Урок № 2

Структурные паттерны

Содержание

   d. Результаты использования паттерна
   e. Практический пример использования паттерна
6. Facade
   a. Цель паттерна
   b. Причины возникновения паттерна
   c. Структура паттерна
   d. Результаты использования паттерна
   e. Практический пример использования паттерна
7. Flyweight
   a. Цель паттерна
   b. Причины возникновения паттерна
   c. Структура паттерна
   d. Результаты использования паттерна
   e. Практический пример использования паттерна
8. Proxy
   a. Цель паттерна
   b. Причины возникновения паттерна
   c. Структура паттерна
   d. Результаты использования паттерна
   e. Практический пример использования паттерна
9. Анализ и сравнение структурных паттернов
10. Практические примеры использования структурных паттернов

1. Понятие структурного паттерна

Исходя из определения информационных систем, является очевидным, что всякое программное решение, так ли иначе, оперирует некоторой совокупностью данных, при анализе которых мы получаем некоторую информацию, позволяющую нам осуществлять принятие бизнес решений и штатных решений по управлению самой информационной системой.

Анализ данных предполагает, что они должны быть организованы в некоторой структуре, которая позволит с одной стороны идентифицировать различные данные (логически отличать их друг от друга), а с другой – организовывать атомарные величины в структурированные объекты и их совокупности. Структурирование данных необходимо для отражения понятий, которыми мы оперируем при анализе, а также отображения отношений, которые образуются между объектами в процессе их появления.

Объектно-ориентированный подход предоставляет нам богатый инструментарий в смысле описания структурированных понятий и отражения отношений, в которых они состоят. Однако организация структуры, связывающей понятия в работающую систему, имеет решающее значение с точки зрения эффективного применения информационной системы, а так же с точки зрения сопровождения полученного в результате процесса разработки продукта.

Паттерны проектирования в общем смысле отражают наилучшие модели решения прикладных проблем, прошедшие проверку временем и практикой применения. Структурные паттерны предлагают модели организации данных в структуры, которые наилучшим образом решают вопрос организации управления данными, которые мы используем в своих прикладных решениях. Следует понимать, что данными можно считать не только атомарные величины, но и все объекты, существующие в пределах нашего приложения.

Далее последует перечисление основных структурных паттернов, признанных мировым сообществом лучшей практикой разработки программного обеспечения.

2. Паттерн Adapter

Цель паттерна

Для лучшего понимания предлагаемого материала, а так же с целью упрощения изложения, нами будет введена следующая терминология, специфичная рассматриваемому вопросу:

под *клиентом (client)* мы будем понимать некоторый класс, который использует (в общем случае агрегирует) некоторый класс, который мы называем *адаптируемым (adaptee)*. Под *адаптером (adapter)* мы будем понимать класс, выполняющий приведение интерфейса адаптируемого класса к интерфейсу, ожидаемому клиентом.

Цель паттерна проектирования Adapter (англ. «адаптер») состоит в том, чтобы привести (адаптировать) интерфейс некоторого адаптируемого класса к интерфейсу, который ожидается клиентом.

Причина возникновения паттерна

Достаточно часто встречается следующая проблема: у нас в наборе и инструментов имеется некоторый класс, который мы хотим использовать в неспецифичной для его структуры задаче. Например, у нас объявлен тип данных, описывающий понятие сетевого устройства и названный нами IPEndPoint, и наделённый такими свойствами как IP- адрес, мак-адрес и имя хоста, которые мы используем в целях некоторого прикладного анализа (например, трассировки перемещения пакетов). Анализ выполняется некоторым классом, который агрегирует множество объектов типа IPEndPoint и называется NetView. Однако мы хотим реализовать графическое представление для процесса и результата анализа, с выводом его на основное окно нашего приложения. Проблема состоит в том, что класс NetView не имеет интерфейса, специфичного для объекта графической подсистемы и, соответственно, не может быть использован оконным классом для выполнения прорисовки. Мы не имеем возможности «переписать» (изменить исходный текст и соответственно структуру) класс NetView, под нужды приложения, поскольку он является частью, используемого нами набора типов из dll-библиотеки предоставленной

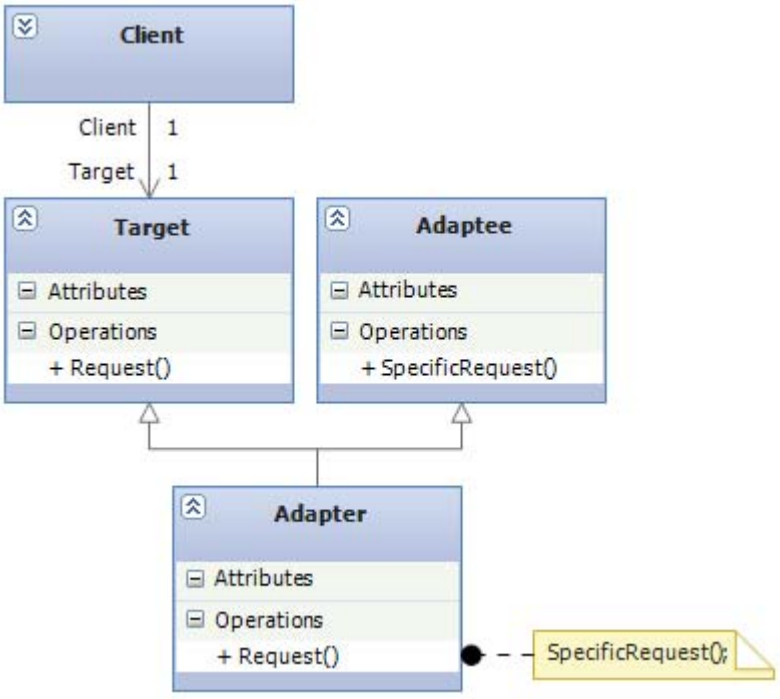
сторонними разработчиками; или мы просто не хотим «перегружать» структуру класса, поскольку она критична для каких-либо задач нашего приложения.

В таком случае логично использовать некоторый класс-посредник, который, используя наследование, позволит привести класс NetView к необходимому типу данных, а так же посредством переопределения методов, специфичных для компонента графической подсистемы, определит интерфейс прорисовки типа данных NetView.

Такой класс-посредник обычно называют адаптером.

Структура паттерна

Структура паттерна Adapter представлена на приведённой слева диаграмме.



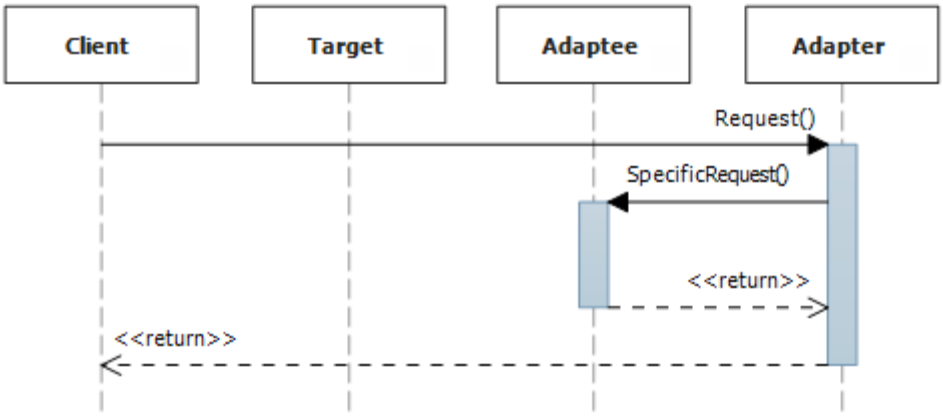
Участвующие элементы:

- *Client* – класс, который использует некоторые вспомогательные типы данных и ожидает, что они имеют стандартный интерфейс взаимодействия (использования) описанный классом Target.
- *Target* – класс, имеющий интерфейс, ожидаемый клиентом.
- *Adaptee* – класс, который необходим для работы клиента, но имеет интерфейс, отличный от того, который ожидается клиентом.
- *Adapter* – класс, выполняющий приведение интерфейса класса Adaptee, к интерфейсу класса Target.

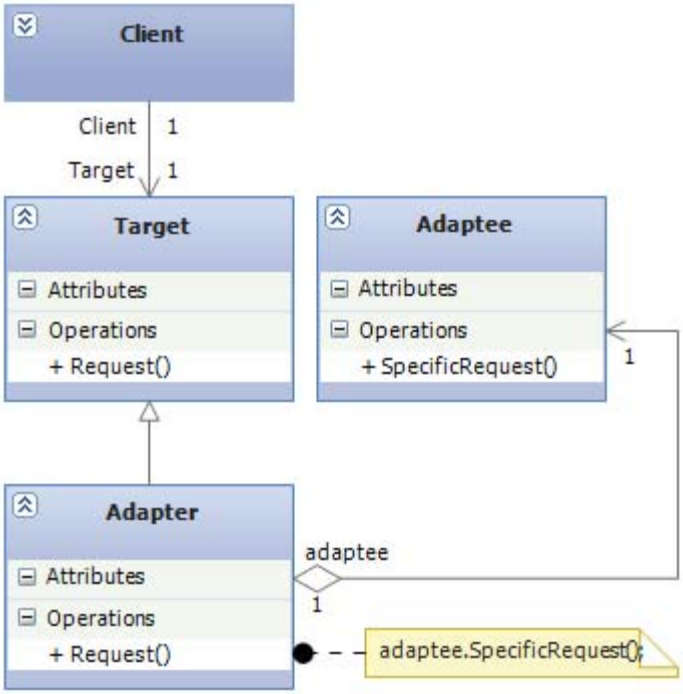
В, предложенной выше структуре, приведение интерфейса выполняется за счёт того, что класс Adapter наследует оба класса Adaptee и Target, а, значит, обладает интерфейсами обоих этих классов. Затем класс Adapter приводит

вызовы методов специфичных для интерфейса класса Target к вызовам соответствующих методов интерфейса класса Apadtee.

На представленной ниже диаграмме последовательности демонстрируется, что вызовы методов объекта класса Adapter сводятся к вызовам методов объекта его базового класса.



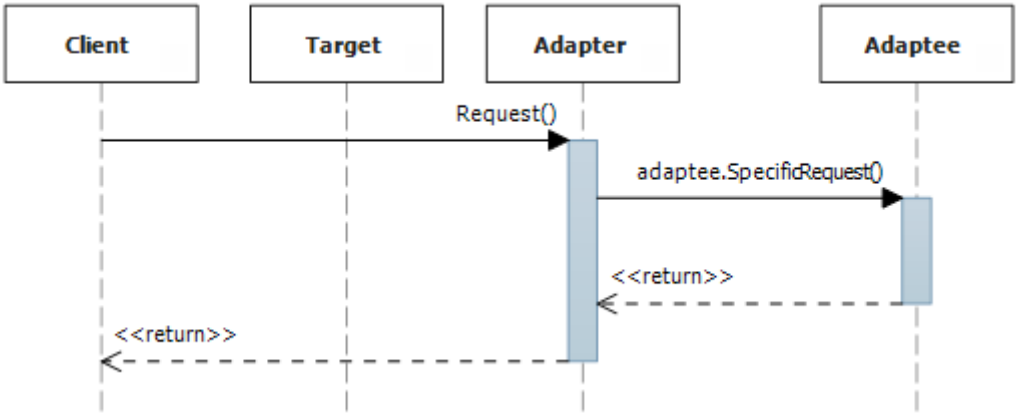
Паттерн Adapter также может быть реализован альтернативным способом, структура которого представлена на приведённой слева диаграмме.



Отличие от предыдущей структуры состоит в том, что в текущей модели классы Adapter и Adaptee находятся не в отношении родства, а в отношении ассоциации, то есть класс Adapter агрегирует класс Adaptee.

Таким образом, приведение интерфейса класса Adaptee к интерфейсу класса Target выполняется за счёт того, что вызовы методов объекта класса Adapter, специфичные для интерфейса класса Target приводятся к вызовам соответствующий методов объекта класса

Adaptee, инкапсулированного в классе Adapter. Нижеприведённая диаграмма последовательности иллюстрирует происходящее.



Результаты использования паттерна

Основной положительный результат использования паттерна проектирования Adapter состоит в том, что мы получаем возможность гибко привести интерфейс некоторого класса к интерфейсу, ожидаемому приложением без изменения структуры самого класса. Это необходимо, как с точки зрения устранения избыточности структуры типов, так и с точки зрения модульности создаваемых приложений.

Избыточность играет отрицательную роль тогда, когда нам необходимо повторно использовать написанный нами код (например, в другом приложении) Такой код называется reusable-кодом. Создание reusable-кода считается хорошей практикой, поскольку уменьшает стоимость и увеличивает скорость разработки. А так же увеличивает гибкость и масштабируемость создаваемых приложений за счёт модульной структуры готового приложения.

Расширять приложение путём добавления новых типов значительно проще, чем полностью заново создавать некоторые модули.

Практический пример использования паттерна

Мы рассмотрим использование паттерна Adapter на примере приложения, выполняющего управление товарооборотом некоторого предприятия. Для осуществления управления товарами мы опишем тип данных Product (продукт/товар), который будет организован в коллекцию товаров при помощи

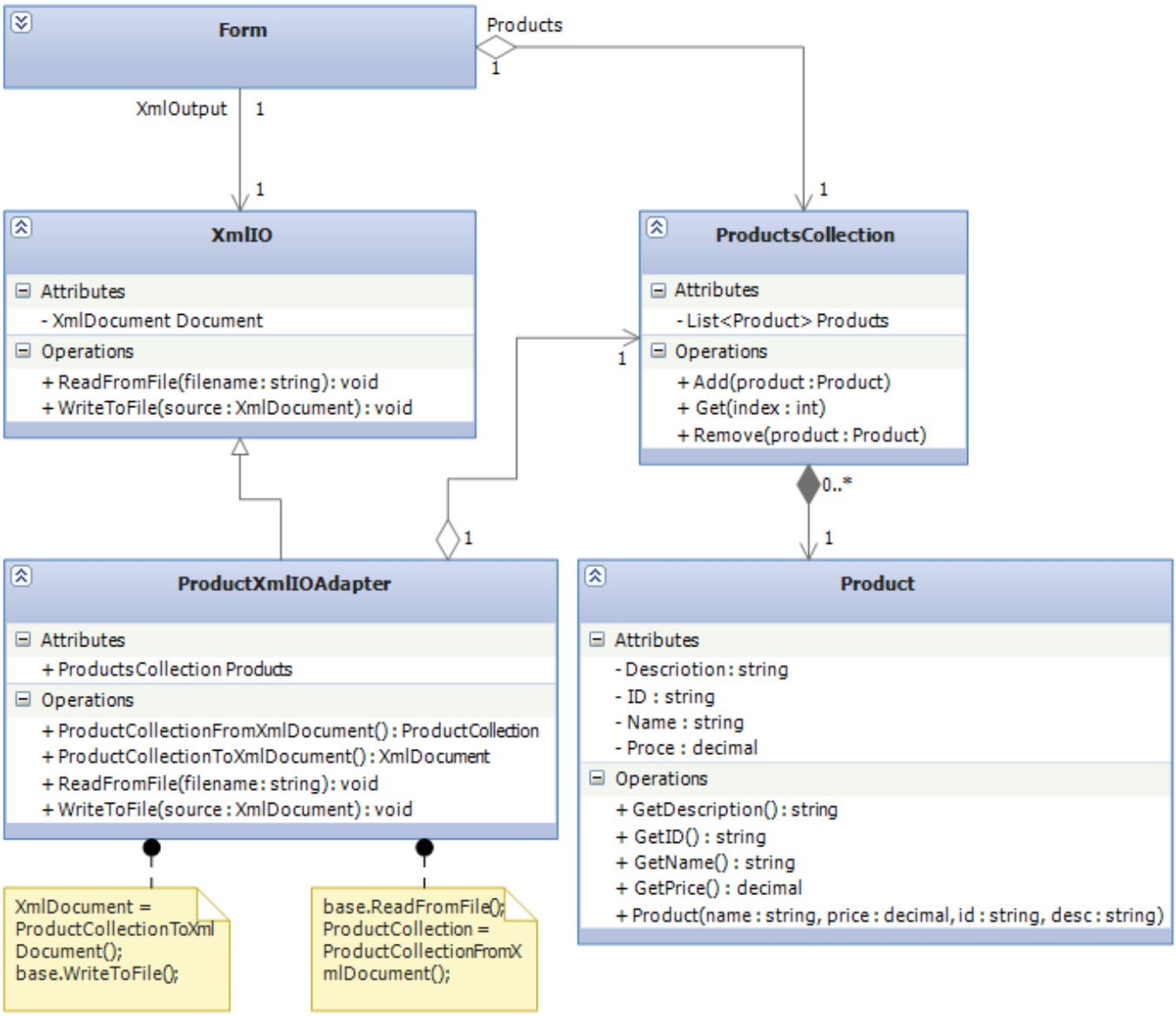
класса ProductsCollection.

Наше приложение так же выполняет управление и другими аспектами работы торгового предприятия, которые мы «опустим» для прозрачности примера.

Для реализации возможности создавать переносимую резервную копию данных, которую можно использовать для практически любых задач мы решили сделать так, чтобы можно было все данные экспортировать в отдельные xml-файлы. Для этого мы реализовали класс XmlIO, которые осуществляет запись и чтение Xml-документа, и объект которого инкапсулируется оконным классом нашего приложения. Для реализации записи в файл коллекции продуктов нами был создан класс ProductXmlIOAdapter, который инкапсулирует коллекцию продуктов, реализует приведение этой коллекции к xml-документу, а так же xml-документа к коллекции.

Таким образом, мы реализуем функцию записи коллекции элементов в файл в xml-формате и в то же время не изменяем структуры исходного типа данных и устраняем избыточность структуры, которая могла появиться вследствие наполнения класса Product дополнительным функционалом. Устранение избыточности необходимо постольку, поскольку тип данных Product может использоваться при приведении информации о продуктах, полученной из некоторой базы данных, к объектному представлению, для выполнения последующего анализа т так далее. При выполнении всех этих действий избыточность структуры типа может создавать дополнительные сложности. Также, подобная модульная структура добавляет гибкости при сопровождении проекта и использовании (создании) reusable-кода.

Далее представлена диаграмма классов иллюстрирующая описанное выше приложение.


