуте (или методе) вызвать меню редактирования атрибуту (метода), как показано на рисунке 8.

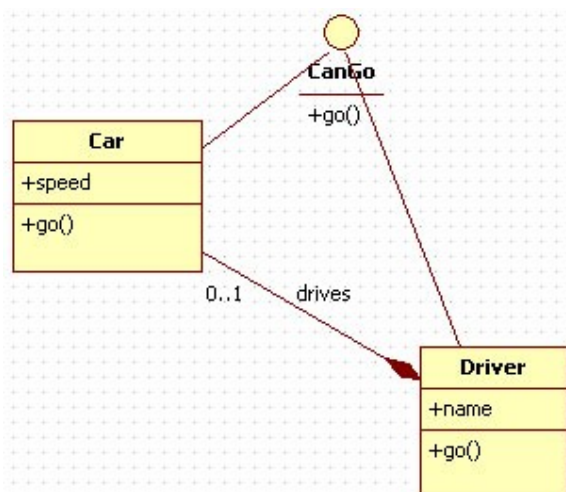


Рисунок 8 – Редактирование метода класса

1.1.4. Генерация кода программы

StarUML позволяет генерировать программный код на основе диаграммы классов. Это дает возможность выполнить проектирование системы с помощью визуального редактора диаграмм, после чего автоматически получить шаблон текста программы, содержащий объявление классов, интерфейсов, атрибутов и методов. В результате программист получает исходные тексты классов и реализует необходимую функциональность согласно спецификации методов, не отвлекаясь на общую структуру проекта.

Для генерации программного кода следует воспользоваться пунктом главного меню Tools / C# / Generate Code. Далее следуют диалоговые окна, в которых нужно выбрать модель и классы, исходный код которых будет генерироваться, папку, в которой он будет сохранен, способ сохранения (все классы в одном файле или отдельный файл для каждого класса), задать формат заголовка файла. Для всех этих параметров можно оставить значения по умолчанию.

Для диаграммы классов, показанной на рисунке 8, будет сгенерирован такой программный код:

```

//
//
//  Generated by StarUML(tm) C# Add-In
//
//  @ Project : TPRPAS
//  @ File Name : Model1.cs
//  @ Date : 27.03.2012
//  @ Author :
//
//

public class Car : CanGo{
    public object speed ;

```

```
        public void go(){
        }
    }

    public class Driver : CanGo{
        public object name ;
        public void go(){
        }
    }

    public interface CanGo {
        void go();
    }
```

Лабораторная работа №1

Исследование способов построения диаграмм классов в языке UML

1. Цель работы

Исследовать способы описания классов в языке UML, определения атрибутов и операций для класса. Изучить виды связей в диаграмме классов, правил описания и использования интерфейсов. Получить практические навыки построения UML-диаграмм классов.

2. Общие положения

2.1. Классы, атрибуты, операции и обязанности

Классом (Class) в ООП называется описание совокупности объектов с общими атрибутами, операциями, отношениями и семантикой. Графически класс изображается в виде прямоугольника.

Атрибут - это именованное свойство класса, включающее описание множества значений, которые могут принимать экземпляры этого свойства. Класс может иметь любое число атрибутов или не иметь их вовсе. Атрибут представляет некоторое свойство моделируемой сущности, общее для всех объектов данного класса. Например, у любой стены есть высота, ширина и толщина; при моделировании клиентов можно задавать фамилию, адрес, номер телефона и дату рождения. Таким образом, атрибут является абстракцией данных объекта или его состояния. В каждый момент времени любой атрибут объекта, принадлежащего данному классу, обладает вполне определенным значением. Атрибуты представлены в разделе, который расположен под именем класса; при этом указываются только их имена.

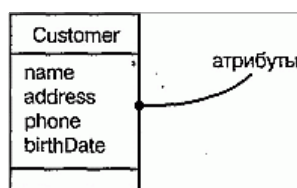


Рисунок 2.1 – Описание атрибутов класса

При описании атрибута можно явным образом указывать его класс и начальное значение, принимаемое по умолчанию, как продемонстрировано на рисунке 2.2.

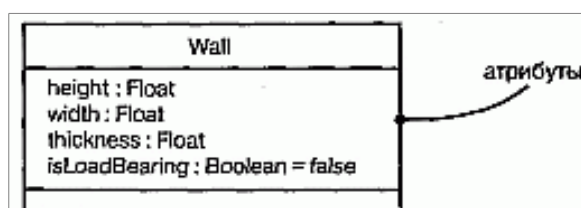


Рисунок 2.2 – Описание типов и начальных значений атрибутов

Операцией называется реализация услуги, которую можно запросить у любого объекта класса для воздействия на поведение. Иными словами, операция – это абстракция того, что позволено делать с объектом. У всех объектов класса имеется общий набор операций. Класс может содержать любое число операций или не содержать их вовсе. Часто (хотя не всегда) обращение к операции объекта изменяет его состояние или его данные. Операции класса изображаются в разделе, расположенном ниже раздела с атрибутами. При этом можно ограничиться только именами, как показано на рисунке 2.3.

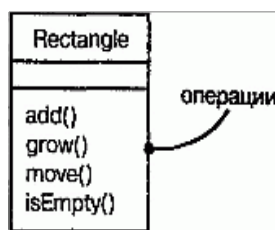


Рисунок 2.3 – Описание операций класса

Имя операции, как и имя класса, может быть произвольной текстовой строкой. На практике для именования операций используют короткий глагол или глагольный оборот, соответствующий определенному поведению объемлющего класса. Каждое слово в имени операции, кроме самого первого, обычно пишут с заглавной буквы, например *move* или *isEmpty*.

Операцию можно описать более подробно, указав ее сигнатуру, в которую входят имена и типы всех параметров, их значения, принятые по умолчанию, а применительно к функциям – тип возвращаемого значения, как показано на рисунке 2.4.

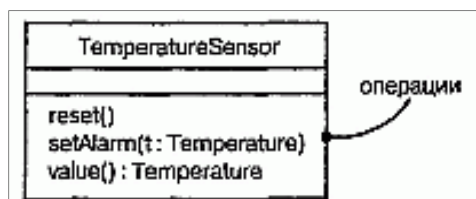


Рисунок 2.4 – Задание сигнатуры операций

Обязанности (Responsibilities) класса – это своего рода контракт, которому он должен подчиняться. Определяя класс, вы постулируете, что все его объекты имеют однотипное состояние и ведут себя одинаково. Выражаясь абстрактно, соответствующие атрибуты и операции как раз и являются теми свойствами, посредством которых выполняются обязанности класса. Например, класс *wall* (Стена) отвечает за информацию о высоте, ширине и толщине. Класс

TemperatureSensor (Датчик Температуры) отвечает за измерение температуры и подачу сигнала тревоги в случае превышения заданного уровня и т.д.

Моделирование классов лучше всего начинать с определения обязанностей сущностей, которые входят в словарь системы. На этом этапе особенно полезны будут такие методики, как применение CRC-карточек и анализ прецедентов. В принципе число обязанностей класса может быть произвольным, но на практике хорошо структурированный класс имеет по меньшей мере одну обязанность; с другой стороны, их не должно быть и слишком много. При уточнении модели обязанности класса преобразуются в совокупность атрибутов и операций, которые должны наилучшим образом обеспечить их выполнение.

Графически обязанности изображают в особом разделе в нижней части пиктограммы класса (см. рисунок 2.5). Их можно указать также в примечании.

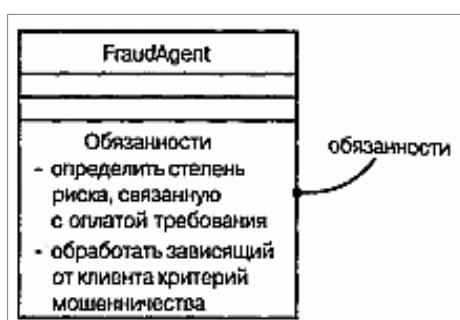


Рисунок 2.5 – Описание обязанностей класса

2.2. Отношения между классами

Кроме внутреннего устройства или структуры классов на соответствующей диаграмме указываются различные отношения между классами. При этом совокупность типов таких отношений фиксирована в языке UML и предопределена семантикой этих типов отношений. Базовыми отношениями или связями в языке UML являются:

- отношение зависимости (dependency relationship);
- отношение ассоциации (association relationship);
- отношение обобщения (generalization relationship).

Каждое из этих отношений имеет собственное графическое представление на диаграмме, которое отражает взаимосвязи между объектами соответствующих классов.

2.2.1. Отношение зависимости

Отношение зависимости в общем случае указывает некоторое семантическое отношение между двумя элементами модели или двумя множествами таких элементов, которое не является отношением ассоциации, обобщения или реализации. Отношение зависимости используется в такой

ситуации, когда некоторое изменение одного элемента модели может потребовать изменения другого зависимого от него элемента модели.

Отношение зависимости графически изображается пунктирной линией между соответствующими элементами со стрелкой на одном из ее концов ("– →" или "← –"). На диаграмме классов данное отношение связывает отдельные классы между собой, при этом стрелка направлена от класса-клиента зависимости к независимому классу или классу-источнику. На рисунке 2.6 изображены два класса: Класс_А и Класс_Б, при этом Класс_Б является источником некоторой зависимости, а Класс_А – клиентом этой зависимости.

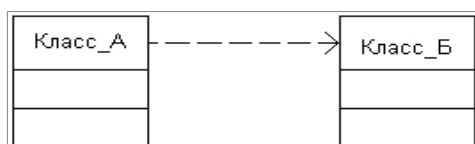


Рисунок 2.6 – Отношения зависимости на диаграмме классов

Стрелка может помечаться необязательным, но стандартным ключевым словом в кавычках и необязательным индивидуальным именем. Для отношения зависимости predeterminedены ключевые слова, которые обозначают некоторые специальные виды зависимостей. Эти ключевые слова (стереотипы) записываются в кавычках рядом со стрелкой, которая соответствует данной зависимости. Примеры стереотипов для отношения зависимости представлены ниже:

- "access" – служит для обозначения доступности открытых атрибутов и операций класса-источника для классов-клиентов;
- "bind" – класс-клиент может использовать некоторый шаблон для своей последующей параметризации;
- "derive" – атрибуты класса-клиента могут быть вычислены по атрибутам класса-источника;
- "import" – открытые атрибуты и операции класса-источника становятся частью класса-клиента, как если бы они были объявлены непосредственно в нем;
- "refine" – указывает, что класс-клиент служит уточнением класса-источника в силу причин исторического характера, когда появляется дополнительная информация в ходе работы над проектом.

