Оглавление

Причины перепроектирования. Каркасы. Паттерны. Отличия каркасов от паттернов. Обзор паттернов проектирования.

Системы необходимо проектировать с учетом их дальнейшего развития. Для проектирования системы, устойчивой к таким изменениям, следует предположить, как она будет изменяться на протяжении отведенного ей времени жизни. Если при проектировании системы не принималась во внимание возможность изменений, то есть вероятность, что в будущем ее придется полностью перепроектировать. Это может повлечь за собой переопределение и новую реализацию классов, модификацию клиентов и повторный цикл тестирования. Перепроектирование отражается на многих частях системы, поэтому непредвиденные изменения всегда оказываются дорогостоящими. Вот некоторые типичные **Причины перепроектирования:**

1. **При создании объекта явно указывается класс.** Задание имени класса привязывает вас к конкретной реализации, а не к конкретному интерфейсу. Это может осложнить изменение объекта в будущем. Чтобы уйти от такой проблемы, создавайте объекты косвенно.

2. **Зависимость от конкретных операций.** Задавая конкретную операцию, вы ограничиваете себя единственным способом выполнения запроса. Если же не включать запросы в код, то будет проще изменить способ удовлетворения запроса, как на этапе компиляции, так и на этапе выполнения.

3. **Зависимость от аппаратной и программной платформ.** Внешние интерфейсы операционной системы и интерфейсы прикладных программ (API) различны на разных программных и аппаратных платформах. Если программа зависит от конкретной платформы, ее будет труднее перенести на другие. Даже на родной платформе такую программу трудно поддерживать. Поэтому при проектировании систем так важно ограничивать платформенные зависимости.

4. **Зависимость от представления или реализации объекта.** Если клиент знает, как объект представлен, хранится или реализован, то при изменении объекта может оказаться необходимым изменить и клиента. Сокрытие этой информации от клиентов поможет уберечься от каскада изменений.

5. **Зависимость от алгоритмов.** Во время разработки и последующего использования алгоритмы часто расширяются, оптимизируются и заменяются. Зависящие от алгоритмов объекты придется переписывать при каждом изменении алгоритма. Поэтому алгоритмы, вероятность изменения которых высока, следует изолировать.

6. Сильная связанность. Сильно связанные между собой классы трудно использовать порознь, так как они зависят друг от друга. Сильная связанность приводит к появлению монолитных систем, в которых нельзя ни изменить, ни удалить класс без знания деталей и модификации других классов. Такую систему трудно изучать, переносить на другие платформы и сопровождать. Слабая связанность повышает вероятность того, что класс можно будет повторно использовать сам по себе. При этом изучение, перенос, модификация и сопровождение системы намного упрощаются.

7. **Расширение функциональности за счет порождения подклассов.** Специализация объекта путем создания подкласса часто оказывается непростым делом. С каждым новым подклассом связаны фиксированные издержки реализации (инициализация, очистка и т.д.). Для определения подкласса необходимо также ясно представлять себе устройство родительского класса. Например, для замещения одной операции может потребоваться заместить и другие. Замещение операции может оказаться необходимым для того, чтобы можно было вызвать унаследованную операцию. Кроме того, порождение подклассов ведет к

комбинаторному росту числа классов, поскольку даже для реализации простого расширения может понадобиться много новых подклассов.

Каркас (фреймворк) – это набор взаимодействующих классов, составляющих повторно используемое проектное решение для конкретного класса программ. Каркас**:**

1. Диктует определенную структуру приложения или модуля.
2. Определяет общую структуру, ее разделение на классы и объекты, основные функции тех и других, методы взаимодействия потоков и классов, потоки управления.

Паттерн (образец проектирования, шаблон проектирования) – описание взаимодействия объектов и классов, адаптированных для решения общей задачи проектирования в конкретном контексте. Паттерн именует, абстрагирует и идентифицирует ключевые аспекты структуры общего решения, которые и позволяют применить его для создания повторно используемого проектного решения. Он выделяет участвующие классы и экземпляры, их роль и отношения, а также функции.

Другими словами, **паттерн** – это многократно используемое решение широко распространенной проблемы, возникающей при разработке программного обеспечения.