Form
Products
1
XmlOutput
1
1
1
XmlIO
Attributes
- XmlDocument Document
Operations
+ ReadFromFile(filename: string): void
+ WriteToFile(source : XmlDocument) : void
ProductsCollection
Attributes
- List<Product> Products
Operations
+ Add(product : Product)
+ Get(index : int)
+ Remove(product : Product)
1
0..*
1
1
ProductXmlIOAdapter
Attributes
+ ProductsCollection Products
Operations
+ ProductCollectionFromXmlDocument() : ProductCollection
+ ProductCollectionToXmlDocument() : XmlDocument
+ ReadFromFile(filename: string): void
+ WriteToFile(source : XmlDocument) : void
Product
Attributes
- Descriotion: string
- ID : string
- Name : string
- Proce : decimal
Operations
+ GetDescription() : string
+ GetID() : string
+ GetName() : string
+ GetPrice() : decimal
+ Product(name : string, price : decimal, id : string, desc : string)
XmlDocument = ProductCollectionToXmlDocument();
base.WriteToFile();
base.ReadFromFile();
ProductCollection = ProductCollectionFromXmlDocument();

3. Паттерн Bridge

Цель паттерна

Цель паттерна Bridge (англ. «мост») состоит в том, чтобы отделить абстракцию от её реализации, для того, чтобы они могли изменяться независимо друг от друга.

Причины возникновения паттерна

Обычно, в случаях, когда некоторая абстракция (обычно абстрактный класс) может иметь несколько конкретных реализаций, используют наследование для определения множества классов, с похожим (в общем случае говорят одинаковым или совместимым) интерфейсом. Абстрактный класс определяет интерфейс для своих потомков, который они реализуют «различными» способами.

Однако такой подход является не всегда достаточно гибким и имеет некоторые слабые стороны, способные привести к избыточности кода, а также создать дополнительные трудности при сопровождении проекта, что значительно увеличит его стоимость. Прямое наследование интерфейса *абстракции* некоторым конкретным классом связывает *реализацию* с *абстракцией* напрямую, что создаёт трудности при дальнейшей модификации *реализации* (её расширении), а так же не позволяет повторно использовать *абстракцию* и её *реализацию* отдельно друг от друга. *Реализация,* как бы, становиться «жёстко связанной» с *абстракцией*.

Паттерн проектирования мост предполагает помещение интерфейса и его реализации в различных иерархиях, что позволяет отделить интерфейс от реализации и использовать их независимо, а так же комбинировать любые варианты *реализации* с различными уточнёнными вариантами *абстракции*.

Структура паттерна

Паттерн проектирования мост представлен следующими структурными элементами:

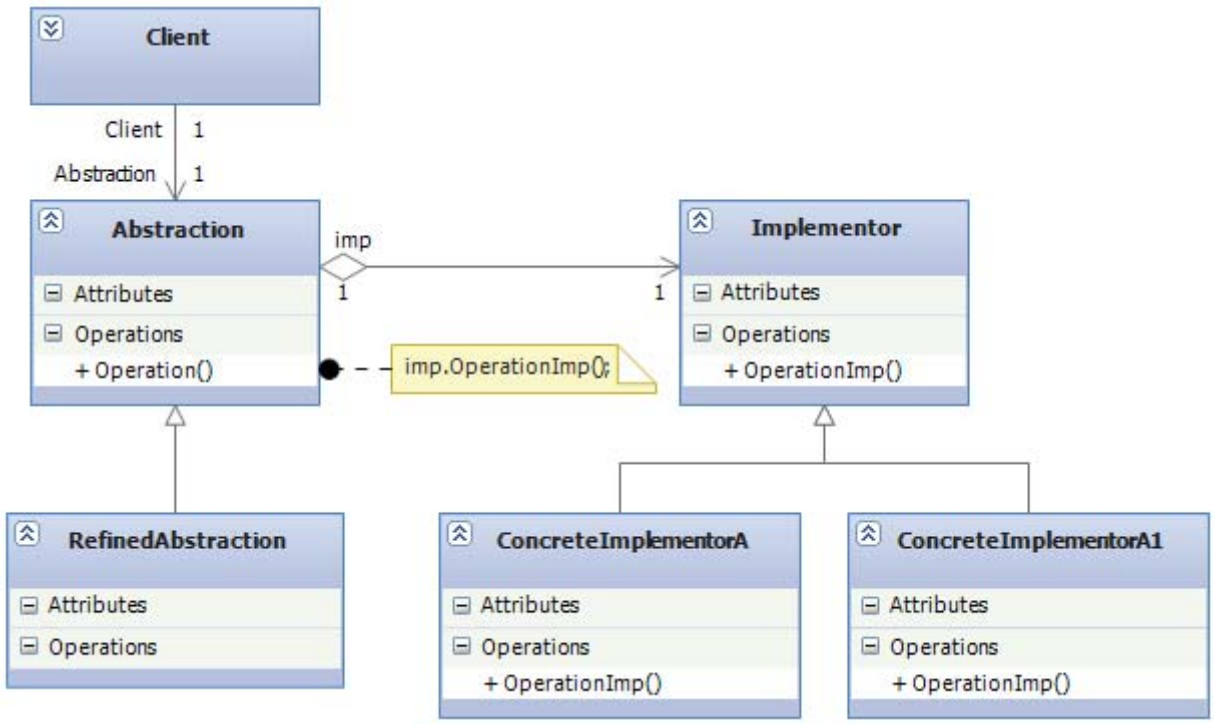
- *Abstraction* (абстракция) – определяет интерфейс абстракции, а также содержит объект исполнителя, который определяет интерфейс

реализации.

- *Implementor* (исполнитель) – определяет интерфейс для классов реализации. Интерфейс исполнителя не обязательно должен соответствовать интерфейсу абстракции. В принципе, интерфейсы, определённые абстракцией и исполнителем, могут быть совершенно разными, что является достаточно гибким. В целом, исполнитель должен определять базовые операции, на которых впоследствии базируется высокоуровневая логика абстракции.
- *RefinedAbstraction* (уточнённая абстракция) – расширяет интерфейс определённый абстракцией.
- *ConcreteImplementor* (конкретизированный исполнитель) – класс, который реализует интерфейс исполнителя и определяет его частную реализацию.

Абстракция и исполнитель совместно образуют «мост», который связывает уточнённую абстракцию с конкретной реализацией.

Структура паттерна проектирования мост представлена ниже в виде диаграммы классов.



Результаты использования паттерна

Основное преимущество которое предоставляет использование паттерна проектирования мост состоит в том, что выполняется логическое и структурное разделение абстракции от её реализации, что делает код более гибким. Так же применение паттерна мост улучшает такое качество кода, как расширяемость, поскольку абстракция и исполнитель находятся в различных иерархических структурах, а значит становиться возможным расширение реализации независимо от абстракции, и наоборот.

Ещё одной причиной в пользу применения паттерна проектирования мост служит тот факт, что он позволяет скрывать детали реализации от клиента (client), то есть от приложения, которое использует абстракцию, что позволяет клиенту быть независимым от того, какая именно реализация была выбрана в том или ином случае.

Практический пример использования паттерна

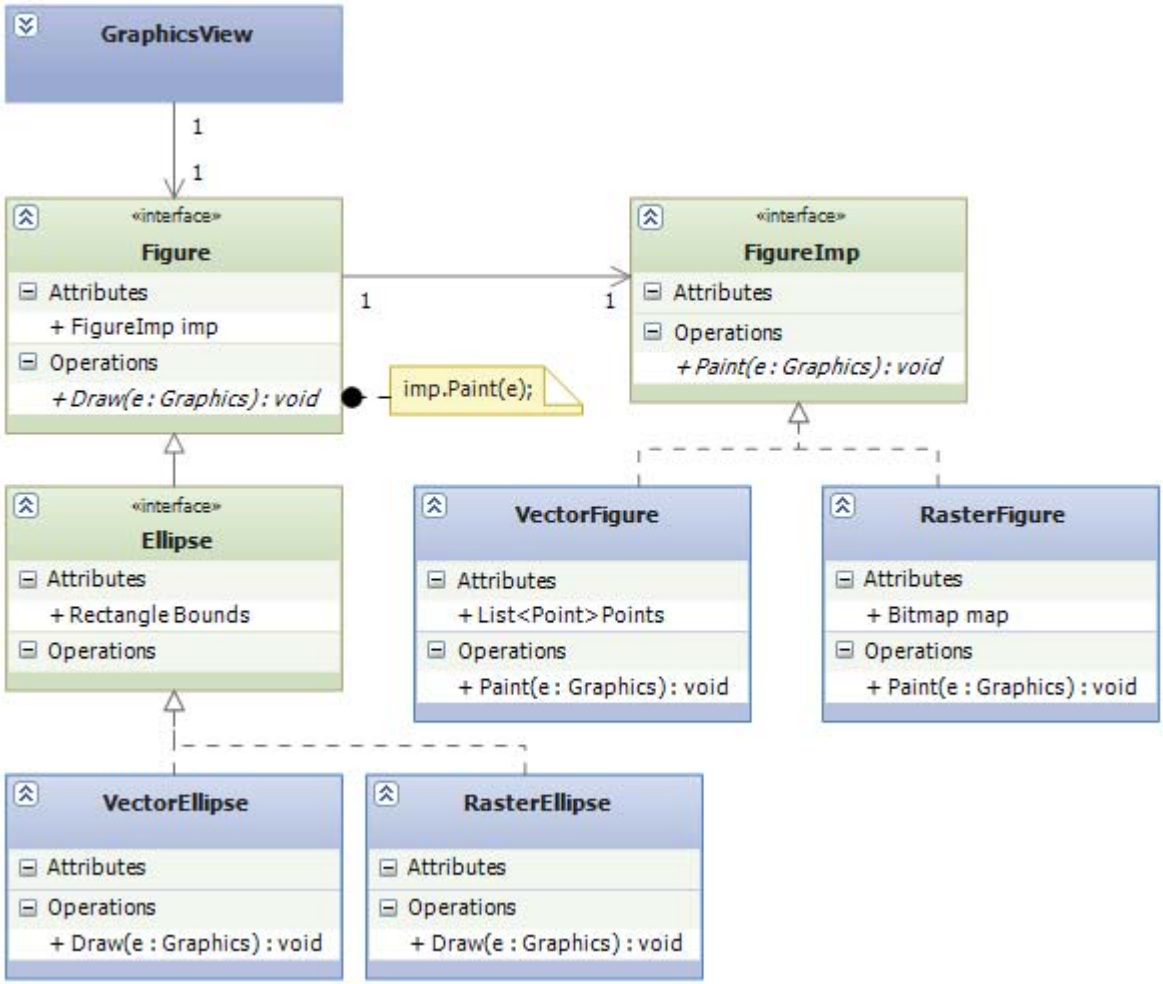
Мы рассмотрим применение паттерна проектирования мост на примере «смешанного» графического редактора (редактора, позволяющего совместно, в рамках одного представления, редактировать растровую и векторную графику).

Модель приложения предполагает наличие некоторого графического представления, оперирующего некоторыми абстрактными фигурами, интерфейс взаимодействия с которыми описывается интерфейсом Figure, играющим роль абстракции в данном примере.

Интерфейс исполнителя определяется интерфейсом FigureImp, от которого мы наследуем классы VectorFigure и RasterFigure – соответственно, описывающих реализации прорисовки векторной и растровой фигур.

После определения реализации мы можем уточнить абстракцию, описав интерфейса некоторой конкретной фигуры (например, эллипса). После этого можно связать уточнённую абстракцию с конкретной реализацией, например, определив классы VectorEllipse и RasterEllipse, описывающие соответственно векторный и растровый эллипсы.

Ниже представлена модель описанного выше приложения.



4. Паттерн Composite

Цель паттерна

Паттерн Composite (компоновщик) предназначен для того, чтобы представить объекты в виде структуры дерева в иерархической связи часть-целое.

Причины возникновения паттерна

Паттерн компоновщик существует потому, что структура типа «дерево» является достаточно распространённой и часто используется для организации данных имеющих регулярную структуру. То есть такую структуру, которая реализует иерархическую зависимость часть-целое, предполагающую, что

элементом некоторой композиции данных может быть не только элементарный элемент, но и такая же композиция.

Самая простая реализация подобной структуры с рекурсивной системой вложенности может быть выражена в виде некоторой системы типов, содержащей классы, описывающие элементарные компоненты, и классов, которые будут использоваться в качестве контейнеров для элементарных компонент.

Но подобный подход имеет серьёзный недостаток, состоящий в том, что код, использующий указанные классы, должен отдельно обрабатывать элементы и их контейнеры, даже если в большинстве случаев они обрабатываются одинаково. Это резко усложняет реализацию приложения.

Паттерн компоновщик предлагает рекурсивную структуру, при которой не придётся принимать решения о том, как обрабатывать отдельные элементы общей совокупности данных.

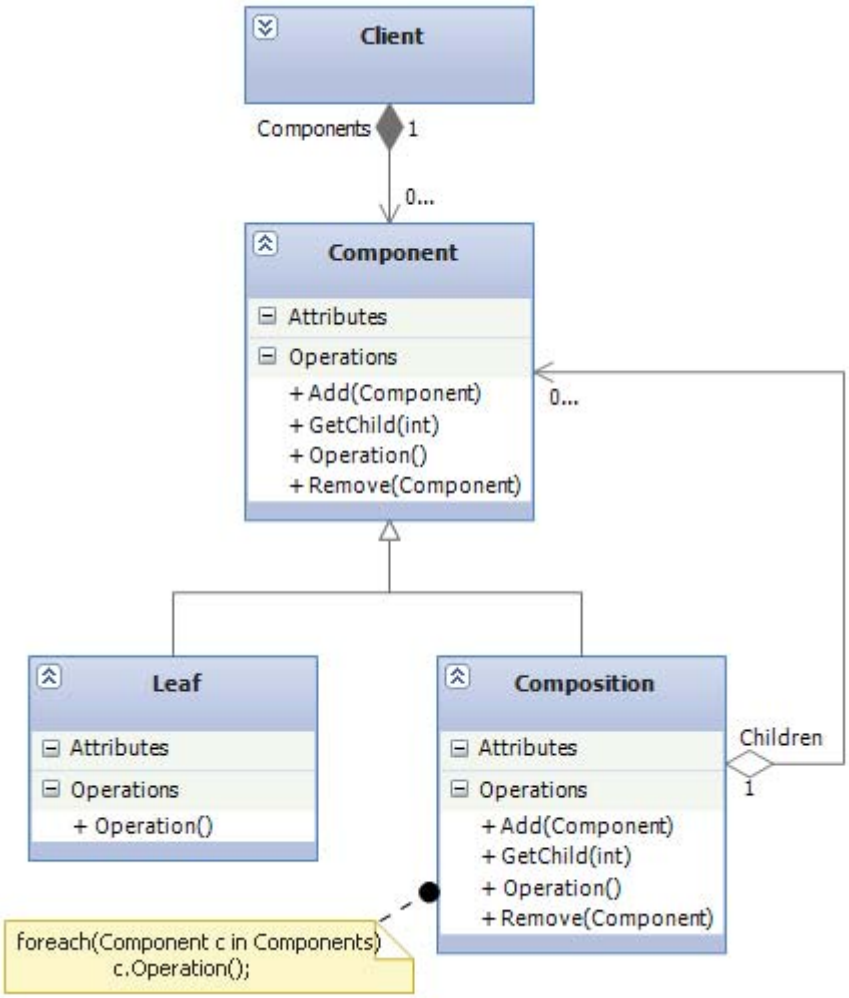
Ключ к пониманию природы компоновщика состоит в определении абстрактного идентичного интерфейса для элементарных компонент и их контейнеров. Таким образом, поскольку контейнеры и элементарные компоненты находятся в отношении родства и имеют общий пользовательский интерфейс, то контейнер может быть элементом другого такого же контейнера, что создаёт удобную регулярную (рекурсивную) структуру.

Структура паттерна

- Component (компонент) – описывает интерфейс для объектов и их композиций; реализует базовое поведение, специфичное и для отдельных элементов и для их композиций; определяет интерфейс доступа к элементам композиции и управления этими элементами, а также определяет интерфейс доступа к родительскому элементу рекурсивной структуры.
- Leaf (лист) – определяет отдельный элемент композиции и описывает поведение «примитивных» (базовых) элементов общей структуры.

- Composition (композиция) – определяет поведение для компонентов, содержащих дочерние элементы, инкапсулирует дочерние элементы, а также реализует операции управления дочерними элементами и доступа к ним, определённые интерфейсом компонента.
- Client (клиент) – управляет элементами композиции через интерфейс компонента.

Структура паттерна компоновщик представлена в виде диаграммы классов, приведённой на рисунке справа.



Результаты использования паттерна

Использование паттерна компоновщик помогает упростить организацию элементов в виде вложенной структуры данных и устраняет избыточность кода при реализации этой задачи. Также компоновщик делает клиентский объект более простым и позволяет ему обрабатывать элементарные компоненты и их композиции одинаково, в силу их унифицированного интерфейса, определённого классом-компонентом.

Компоновщик упрощает процесс добавления новых компонент. Новые классы, независимо от того, являются они элементарными компонентами или композициями, будут работать с уже существующей на момент их создания

структурой. Другими словами нет необходимости изменять клиента при создании новых компонент.

Практический пример использования паттерна

Мы рассмотрим применение паттерна проектирования компоновщик на базе организации множества элементов графического интерфейса пользователя, которое имеет регулярную структуру. Другими словами, всякое окно может содержать элементы управления, но также и другие окна. Окна можно упрощённо определить как композиции элементов управления.

Для организации компоновщика под определённую нами задачу мы объявляем абстрактный класс UIElement, описывающий некоторый элемент графического интерфейса пользователя и определяющий интерфейс базового поведения элемента графического интерфейса и интерфейс управления дочерними элементами композиции.