2.2.2. Отношение ассоциации

Отношение ассоциации соответствует наличию некоторой связи между классами. Данное отношение обозначается сплошной линией с дополнительными специальными символами, которые характеризуют отдельные свойства конкретной ассоциации. В качестве дополнительных специальных символов могут использоваться имя ассоциации, а также имена и

кратность классов-ролей ассоциации. Имя ассоциации является необязательным элементом ее обозначения. Если оно задано, то записывается с заглавной (большой) буквы рядом с линией соответствующей ассоциации.

Наиболее простой случай данного отношения – бинарная ассоциация. Она связывает в точности два класса и, как исключение, может связывать класс с самим собой. Для бинарной ассоциации на диаграмме может быть указан порядок следования классов с использованием треугольника в форме стрелки рядом с именем данной ассоциации. Направление этой стрелки указывает на порядок классов, один из которых является первым (со стороны треугольника), а другой — вторым (со стороны вершины треугольника). Отсутствие данной стрелки рядом с именем ассоциации означает, что порядок следования классов в рассматриваемом отношении не определен.

В качестве простого примера отношения бинарной ассоциации рассмотрим отношение между двумя классами – классом "Компания" и классом "Сотрудник" (рисунок 2.7). Они связаны между собой бинарной ассоциацией "Работа", имя которой указано на рисунке рядом с линией ассоциации. Для данного отношения определен порядок следования классов, первым из которых является класс "Сотрудник", а вторым – класс "Компания". Отдельным примером или экземпляром данного отношения может являться пара значений (Петров И.И., "Рога-и-Копыта"). Это означает, что сотрудник Петров И.И. работает в компании "Рога-и-Копыта".



Рисунок 2.7 – Отношение бинарной ассоциации между классами

Ассоциации более высокой арности в общем случае называются N -арной ассоциацией (читается — "эн арная ассоциация"). Такая ассоциация связывает некоторым отношением 3 и более классов, при этом один класс может участвовать в ассоциации более чем один раз. Класс ассоциации имеет определенную роль в соответствующем отношении, что может быть явно указано на диаграмме. Каждый экземпляр N -арной ассоциации представляет собой N -арный кортеж значений объектов из соответствующих классов. Бинарная ассоциация является частным случаем N -арной ассоциации, когда значение $N=2$, и имеет свое собственное обозначение.

N -арная ассоциация графически обозначается ромбом, от которого ведут линии к символам классов данной ассоциации. В этом случае ромб соединяется с символами соответствующих классов сплошными линиями. Обычно линии проводятся от вершин ромба или от середины его сторон. Имя N -арной ассоциации записывается рядом с ромбом соответствующей ассоциации.

Порядок классов в N -арной ассоциации, в отличие от порядка множеств в отношении, на диаграмме не фиксируется. Некоторый класс может быть присоединен к ромбу пунктирной линией. Это означает, что данный класс обеспечивает поддержку свойств соответствующей N -арной ассоциации, а сама N -арная ассоциация имеет атрибуты, операции и/или ассоциации. Другими словами, такая ассоциация, в свою очередь, является классом с соответствующим обозначением в виде прямоугольника и является самостоятельным элементом языка UML – ассоциативным классом (Association Class). N -арная ассоциация не может содержать символ агрегации ни для какой из своих ролей.

2.2.3. Отношение обобщения

Отношение обобщения является обычным таксономическим отношением между более общим элементом (родителем или предком) и более частным или специальным элементом (дочерним или потомком). Данное отношение может использоваться для представления взаимосвязей между пакетами, классами, вариантами использования и другими элементами языка UML.

Применительно к диаграмме классов данное отношение описывает иерархическое строение классов и наследование их свойств и поведения. При этом предполагается, что класс-потомок обладает всеми свойствами и поведением класса-предка, а также имеет свои собственные свойства и поведение, которые отсутствуют у класса-предка. На диаграммах отношение обобщения обозначается сплошной линией с треугольной стрелкой на одном из концов (рисунок 2.8). Стрелка указывает на более общий класс (класс-предок или суперкласс), а ее отсутствие — на более специальный класс (класс-потомок или подкласс).



Рисунок 2.8 – Отношения обобщения на UML-диаграмме классов

2.3. Интерфейсы

При создании программного обеспечения важно строить системы с четким разделением задач, так чтобы при развитии системы изменения в одной ее части не затронули другие. Для достижения этой цели необходимо специфицировать стыковочные узлы системы, к которым присоединяются эти независимо изменяемые части.

В языке UML для моделирования стыковочных узлов в системе используются интерфейсы. *Интерфейс* (Interface) – это набор операций, используемых для специфицирования услуг, предоставляемых классом или компонентом. Интерфейс определяет границу между спецификацией того, что делает абстракция, и реализацией того, как она это делает.

Объявляя интерфейс, вы получаете возможность постулировать желаемое поведение абстракции, не зависящее от ее реализации. Клиенты будут спокойно полагаться на объявленный интерфейс, а вы в дальнейшем можете создать или купить любую его реализацию. Важно только, чтобы она выполняла обязательства, объявленные интерфейсом.

Типом (Type) называют стереотип класса, используемый для определения области значений объектов вместе с применимыми к ним операциями (но не методами).

Роль (Role) – это поведение сущности в данном контексте.

Интерфейсы используются для визуализации, специфицирования, конструирования и документирования стыковочных узлов между составными частями системы. Типы и роли обеспечивают механизм моделирования статического и динамического соответствия абстракции интерфейсу в конкретном контексте.

Хорошо структурированный интерфейс предусматривает четкую границу между внешним и внутренним представлениями абстракции, делая возможным ее понимание и работу с ней без необходимости углубляться в детали реализации. Кроме того, правильный подбор интерфейсов позволяет использовать имеющиеся стандартные библиотеки, компоненты и каркасы (frameworks) для их реализации, не программируя все это самостоятельно. Обнаружив впоследствии лучшую реализацию, вы сможете заменить старую, не беспокоя пользователей.

Почти все современные языки программирования поддерживают концепцию интерфейсов. Интерфейсы важны не только для отделения спецификации от реализации класса или компонента, но и для внешнего описания пакета или подсистемы в случае увеличения размеров системы.

Графически интерфейс изображается в виде кружочка; в развернутой форме его можно представить как стереотипный класс, чтобы раскрыть операции и другие свойства. Представление интерфейсов в языке UML показано на рисунке 2.9. Такая нотация позволяет отдельно визуализировать описание абстракции и ее реализацию.

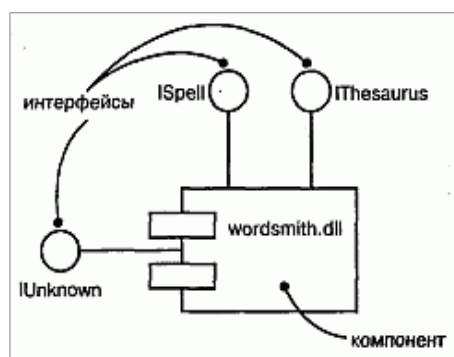


Рисунок 2.9 – Интерфейсы на UML-диаграмме

У любого интерфейса должно быть имя, отличающее его от остальных. Имя интерфейса представляет собой текстовую строку. Взятое само по себе, оно

называется простым. Составное имя образуется путем добавления в его начало имени пакета, в который входит данный интерфейс. Имя интерфейса должно быть уникальным внутри объемлющего пакета. При изображении интерфейса можно ограничиться только его именем, как показано на рисунке 2.10.

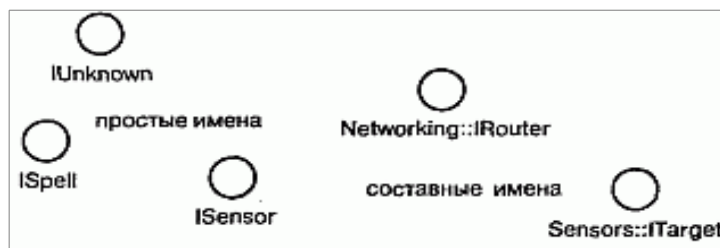


Рисунок 2.10 – Простые и составные имена интерфейсов

Как уже отмечалось выше, интерфейс - это именованный набор операций для специфирования сервиса, предоставляемого классом или компонентом. В отличие от классов и типов, интерфейсы не описывают структуры (поэтому не могут содержать атрибуты) и реализации (а значит, не содержат реализующих операции методов). Как и класс, интерфейс может включать любое число операций, которые могут быть дополнены свойствами видимости, параллельности, стереотипами, помеченными значениями и ограничениями.

Изображая интерфейс в виде кружочка, вы по определению подавляете показ этих операций. Однако, если это необходимо для понимания модели, можно изобразить интерфейс как стереотипный класс, перечислив операции в соответствующем разделе. При этом можно показывать только имя операции или же предоставить развернутое описание с сигнатурой и другими свойствами, как показано на рисунке 2.11.

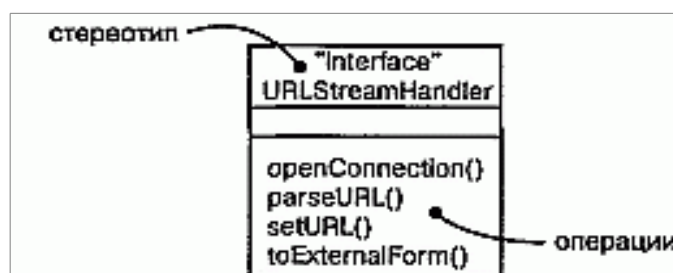


Рисунок 2.11 – Развернутое описание интерфейса

Подобно классам, интерфейсы могут принимать участие в отношениях обобщения, ассоциации и зависимости. Кроме того, для них определено еще и отношение реализации, представляющее собой семантическое отношение между двумя классификаторами, один из которых определяет ряд обязательств (контракт), а другой обеспечивает их выполнение.