По мере накопления опыта программисты распознают сходство возникающих новых проблем с решаемыми ранее. Решение похожих проблем представляют собой повторяющиеся шаблоны, зная которые можно сразу использовать готовые решения, не тратя время на предварительный анализ проблемы. Ещё важнее то, что однажды сформулированный шаблон может использоваться и другими программистами. Это позволяет передать опыт начинающим программистам, а также помогает исключить ситуации, когда обсуждаются различные варианты решения проблемы и неожиданно выясняется, что имеется в виду одно и тоже решение, но сформулированное по-разному.

Отличия каркасов от паттернов:

1. **Паттерны более абстрактны, чем каркасы.** В код могут быть включены целые каркасы, но только экземпляры паттернов. Каркасы можно писать на разных языках программирования и не только изучать, но и непосредственно исполнять и повторно использовать. В противоположность этому паттерны проектирования, необходимо реализовывать всякий раз, когда в них возникает необходимость. Паттерны объясняют намерения проектировщика, компромиссы и последствия выбранного дизайна.

2. **Как архитектурные элементы паттерны мельче, чем каркасы.** Типичный каркас содержит несколько паттернов. Обратное утверждение неверно.

3. **Паттерны менее специализированны, чем каркасы.** Каркас всегда создается для конкретной предметной области. В принципе, каркас графического редактора можно использовать для моделирования, но его никогда не спутаешь с каркасом, предназначенным специально для моделирования. Напротив, паттерны разрешается использовать в приложениях почти любого вида. Хотя, безусловно, существуют и более специализированные паттерны (скажем, паттерны для распределенных систем или параллельного программирования), но даже они не диктуют выбор архитектуры в той же мере, что и каркасы.

- Паттерны проектирования различаются степенью детализации и уровнем абстракции и должны быть каким-то образом организованы. Мы будем классифицировать паттерны по цели, т.е. по назначению паттерна. В связи с этим выделяются **порождающие паттерны**, **структурные паттерны, паттерны поведения и системные (архитектурные) паттерны**. Первые связаны с процессом создания объектов. Вторые имеют отношение к композиции объектов и классов. Паттерны поведения характеризуют то, как классы или объекты взаимодействуют между собой. Системные

(архитектурные**)** паттерны **–** отвечают за представление приложения на архитектурном уровне, описывают процессы внутри приложения и взаимодействие разных приложений.

Порождающие паттерны

- **Abstract Factory** – Абстрактная фабрика. Предоставляет интерфейс для создания семейств взаимосвязанных или взаимозависимых объектов, не специфицируя их конкретных классов.
- **Builder** – Строитель. Отделяет конструирование сложного объекта от его представления, так что в результате одного и того же процесса конструирования могут получаться разные представления.
- **Factory Method** – Фабричный метод. Определяет интерфейс для создания объекта, но оставляет подклассам решение о том, какой класс инстанцировать. Фабричный метод позволяет классу делегировать инстанцирование подклассам.
- **Prototype** – Прототип. Задает виды создаваемых объектов с помощью экземпляра-прототипа и создает новые объекты путем копирования этого прототипа.
- **Singleton** – Одиночка. Гарантирует, что у класса есть только один экземпляр, и предоставляет к нему глобальную точку доступа.

Структурные паттерны

- **Adapter** – Адаптер**.** Преобразует интерфейс одного класса в интерфейс другого, который ожидают клиенты. Адаптер обеспечивает совместную работу классов с несовместимыми интерфейсами, которая без него была бы невозможна.
- **Bridge –** Мост. Отделяет абстракцию от ее реализации так, чтобы то и другое можно было изменять независимо.

- **Composite** – Компоновщик. Компонует объекты в древовидные структуры для представления иерархий часть-целое. Позволяет клиентам единообразно трактовать индивидуальные и составные объекты.
- **Decorator –** Декоратор. Динамически добавляет объекту новые обязанности. Является гибкой альтернативой порождению подклассов с целью расширения функциональности.
- **Façade –** Фасад. Предоставляет унифицированный интерфейс вместо набора интерфейсов некоторой подсистемы. Фасад определяет интерфейс более высокого уровня, который упрощает использование подсистемы.
- **Flyweight –** Приспособленец. Использует разделение для эффективной поддержки множества мелких объектов.
- **Proxy –** Заместитель. Является суррогатом другого объекта и контролирует доступ к нему.