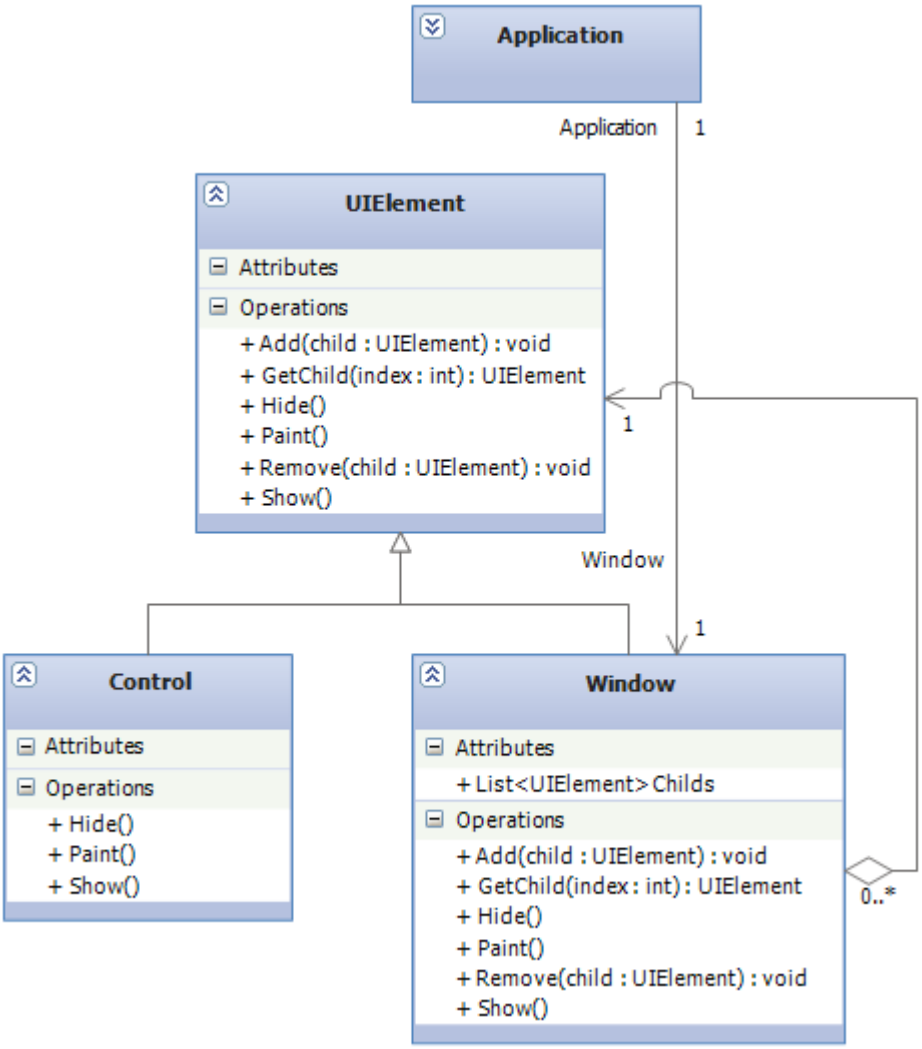
К базовому поведению элемента графического интерфейса относятся методы Hide, Paint и Show, которые соответственно реализуют возможность показывать элемент, перерисовывать его графическое представление и прятать.

Для управления-доступа к дочерним элементам композиции определяются методы Add, GetChild и Remove, которые соответственно позволяют добавлять новый дочерний элемент, получить его и удалить элемент из композиции.

Класс Control, описывает элемент управления, который играет роль «примитивного» элемента реализуемого нами компоновщика и реализует базовое поведение. Класс Window представляет собой композицию элементов управления, но также реализует и базовое поведение, поскольку имеет некоторое элементарное графическое представление.

Класс Application играет роль клиента, который ассоциирован с некоторым окном, то есть использует объект класса Window и интерфейс, описанный классом UIElement.

Модель описанного приложения представлена на диаграмме классов на приведённом ниже рисунке.



5. Паттерн Decorator

Цель паттерна

Цель паттерна Decorator (англ. «декоратор») состоит в том, чтобы реализовать возможность динамического добавления функционала к объекту, а так же распределить ответственность за выполнение отдельных функций между отдельными классами.

Так же декоратор представляет собой альтернативу наследованию в смысле расширения функционала объектов.

Причины возникновения паттерна

Достаточно распространённым подходом является разделение

ответственности за выполнение отдельных операций между отдельными классами, поскольку это создаёт структуру, при которой каждый класс инкапсулирует логику управления только теми обязанностями, которые на него возложены. И значение имеет не только модульность, упрощающая разработку, но и возможность закрывать классу доступ к операциям, которыми он управлять не должен по его сути.

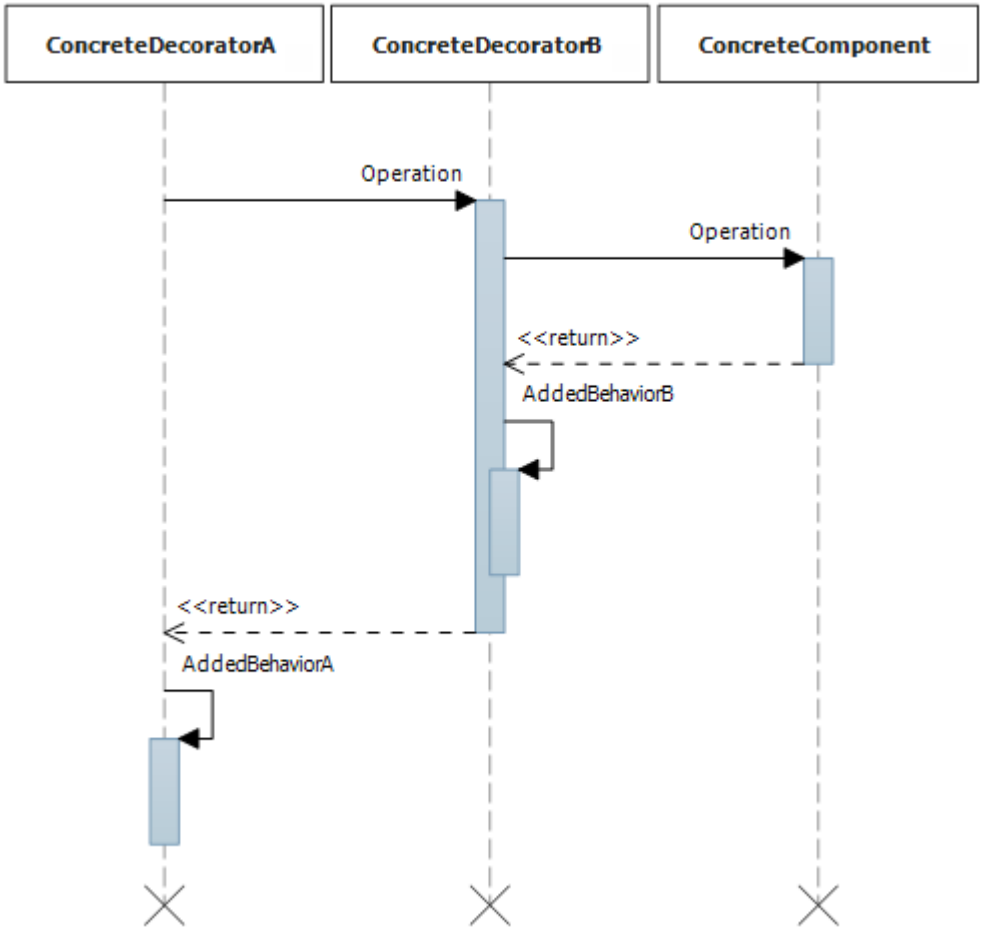
Одним из методов, позволяющим распределить обязанности по выполнению некоторой общей задачи между отдельными классами является наследование. Однако такой подход является негибким, поскольку при наследовании добавленная функция статически закрепляется для всех потомков этого класса. И для того, чтобы создавать различные типы (с разным набором функциональности), необходимо выполнять наследование «во всех возможных» комбинациях.

Декоратор же позволяет не только гибко комбинировать функциональность объекта по его внутренней логике, но и динамически изменять набор функций объекта во время выполнения, поскольку добавление функции сводится к созданию компонентного объекта необходимого класса.

Структура паттерна

Структура паттерна Decorator представлена следующими элементами:

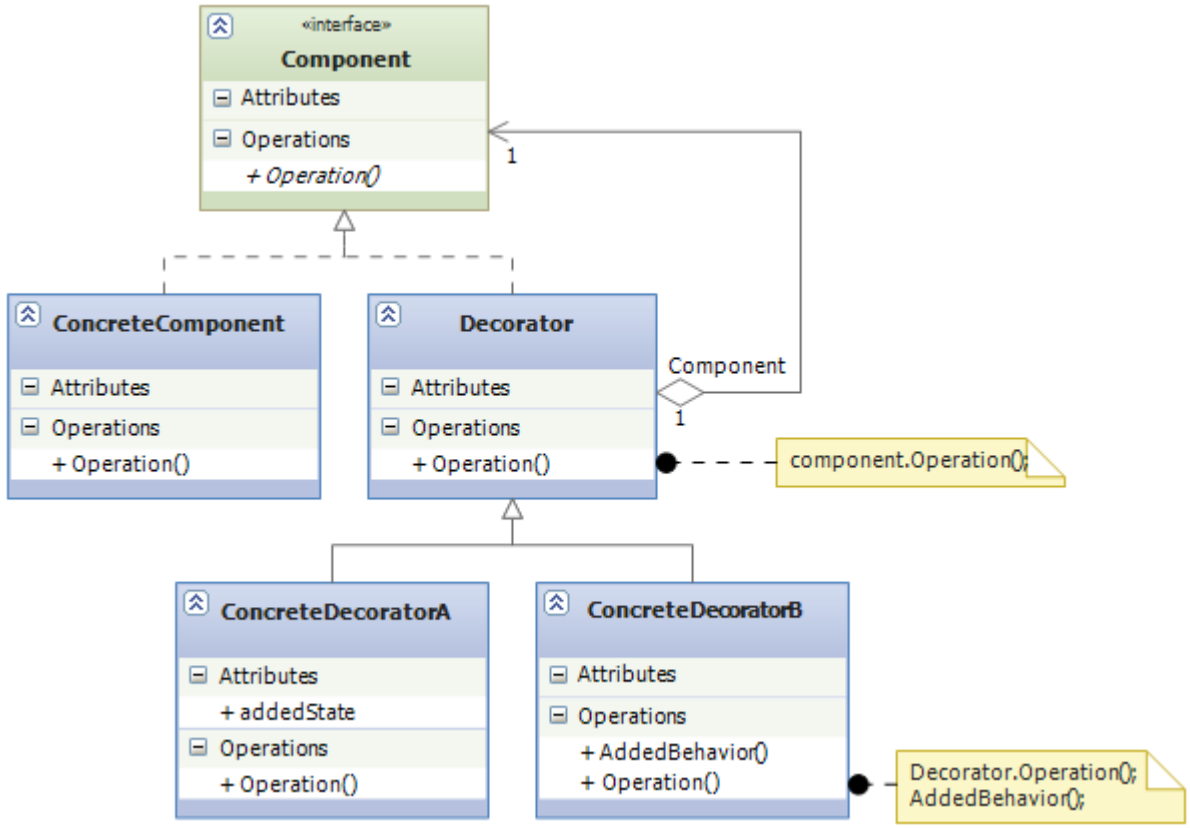
- Component – представляет собой абстракцию некоторого элемента, для которого будут определены «декорации» (оформление). Конечно же, под «декорированием» понимается не визуальное оформление, а добавление специфических функций. Component содержит объявление абстрактной операции, к конкретной реализации которой декоратором, впоследствии, будет добавлен «новый» функционал.
- ConcreteComponent – представляет собой реализацию компонента.
- Decorator – класс, который наследует и агрегирует компонент. Он переопределяет реализацию операции таким образом, чтобы выполнить функцию, инкапсулированную в компоненте, а затем добавить новый функционал.



Поскольку декоратор является частным случаем компонента (наследует Component), то в качестве инкапсулированного компонента может быть использован другой декоратор, что позволяет осуществлять рекурсивную последовательность вызовов переопределённой операции с постепенным «накоплением» функционала (как показано выше на диаграмме последовательности).

ConcreteDecoratorA и ConcreteDecoratorB – представляют собой частные реализации декоратора для компонента.

Структура паттерна проектирования декоратор (Decorator) иллюстрируется представленной ниже диаграммой классов.



Результаты использования паттерна

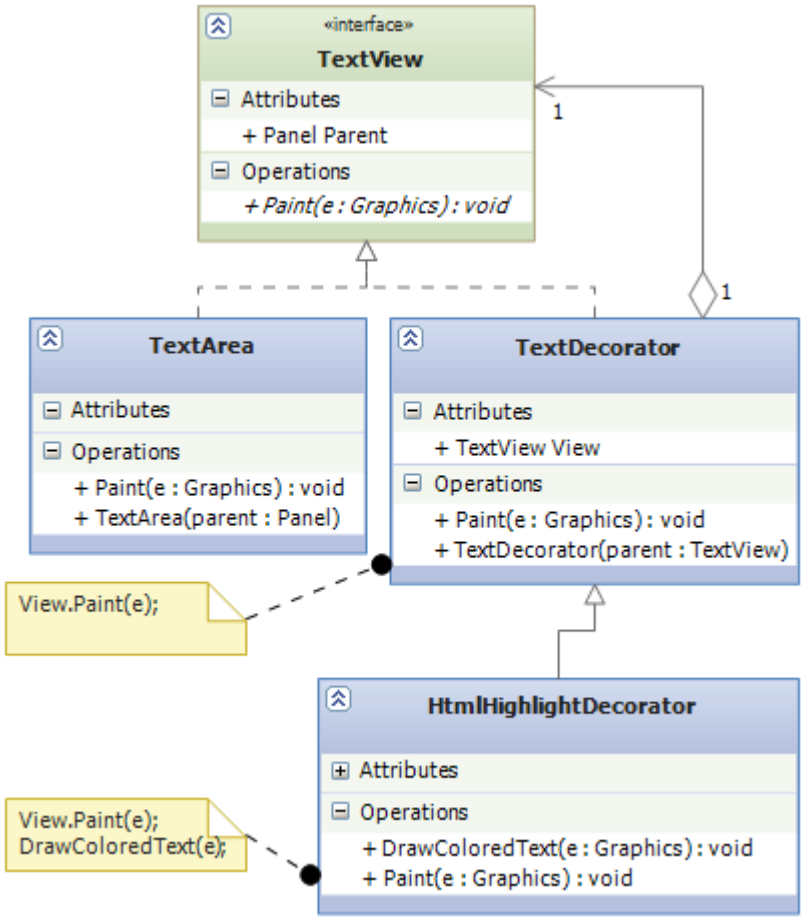
Использование паттерна, как уже выше упоминалось, позволяет более гибко, нежели при использовании наследования реализовать распределение обязанностей по выполнению некоторой сложной задачи между несколькими классами.

С другой стороны паттерн проектирования декоратор предотвращает перенасыщение иерархии классов, поскольку позволяет избегать необходимости создавать классы, которые бы сочетали в себе функционал во всех необходимых комбинациях.

Практический пример использования паттерна

В качестве примера мы рассмотрим модель текстового редактора с подсветкой html-синтаксиса.

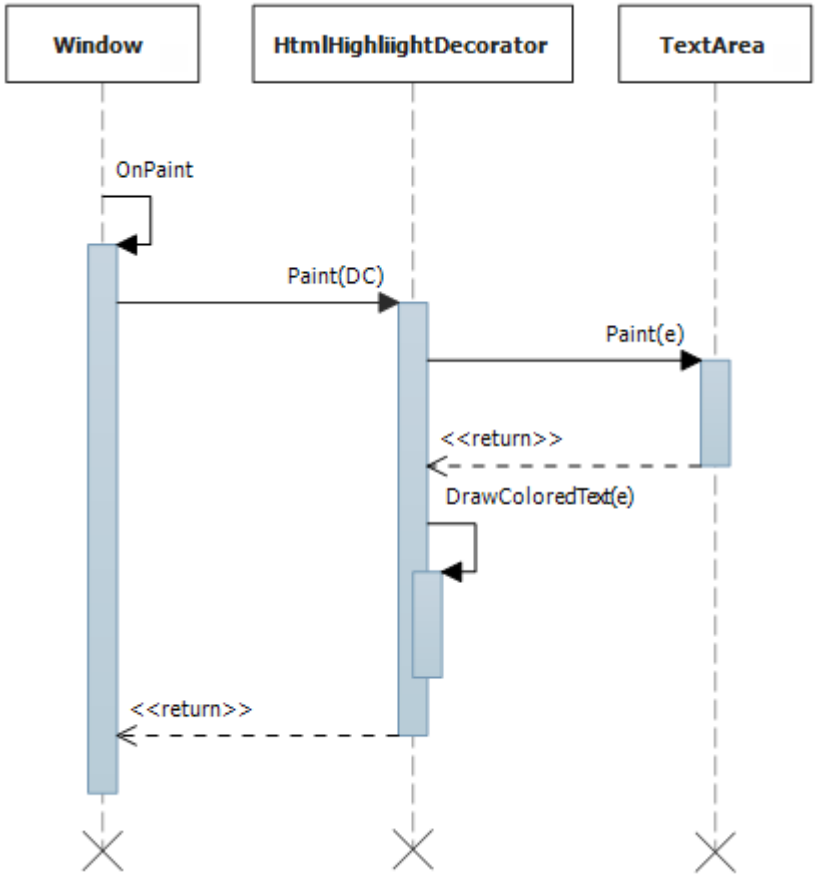
Предполагается использовать класс текстового представления, для реализации логики представления текстовой информации пользователю. Модель реализации текстового представления приведена ниже.



Класс текстового представления (TextView) наследуется, с одной стороны, классом конкретной реализации представления в виде текстовой области (класс TextArea), поддерживающей редактирование. Для поддержки возможности расширения функционала текстового представления, нами объявляется класс TextDecorator, который, собственно, реализует паттерн Decorator. Для реализации возможности подсветки html-синтаксиса нами создаётся конкретная реализация класса TextDecorator, которая инкапсулирует логику осуществления подсветки html-синтаксиса в текстовом представлении.