Интерфейс специфицирует контракт класса или компонента, но не накладывает никаких ограничений на реализацию. Класс или компонент могут реализовывать несколько интерфейсов. При этом они обязуются выполнить все

свои контракты, то есть должны содержать методы, корректно реализующие объявленные интерфейсом операции. Точно так же класс или компонент может зависеть от нескольких интерфейсов. При этом он ожидает, что объявленные контракты будут выполнены каким-то набором реализующих их компонентов. Именно поэтому говорят, что интерфейс представляет собой стыковочный узел в системе. Он определяет условия контракта, после чего обе стороны – клиент и поставщик – могут действовать независимо друг от друга, полностью полагаясь на взаимные обязательства.

Как видно из рисунка 2.12, связь интерфейса с реализующим его элементом можно графически представить двумя способами.

Во-первых, существует простая форма: отношения между интерфейсом и его реализацией изображаются кружочком с одной стороны класса или компонента. Этот способ удобен, когда требуется показать стыковочные узлы системы. Однако операции и сигналы, предоставляемые интерфейсом, таким способом визуализировать нельзя.

Во-вторых, можно воспользоваться расширенной формой, когда интерфейс представляют в виде стереотипного класса и связывают отношением реализации с классификатором или компонентом. При этом разрешается показать его операции и другие свойства. В UML отношение реализации изображают в виде пунктирной линии с большой незакрашенной стрелкой, направленной в сторону интерфейса. В этой нотации суммируются обобщение и зависимость.

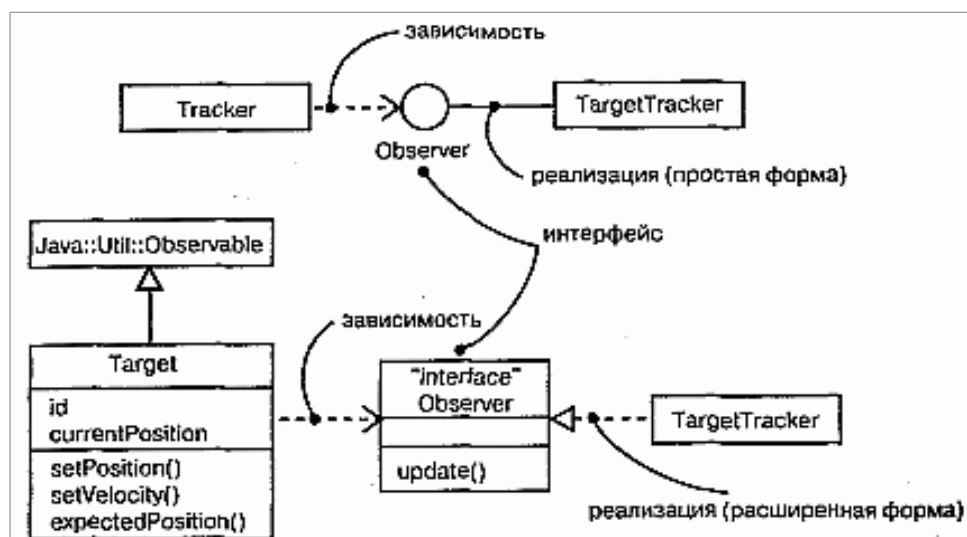


Рисунок 2.12 – Способы визуализация интерфейса

В обоих случаях класс или компонент, использующий интерфейс, соединяют с ним отношением зависимости.

Интерфейсы напоминают абстрактные классы, в частности потому, что ни у тех, ни у других нет непосредственных экземпляров. Но все же между ними имеется достаточно различий, чтобы считать их отдельными элементами моделирования. У абстрактного класса могут быть атрибуты, а у интерфейса – нет. Кроме того, интерфейсы пересекают границы модели. Например, один и

тот же интерфейс может быть реализован как классом (логическая абстракция), так и компонентом (физическая абстракция, материализующая класс).

Моделируя интерфейс на языке UML, следует помнить, что он должен описывать некоторый стыковочный узел системы, отделяя спецификацию от реализации. Хорошо структурированный интерфейс характеризуется следующими свойствами:

- одновременно обладает простотой и завершенностью, то есть содержит все операции, необходимые и достаточные для специфицирования конкретного сервиса;

- понятен, то есть содержит информацию, достаточную для его применения, и не требует для этого обращения к его реализации и примерам использования;

- содержит информацию, достаточную для понимания пользователем основных свойств, но не перегружен сведениями, касающимися деталей всех операций.

3. Порядок выполнения работы

3.1. Выбрать моделируемую предметную область из приведенного ниже списка:

1. Обслуживание клиента в банке;
2. Обучение студента в университете;
3. Строительство дома;
4. Система для учета успеваемости студентов университета;
5. Учет работников предприятия (отдел кадров);
6. Фирма по продаже компьютерной техники;
7. Система для корпоративной библиотеки;
8. Офис юридической компании;
9. Редакция журнала или газеты;
10. Работа страховой компании;
11. Работа банкомата.

3.2. Для выбранной предметной области выделить основные классы, проанализировать обязанности классов. На основе анализа обязанностей определить атрибуты и операции классов.

3.3. Определить отношения между классами (необходимо использовать три типа отношений: ассоциации, зависимости и обобщения).

3.3. Разработать необходимые интерфейсы и связать их с разработанными выше классами.

3.4. Построить UML-диаграмму, содержащую разработанные выше классы, интерфейсы и связи между ними. Каждый элемент диаграммы должен сопровождаться комментариями.

3.5. На основе UML-диаграммы выполнить синтез программного кода, скомпилировать программу и продемонстрировать ее работоспособность.

4. Содержание отчета

- 4.1. Цель работы.
- 4.2. Постановка задачи с указанием моделируемой предметной области.
- 4.3. Словесное описание основных классов предметной области и порядка их взаимодействия.
- 4.4. UML-диаграмма классов с комментариями.
- 4.5. Текст программы.
- 4.6. Выводы по работе.

5. Контрольные вопросы

- 5.1. Что называется классом?
- 5.2. Какие элементы включает класс?
- 5.3. Какие типы отношений могут существовать между классами?
- 5.4. Понятие зависимости между классами и основные типы зависимостей.
- 5.5. Что понимается под отношением ассоциации, какие существуют типы ассоциаций?
- 5.6. Что означает отношение обобщения между классами?
- 5.7. Что понимается такое интерфейс, для чего он предназначен?

Лабораторная работа №2

Исследование способов применения структурных паттернов проектирования

1. Цель работы

Исследовать возможность использования структурных паттернов проектирования. Получить практические навыки применения структурных паттернов при объектно-ориентированном проектировании.

2. Основные положения

2.1. Паттерны проектирования

Паттерны (шаблоны) проектирования [2] представляют собой инструмент, который позволяет документировать опыт разработки объектно-ориентированных программ. В основе использования паттернов лежит следующая идея: при проектировании каждый проект не разрабатывается с нуля, а используется опыт предыдущих проектов. То есть паттерны проектирования упрощают повторное использование удачных проектных и архитектурных решений. Представление прошедших проверку временем методик в виде паттернов проектирования облегчает доступ к ним со стороны разработчиков новых систем.

Во многих объектно-ориентированных системах встречаются повторяющиеся паттерны, состоящие из классов и взаимодействующих объектов. С их помощью решаются конкретные задачи проектирования, в результате чего объектно-ориентированный дизайн становится более гибким, элегантным, и им можно воспользоваться повторно. Проектировщик, знакомый с паттернами, может сразу же применять их к решению новой задачи, не пытаясь каждый раз изобретать велосипед.

Описание каждого паттерна принято разбивать на следующие разделы:

- **Название и классификация паттерна.** Название паттерна должно четко отражать его назначение. Классификация паттернов проводится в соответствии со схемой, которая будет рассмотрена ниже.

- **Назначение.** Лаконичный ответ на следующие вопросы: каковы функции паттерна, его обоснование и назначение, какую конкретную задачу проектирования можно решить с его помощью.

- **Известен также под именем.** Другие распространенные названия паттерна, если таковые имеются.

- **Мотивация.** Сценарий, иллюстрирующий задачу проектирования и то, как она решается данной структурой класса или объекта. Благодаря мотивации можно лучше понять последующее, более абстрактное описание паттерна.

- **Применимость.** Описание ситуаций, в которых можно применять данный паттерн. Примеры проектирования, которые можно улучшить с его помощью. Распознавание таких ситуаций.

- **Структура.** Графическое представление классов в паттерне с использованием нотации, основанной на методике Object Modeling Technique (OMT). Могут использоваться также диаграммы взаимодействий для иллюстрации последовательностей запросов и отношений между объектами.

- **Участники.** Классы или объекты, задействованные в данном паттерне проектирования, и их функции.

- **Отношения.** Взаимодействие участников для выполнения своих функций.

- **Результаты.** Насколько паттерн удовлетворяет поставленным требованиям? Результаты применения, компромиссы, на которые приходится идти. Какие аспекты поведения системы можно независимо изменять, используя данный паттерн?

- **Реализация.** Сложности и так называемые подводные камни при реализации паттерна. Советы и рекомендуемые приемы. Есть ли у данного паттерна зависимость от языка программирования?

- **Пример кода программы.** Фрагмент программного кода, иллюстрирующий вероятную реализацию на языках C++ или Smalltalk.

- **Известные применения.** Возможности применения паттерна в реальных системах. Даются, по меньшей мере, два примера из различных областей.

- **Родственные паттерны.** Связь других паттернов проектирования с данным. Важные различия. Использование данного паттерна в сочетании с другими.

2.2. Порядок использования паттернов проектирования

1. Прочитать описание паттерна (см. ниже), чтобы получить о нем общее представление. Особое внимание обратить на разделы «Применимость» и «Результаты». Убедиться, что выбранный паттерн действительно подходит для решения данной задачи.

2. Изучить разделы описания паттерна «Структура», «Участники» и «Отношения». Детально проанализировать назначение упоминаемых в паттерне классов и объектов и то, как они взаимодействуют друг с другом.

3. Посмотреть на раздел «Пример кода», где приведен конкретный пример использования паттерна в программе. Изучение программного кода поможет понять, как нужно реализовывать паттерн.

4. Выбрать для участников паттерна подходящие имена. Имена участников паттерна обычно слишком абстрактны, чтобы употреблять их непосредственно в коде. Тем не менее, бывает полезно включить имя участника как имя в программе. Это помогает сделать паттерн более очевидным при реализации. Например, при использовании паттерна *Стратегия* в алгоритме размещения текста, классы могли бы называться SimpleLayoutStrategy или TexLayoutStrategy.