Образцы поведения

- **Chain of Responsibility –** Цепочка обязанностей. Можно избежать жесткой зависимости отправителя запроса от его получателя, при этом запросом начинает обрабатываться один из нескольких объектов. Объекты-получатели связываются в цепочку, и запрос передается по цепочке, пока какой-то объект его не обработает.
- **Command –** Команда. Инкапсулирует запрос в виде объекта, позволяя тем самым параметризовывать клиентов типом запроса, устанавливать очередность запросов, протоколировать их и поддерживать отмену выполнения операций.
- **Interpreter** – Интерпретатор. Для заданного языка определяет представление его грамматики, а также интерпретатор предложений языка, использующий это представление.

- **Iterator –** Итератор. Дает возможность последовательно обойти все элементы составного объекта, не раскрывая его внутреннего представления.
- **Mediator –** Посредник. Определяет объект, в котором инкапсулировано знание о том, как взаимодействуют объекты из некоторого множества. Способствует уменьшению числа связей между объектами, позволяя им работать без явных ссылок друг на друга. Это, в свою очередь, дает возможность независимо изменять схему взаимодействия.
- **Memento –** Хранитель. Позволяет, не нарушая инкапсуляции, получить и сохранить во внешней памяти внутреннее состояние объекта, чтобы позже объект можно было восстановить точно в таком же состоянии.
- **Observer –** Наблюдатель. Определяет между объектами зависимость типа один-ко-многим, так что при изменении состояния одного объекта все зависящие от него получают извещение и автоматически обновляются.
- **State –** Состояние. Позволяет объекту варьировать свое поведение при изменении внутреннего состояния. При этом создается впечатление, что поменялся класс объекта.
- **Strategy –** Стратегия. Определяет семейство алгоритмов, инкапсулируя их все и позволяя подставлять один вместо другого. Можно менять алгоритм независимо от клиента, который им пользуется.
- **Template Method –** Шаблонный метод. Определяет скелет алгоритма, перекладывая ответственность за некоторые его шаги на подклассы. Позволяет подклассам переопределять шаги алгоритма, не меняя его общей структуры.
- **Visitor –** Посетитель. Представляет операцию, которую надо выполнить над элементами объекта. Позволяет определить новую операцию, не меняя классы элементов, к которым он применяется.

Паттерн Abstract Factory.

Название

Абстрактная фабрика – паттерн, порождающий объекты.

Назначение

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

Известен также под именем

Kit (инструментарий).

Abstract Factory: 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.

Мы можем решить эту проблему, определив абстрактный класс WidgetFactory, 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.

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.