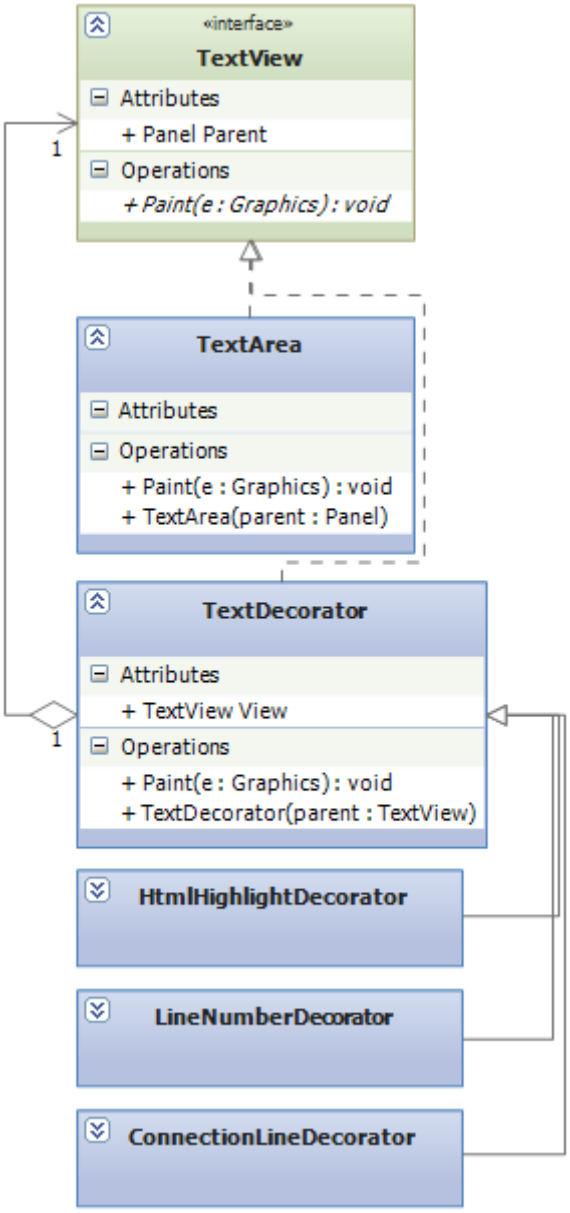
При осуществлении прорисовки окна, в котором используется текстовое представление, (обработчик события Paint) создаётся изображение по размеру клиентской области приложения, для которого инициализируется графический контекст. Объект графического контекста созданного изображения передаётся в метод Paint объекта класса HtmlHighlightDecorator, который вызывает метод Paint объекта класса TextArea, а впоследствии и метод DrawColoredText,

который отвечает за прорисовку подсвеченного текста. Последовательность вызовов иллюстрируется на диаграмме последовательности, приведённой ниже.



Таким образом, после возвращения управления в обработчик события Paint оконного класса, на созданном в обработчике изображении будет прорисован текст с подсвеченным синтаксисом. Последним действием мы прорисовываем

```
4       <component name=
"Microsoft-Windows-International-Core-WinPE"
processorArchitecture="x86" publicKeyToken=
"31bf3856ad364e35" language="neutral" versionScope=
"nonSxS" xmlns:wcm=
"http://schemas.microsoft.com/WMIConfig/2002/State"
xmlns:xsi=
"http://www.w3.org/2001/XMLSchema-instance">
5         <SetupUILanguage>
6           <UILanguage>ru-RU</UILanguage>
7         </SetupUILanguage>
8         <InputLocale>en-US; ru-RU</InputLocale>
9         <SystemLocale>ru-RU</SystemLocale>
10        <UILanguage>ru-RU</UILanguage>
11        <UserLocale>ru-RU</UserLocale>
12      </component>
```



полученное изображение на графическом контексте окна, с которым связано текстовое представление.

Подобная реализация позволит впоследствии гибко добавлять и убирать различные декорирующие эффекты, свойственные текстовым редакторам. Например, номера строк, соединяющие линии для парных элементов, как показано на представленном выше рисунке.

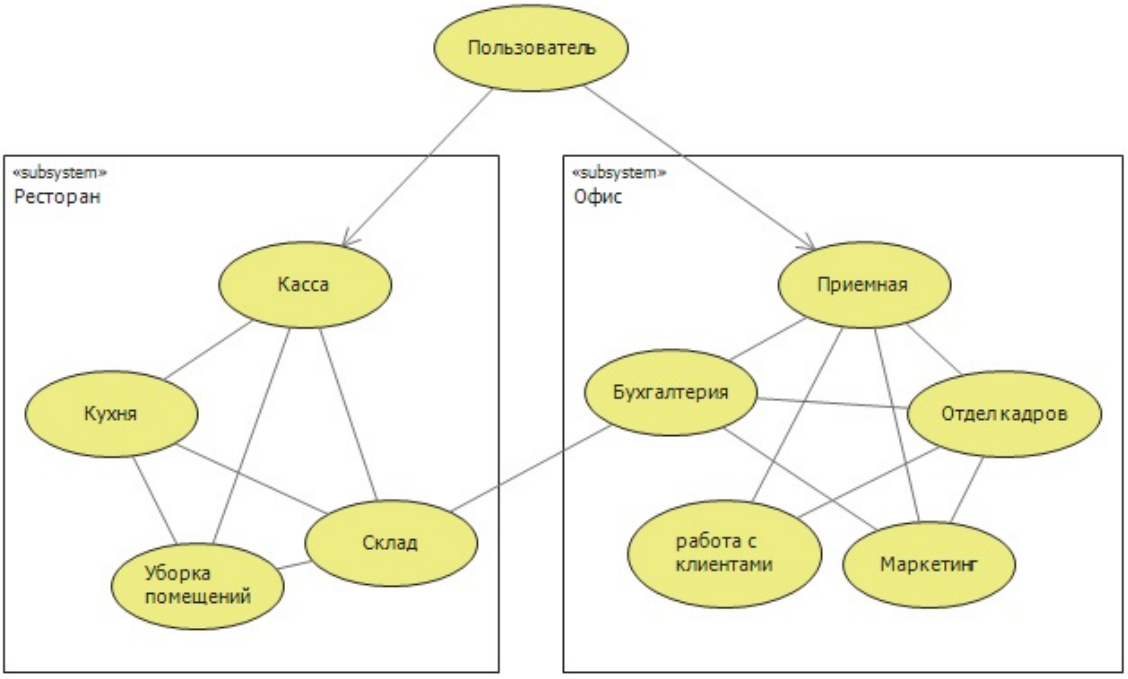
Для этого достаточно объявить несколько декораторов, и добавить их для текстового представления в оконном классе, с которым связано текстовое представление.

Обобщённая модель приведена слева.

6. Паттерн Facade

Паттерн Facade (фасад) предназначен в большей мере для инкапсуляции (сокрытия) содержимого и разделения логических частей на независимые подсистемы. В «реальном» мире много где применяется структура паттерна фасад, хорошим примером служит любой ресторан быстрого питания, где вы просто подходите к кассе и заказываете еду, а дальнейшая структура забегаловки вас, как правило, не интересует. Но в забегаловке также есть офис, в который вы тоже можете прийти, и обратится к нему по «узкому»

интерфейсу (например, стать постоянным клиентом). Таким образом, ресторан получает две минимально зависящие друг от друга подсистемы, которые вместе складываются в одну большую систему.



Цель паттерна

Целью паттерна фасад является определенная структуризация классов, в подсистемы, доступ к каждой из них происходит через один интерфейс. Так же при построении архитектуры приложения желательно его разделять на отдельные под системы, которые должны быть максимально независимы.

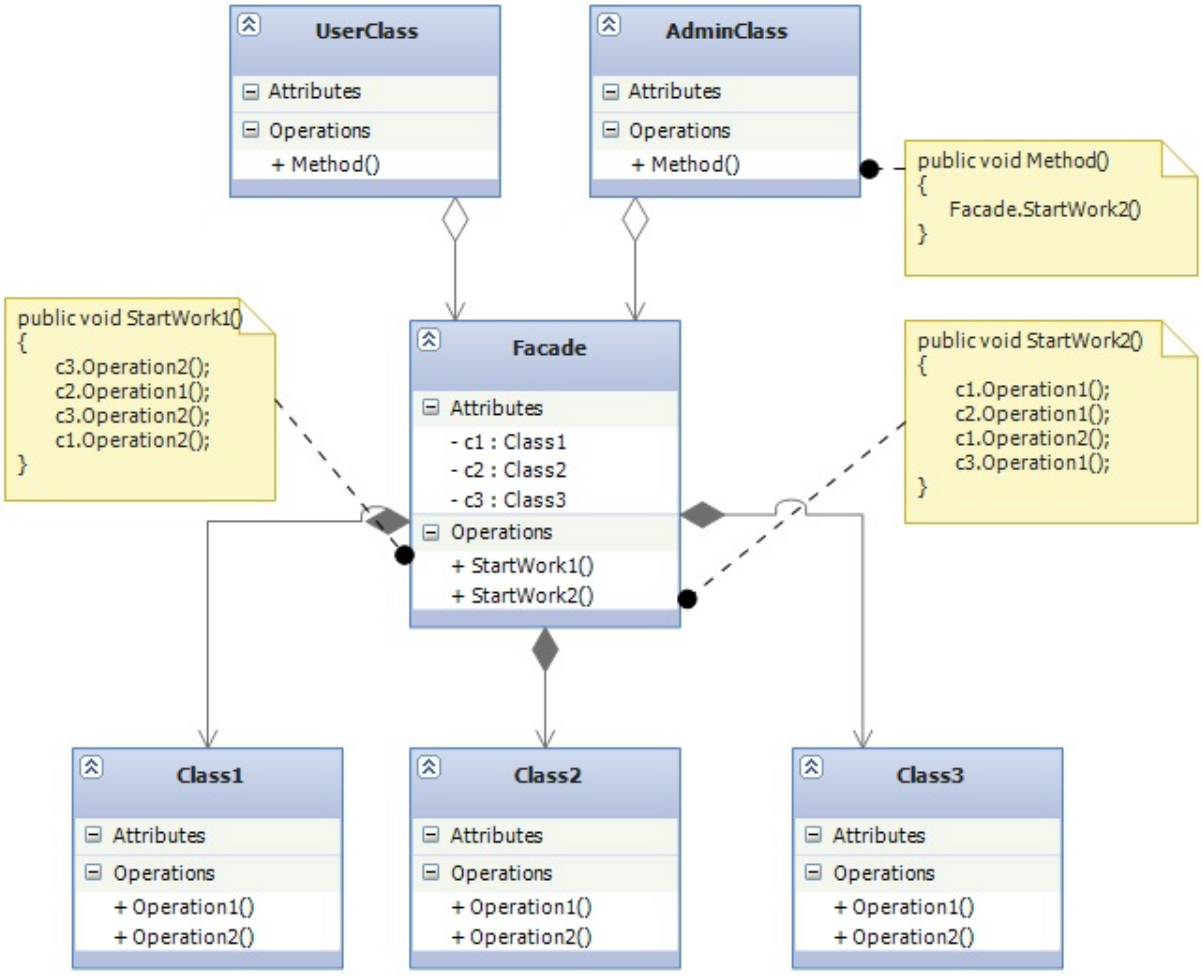
Каждая часть программы (подсистема) будет иметь свой класс фасад, через который будет производиться управление этой частью программы.

Причины возникновения паттерна

Спроектировать и потом построить огромное приложение «сразу» очень сложно, архитекторы решили эту задачу разбиением программы на более мелкие подсистемы, это позволяет строить какую-то одну часть программы не отвлекаясь на другие ее аспекты, после построения переходить к написанию следующей подсистемы. А если строить системы не зависимыми друг от друга можно получить «безопасную» заменимость ее компонентов или целых

подсистем, также, получая независимую подсистему, вы можете использовать ее повторно в других проектах, и наконец если в команде программистов более одного человека это позволит поручить каждому(или нескольким) из них написать отдельную подсистему, а потом просто собрать из в целое приложение.

Но для того чтоб воспользоваться незнакомой вам подсистемой приходилось перечитывать немало документации об этой структуре, и разбираться как ней пользоваться. Для решения этой проблемы, был создан паттерн Фасад(Facade). Который подразумевает, что для каждой подсистемы, будет создан свой класс, который является своего рода хранителем подсистемы. Через такой класс будет легко взаимодействовать со структурой.

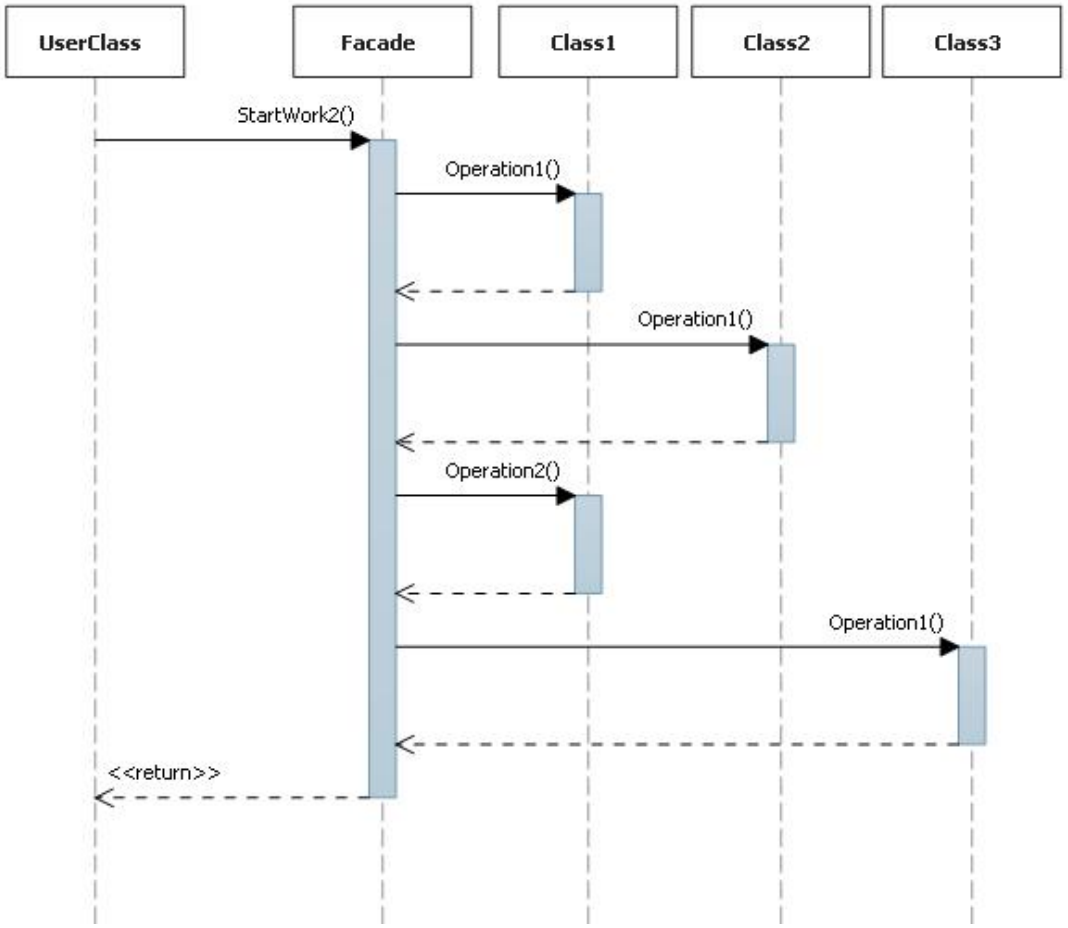


Структура паттерна

Паттерн фасад подразумевает существование некоторой подсистемы в программе, для которой создается класс фасад. Через класс фасада инициируется выполнение операций из классов подсистемы.

Стоит так же отметить то, что клиенты (пользователи подсистемы) не должны иметь доступа к классам подсистемы.

Все объекты подсистемы, если это возможно должны храниться в классе фасад. При вызове клиентами метода из объекта Facade, он начинает работу с классами подсистемы.



Описанный выше пример, не является правилом. В паттерне фасад могут иметься более или менее чем три класса внутри подсистемы. Объект класса фасад может не только вызывать методы из элементов подсистемы, он может

их настраивать, присваивать их свойствам значения и так далее.

Необязательно класс фасада должен хранить в себе объекты классов подсистемы.

Доступ к объектам подсистемы можно обустроить через паттерн Proxy. Также иногда программисты делают класс фасада статическим, для того чтоб к нему можно было обратится с любой точки программы.

Результаты использования паттерна

Строить программу, начиная с написания отдельных независимых подсистем проще, чем писать сразу всё приложение. Вдобавок, мы получаем готовый блок программы, который можно использовать и в других приложениях. Также можно произвести замену одного из компонентов подсистемы, не нарушая общей структуры приложения.

Если для каждой подсистемы создавать свой класс фасада, пользоваться такой системой будет проще, так как вся работа этой структуры инкапсулирована (спрятана) в классе фасад.

Используя другие паттерны вместе с паттерном Facade можно достичь большей производительности, гибкости, безопасности приложения.