5. Определить классы. Объявить их интерфейсы, установить отношения наследования и определить переменные экземпляра, которыми будут представлены данные объекты и ссылки на другие объекты. Выявить

имеющиеся в вашем приложении классы, на которые паттерн оказывает влияние, и соответствующим образом модифицировать их.

6. Определить имена операций, встречающихся в паттерне. Здесь, как и в предыдущем случае, имена обычно зависят от приложения. При этом следует руководствоваться теми функциями и взаимодействиями, которые ассоциированы с каждой операцией. Кроме того, нужно быть последовательным при выборе имен. Например, для обозначения **фабричного метода** можно было бы всюду использовать префикс `Create-`.

7. Реализовать операции, которые выполняют обязанности и отвечают за отношения, определенные в паттерне. Советы о том, как это лучше сделать, можно найти в разделе «Реализация». Поможет и «Пример кода».

2.3. Структурные паттерны

В структурных паттернах рассматривается вопрос о том, как из классов и объектов образуются более крупные структуры. Структурные паттерны уровня класса используют наследование для составления композиций из интерфейсов и реализаций. Простой пример – использование множественного наследования для объединения нескольких классов в один. В результате получается класс, обладающий свойствами всех своих родителей. Особенно полезен этот паттерн, когда нужно организовать совместную работу нескольких независимо разработанных библиотек.

Другой пример паттерна уровня класса – *Адаптер*. В общем случае Адаптер делает интерфейс одного класса (адаптируемого) совместимым с интерфейсом другого, обеспечивая тем самым унифицированную абстракцию разнородных интерфейсов. Это достигается за счет закрытого наследования адаптируемому классу. После этого адаптер выражает свой интерфейс в терминах операций адаптируемого класса.

Вместо композиции интерфейсов или реализаций структурные паттерны уровня объекта komponуют объекты для получения новой функциональности. Дополнительная гибкость в этом случае связана с возможностью изменить композицию объектов во время выполнения, что недопустимо для статической композиции классов.

Примером структурного паттерна уровня объектов является *Компоновщик*. Он описывает построение иерархии классов для двух видов объектов: примитивных и составных. Последние позволяют создавать произвольно сложные структуры из примитивных и других составных объектов.

В паттерне *Заместитель* объект берет на себя функции другого объекта. У Заместителя есть много применений. Он может действовать как локальный представитель объекта, находящегося в удаленном адресном пространстве. Или представлять большой объект, загружаемый по требованию. Или ограничивать доступ к критически важному объекту. Заместитель вводит дополнительный косвенный уровень доступа к отдельным свойствам объекта. Поэтому он может ограничивать, расширять или изменять эти свойства.

Паттерн **Приспособленец** определяет структуру для совместного использования объектов. Владельцы разделяют объекты, по меньшей мере, по двум причинам: для достижения эффективности и непротиворечивости. Приспособленец акцентирует внимание на эффективности использования памяти. В приложениях, в которых участвует очень много объектов, должны снижаться накладные расходы на хранение. Значительной экономии можно добиться за счет разделения объектов вместо их дублирования. Но объект может быть разделяемым, только если его состояние не зависит от контекста. У объектов-приспособленцев такой зависимости нет. Любая дополнительная информация передается им по мере необходимости. В отсутствие контекстных зависимостей объекты-приспособленцы могут легко разделяться.

Если паттерн Приспособленец дает способ работы с большим числом мелких объектов, то **Фасад** показывает, как один объект может представлять целую подсистему. Фасад представляет набор объектов и выполняет свои функции, перенаправляя сообщения объектам, которых он представляет. Паттерн **Мост** отделяет абстракцию объекта от его реализации, так что их можно изменять независимо.

Паттерн **Декоратор** описывает динамическое добавление объектам новых обязанностей. Это структурный паттерн, который рекурсивно компонует объекты с целью реализации заранее неизвестного числа дополнительных функций. Например, объект-декоратор, содержащий некоторый элемент пользовательского интерфейса, может добавить к нему оформление в виде рамки или тени либо новую функциональность, например возможность прокрутки или изменения масштаба. Два разных оформления прибавляются путем простого вкладывания одного декоратора в другой. Для достижения этой цели каждый объект-декоратор должен соблюдать интерфейс своего компонента и перенаправлять ему сообщения. Свои функции (скажем, рисование рамки вокруг компонента) декоратор может выполнять как до, так и после перенаправления сообщения.

2.3.1. Паттерн «Адаптер»

Название и классификация паттерна

Адаптер – паттерн, структурирующий классы и объекты.

Назначение

Преобразует интерфейс одного класса в интерфейс другого, который ожидают клиенты. *Адаптер* обеспечивает совместную работу классов с несовместимыми интерфейсами, которая без него была бы невозможна.

Известен также под именем

Wrapper (обертка).

Мотивация

Иногда класс из инструментальной библиотеки, спроектированный для повторного использования, не удастся использовать только потому, что его интерфейс не соответствует тому, который нужен конкретному приложению.

Рассмотрим, например, графический редактор, благодаря которому пользователи могут рисовать на экране графические элементы (линии, многоугольники, текст и т.д.) и организовывать их в виде картинок и диаграмм. Основной абстракцией графического редактора является графический объект, который имеет изменяемую форму и изображает сам себя. Интерфейс графических объектов определен абстрактным классом `Shape`. Редактор определяет подкласс класса `Shape` для каждого вида графических объектов: `LineShape` для прямых, `PolygonShape` для многоугольников и т.д.

Классы для элементарных геометрических фигур, например `LineShape` и `PolygonShape`, реализовать сравнительно просто, поскольку заложенные в них возможности рисования и редактирования крайне ограничены. Но подкласс `TextShape`, умеющий отображать и редактировать текст, уже значительно сложнее, поскольку даже для простейших операций редактирования текста нужно нетривиальным образом обновлять экран и управлять буферами. В то же время, возможно, существует уже готовая библиотека для разработки пользовательских интерфейсов, которая предоставляет развитый класс `TextView`, позволяющий отображать и редактировать текст. В идеале мы хотели бы повторно использовать `TextView` для реализации `TextShape`, но библиотека разрабатывалась без учета классов `Shape`, поэтому заставить объекты `TextView` и `Shape` работать совместно не удастся.

Так каким же образом существующие и независимо разработанные классы вроде `TextView` могут работать в приложении, которое спроектировано под другой, несовместимый интерфейс? Можно было бы так изменить интерфейс класса `TextView`, чтобы он соответствовал интерфейсу `Shape`, только для этого нужен исходный код. Но даже если он доступен, то вряд ли разумно изменять `TextView`; библиотека не должна приспособливаться к интерфейсам каждого конкретного приложения.

Вместо этого мы могли бы определить класс `TextShape` так, что он будет адаптировать интерфейс `TextView` к интерфейсу `Shape`. Это допустимо сделать двумя способами:

- наследуя интерфейс от `Shape`, а реализацию от `TextView`;

- включив экземпляр `TextView` в `TextShape` и реализовав `TextShape` в терминах интерфейса `TextView`. Два данных подхода соответствуют вариантам паттерна **Адаптер** в его классовой и объектной ипостасях. Класс `TextShape` мы будем называть адаптером.

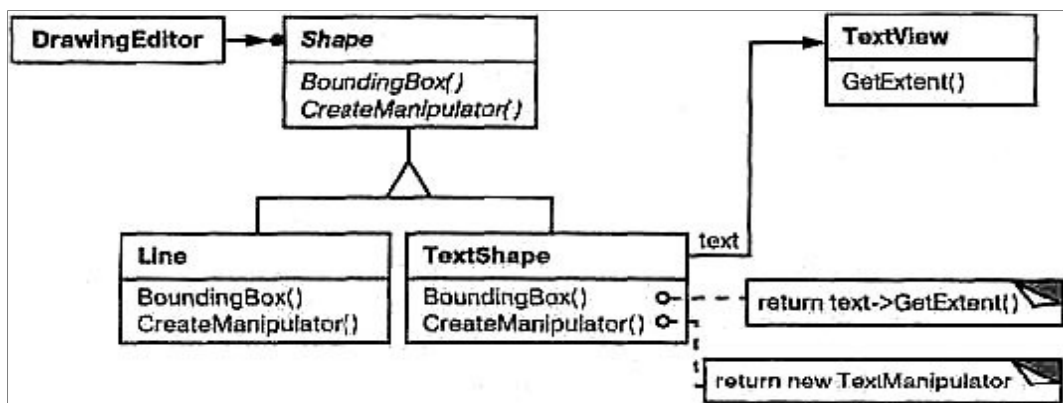


Рисунок 2.1 – Пример использования паттерна Адаптер

На рисунке 2.1 показан адаптер объекта. Видно, как запрос `BoundingBox`, объявленный в классе `Shape`, преобразуется в запрос `GetExtent`, определенный в классе `TextView`. Поскольку класс `TextShape` адаптирует `TextView` к интерфейсу `Shape`, графический редактор может воспользоваться классом `TextView`, хотя тот и имеет несовместимый интерфейс.

Часто адаптер отвечает за функциональность, которую не может предоставить адаптируемый класс. На диаграмме показано, как адаптер выполняет такого рода функции. У пользователя должна быть возможность перемещать любой объект класса `Shape` в другое место, но в классе `TextView` такая операция не предусмотрена. `TextShape` может добавить недостающую функциональность, самостоятельно реализовав операцию `CreateManipulator` класса `Shape`, которая возвращает экземпляр подходящего подкласса `Manipulator`.

`Manipulator` – это абстрактный класс объектов, которым известно, как анимировать `Shape` в ответ на такие действия пользователя, как перетаскивание фигуры в другое место. У класса `Manipulator` имеются подклассы для различных фигур. Например, `TextManipulator` – подкласс для `TextShape`. Возвращая экземпляр `TextManipulator`, объект класса `TextShape` добавляет новую функциональность, которой в классе `TextView` нет, а классу `Shape` требуется.

Применимость

Паттерн *Адаптер* следует применять, когда:

- необходимо использовать существующий класс, но его интерфейс не соответствует вашим потребностям;
- нужно создать повторно используемый класс, который должен взаимодействовать с заранее неизвестными или не связанными с ним классами, имеющими несовместимые интерфейсы;
- (только для адаптера объектов!) нужно использовать несколько существующих подклассов, но непрактично адаптировать их интерфейсы путем

порождения новых подклассов от каждого. В этом случае адаптер объектов может приспособлять интерфейс их общего родительского класса.