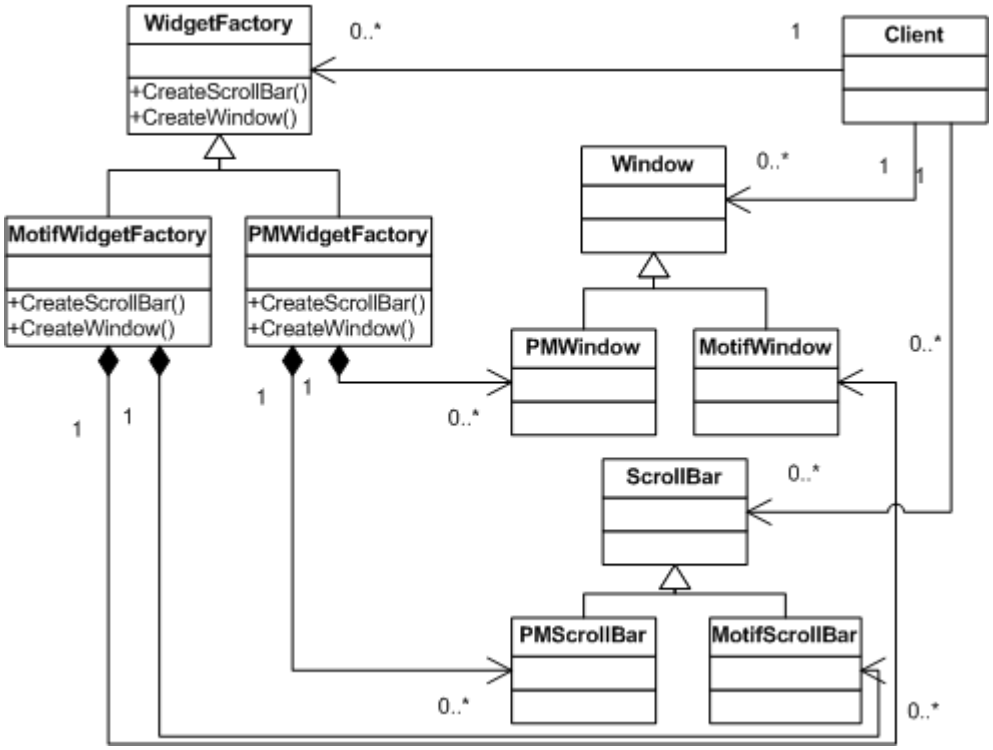

Рисунок 1

Abstract Factory: применимость

Используйте паттерн абстрактная фабрика, 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 вам необходимо обеспечить выполнение этого ограничения;
- Система должна конфигурироваться одним из семейств составляющих ее объектов;
- Вы хотите предоставить библиотеку объектов, раскрывая только их интерфейсы, но не реализацию.

Abstract Factory: структура

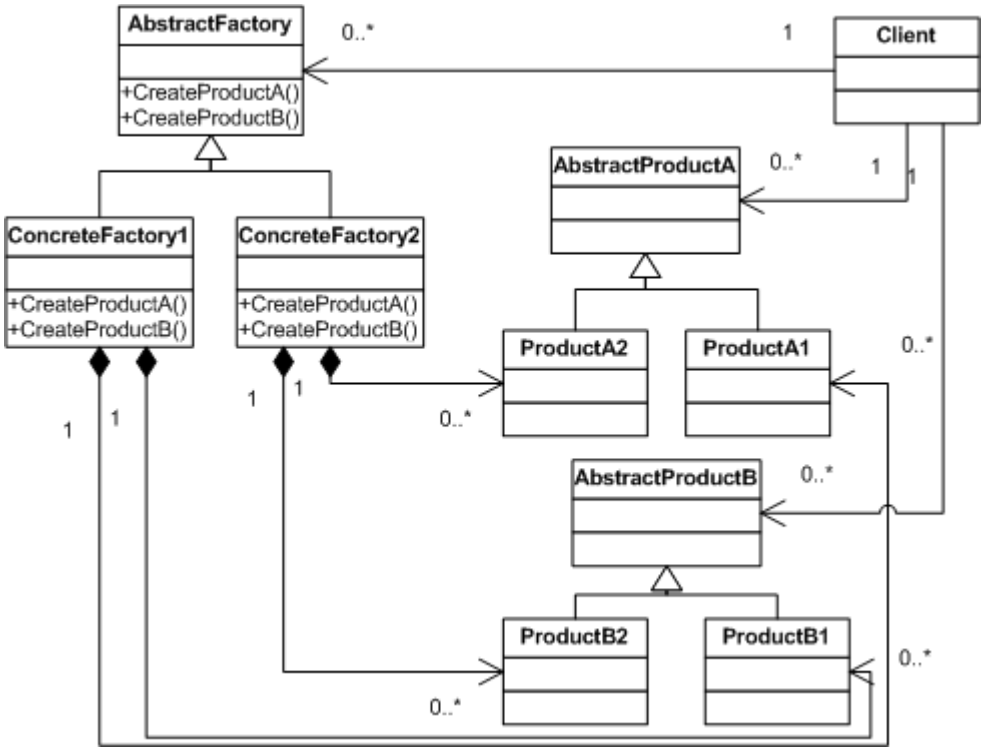

Рисунок 2

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 и 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.

Реализация

1. **Фабрики как объекты, существующие в единственном экземпляре.** Как правило, приложению нужен только один экземпляр класса ConcreteFactory на каждое семейство продуктов. Поэтому для реализации лучше всего применить паттерн одиночка;
2. **Создание продуктов.** Класс AbstractFactory объявляет только интерфейс для создания продуктов. Фактическое их создание - дело подклассов ConcreteProduct. Чаще всего для этой цели определяется фабричный метод для каждого продукта (см. паттерн фабричный метод). Конкретная фабрика специфицирует свои продукты путем замещения фабричного метода для каждого из них. Хотя такая реализация проста, она требует создавать новый подкласс конкретной фабрики для каждого семейства продуктов, даже если они почти ничем не отличаются. Если семейств продуктов может быть много, то конкретную фабрику удастся реализовать с помощью паттерна прототип. В этом случае она инициализируется экземпляром-прототипом каждого продукта в семействе и создает новый продукт путем клонирования этого