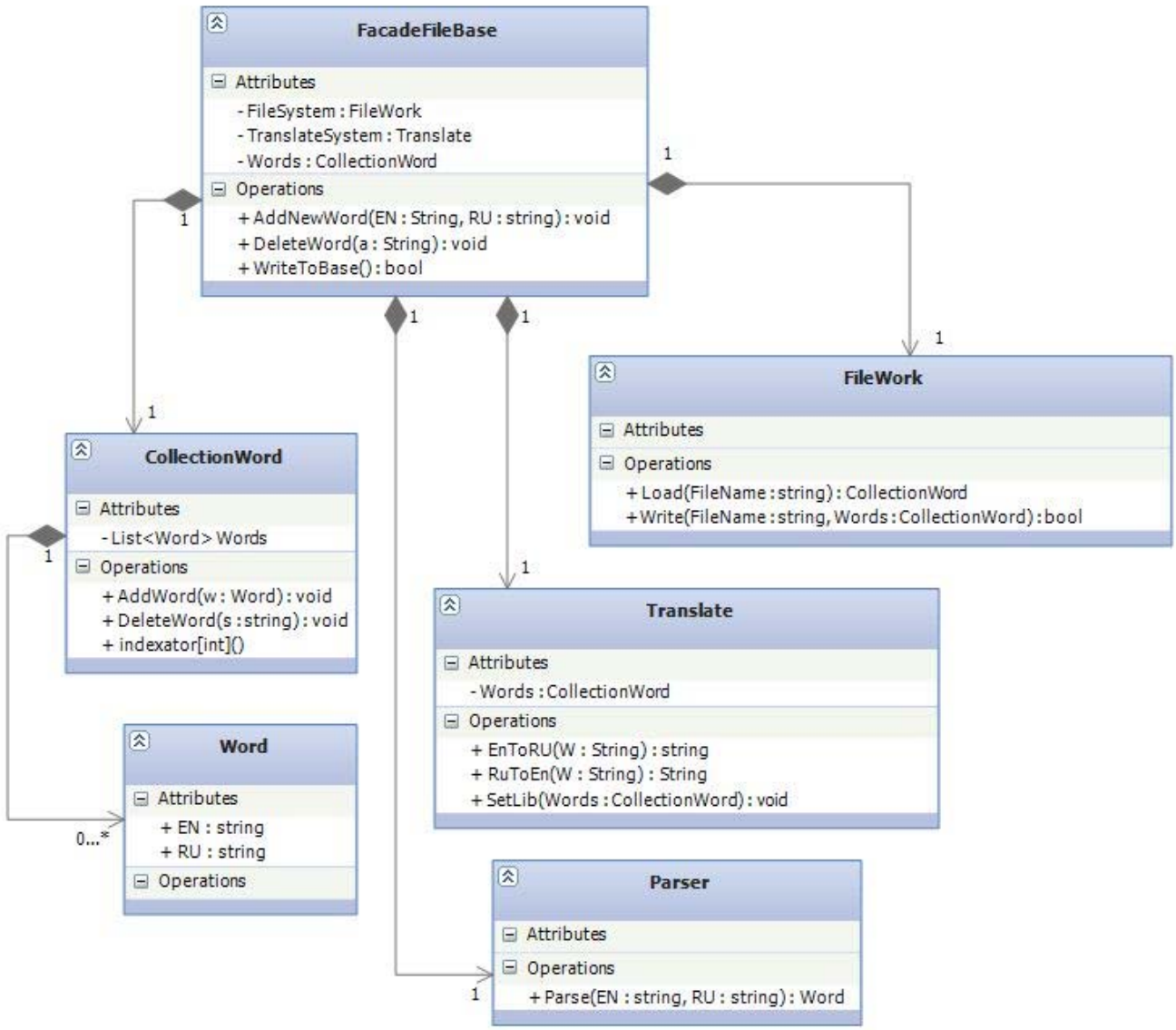
Практический пример

Рассмотрим паттерн фасад в примере следующего приложения: простой переводчик слов, пользователь вводит слово на английском и в результате получает его перевод на русском.

Приложение должно предоставлять возможность работать со словарем заранее записанных слов (сохранять, загружать) с файла. Также приложение должно уметь добавлять новые слова в текстовую базу данных.

В программе предусмотрена подсистема для работы со словарем, доступ к которой будет осуществятся через объект класса фасад.

Далее представлена диаграмма классов этого приложения.



Пользователь будет управлять системой через класс фасад(FacadeFileBase) который в свою очередь вызывает объекты подсистемы.

7. Паттерн Flyweight

Цель паттерна

Целью паттерна является более экономное использование памяти компьютера, достигается это более правильным способом работы с большим количеством мелких объектов.

Для понимания паттерна необходимо научиться разделять внутренние и внешние свойства объекта. Внутренне свойство объекта это свойство, которое хранить объект внутри себя, другими словами это экземпляр класса внутри которого создано свойство (переменная) и храниться до тех пор, пока «живет» объект. Внешнее свойство это свойство, которое передается объекту как аргумент одного (или нескольких) метода, и существует, пока выполняется метод.

Суть паттерна заключается в том чтобы «переписать» внутренние свойства во внешние, и после не создавать много объектов, а создать один который будет «отображаться» по разному, в зависимости от того какие внешние свойства к нему будут применены.

Причины возникновения паттерна

После «перехода» на ООП стало проще программировать так как все объекты «как настоящие», но это сопряжено с некоторым количеством трудностей. Создавая модель приложения, хочется расположить объекты по принципам объектно-ориентированного программирования, например стул состоит из ножек и сидения, а сидение состоит еще из нескольких деталей. Если описывать такую архитектуру слишком «глубоко» может возникнуть ситуация когда объектов будет слишком много, от чего приложение с такой архитектурой будет «тормозить». Что же делать когда необходимый уровень «глубины» недостижим для компьютера? Ответ: разделить логическое понимание приложение от физического, вить в отличии от реальной жизни в

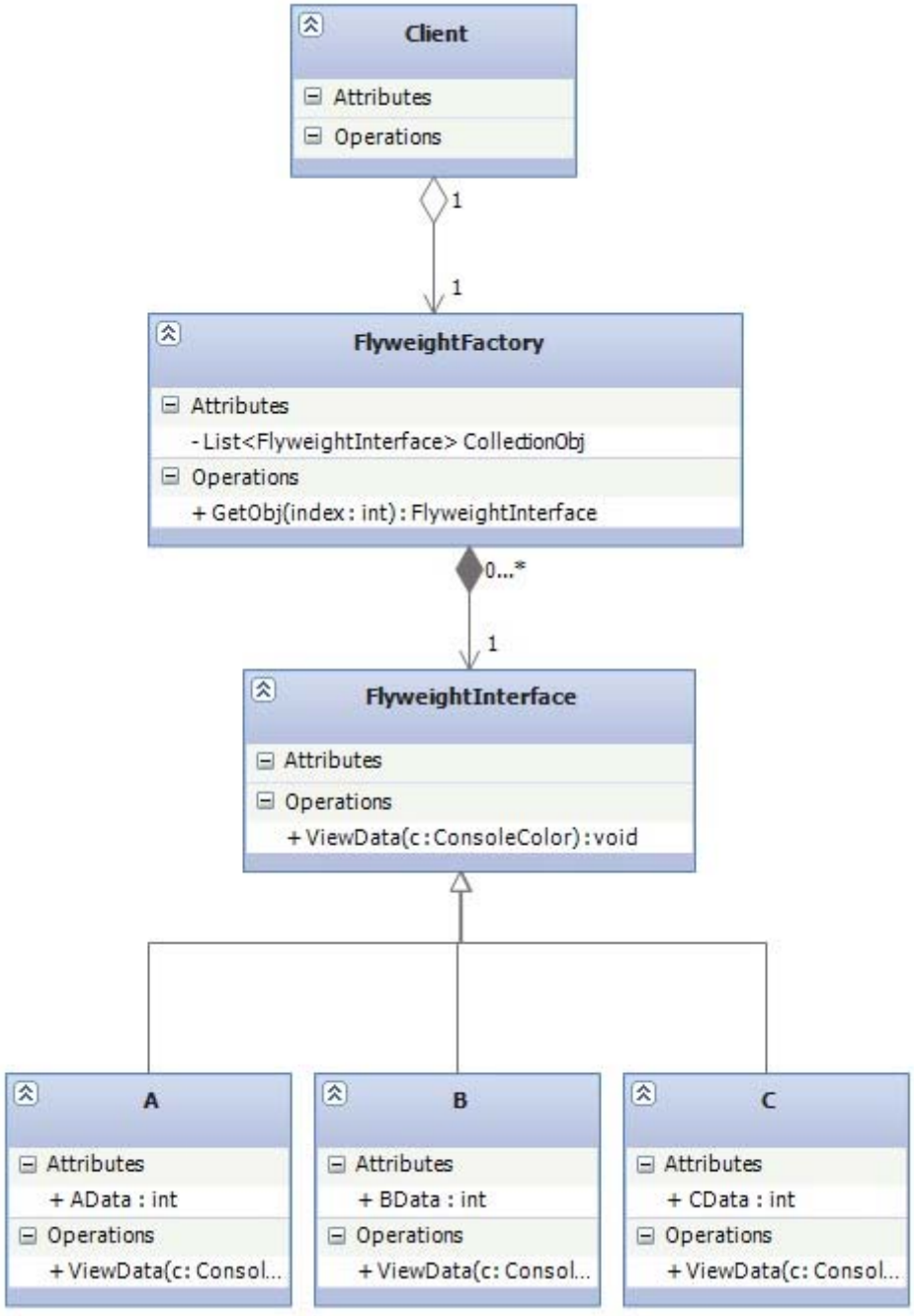
программировании можно встретить моменты когда одна и та же сущность находиться «рисуется» в двух (или более) местах. Так например Логически у нас есть стул, состоящий из ножек и сидения, а физически в программе создан объект ножки которая фигурирует в разных местах. Для того чтобы объект фигурировал в разных местах достаточно просто при вызове из объекта метода «отобразить» передавать в него, некоторое количество параметров которые и будут указывать где и как отображать этот элемент. Используя такой подход к построению архитектуры программист получит приложение которое будет достаточно экономно использовать ресурсы компьютера.

Структура паттерна

Объект который «фигурирует» в нескольких местах одновременно называется приспособленцем. Доступ к таким объектам предоставляется посредством фабрики. Клиент создает объекты приспособленцев с помощью фабрики, вызывая метод, который возвращает объект приспособленца и принимает идентификатор объекта, который ему необходимо вернуть.

Фабрика хранит в себе (коллекцию, массив, отдельно) объектов приспособленцев. Важно что объекты должны создаваться при первом запросе клиента, а при последующих запросах им должен возвращаться уже созданный объект который как было сказано выше должен храниться внутри фабрики.

При вызове метода GetObj пользователь передает в него идентификатор объекта, который ему необходимо получить, если объект с таким индексом уже создан, его необходимо вернуть как результат работы метода, но если объект не создан, его необходимо создать, и записать в хранилище фабрики, после чего вернуть как его как результат работы метода.



Далее для ознакомления представлен метод GetObj.

```
public IFlyweightInterface GetObj(int Index)
{
    // Берем объект с коллекции по индексу.
    IFlyweightInterface D = CollectionObj[Index] as IFlyweightInterface;

    // Проверка, если объект не существует.
    if (D == null)
    {
        // Создание объектов.
```

```
            switch (Index)
            {
                case 0:
                    D = new A();
                    break;
                case 1:
                    D = new B();
                    break;
                case 2:
                    D = new C();
                    break;
            }
            // После создание добавляем объект в коллекция
            CollectionObj.Add(D, Index);
        }

        // Возвращаем полученный из коллеции или только что созданный объект.
        return D;
    }
```

Классы A,B,C наследуются от базового класса, который описывает интерфейс взаимодействия со всеми типами приспособленцев.

Результаты использования паттерна

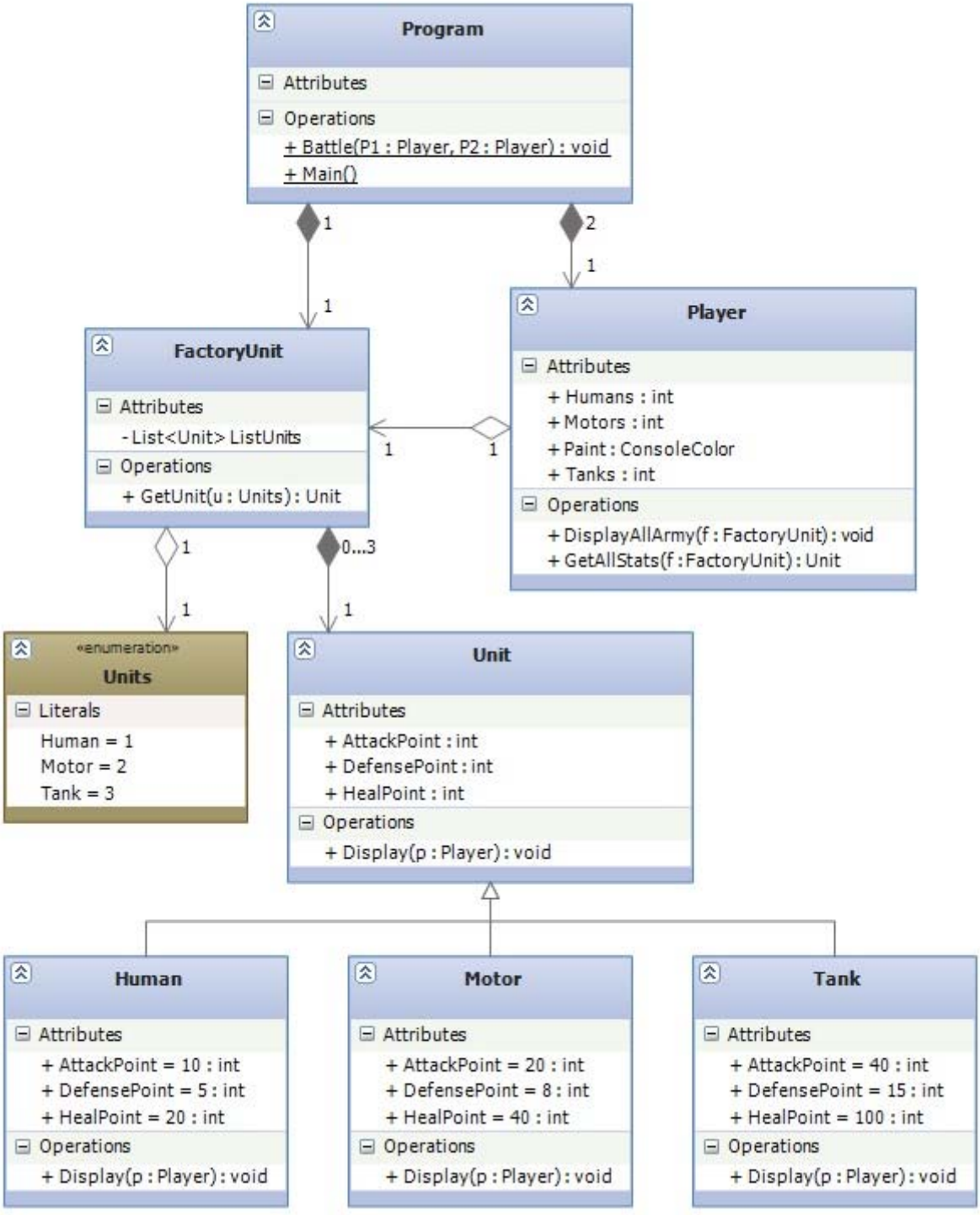
Если применять этот паттерн к своему приложению, вы получите действительно сильную экономию памяти, даже учитывая то, что при получении доступа к объекту будет теряться время в методе фабрики (в нашем случае GetObj). Но экономия памяти получится лишь в том случае, если в объектах по минимуму будут содержаться внутренние свойства, а при вызове будут передаваться внешние. Внешние свойства должны вычисляться алгоритмами, в этом и состоит основная идея паттерна, «экономить память за счет вычислительной мощности».

Получать доступ к объектам приспособленцев стоит только через методы фабрики, так как это добавит гибкости и возможности настроить объект в методах фабрики.

Применять паттерн следует тогда когда в вашем приложении действительно большое количество объектов, и когда их состояние можно описать внешними свойствами, и внешние свойства можно описать и присвоить с помощью вычислений.

Практический пример

Рассмотрим консольное приложение, которое будет отображать, и просчитывать итог сражения двух армий.



Классы Human, Motor и Tank, отличаются друг от друга только значениями внутренних свойств, и методом, отображающим на экране боевую единицу.

Объект класса FactoryUnit будет хранить в себе от нуля до трех объектов в виде Unit в специально для этого созданной коллекции ListUnits. Классы клиентов смогу получить доступ, к объекту вызвав из фабрики метод и передав в него объект перечисления который и укажет какой конкретно Unit надо вернуть как результат вызова метода.

Класс игрока(Player) будет использовать фабрику для получения и отображения свойств объектов. Но пользоваться все объекты игроков будут на самом деле только 3-мяэкземплярами классов потомков Unit это и должно экономить память.

Основной класс Program содержит метод Main, в котором происходит управление всей программой, а так же метод Battle отвечающий за математику которая покажет какой был исход битвы армий двух игроков.

8. Паттерн Proxy

Паттерн Proxy часто именую как паттерн суррогат. Далее основной класс, о котором будет идти речь, именуется суррогатом, или прокси, оба термина являются верными и описывают один и тот же класс / объект.

Прокси это – некоторый объект, обеспечивающий транзитный доступ к другому объекту.

Суррогат это – некоторый объект, который внешне ничем не отличается от основного, но внутренней функциональности в нем нет.

Объект Суррогата или прокси внешне ничем не отличается от объекта, который он представляет, но вся его внутренняя функциональность лишь является некоторым туннелем, через который классы клиента получают доступ к основным объектам.

Так же объект суррогата или прокси можно называть заместителем объекта.

Цель паттерна

Целью паттерна является создание системы доступа к объекту через специальный объект суррогата.

Это позволяет добиться большей гибкости работы с объектом целевого класса.

Если объект целевого класса после создания будет занимать много места и нет гарантии того что объект будет необходим в работе приложения, можно создавать целевой объект только когда клиент первый раз вызывает метод из объекта суррогата это ускорит быстродействие приложения в случае если объект не будет использоваться. А если объект и будет создан, то на этапе выполнения программы это будет не так «болезненно» для пользователя.

Такая система доступа позволит так же использовать целевой класс с большей защитой (в суррогате можно выполнять различные проверки принимаемых параметров). Например, если целевой класс это калькулятор, просто принимает в аргументы метода два параметра и выполняет базовые математические операции, в суррогате класса калькулятор можно выполнить проверку чтобы не получилось, что калькулятору будет передана задача, «поделить на нуль».

Можно, использовать структуру паттерна прокси для создания так называемого «Посла» в другое пространство имен. Например, целевой класс описан в пространстве имен, которое по каким-то причинам не возможно (или неудобно) подключить, в таком случае можно создать суррогат, который будет находиться в «нужном» пространстве имен, а внутри суррогата будет создан объект целевого класса из другого пространства имен.