Структура

Адаптер класса использует множественное наследование для адаптации одного интерфейса к другому. Структура адаптера класса показана на рисунке 2.2.

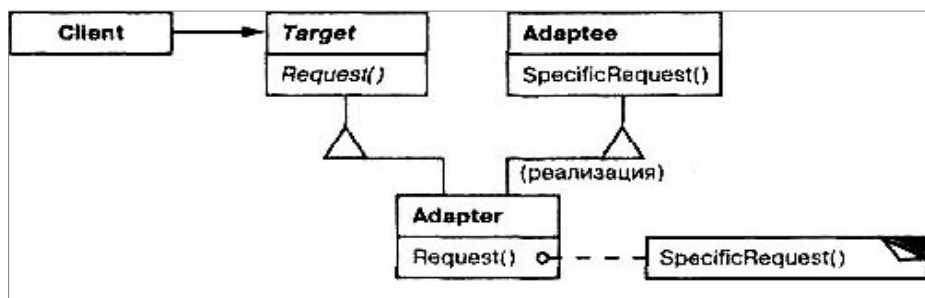


Рисунок 2.2 – Структура адаптера класса

Адаптер объекта применяет композицию объектов. Структура адаптера уровня объектов показана на рисунке 2.3.

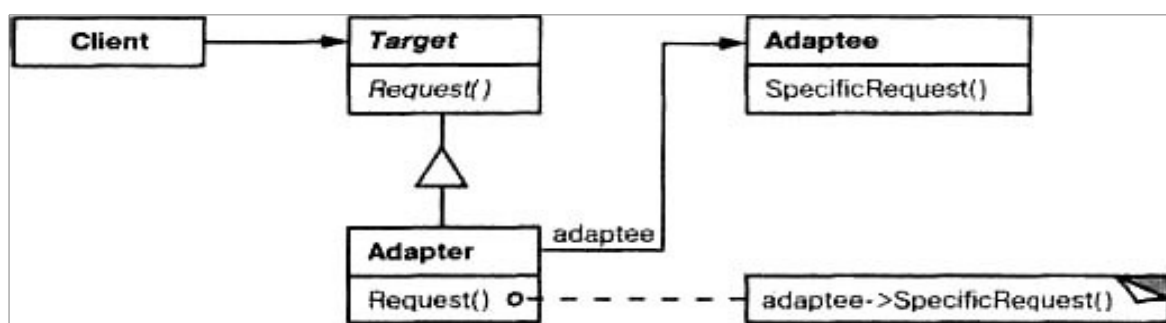


Рисунок 2.3 – Структура адаптера объекта

Участники

- Target (Shape) – целевой. Определяет зависящий от предметной области интерфейс, которым пользуется Client;
- Client (DrawingEditor) – клиент: вступает во взаимоотношения с объектами, удовлетворяющими интерфейсу Target;
- Adaptee (Textview) – адаптируемый: определяет существующий интерфейс, который нуждается в адаптации;
- Adapter (Text Shape) – адаптер: адаптирует интерфейс Adaptee к интерфейсу Target.

Отношения

Клиенты вызывают операции экземпляра адаптера Adapter. В свою очередь адаптер вызывает операции адаптируемого объекта или класса Adaptee, который и выполняет запрос.

Результаты

Результаты применения адаптеров объектов и классов различны. Адаптер класса:

- адаптирует Adaptee к Target, перепоручая действия конкретному классу Adaptee. Поэтому данный паттерн не будет работать, если мы захотим одновременно адаптировать класс и его подклассы;

- позволяет адаптеру Adapter заместить некоторые операции адаптируемого класса Adaptee, так как Adapter есть не что иное, как подкласс Adaptee;

- вводит только один новый объект. Чтобы добраться до адаптируемого класса, не нужно никакого дополнительного обращения по указателю.

Адаптер объектов:

- позволяет одному адаптеру Adapter работать со многим адаптируемыми объектами Adaptee, то есть с самим Adaptee и его подклассами (если таковые имеются). Адаптер может добавить новую функциональность сразу всем адаптируемым объектам;

- затрудняет замещение операций класса Adaptee. Для этого потребуется породить от Adaptee подкласс и заставить Adapter ссылаться на этот подкласс, а не на сам Adaptee.

Пример кода программы

Ниже приводится краткий обзор реализации адаптеров класса и объекта для примера, рассмотренного в разделе «Мотивация».

```
class Shape {
public:
    Shape();
    virtual void BoundingBox(Points bottomLeft, Point& topRight)
const;
    virtual Manipulator* CreateManipulator() const;
};
class TextView {
public:
    TextView();
    void GetOrigin(Coord& x, Coords y) const;
    void GetExtent(Coord& width, Coords height) const;
    virtual bool IsEmpty() const;
};
```

В классе Shape предполагается, что ограничивающий фигуру прямоугольник определяется двумя противоположными углами. Напротив, в классе TextView он характеризуется начальной точкой, высотой и шириной. В

классе Shape определена также операция CreateManipulator() для создания объекта-манипулятора класса Manipulator, который знает, как анимировать фигуру в ответ на действия пользователя. В TextView эквивалентной операции нет. Класс TextShape является адаптером между двумя этими интерфейсами.

Для адаптации интерфейса адаптер класса использует множественное наследование. Принцип адаптера класса состоит в наследовании интерфейса по одной ветви и реализации – по другой. В С++ интерфейс обычно наследуется открыто, а реализация – закрыто. Мы будем придерживаться этого соглашения при определении адаптера TextShape:

```
class TextShape : public Shape, private TextView {
public:
    TextShape();
    virtual void BoundingBox(Point& bottomLeft, Points topRight)
const;
    virtual bool IsEmpty() const;
    virtual Manipulator* CreateManipulator() const;
};
```

Операция BoundingBox преобразует интерфейс TextView к интерфейсу Shape:

```
void TextShape::BoundingBox (Points bottomLeft, Point& topRight)
const
{
    Coord bottom, left, width, height;
    GetOrigin(bottom, left);
    GetExtent(width, height);
    bottomLeft = Point(bottom, left);
    topRight = Point(bottom + height, left + width);
}
```

На примере операции IsEmpty демонстрируется прямая переадресация запросов, общих для обоих классов:

```
bool TextShape::IsEmpty () const
{
    return TextView::IsEmpty();
}
```

Наконец, определим операцию CreateManipulator (отсутствующую в классе TextView) с нуля. Предположим, класс TextManipulator, который поддерживает манипуляции с TextShape, уже реализован:

```
Manipulator* TextShape::CreateManipulator () const
{
    return new TextManipulator(this);
}
```

```
}
```

Адаптер объектов применяет композицию объектов для объединения классов с разными интерфейсами. При таком подходе адаптер TextShape содержит указатель на TextView:

```
class TextShape : public Shape {
public:
    TextShape(TextView*);
    virtual void BoundingBox(Point& bottomLeft, Points topRight)
const;
    virtual bool IsEmpty() const;
    virtual Manipulator* CreateManipulator() const;
private:
    TextView* _text;
};
```

Объект TextShape должен инициализировать указатель на экземпляр TextView. Делается это в конструкторе. Кроме того, он должен вызывать операции объекта TextView всякий раз, как вызываются его собственные операции.

В этом примере мы предположим, что клиент создает объект TextView и передает его конструктору класса TextShape:

```
TextShape::TextShape (TextView* t)
{
    _text = t;
}
void TextShape::BoundingBox (Points bottomLeft, Point& topRight)
const
{
    Coord bottom, left, width, height;
    _text->GetOrigin(bottom, left);
    _text->GetExtent(width, height);
    bottomLeft = Point(bottom, left);
    topRight = Point(bottom + height, left + width);
}

bool TextShape::IsEmpty () const
{
    return _text->IsEmpty();
}
```

Реализация CreateManipulator не зависит от версии адаптера класса, поскольку реализована с нуля и не использует повторно никакой функциональности TextView:

```
Manipulator* TextShape::CreateManipulator () const
{
```

```

    return new TextManipulator(this);
}

```

Сравним этот код с кодом адаптера класса. Для написания адаптера объекта нужно потратить чуть больше усилий, но зато он оказывается более гибким. Например, вариант адаптера объекта `TextShape` будет прекрасно работать и с подклассами `TextView`: клиент просто передает экземпляр подкласса `TextView` конструктору `TextShape`.

3. Порядок выполнения работы

3.1. Ознакомиться с основными преимуществами объектно-ориентированного проектирования на основе паттернов, изучить порядок проектирования с использованием паттернов. Изучить назначение и структуру паттерна **Адаптер** (выполнить в ходе самостоятельной подготовки).

3.2. Применительно к предметной области, заданной в лабораторной работе № 1, проанализировать возможность использования паттерна **Адаптер**. Для этого на диаграмме классов, разработанной в ходе работы № 1, найти класс-клиент и адаптируемый класс, функциональностью которого должен воспользоваться клиент.

3.3. Выполнить перепроектирование диаграммы классов, разработанной в лабораторной работе №1, используя паттерн **Адаптер**.

3.4. Сравнить полученные диаграммы классов, сделать выводы и целесообразности использования паттерна **Адаптер** для данной системы.

3.5. На основе UML-диаграммы выполнить синтез программного кода, скомпилировать программу и продемонстрировать ее работоспособность.

4. Содержание отчета

4.1. Цель работы.

4.2. Постановка задачи с указанием моделируемой предметной области.

4.3. Словесное описание мотивации применения паттерна **Адаптер** при проектировании данной системы.

4.4. UML-диаграмма классов с комментариями.

4.5. Текст программы.

4.6. Выводы по работе.

5. Контрольные вопросы

5.1. Что понимается под паттерном объектно-ориентированного проектирования?

5.2. Из каких разделов состоит описание паттерна?

5.3. Каков общий порядок применения паттернов проектирования?

5.4. Для чего предназначены структурные паттерны проектирования?

5.5. Какие задачи решает паттерн «Адаптер»?

5.6. Какие классы входят в состав паттерна «Адаптер», каковы их обязанности?

Лабораторная работа №3

Исследование способов применения поведенческих паттернов проектирования

1. Цель работы

Исследовать возможность использования поведенческих паттернов проектирования. Получить практические навыки применения паттернов поведения при объектно-ориентированном проектировании.