прототипа. Подход на основе прототипов устраняет необходимость создавать новый класс конкретной фабрики для каждого нового семейства продуктов;

3. **Определение расширяемых фабрик.** Класс AbstractFactory обычно определяет разные операции для каждого вида изготавливаемых продуктов. Виды продуктов кодируются в сигнатуре операции. Для добавления нового вида продуктов нужно изменить интерфейс класса AbstractFactory и всех зависящих от него классов. Более гибкий, но не такой безопасный способ - добавить параметр к операциям, создающим объекты. Данный параметр определяет вид создаваемого объекта. Это может быть идентификатор класса, целое число, строка или что-то еще, однозначно описывающее вид продукта. При таком подходе классу AbstractFactory нужна только одна операция Make с параметром, указывающим тип создаваемого объекта. Такой вариант проще использовать в динамически типизированных языках вроде Smalltalk, нежели в статически типизированных, каким является C++. Воспользоваться им в C++ можно только, если у всех объектов имеется общий абстрактный базовый класс или если объекты-продукты могут быть безопасно приведены к корректному типу клиентом, который их запросил. Но даже если приведение типов не нужно, остается принципиальная проблема: все продукты возвращаются клиенту одним и тем же абстрактным интерфейсом с уже определенным типом возвращаемого значения. Клиент не может ни различить классы продуктов, ни сделать какие-нибудь предположения о них. Если клиенту нужно выполнить операцию, зависящую от подкласса, то она будет недоступна через абстрактный интерфейс. Хотя клиент мог бы выполнить динамическое приведение типа, это небезопасно и необязательно заканчивается успешно. Здесь мы имеем классический пример компромисса между высокой степенью гибкости и расширяемостью интерфейса.

Применение в Java API

Шаблон AbstractFactory используется в Java API с целью реализации класса java.awt.Toolkit. Это класс абстрактной фабрики, применяемой для создания

объектов, работающих с «родной» системой управления окнами. Используемый класс конкретной фабрики задается кодом инициализации и единственный объект конкретной фабрики возвращается его методом getDefaultToolkit.

Паттерн Builder.

Название

Строитель – паттерн, порождающий объекты.

Назначение

Отделяет конструирование сложного объекта от его представления, так что в результате одного и того же процесса конструирования могут получаться разные представления.

Builder: мотивация

Программа, в которую заложена возможность распознавания и чтения документа в формате RTF (Rich Text Format), должна также “уметь” преобразовывать его во многие другие форматы, например в простой ASCII-текст или в представление, которое можно отобразить в виджете для ввода текста. Однако число вероятных преобразований заранее неизвестно. Поэтому должна быть обеспечена возможность без труда добавлять новый конвертор.

Таким образом, нужно сконфигурировать класс RTFReader с помощью объекта TextConverter, который мог бы преобразовывать RTF в другой текстовый формат. При разборе документа в формате RTF класс RTFReader вызывает TextConverter для выполнения преобразования. Всякий раз, как RTFReader распознает лексему RTF (простой текст или управляющее слово), для ее преобразования объекту TextConverter посылается запрос. Объекты TextConverter отвечают как за преобразование данных, так и за представление лексемы в конкретном формате.

Подклассы TextConverter специализируются на различных преобразованиях и форматах. Например, ASCIIConverter игнорирует запросы на преобразование чего бы то ни было, кроме простого текста. С другой стороны, TeXConverter будет

реализовывать все запросы для получения представления в формате редактора TJX, собирая по ходу необходимую информацию о стилях. А TextWidgetConverter станет строить сложный объект пользовательского интерфейса, который позволит пользователю просматривать и редактировать текст.