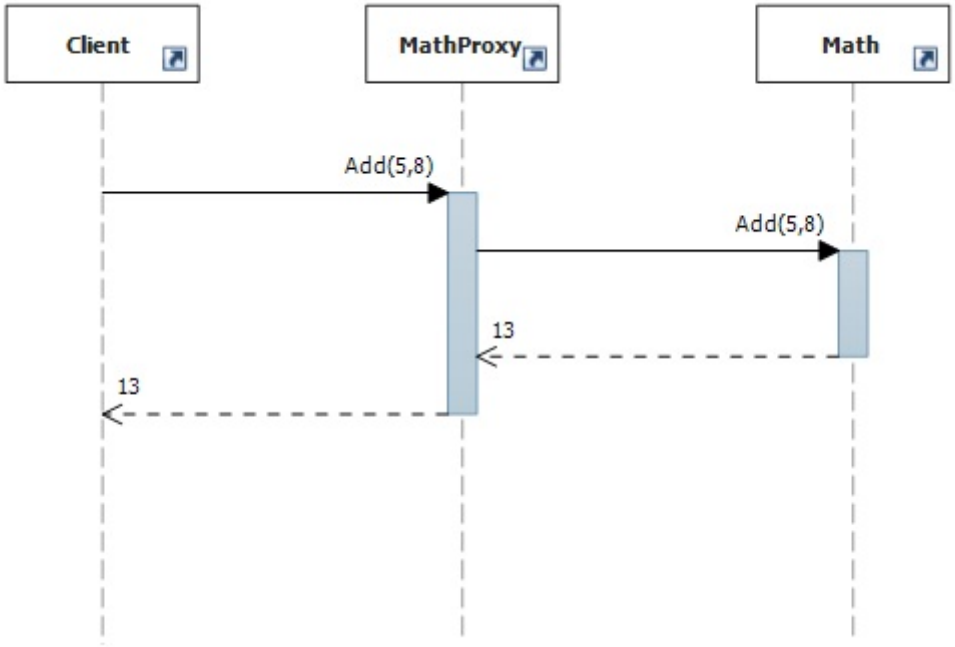
Причины возникновения паттерна

Код одних программистов должен быть понятен другим программистам, касательно как синтаксических «традиций» так и структурных. Если строить приложение, используя паттерн прокси для доступа к объектам и в объектах суррогатов заниматься проверкой данных, а в самом целевом классе выполнять логику. Потом достаточно легко объяснить другому программисту, где выполняются проверки, а где основная логика. Вообще используя паттерны довольно легко объяснить структуру приложения.

Наверное, вам уже приходилось видеть приложения, которые после начала запуска, очень долго «думают» и только по истечении 5 или более секунд начинают отображать свой интерфейс. Так получается, потому что приложение сразу при старте начинает загружать все свои модули, создает все экземпляры классов, которые понадобятся (или не понадобятся) в работе приложения. Для частичного решения такой проблемы, можно создать суррогат «тяжелого» объекта. Создавать такой объект при старте приложения, суррогат это лишь муляж который не является таким «тяжелым» как целевой объект и потому создается быстрее, а «тяжелый» объект будет создан только, тогда когда он будет действительно необходим. А необходимость появляется, когда классы клиента будут работать с объектом суррогата, который все запросы будет перенаправлять на только что созданный целевой объект.

Структура паттерна

Паттерн Proxy подразумевает наличие некоторого объекта(далее «целевой класс»), к которому будет осуществляться доступ с помощью специально созданного объекта суррогата.



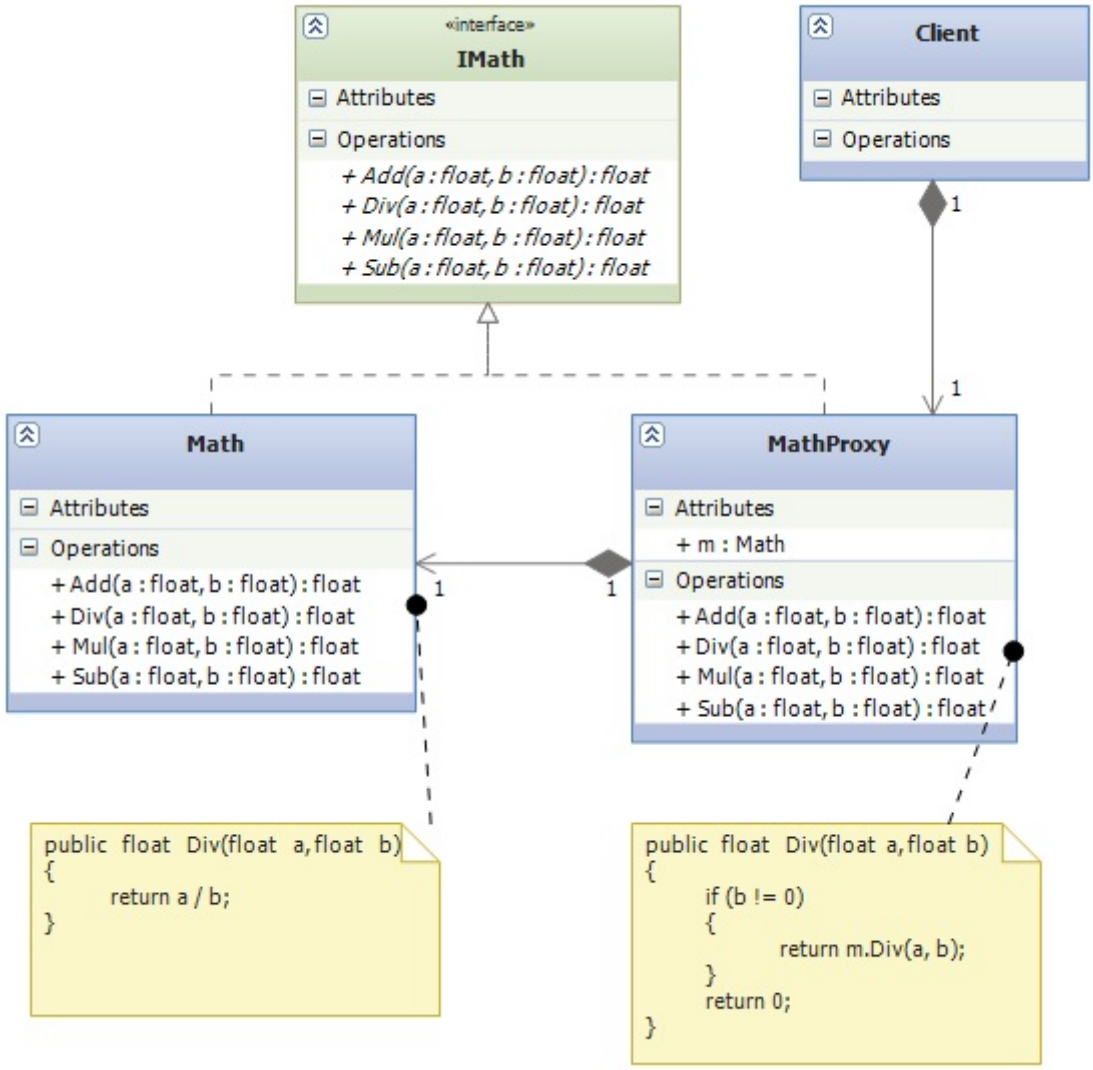
Но объект суррогата не должен отличаться внешне от объекта целевого класса, потому оба класса должны быть унаследованы от одного интерфейса.

В приведённом далее примере целевой объект Math создается сразу с объектом суррогата, так как класс math не является «тяжелым» и будет создаваться быстро.

Пример показывает, как с помощью объекта суррогата можно выполнять различные проверки перед непосредственной передачей данных целевому классу.

Результаты использования паттерна

Реализовав доступ к паттерну через объект суррогата, мы получаем дополнительный уровень работы с целевым объектом. На этом уровне мы можем заниматься оптимизацией, создавая целевые объекты только по мере того как они будут использоваться. Можно заниматься проверкой данных перед передачей в целевой объект. В объекте суррогата можно даже вести логированние доступа к объекту (например, высчитывать сколько раз был вызван метод суммирования на калькуляторе).

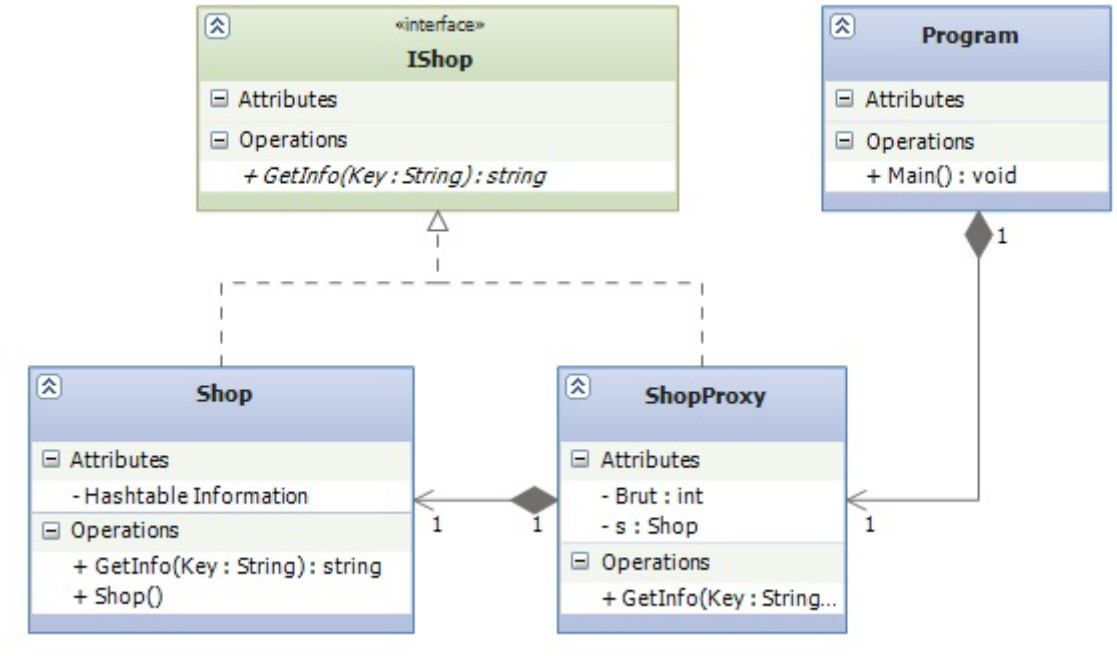


Практический пример

Для лучшего понимания паттерна ознакомимся со структурой приложения «магазин информации». В архитектуре приложения имеется класс Shop к которому осуществляется доступ через класс ProxyShop. Касс Shop имеет метод который принимает строку с ключом, а потом если возможно возвращает строку с информацией.

Класс ProxyShop защищает класс Shop от возможности его «брутфорсить» также объект Shop создается только тогда когда он будет необходим в программе, другими словами тогда когда кто-то попытается получить информацию по ключу.

Ниже представлена диаграмма классов этого приложения.



метод GetInfo из ShopProxy выполняет создание объекта и контроль доступа.

```
public string GetInfo(string Key)
{
    if (Brut < 3)
    {
        if (s == null)
        {
            s = new Shop();
        }

        string Ret = s.GetInfo(Key);

        if (Ret == "null")
        {
            Brut++;
            return "Код неверный ошибка номер: " + Brut;
        }
        Brut = 0;
        return Ret;
    }
    return "У вас нет доступа";

}
```

А метод GetInfo из Shop просто вовращает значение по ключу.

```
public string GetInfo(string Key)
{
    String Resoult = Information[Key] as string;
    if (Resoult == null)
    {
        return "null";
    }
    else
    {
        return Resoult;
    }
}
```

9. Анализ и сравнение структурных паттернов

Вы уже изучили все из структурных паттернов:

- Adapter — паттерн который позволяет «адаптировать» объект под другой интерфейс, для доступа к нему. Например в объекте имеется метод Operation1 а нам необходимо сделать так чтоб он назывался OperationA. Целью паттерна Adapter является «редактирование» интерфейса доступа к целевому объекту.
- Bridge — паттерн позволяет отделить абстракцию от реализации, когда имеется иерархия объектов которая описана абстрактно и позже будет реализоваться иерархия под конкретную систему. Целью паттерна Bridge является построение отдельно абстракции и реализации и предоставление клиенту абстракции с помощью, которой он сможет управлять реализацией.
- Composite — структурный паттерн который выстраивает объекты по типу дерева. Целью этого паттерна является построение древовидной структуры для хранения объектов.
- Decorator — паттерн позволяющий структурировать таким образом что несколько объектов отображаются как один. И количество внутренних объектов может изменяться «на лету».
- Facade — структурный паттерн который рассказывает как строить

большие приложения из мелких не зависимых подсистем. Целью паттерна является создание узкого и понятного интерфейса для работы с подсистемой.

- Flyweight — паттерн позволяющий экономить место в случае если в программе имеется множество мелких объектов и их состояние можно вынести во внешние свойства которые вычисляются алгоритмами.
- Proxy — подразумевает создание объекта суррогата для целевого объекта чем может обеспечить большую гибкость, экономию памяти, защиту.

Подходить к использованию паттернов нужно отталкиваясь от их назначения, потому что с первого взгляда может показаться что паттерны многим похожи, но так кажется только потому что у некоторых из них похожая структура. Но у каждого паттерна есть цель использования. Например, вам может показаться паттерны фасад и прокси, очень похожи, но предназначены они для разных целей, цель паттерна фасад, создать узкий интерфейс работы с подсистемой, а прокси позволяет защитить целевой объект.

Структурно паттерны Composite и Decorator тоже очень похожи, но необходимо отталкиваться от целей применения, так как паттерн Composite структурирует объекты, а паттерн Decorator позволяет создать отдельные виды функциональности и создать из них «один» объект.

Часто приходится видеть, как люди не видят разницы между паттерном фасад(Facade) и паттерном адаптер(Adapter) потому что кажется, что цель у них одна и так же, это изменить или модифицировать интерфейс доступа к объекту или объектам. Но адаптер позволяет работать со старым интерфейсом доступа к объекту и использовать новый, а паттерн Façade создает новый интерфейс, более целенаправленный.

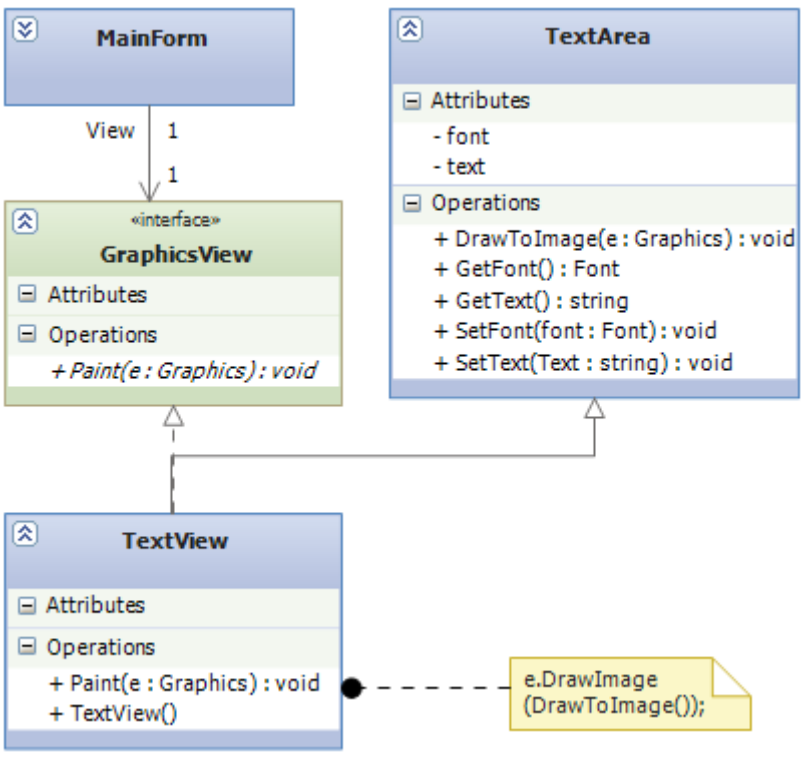
Важно отметить тот факт что паттерн это не эталон структуры программы, на практике приходится слышать, «ну вот у нас здесь архитектура как в патерне Flyweight но в фабрике разные типы объектов возвращаются из разных методов».

10. Практические примеры использования структурных паттернов

В качестве первого примера мы рассмотрим приложение, использующее структурные паттерны Adapter и Decorator. Приложение будет представлять собой текстовый редактор, с возможностью открытия текстовых файлов. Но оно не будет использовать элементы управления, предназначенные для редактирования текста, предоставляемые стандартной системой типов .NET Framework (такие как TextBox и RichTextBox). Собственно назначение практического примера состоит в том, чтобы проиллюстрировать подход к организации элемента управления, выполняющего редактирование текста.

Паттерн адаптер будет использоваться для того, чтобы связать класс, выполняющий управление текстом (под управлением понимается получение, хранение и редактирование) с некоторым графическим контекстом – класс TextArea (англ. Text area – текстовая область). А обязанности по прорисовке текста возложить на отдельный класс, названный нами TextView (англ. Text view – текстовое представление).

Модель связывания текстовой области с графически контекстом при использовании паттерна адаптер представлена на диаграмме справа.



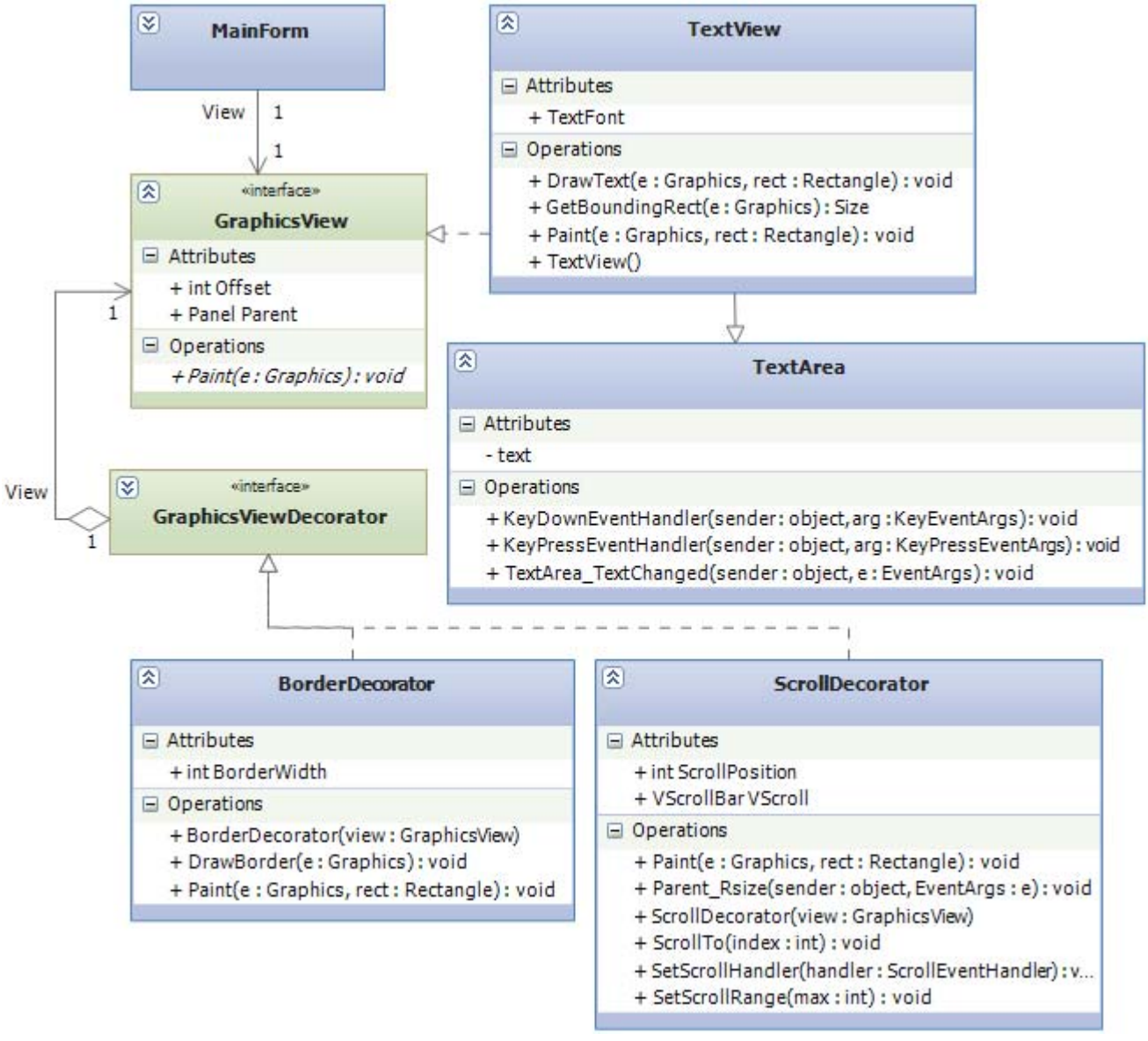
Объявляется интерфейс GraphicsView, который определяет стандартный интерфейс элемента, имеющего графическое представление.