2. Основные положения

2.1. Паттерны поведения

Паттерны поведения связаны с алгоритмами и распределением обязанностей между объектами. Речь в них идет не только о самих объектах и классах, но и о типичных способах взаимодействия. Паттерны поведения характеризуют сложный поток управления, который трудно проследить во время выполнения программы. Внимание акцентировано не на потоке управления как таковом, а на связях между объектами.

В паттернах поведения уровня класса используется наследование – чтобы распределить поведение между разными классами. Из них более простым и широко распространенным является **шаблонный метод**, который представляет собой абстрактное определение алгоритма. Алгоритм здесь определяется пошагово. На каждом шаге вызывается либо примитивная, либо абстрактная операция. Алгоритм усложняется за счет подклассов, где определены абстрактные операции.

Другой паттерн поведения уровня класса – **интерпретатор**, который представляет грамматику языка в виде иерархии классов и реализует интерпретатор как последовательность операций над экземплярами этих классов.

В паттернах поведения уровня объектов используется не наследование, а композиция. Некоторые из них описывают, как с помощью кооперации множество равноправных объектов справляется с задачей, которая ни одному из них не под силу. Важно здесь то, как объекты получают информацию о существовании друг друга. Объекты-коллеги могут хранить ссылки друг на друга, но это увеличит степень связанности системы. При максимальной степени связанности каждому объекту пришлось бы иметь информацию обо всех остальных. Эту проблему решает паттерн **посредник**. Посредник, находящийся между объектами-коллегами, обеспечивает косвенность ссылок, необходимую для разрывания лишних связей.

Паттерн **цепочка обязанностей** позволяет и дальше уменьшать степень связанности. Он дает возможность посылать запросы объекту не напрямую, а по цепочке «объектов-кандидатов». Запрос может выполнить любой «кандидат», если это допустимо в текущем состоянии выполнения программы.

Число кандидатов заранее не определено, а подбирать участников можно во время выполнения.

Паттерн **наблюдатель** определяет и отвечает за зависимости между объектами. Классический пример наблюдателя встречается в схеме модель/вид/контроллер языка Smalltalk, где все виды модели уведомляются о любых изменениях ее состояния.

Прочие паттерны поведения связаны с инкапсуляцией поведения в объекте и делегированием ему запросов. Паттерн **стратегия** инкапсулирует алгоритм объекта, упрощая его спецификацию и замену. Паттерн **команда** инкапсулирует запрос в виде объекта, который можно передавать как параметр, хранить в списке истории или использовать как-то иначе. Паттерн **состояние** инкапсулирует состояние объекта таким образом, что при изменении состояния объект может изменять поведение. Паттерн **посетитель** инкапсулирует поведение, которое в противном случае пришлось бы распределять между классами, а паттерн **итератор** абстрагирует способ доступа и обхода объектов из некоторого агрегата.

2.2. Паттерн «Цепочка обязанностей»

Назначение

Позволяет избежать привязки отправителя запроса к его получателю, давая шанс обработать запрос нескольким объектам. Связывает объекты-получатели в цепочку и передает запрос вдоль этой цепочки, пока его не обработают.

Мотивация

Рассмотрим контекстно-зависимую оперативную справку в графическом интерфейсе пользователя, который может получить дополнительную информацию по любой части интерфейса, просто щелкнув на ней мышью. Содержание справки зависит от того, какая часть интерфейса и в каком контексте выбрана. Например, справка по кнопке в диалоговом окне может отличаться от справки по аналогичной кнопке в главном окне приложения. Если для некоторой части интерфейса справки нет, то система должна показать информацию о ближайшем контексте, в котором она находится, например о диалоговом окне в целом.

Поэтому естественно было бы организовать справочную информацию от более конкретных разделов к более общим. Кроме того, ясно, что запрос на получение справки обрабатывается одним из нескольких объектов пользовательского интерфейса, каким именно – зависит от контекста и имеющейся в наличии информации.

Проблема в том, что объект, инициирующий запрос (например, кнопка), не располагает информацией о том, какой объект в конечном итоге предоставит справку. Необходим какой-то способ отделить кнопку-инициатор запроса от объектов, владеющих справочной информацией. Как этого добиться, показывает паттерн **цепочка обязанностей**.

Идея заключается в том, чтобы разорвать связь между отправителями и получателями, дав возможность обработать запрос нескольким объектам. Запрос перемещается по цепочке объектов, пока один из них не обработает его. Первый объект в цепочке получает запрос и либо обрабатывает его сам, либо направляет следующему кандидату в цепочке, который ведет себя точно так же. У объекта, отправившего запрос, отсутствует информация об обработчике. Говорят, что у запроса есть анонимный получатель (implicit receiver).

Предположим, что пользователь запрашивает справку по кнопке Print (печать). Она находится в диалоговом окне PrintDialog, содержащем информацию об объекте приложения, которому принадлежит (диаграмма объектов показана на рисунке 2.1).

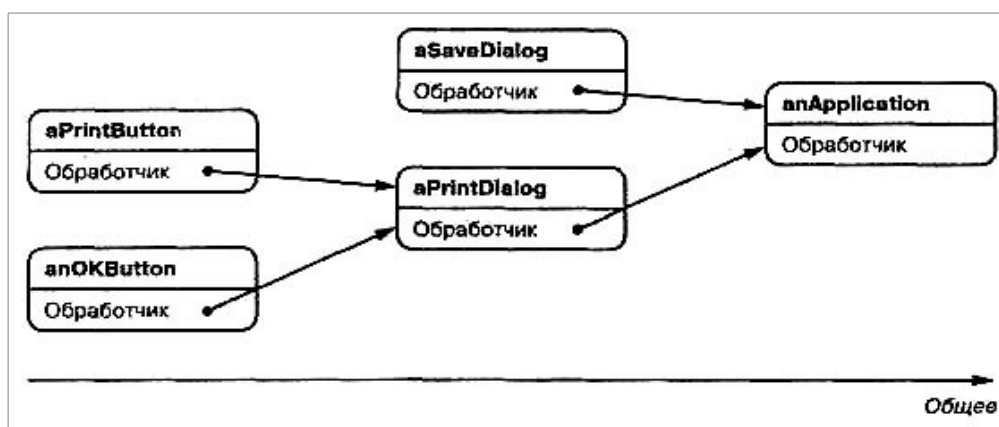


Рисунок 2.1 – Диаграмма объектов

В данном случае ни кнопка aPrintButton, ни окно aPrintDialog не обрабатывают запрос, он достигает объекта anApplication, который может его обработать или игнорировать. У клиента, инициировавшего запрос, нет прямой ссылки на объект, который его в конце концов обработает.

Диаграмма классов показана на рисунке 2.2. Чтобы отправить запрос по цепочке и гарантировать анонимность получателя, все объекты в цепочке имеют единый интерфейс для обработки запросов и для доступа к своему преемнику (следующему объекту в цепочке). Например, в системе оперативной справки можно было бы определить класс HelpHandler (предок классов всех объектов-кандидатов или подмешиваемый класс (mixin class)) с операцией handleHelp. Тогда классы, которые будут обрабатывать запрос, смогут его передать своему родителю.

Для обработки запросов на получение справки классы Button, Dialog и Application пользуются операциями HelpHandler. По умолчанию операция handleHelp просто перенаправляет запрос своему преемнику. В подклассах эта операция замещается, так что при благоприятных обстоятельствах может выдаваться справочная информация. В противном случае запрос отправляется дальше посредством реализации по умолчанию.

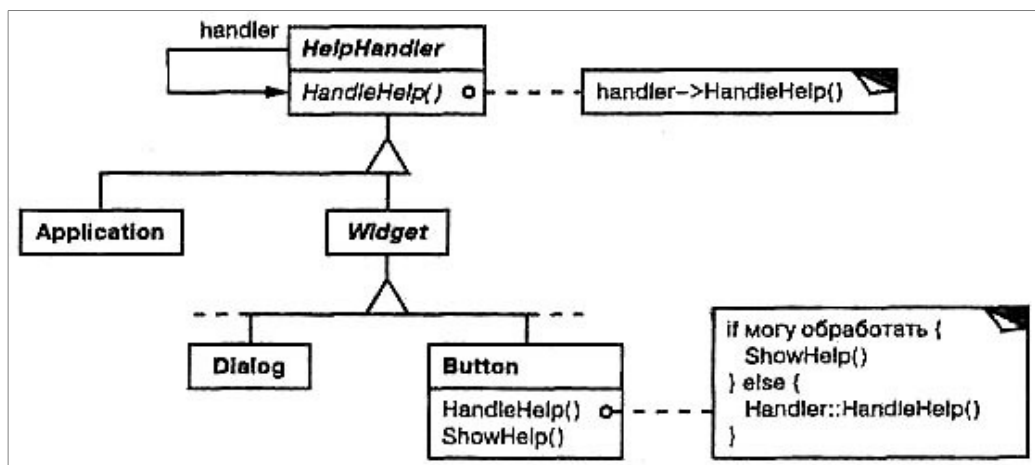


Рисунок 2.2 – Диаграмма классов

Применимость

Паттерн «Цепочка обязанностей» целесообразно применять, когда:

- есть более одного объекта, способного обработать запрос, причем настоящий обработчик заранее неизвестен и должен быть найден автоматически;
- нужно отправить запрос одному из нескольких объектов, не указывая явно, какому именно;
- набор объектов, способных обработать запрос, должен задаваться динамически.