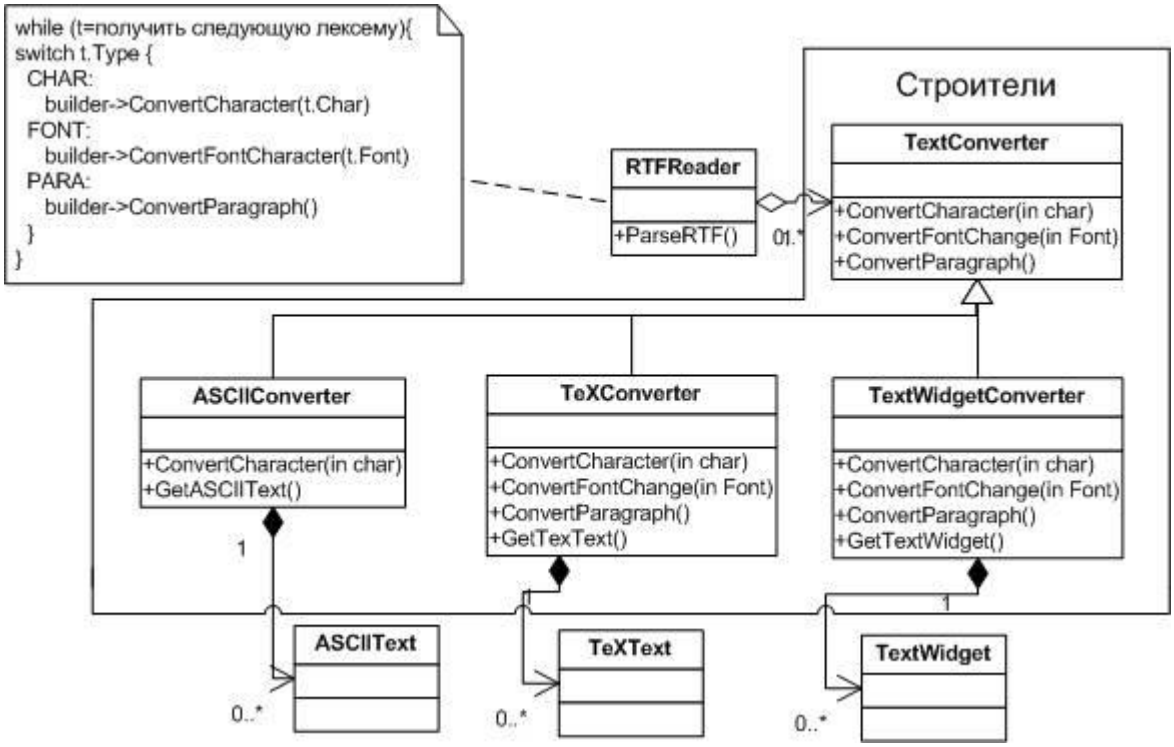

Рисунок 3

Класс каждого конвертора принимает механизм создания и сборки сложного объекта и скрывает его за абстрактным интерфейсом. Конвертор отделен от загрузчика, который отвечает за синтаксический разбор RTF-документа.

В паттерне строитель абстрагированы все эти отношения. В нем любой класс конвертора называется строителем, а загрузчик - распорядителем. В применении к рассмотренному примеру строитель отделяет алгоритм интерпретации формата текста (то есть анализатор RTF-документов) от того, как создается и представляется документ в преобразованном формате. Это позволяет повторно использовать алгоритм разбора, реализованный в RTFReader, для создания разных текстовых представлений RTF-документов; достаточно передать в RTFReader различные подклассы класса TextConverter.

Builder: применимость

Используйте паттерн строитель, когда:

- Алгоритм создания сложного объекта не должен зависеть от того, из каких частей состоит объект и как они стыкуются между собой;
- Процесс конструирования должен обеспечивать различные представления конструируемого объекта.