Класс TextView наследует, с одной стороны, GraphicsView, а

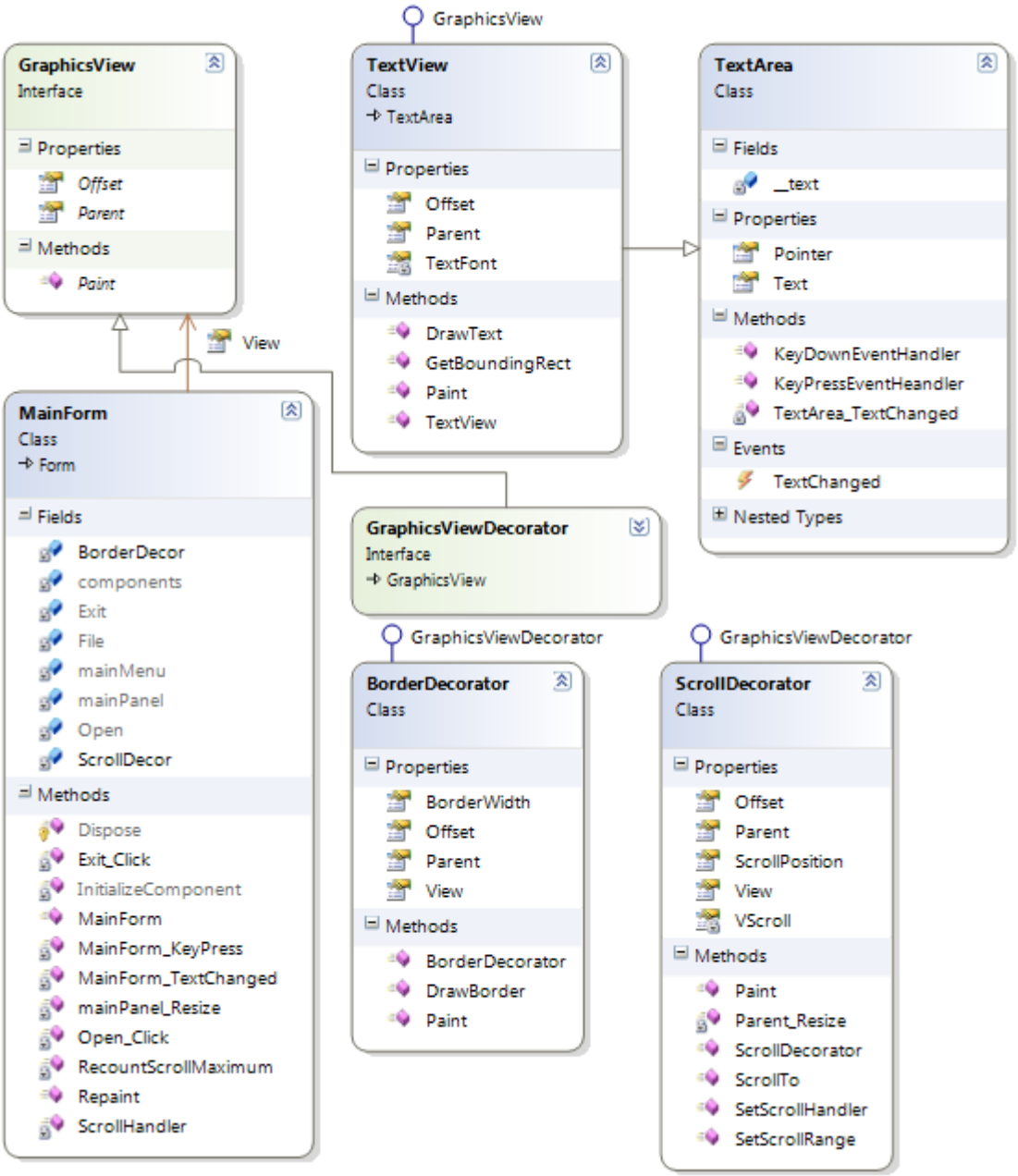
с другой – TextArea.

Далее, для оформления элемента управления мы используем паттерн декоратор, в частности нами будем определено два декоратора: первый будет ответственен за то, чтобы прорисовывать рамку вокруг текстовой области, а второй – за то, чтобы добавлять полосу прокрутки к текстовой области и выполнять прорисовку текстового представления с учётом смещения полосы прокрутки.

Общая модель представлена ниже.



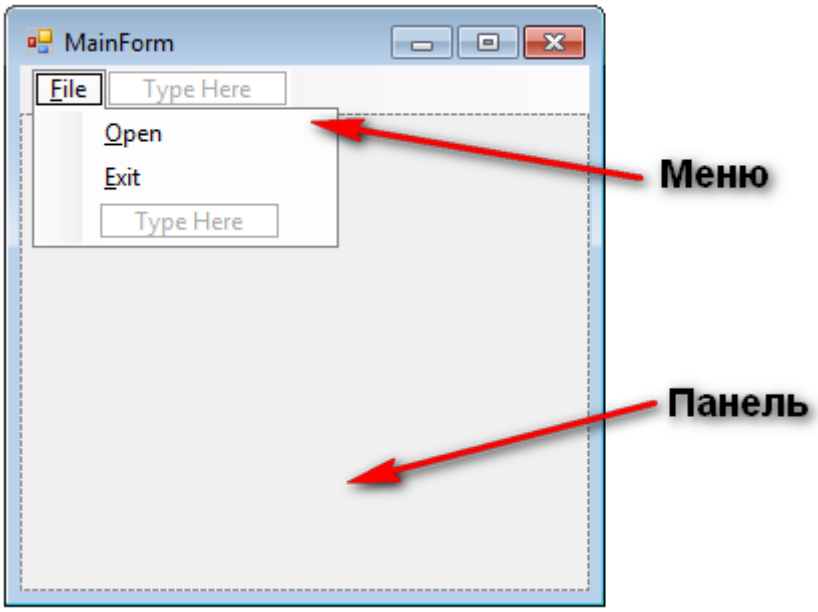
После объявления классов в соответствии с определённой выше моделью, мы получим следующую структуру классов, которая представлена на диаграмме классов ниже.



Следует учитывать, что модель не всегда полностью согласуется с реализацией, и поэтому между UML-диаграммой и диаграммой классов готового проекта может быть существенная разница. Но общий подход останется неизменным.

Для того, чтобы добавить диаграмму классов в проект, необходимо в окне

SolutionExplorer вызвать контекстное меню проекта и выбрать создание диаграммы классов.



Форму необходимо оформить следующим образом: добавить меню и панель, свойство Dock, которой необходимо установить в значение Fill.

Должно получиться нечто вроде того, что представлено на рисунке слева.

Далее будет представлен код приложения по классам, представленным на диаграмме классов.

Ниже представлен интерфейс графического элемента, или элемента, который может быть прорисован на графическом контексте.

```
public interface GraphicsView
{
    Panel Parent { get; set; }
    int Offset { get; set; }
    void Paint(System.Drawing.Graphics e, Rectangle updateRectangle);
}
```

Интерфейс GraphicViewDecorator наследует графическую область и отличается только тем, что определяет свойство View типа GraphicsView, поскольку декоратор должен иметь ссылку на декорируемое представление.

```
public interface GraphicsViewDecorator : GraphicsView
{
    GraphicsView View { get; set; }
}
```

Далее следует описание текстовой области, которая имеет методы, для

обработки добавления текста в поле __text, которое осуществляется через обработчик оконного события KeyPressed. Обработчик события KeyDown необходим для того, чтобы обрабатывать перемещение внутреннего указателя текстовой области путём нажатия кнопок Left и Right.

```
public class TextArea
{
    public delegate void TextChangedHandler(object sender, EventArgs e);
    public event TextChangedHandler TextChanged =
                          new TextChangedHandler(TextArea_TextChanged);
    string __text = "";
    public int Pointer { get; set; }
    public string Text
    {
        get
        {
            return __text;
        }
        set
        {
            __text = value;
            TextChanged(this, new EventArgs());
        }
    }
    static void TextArea_TextChanged(object sender, EventArgs e)
    {
    }
    public virtual void KeyPressEventHeandler(
                                object sender,
                                KeyPressEventArgs arg)
    {
        if ((int)arg.KeyChar == 8)
        {
            if (Text.Length > 0)
            {
                Text = Text.Remove(Pointer-1, 1);
                Pointer--;
            }
        }
        else if (!Char.IsControl(arg.KeyChar))
        {
            Text = Text.Insert(Pointer++, arg.KeyChar.ToString());
        }
    }
    public void KeyDownEventHandler(object sender, KeyEventArgs arg)
    {
        switch (arg.KeyData)
        {
            case Keys.Left:
                if (Pointer > 0)
                    Pointer--;
                break;
            case Keys.Right:
```

```
                    if (Pointer < Text.Length)
                        Pointer++;
                    break;
            }
        }
    }
```

Класс TextView является классом-адаптером и имеет описание метода прорисовки текста на графическом контексте, а так же метод Paint, которы в последствии будет вызываться декоратором.

```
    public class TextView : TextArea, GraphicsView
    {
        public Panel Parent { get; set; }
        public int Offset { get; set; }
        private Font TextFont { get; set; }
        public TextView(string Text, Panel parent)
            :base()
        {
            Parent = parent;
            this.Text = Text;
            TextFont = new Font("Verdana", 10, FontStyle.Regular);
        }
        public SizeF GetBoundingRect(Graphics e)
        {
            return e.MeasureString(
                        Text,
                        TextFont,
                        Parent.ClientRectangle.Width,
                        StringFormat.GenericDefault);
        }
        public void DrawText(Graphics e,Rectangle rect)
        {
            e.DrawString(
                        Text,
                        TextFont,
                        Brushes.Black,
                        new Rectangle(
                              rect.X,
                              rect.Y - Offset,
                              rect.Width,
                              rect.Height + Offset));
        }
        public void Paint(System.Drawing.Graphics e,Rectangle updateRectangle)
        {
            DrawText(e, updateRectangle);
        }
    }
```

Класс BorderDecorator предназначен для того, чтобы добавлять рамку к

графическому представлению (классу, наследующему GraphicsView). Он имеет для этого методы DrawBorder (который собственно и прорисовывает рамку на некотором графическом контексте, переданном в качестве параметра) и метод Paint, унаследованный от интерфейса GraphicsView. Метод Paint последовательно вызывает аналогичный метод агрегируемого графического представления (того представление, которое декорируется классом), и метод DrawBorder.

```
public class BorderDecorator : GraphicsViewDecorator
{
    public GraphicsView View { get; set; }
    public Panel Parent { get; set; }
    public int Offset { get; set; }
    public BorderDecorator(GraphicsView view)
    {
        View = view;
        Parent = View.Parent;
        BorderWidth = 2;
        Offset += BorderWidth;
    }
    public int BorderWidth { get; set; }
    public void Paint(System.Drawing.Graphics e, Rectangle rect)
    {
        View.Paint(e, View.Parent.ClientRectangle);
        DrawBorder(e);
    }
    public void DrawBorder(Graphics e)
    {
        e.DrawRectangle(
                    new Pen(Brushes.LightGray,BorderWidth),
                    Parent.ClientRectangle);
    }
}
```

Класс ScrollDecorator отвечает за добавление полосы прокрутки к оформляемому текстовому представлению, и за реакцию текстового представления на перемещение ползунка по полосе прокрутки, для чего зарегистрированы соответствующие обработчики событий. Также в класс добавлены объявления вспомогательных методов: метод SetScrollRange необходим для того, чтобы изменять максимальное значение полосы прокрутки при изменении объёма текста в декорируемом текстовом представлении, а метод SetScrollHandler – для установки пользовательских обработчиков на событие изменения положения ползунка на полосе прокрутки.

```
public class ScrollDecorator : GraphicsViewDecorator
{
    VScrollBar VScroll { get; set; }
    public GraphicsView View { get; set; }
    public Panel Parent { get; set; }
    public int Offset { get; set; }
    public int ScrollPosition { get; set; }
    public ScrollDecorator(GraphicsView view)
    {
        View = view;
        Parent = View.Parent;
        VScroll = new VScrollBar();
        Parent.Controls.Add(VScroll);
        Parent.Resize += Parent_Resize;
        Parent_Resize(this, new EventArgs());
    }
    public void SetScrollHandler(ScrollEventHandler handler)
    {
        VScroll.Scroll += handler;
    }

    void Parent_Resize(object sender, EventArgs e)
    {
        VScroll.Location =
              new Point(
                    Parent.ClientRectangle.Width - VScroll.Width - View.Offset,
                    View.Offset);
        VScroll.Height = Parent.ClientRectangle.Height - View.Offset*2;
    }
    public void ScrollTo(int location)
    {
        ScrollPosition = location;
        Paint(View.Parent.CreateGraphics(),View.Parent.ClientRectangle);
    }
    public void SetScrollRange(int max)
    {
        VScroll.Maximum = max;
    }
    public void Paint(System.Drawing.Graphics e, Rectangle rect)
    {
        View.Paint(e, View.Parent.ClientRectangle);
    }
}
```

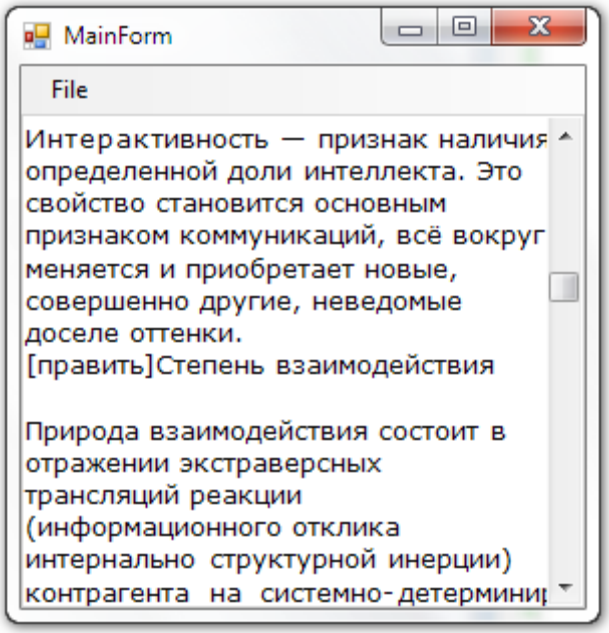
Далее представлен текст класса главной формы приложения. В нём добавлено поведение вспомогательного пользовательского интерфейса (обработчики нажатия на пункты меню окна), а также добавлен обработчик события изменения текста в текстовом представлении, реакцией на которое становиться изменение максимального значения полосы прокрутки. Этот обработчик вынесен в текст класса MainForm, постольку ScrollDecorator может

быть потенциально прикреплён к любому представлению, а это действие зависимо от контекста применения.

```
public partial class MainForm : Form
{
    public GraphicsView View { get; set; }
    BorderDecorator BorderDecor;
    ScrollDecorator ScrollDecor;
    public MainForm()
    {
        InitializeComponent();
        this.BackColor = mainPanel.BackColor = Color.White;
        View = new TextView("Hello world", mainPanel);
        this.KeyPress += (View as TextView).KeyPressEventHeandler;
        this.KeyDown += (View as TextView).KeyDownEventHandler;
        this.KeyPress += MainForm_KeyPress;
        BorderDecor = new BorderDecorator(View);
        ScrollDecor = new ScrollDecorator(BorderDecor);
        (View as TextArea).TextChanged += MainForm_TextChanged;
        ScrollDecor.SetScrollHandler(ScrollHandler);
    }
    void ScrollHandler(object sender, ScrollEventArgs e)
    {
        View.Offset = e.NewValue;
        Repaint(this, new PaintEventArgs(
                    mainPanel.CreateGraphics(),
                    mainPanel.ClientRectangle));
    }
    private void RecountScrollMaximum()
    {
        int max = (int)(View as TextView).GetBoundingRect(
                                        mainPanel.CreateGraphics()).Height;
        ScrollDecor.SetScrollRange(max);
        if (ScrollDecor.ScrollPosition > max)
            ScrollDecor.ScrollTo(max);
    }
    void MainForm_TextChanged(object sender, EventArgs e)
    {
        RecountScrollMaximum();
    }
    void MainForm_KeyPress(object sender, KeyPressEventArgs e)
    {
        Repaint(
              mainPanel,
              new PaintEventArgs(
                    mainPanel.CreateGraphics(),
                    mainPanel.ClientRectangle));
    }
    public void Repaint(object sender, PaintEventArgs e)
    {
        Image img = new Bitmap(this.ClientSize.Width, this.ClientSize.Height);
        Graphics DC = Graphics.FromImage(img);
        DC.Clear(BackColor);
        BorderDecor.Paint(DC, ClientRectangle);
        e.Graphics.DrawImage(img, 0,0);
```

```
            DC.Dispose();
            img.Dispose();
        }
        private void mainPanel_Resize(object sender, EventArgs e)
        {
            Repaint(
                  mainPanel,
                  new PaintEventArgs(
                        mainPanel.CreateGraphics(),
                        mainPanel.ClientRectangle));
        }

        private void Open_Click(object sender, EventArgs e)
        {
            OpenFileDialog dlg = new OpenFileDialog();
            dlg.Multiselect = false;
            dlg.Filter = "Text files|*.txt";
            if (dlg.ShowDialog() == DialogResult.OK)
            {
                (View as TextView).Text = System.IO.File.ReadAllText(
                                                        dlg.FileName,
                                                        Encoding.Default);
                RecountScrollMaximum();
                mainPanel.Invalidate();
            }
        }
        private void Exit_Click(object sender, EventArgs e)
        {
            Application.Exit();
        }
```

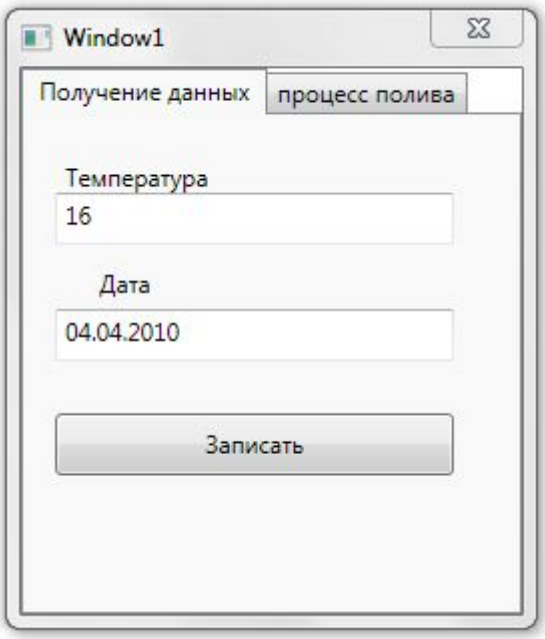
В результате успешного создания кода должно получиться приложение, выглядящее приблизительно так, как показано на изображении справа.