Структура

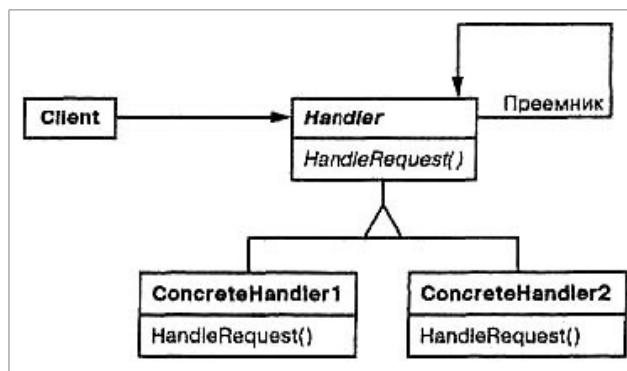


Рисунок 2.3 – Диаграмма классов паттерна «Цепочка обязанностей»

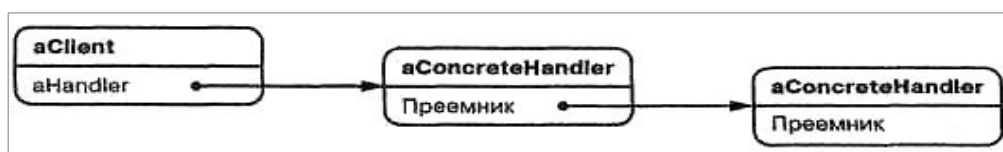


Рисунок 2.4 – Типичная структура объектов

Участники

Handler (Handler) – обработчик:

- определяет интерфейс для обработки запросов;
- реализует связь с преемником (необязательно);

ConcreteHandler (PrintButton, PrintDialog) – конкретный обработчик:

- обрабатывает запрос, за который отвечает;
- имеет доступ к своему преемнику;
- если ConcreteHandler способен обработать запрос, то так и делает, если

не может, то направляет его своему преемнику;

Client – клиент:

- отправляет запрос некоторому объекту ConcreteHandler в цепочке.

Отношения

Когда клиент инициирует запрос, он продвигается по цепочке, пока некоторый объект ConcreteHandler не возьмет на себя ответственность за его обработку.

Результаты

Паттерн цепочка обязанностей имеет следующие достоинства и недостатки:

- ослабление связанности. Этот паттерн освобождает объект от необходимости «знать», кто конкретно обработает его запрос. Отправителю и получателю ничего неизвестно друг о друге, а включенному в цепочку объекту – о структуре цепочки.

Таким образом, цепочка обязанностей помогает упростить взаимосвязи между объектами. Вместо того чтобы хранить ссылки на все объекты, которые могут стать получателями запроса, объект должен располагать информацией лишь о своем ближайшем преемнике;

- дополнительная гибкость при распределении обязанностей между объектами. Цепочка обязанностей позволяет повысить гибкость распределения обязанностей между объектами. Добавить или изменить обязанности по обработке запроса можно, включив в цепочку новых участников или изменив ее каким-то другим образом. Этот подход можно сочетать со статическим порождением подклассов для создания специализированных обработчиков;

- получение запроса не гарантировано. Поскольку у запроса нет явного получателя, то нет и гарантий, что он вообще будет обработан: он может достичь конца цепочки и пропасть. Необработанным запрос может оказаться и в случае неправильной конфигурации цепочки.

Реализация

При рассмотрении цепочки обязанностей следует обратить внимание на следующие моменты:

- 1) реализация цепочки преемников. Есть два способа реализовать такую цепочку:

- определить новые связи (обычно это делается в классе Handler, но можно

и в ConcreteHandler);

- использовать существующие связи.

2) соединение преемников. Если готовых ссылок, пригодных для определения цепочки, нет, то их придется ввести. В таком случае класс Handler не только определяет интерфейс запросов, но еще и хранит ссылку на преемника.

Следовательно у обработчика появляется возможность определить реализацию операции HandleRequest по умолчанию – перенаправление запроса преемнику (если таковой существует). Если подкласс ConcreteHandler не заинтересован в запросе, то ему и не надо замещать эту операцию, поскольку по умолчанию запрос как раз и отправляется дальше.

Определение базового класса HelpHandler, в котором хранится указатель на преемника, имеет вид:

```
class HelpHandler {
public:
    HelpHandler(HelpHandler* s) : _successor(s) { }
    virtual void HandleHelp();
private:
    HelpHandler* _successor;
};
void HelpHandler::HandleHelp () {
    if (_successor) {
        _successor->HandleHelp();
    }
}
```

3) представление запросов. Представлять запросы можно по-разному. В простейшей форме, например в случае класса HandleHelp, запрос жестко кодируется как вызов некоторой операции. Это удобно и безопасно, но переадресовывать тогда можно только фиксированный набор запросов, определенных в классе Handler.

Альтернатива – использовать одну функцию-обработчик, которой передается код запроса (скажем, целое число или строка). Так можно поддерживать заранее неизвестное число запросов. Единственное требование состоит в том, что отправитель и получатель должны договориться о способе кодирования запроса.

Это более гибкий подход, но при реализации нужно использовать условные операторы для раздачи запросов по их коду. Кроме того, не существует безопасного с точки зрения типов способа передачи параметров, поэтому упаковывать и распаковывать их приходится вручную. Очевидно, что это не так безопасно, как прямой вызов операции.

Чтобы решить проблему передачи параметров, допустимо использовать отдельные объекты-запросы, в которых инкапсулированы параметры запроса.

Класс Request может представлять некоторые запросы явно, а их новые типы описываются в подклассах. Подкласс может определить другие параметры. Обработчик должен иметь информацию о типе запроса (какой именно подкласс Request используется), чтобы разобрать эти параметры.

Для идентификации запроса в классе Request можно определить функцию доступа, которая возвращает идентификатор класса. Вместо этого получатель мог бы воспользоваться информацией о типе, доступной во время выполнения, если язык программирования поддерживает такую возможность.

Приведем пример функции диспетчеризации, в которой используются объекты для идентификации запросов. Операция GetKind(), указанная в базовом классе Request, определяет вид запроса:

```
void Handler::HandleRequest (Request* theRequest) {
    switch (theRequest->GetKind()) {
        case Help:
            // привести аргумент к подходящему типу
            HandleHelp((HelpRequest*) theRequest);
            break;
        case Print:
            HandlePrint((PrintRequest*) theRequest);
            // ...
            break;
        default:
            // ...
            break;
    }
}
```

Подклассы могут расширить схему диспетчеризации, переопределив операцию HandleRequest. Подкласс обрабатывает лишь те запросы, в которых заинтересован, а остальные отправляет родительскому классу. В этом случае подкласс именно расширяет, а не замещает операцию HandleRequest.

Подкласс ExtendedHandler расширяет операцию HandleRequest(), определенную в классе Handler, следующим образом:

```
class ExtendedHandler : public Handler {
public:
    virtual void HandleRequest(Request* theRequest);
    // . . .
};

void ExtendedHandler::HandleRequest (Request* theRequest) {
    switch (theRequest->GetKind()) {
        case Preview:
            // обработать запрос Preview
            break;
        default:
            // дать классу Handler возможность обработать
            // остальные запросы
            Handler::HandleRequest(theRequest);
    }
}
```

3. Порядок выполнения работы

3.1. Изучить назначение и структуру паттерна **Цепочка обязанностей** (выполнить в ходе самостоятельной подготовки).

3.2. Применительно к предметной области, заданной в лабораторной работе №1, проанализировать возможность использования паттерна **Цепочка обязанностей**. Для этого на диаграмме классов, разработанной в ходе работы № 1, найти класс-клиент, запрос от которого необходимо передавать по цепочке объектов, и классы-получатели запросов, объекты которых целесообразно объединять в цепочку.

3.3. Выполнить перепроектирование диаграммы классов, разработанной в лабораторной работе №1, используя паттерн **Цепочка обязанностей**.

3.4. Сравнить полученные диаграммы классов, сделать выводы и целесообразности использования паттернов проектирования для данной системы.

3.5. На основе UML-диаграммы выполнить синтез программного кода, скомпилировать программу и продемонстрировать ее работоспособность.

4. Содержание отчета

4.1. Цель работы.

4.2. Постановка задачи с указанием моделируемой предметной области.

4.3. Словесное описание мотивации применения паттерна **Цепочка обязанностей** при проектировании данной системы.

4.4. UML-диаграмма классов с комментариями.

4.5. Текст программы.

4.6. Выводы по работе.

5. Контрольные вопросы

5.1. Для чего предназначены поведенческие паттерны проектирования?

5.5. Какие задачи решает паттерн «Цепочка обязанностей»?

5.6. Какие классы входят в состав паттерна «Цепочка обязанностей», каковы их обязанности?

Лабораторная работа №4

Исследование способов применения порождающих паттернов проектирования

1. Цель работы

Исследовать возможность использования порождающих паттернов проектирования. Получить практические навыки применения порождающих паттернов при объектно-ориентированном проектировании.

2. Основные положения

2.1. Порождающие паттерны

Порождающие паттерны проектирования абстрагируют процесс инстанцирования. Они помогут сделать систему независимой от способа создания, композиции и представления объектов. Паттерн, порождающий классы, использует наследование, чтобы варьировать инстанцируемый класс, а паттерн, порождающий объекты, делегирует инстанцирование другому объекту.

Эти паттерны оказываются важны, когда система больше зависит от композиции объектов, чем от наследования классов. Получается так, что основной упор делается не на жестком кодировании фиксированного набора поведений, а на определении небольшого набора фундаментальных поведений, с помощью композиции которых можно получать любое число более сложных. Таким образом, для создания объектов с конкретным поведением требуется нечто большее, чем простое инстанцирование класса.

Для порождающих паттернов актуальны две темы. Во-первых, эти паттерны инкапсулируют знания о конкретных классах, которые применяются в системе.

Во-вторых, скрывают детали того, как эти классы создаются и стыкуются. Единственная информация об объектах, известная системе, – это их интерфейсы, определенные с помощью абстрактных классов. Следовательно, порождающие паттерны обеспечивают большую гибкость при решении вопроса о том, что создается, кто это создает, как и когда. Можно собрать систему из «готовых» объектов с самой различной структурой и функциональностью статически (на этапе компиляции) или динамически (во время выполнения).

2.2. Паттерн «Абстрактная фабрика»

Назначение

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

Мотивация

Рассмотрим инструментальную программу для создания пользовательского интерфейса, поддерживающего разные стандарты внешнего облика, например Motif и Presentation Manager. Внешний облик определяет визуальное представление и поведение элементов пользовательского интерфейса («виджетов») – полос прокрутки, окон и кнопок. Чтобы приложение можно было перенести на другой стандарт, в нем не должен быть жестко закодирован внешний облик виджетов.

Если инстанцирование классов для конкретного внешнего облика разбросано по всему приложению, то изменить облик впоследствии будет нелегко.