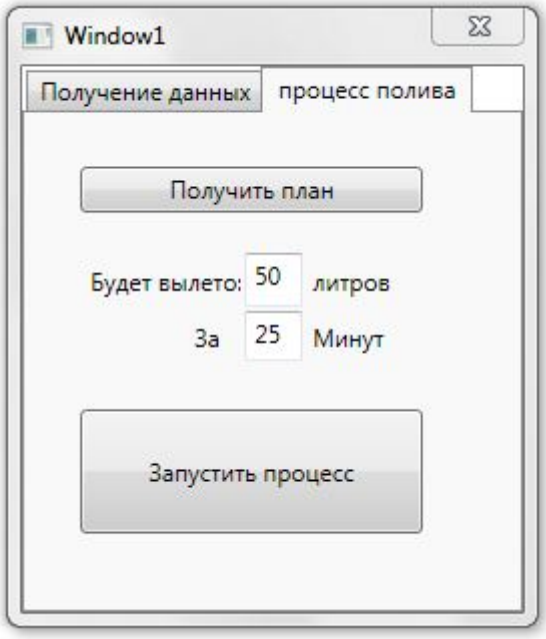


Необходимо отметить тот факт, что паттерны сами по себе не «дают серьёзной выгоды». Секрет успешной и эффективной разработки состоит в том, чтобы комбинировать паттерны и применять их там, где их применение способно принести плоды.

Программа слежения за климатом в теплице

Данное приложение использует два паттерна, это паттерн прокси для создания суррогата доступа к классу который занимается поливом (где должна находиться работа с «железом») и паттерн фасад создающий объект узкого доступа к подсистеме которая занимается вычислениями «при какой температуре, сколько воды наливать».

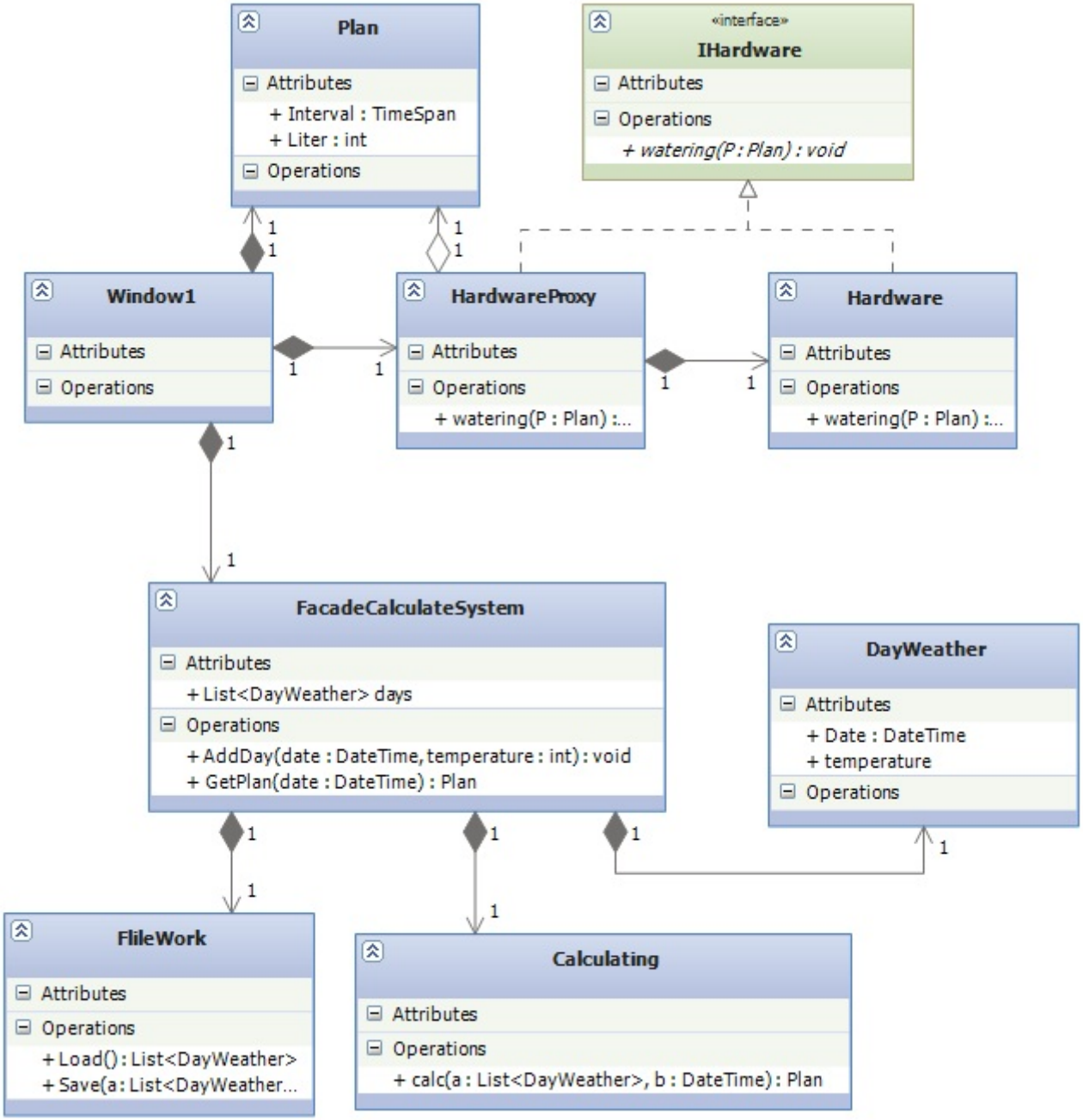




Рассмотрим все классы:

- Window1 – Основное окно приложения через которое будет происходить управление программой.
- Plan — объект такого класса описывает сколько воды необходимо вылить растениям, и за какое время.
- Ihardware – интерфейс в котором имеется метод watering который принимает в аргументы объект класса Plan. С помощью этого метода мы будем запускать процесс поливания растений.
- Hardware – Класс который отвечает за работу с устройствами которые выполняют полив растений. Так как у нас таких устройств нет мы будет видеть MessageBox.

- ProxyHardware — Класс который выполняет транзит команд к классу занимающемуся работой с устройствами. Объект HardwareProxy занимается проверкой условий.



```
        Hardware hard = null;
            public void Watering(Plan a)
            {
                if (hard == null)
                {
                    hard = new Hardware();
                }

                if ((a.Litters / a.Interval.TotalMinutes) < 1)
                {
                    MessageBox.Show("Ошибка: слишком маленькое давление");
                }
                else if ((a.Litters / a.Interval.TotalMinutes) > 4)
                {
                    MessageBox.Show("Ошибка: слишком большое давление");
                }
                else
                {
                    // Вызов работы с оборудованием.
                    hard.Watering(a);
                }

            }
```

- DayWeather — специальный класс объекты которого хранят данные о погоде и времени когда эта погода была записана.
- FileWork — Класс который инкапсулирует в себе всю работу с записью или загрузкой данных о погоде на жесткий диск.
- Calculating — Объект этого класса будет заниматься порождением объектов Plan исходя из коллекции объектов которые содержат данные о погоде за каждый день.

```
    /// <summary>
    /// Подсчет плана по трем минувшим дням
    /// </summary>
    /// <param name="a">Коллекция Минувших денй с погодой.</param>
    /// <param name="date">Сегодняшние время.</param>
    /// <returns>План для полива</returns>
    public Plan Calc(List<DayWeather> a, DateTime date)
    {
        for (int i = 0; i < a.Count; i++)
        {
            if (a[i].Date == date && i>2)
            {
                int Dt = a[i].Temperature +
                    a[i - 1].Temperature +
                    a[i - 2].Temperature;
                Dt = Dt / 3;
                int tim = Dt / 2;
```

```
                Plan P= new Plan();
                P.Interval = new TimeSpan(0, 0, tim, 0);
                P.Litters = Dt;
                return P;
            }

        }
        MessageBox.Show("Ошибка прогнозировавния");
        Plan P2 = new Plan();
        P2.Litters = 2;
        P2.Interval = new TimeSpan(0, 0, 1, 0);
        return P2;
    }
```

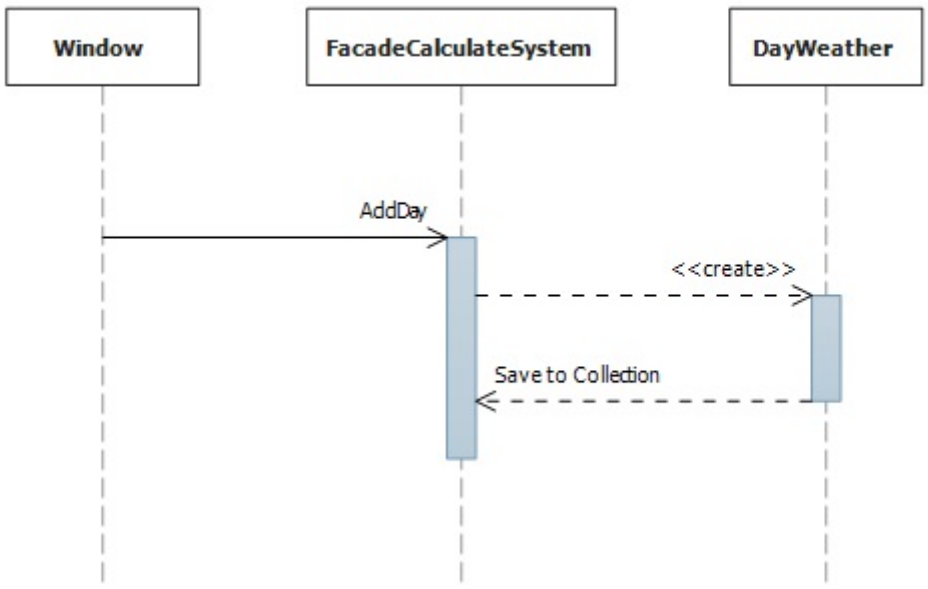
- FacadeCalculateSystem — Класс предоставляющий направленный интерфейс работы с системой хранения и подсчета данных о климате.

```
class FacadeCalculateSystem
{
    private List<DayWeather> Days;
    private FileWork filework;
    private Calculating Calul;

    public FacadeCalculateSystem()
    {
        filework = new FileWork();
        Days = filework.Load();
        if (Days == null)
        {
            Days = new List<DayWeather>();
            filework.Save(Days);
            MessageBox.Show("Файл с погодой отсуцтвовал");
        }
        Calul = new Calculating();
    }

    /// <summary>
    /// Метод добавляет данные о погоде за день.
    /// </summary>
    /// <param name="Date">День о котором идет речь</param>
    /// <param name="Temperature">Температура в этот день</param>
    public void AddDay(DateTime Date, int Temperature)
    {
        DayWeather Day = new DayWeather();
        Day.Date = Date;
        Day.Temperature = Temperature;
        Days.Add(Day);
        filework.Save(Days);
    }

    /// <summary>
    /// Метод получает план по дате
    /// </summary>
    /// <param name="Date">Дата дня по которому нужен план</param>
    /// <returns>План действия для поливалки</returns>
```

```
        public Plan GetPlan(DateTime Date)
        {
            return Calul.Calc(Days, Date);
        }

    }
```

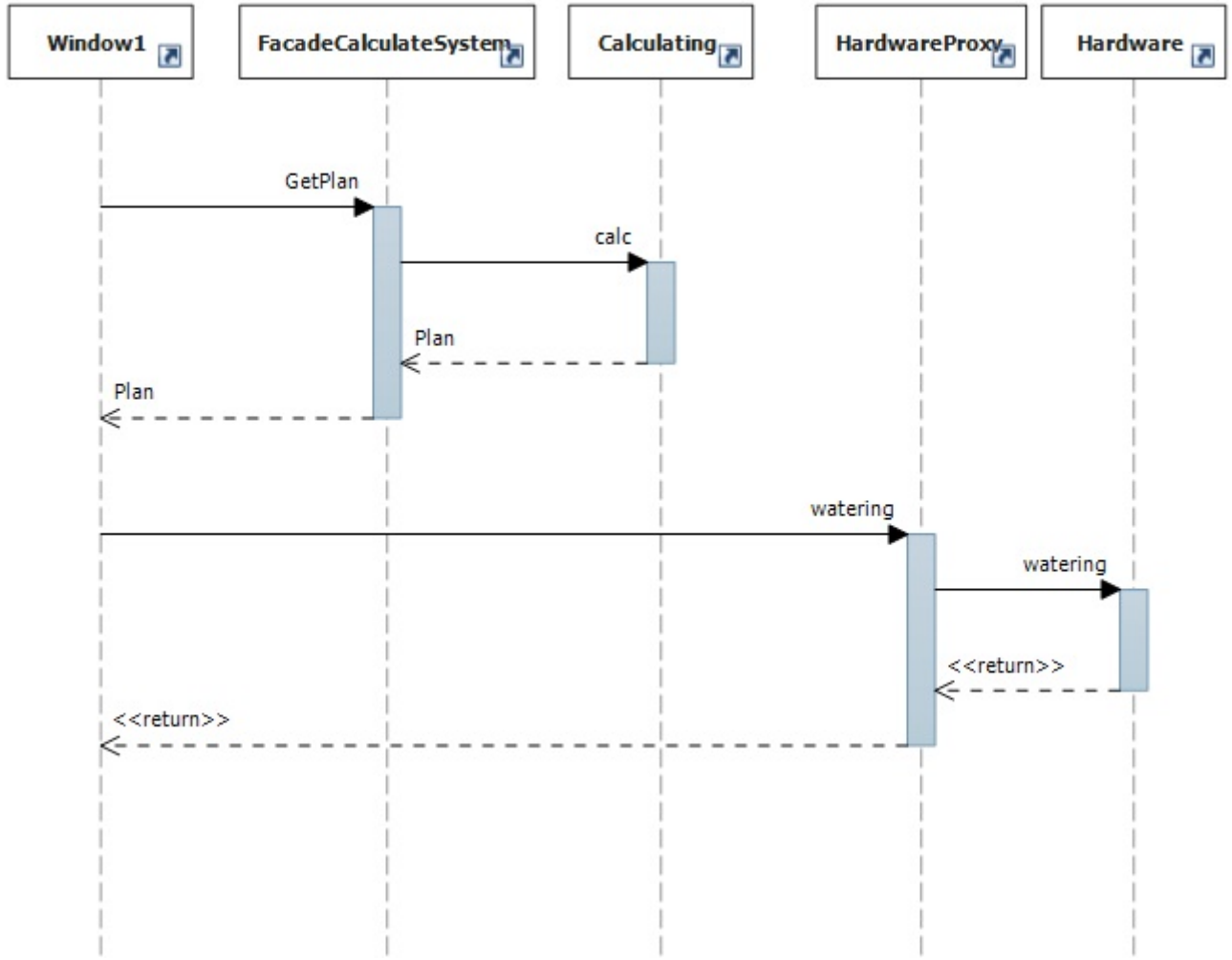
При записи температуры в базу алгоритм работы программы проходит следующим образом:

Пользователь вводит в программу значение температуры в теплице, после чего окно транслирует этот запрос классу FacadeCalculateSystem который создает из этих данных объект DayWeather записывает новый объект в коллекцию, и сохраняет коллекцию в файл.



Когда пользователь пытается приказать программе выполнить полив происходит выполнение следующего алгоритма.

Объект класса Window1 отправляет запрос к объекту FacadeCalculateSystem вызывая метод GetPlan(), класс фасад перенаправляет эту задачу на объект класса Calculating, который выполняет подсчет, и возвращает объект класса Plan, после чего FacadeCalculateSystem тоже возвращает Plan в класс окна приложения. После выполнения выше описанной операции класс Window отправляет запрос классу ProxyHardware чтобы он выполнил действия описанные в плане, Объект прокси перепроверяет данные, и если все хорошо, перенаправляет вызов на объект Hardware который и выполняет логику полива.



11. Домашнее задание

В качестве домашнего задания необходимо разработать три модели приложений, каждое из которых должно реализовывать хотя бы один структурный паттерн проектирования, описанный в текущем уроке. Каждая модель должна быть представлена в виде одной диаграммы классов, отражающей структуру классов, используемых этой моделью, а также связи, в которых состоят эти типы данных.

Одну из созданных моделей (на выбор) необходимо реализовать в виде работающего приложения.