Можно решить эту проблему, определив абстрактный класс `WidgetFactory` (рисунок 2.1), в котором объявлен интерфейс для создания всех основных видов виджетов. Есть также абстрактные классы для каждого отдельного вида и конкретные подклассы, реализующие виджеты с определенным внешним обликом. В интерфейсе `WidgetFactory` имеется операция, возвращающая новый объект-виджет для каждого абстрактного класса виджетов. Клиенты вызывают эти операции для получения экземпляров виджетов, но при этом ничего не знают о том, какие именно классы используют. Стало быть, клиенты остаются независимыми от выбранного стандарта внешнего облика.

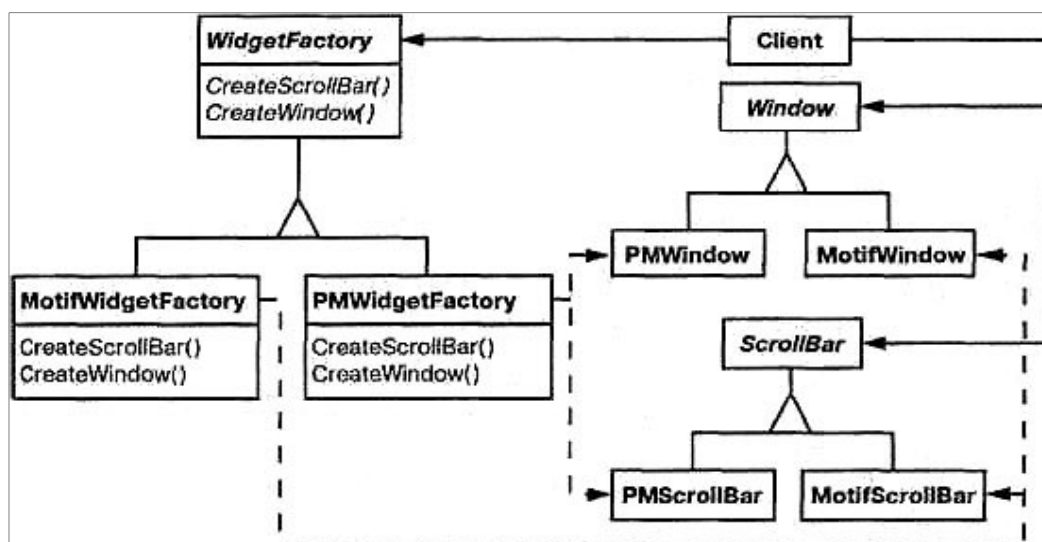


Рисунок 2.1 – Диаграмма классов

Для каждого стандарта внешнего облика существует определенный подкласс `WidgetFactory`. Каждый такой подкласс реализует операции, необходимые для создания соответствующего стандарту виджета. Например, операция `CreateScrollBar` в классе `MotifWidgetFactory` инстанцирует и возвращает полосу прокрутки в стандарте Motif, тогда как соответствующая операция в классе `PMWidgetFactory` возвращает полосу прокрутки в стандарте Presentation Manager. Клиенты создают виджеты, пользуясь исключительно интерфейсом `WidgetFactory`, и им ничего не известно о классах, реализующих виджеты для конкретного стандарта.

Другими словами, клиенты должны лишь придерживаться интерфейса, определенного абстрактным, а не конкретным классом.

Класс `WidgetFactory` также устанавливает зависимости между конкретными классами виджетов. Полоса прокрутки для Motif должна использоваться с кнопкой и текстовым полем Motif, и это ограничение поддерживается автоматически, как следствие использования класса `MotifWidgetFactory`.

Применимость

Паттерн **абстрактная фабрика** следует использовать, 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.

Структура

Диаграмма классов для паттерна **Абстрактная фабрика** показана на рисунке 2.2.

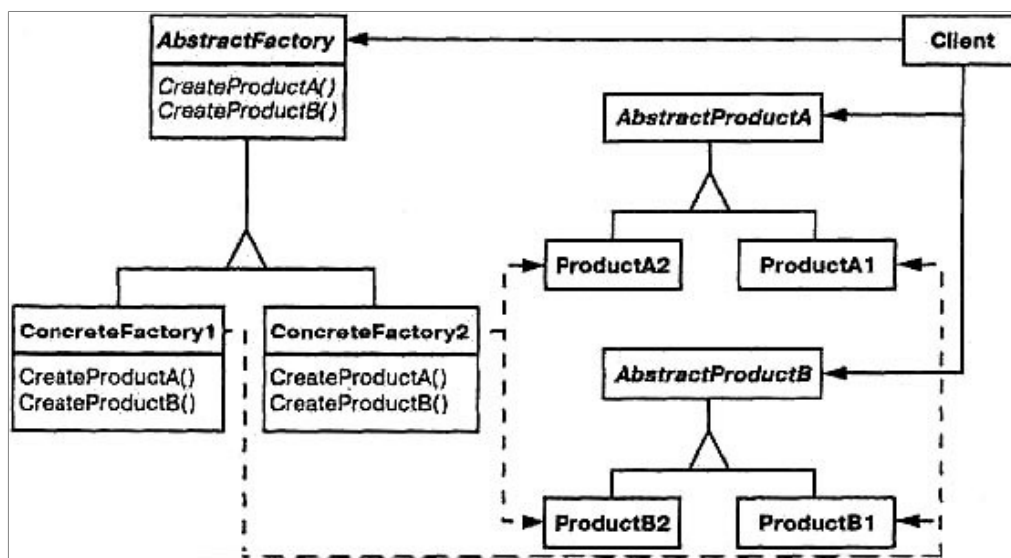


Рисунок 2.2 – Паттерн Абстрактная фабрика

Участники

- `AbstractFactory` (`WidgetFactory`) – абстрактная фабрика. Объявляет интерфейс для операций, создающих абстрактные объекты-продукты;
- `ConcreteFactory` (`MotifWidgetFactory`, `PMWidgetFactory`) – конкретная фабрика. Реализует операции, создающие конкретные объекты-продукты;

- `AbstractProduct (Window, ScrollBar)` – абстрактный продукт.

Объявляет интерфейс для типа объекта-продукта;

- `ConcreteProduct (MotifWindow, MotifScrollBar)` – конкретный продукт. Определяет объект-продукт, создаваемый соответствующей конкретной фабрикой; реализует интерфейс `AbstractProduct`;

- `Client` – клиент. Пользуется исключительно интерфейсами, которые объявлены в классах `AbstractFactory` и `AbstractProduct`.

Отношения

- Обычно во время выполнения создается единственный экземпляр класса `ConcreteFactory`. Эта конкретная фабрика создает объекты-продукты, имеющие вполне определенную реализацию. Для создания других видов объектов клиент должен воспользоваться другой конкретной фабрикой;

- `AbstractFactory` передоверяет создание объектов-продуктов своему подклассу `ConcreteFactory`.

Результаты

Паттерн **абстрактная фабрика** обладает следующими плюсами и минусами:

- изолирует конкретные классы. Помогает контролировать классы объектов, создаваемых приложением. Поскольку фабрика инкапсулирует ответственность за создание классов и сам процесс их создания, то она изолирует клиента от деталей реализации классов. Клиенты манипулируют экземплярами через их абстрактные интерфейсы. Имена изготавливаемых классов известны только конкретной фабрике, в коде клиента они не упоминаются;

- упрощает замену семейств продуктов. Класс конкретной фабрики появляется в приложении только один раз: при инстанцировании. Это облегчает замену используемой приложением конкретной фабрики. Приложение может изменить конфигурацию продуктов, просто подставив новую конкретную фабрику. Поскольку абстрактная фабрика создает все семейство продуктов, то и заменяется сразу все семейство. В нашем примере пользовательского интерфейса перейти от виджетов `Motif` к виджетам `Presentation Manager` можно, просто переключившись на продукты соответствующей фабрики и заново создав интерфейс;

- гарантирует сочетаемость продуктов. Если продукты некоторого семейства спроектированы для совместного использования, то важно, чтобы приложение в каждый момент времени работало только с продуктами единственного семейства. Класс `AbstractFactory` позволяет легко соблюсти это ограничение;

- поддержать новый вид продуктов трудно. Расширение абстрактной фабрики для изготовления новых видов продуктов – непростая задача. Интерфейс `AbstractFactory` фиксирует набор продуктов, которые можно создать. Для поддержки новых продуктов необходимо расширить интерфейс

фабрики, то есть изменить класс `AbstractFactory` и все его подклассы. Решение этой проблемы мы обсудим в разделе «Реализация».

3. Порядок выполнения работы

3.1. Изучить назначение и структуру паттерна **Абстрактная фабрика** (выполнить в ходе самостоятельной подготовки).

3.1. Применительно к предметной области, заданной в лабораторной работе №1, проанализировать возможность использования паттерна **Абстрактная фабрика**. Для этого на диаграмме классов, разработанной в ходе работы № 1, выделить семейства взаимосвязанных и совместно используемых классов, которые должны инстанцироваться совместно и при этом инстанцирующий их клиент не должен быть привязан к конкретным именам классов (пример приведен в разделе 2.2.).

3.3. Выполнить перепроектирование диаграммы классов, разработанной в лабораторной работе №1, используя паттерн **Абстрактная фабрика**.

3.4. Сравнить полученные диаграммы классов, сделать выводы и целесообразности использования паттернов проектирования для данной системы.

3.5. На основе UML-диаграммы выполнить синтез программного кода, скомпилировать программу и продемонстрировать ее работоспособность.

4. Содержание отчета

4.1. Цель работы.

4.2. Постановка задачи с указанием моделируемой предметной области.

4.3. Словесное описание мотивации применения паттерна **Абстрактная фабрика** при проектировании данной системы.

4.4. UML-диаграмма классов с комментариями.

4.5. Текст программы.

4.6. Выводы по работе.

5. Контрольные вопросы

5.1. Для чего предназначены поведенческие паттерны проектирования?

5.5. Какие задачи решает паттерн «Абстрактная фабрика»?

5.6. Какие классы входят в состав паттерна «Абстрактная фабрика», каковы их обязанности?

Библиографический список

1. Рамбо Дж. UML 2.0. Объектно-ориентированное моделирование и разработка / Дж. Рамбо, М. Блаха.– М. и др. : «Питер», 2007.– 544 с.
2. Гамма Э. Приемы объектно-ориентированного проектирования. Паттерны проектирования / Э. Гамма, Р. Хелм, Р. Джонсон, Дж. Влиссидес.– СПб.: «Питер», 2009.– 366 с.

Заказ № _____ от « » _____ 2012 г. Тираж _____ экз.
Изд-во СевНТУ