Builder: структура

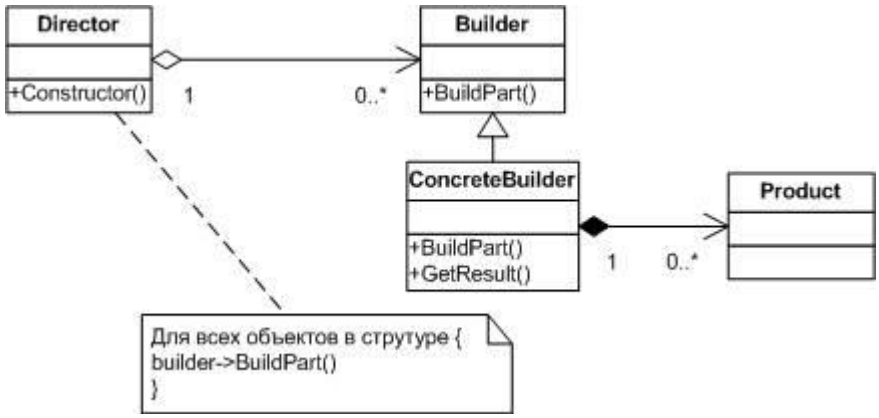

Рисунок 4

Участники

- **Builder (TextConverter)** – строитель - задает абстрактный интерфейс для создания частей объекта Product;
- **ConcreteBuilder (ASCIIConverter, TeXConverter, TextWidgetConverter)** - конкретный строитель - конструирует и собирает вместе части продукта посредством реализации интерфейса Builder;
- определяет создаваемое представление и следит за ним;
- предоставляет интерфейс для доступа к продукту (например, GetASCI IText, GetTextWidget);
- **Director (RTFReader)** – распорядитель - конструирует объект, пользуясь интерфейсом Builder;
- **Product (ASCIIText, TeXText, TextWidget)** – продукт представляет сложный конструируемый объект. ConcreteBuilder строит внутреннее представление продукта и определяет процесс его сборки;

- включает классы, которые определяют составные части, в том числе интерфейсы для сборки конечного результата из частей.

Отношения

Клиент создает объект-распорядитель Director и конфигурирует его нужным объектом-строителем Builder. Распорядитель уведомляет строителя о том, что нужно построить очередную часть продукта. Строитель обрабатывает запросы распорядителя и добавляет новые части к продукту. Клиент забирает продукт у строителя.

Реализация

Обычно существует абстрактный класс Builder, в котором определены операции для каждого компонента, который распорядитель может «попросить» создать. По умолчанию эти операции ничего не делают. Но в классе конкретного строителя ConcreteBuilder они замещены для тех компонентов, в создании которых он принимает участие.

1. **Интерфейс сборки и конструирования.** Строители конструируют свои продукты шаг за шагом. Поэтому интерфейс класса Builder должен быть достаточно общим, чтобы обеспечить конструирование при любом виде конкретного строителя. Ключевой вопрос проектирования связан с выбором модели процесса конструирования и сборки. Обычно бывает достаточно модели, в которой результаты выполнения запросов на конструирование просто добавляются к продукту. В примере с RTF-документами строитель преобразует и добавляет очередную лексему к уже конвертированному тексту. Но иногда может потребоваться доступ к частям сконструированного к данному моменту продукта. Примером являются древовидные структуры, скажем, деревья синтаксического разбора, которые строятся снизу вверх. В этом случае строитель должен был бы вернуть узлы-потомки распорядителю, который затем передал бы их назад строителю, чтобы тот мог построить родительские узлы;

2. **Почему нет абстрактного класса для продуктов.** В типичном случае продукты, изготавливаемые различными строителями, имеют настолько разные представления, что изобретение для них общего родительского класса ничего не дает. В примере с RTF-документами трудно представить себе общий интерфейс у объектов ASCIIText и TextWidget, да он и не нужен. Поскольку клиент обычно конфигурирует распорядителя подходящим конкретным строителем, то, надо полагать, ему известно, какой именно подкласс класса Builder используется и как нужно обращаться с произведенными продуктами;
3. **Пустые методы класса Builder по умолчанию.** В C++ методы строителя намеренно не объявлены чисто виртуальными функциями-членами. Вместо этого они определены как пустые функции, что позволяет подклассу замещать только те операции, в которых он заинтересован.

Паттерн Factory Method.

Название и классификация образца

Фабричный метод – паттерн, порождающий классы.

Назначение

Определяет интерфейс для создания объекта, но оставляет подклассам решение о том, какой класс инстанцировать. Фабричный метод позволяет классу делегировать инстанцирование подклассам.

Известен также под именем

Virtual Constructor (виртуальный конструктор).

Factory Method: мотивация

Каркасы пользуются абстрактными классами для определения и поддержания отношений между объектами. Кроме того, каркас часто отвечает за создание самих объектов.

Рассмотрим каркас для приложений, способных представлять пользователю сразу несколько документов. Две основных абстракции в таком каркасе – это классы

Application и Document. Оба класса абстрактные, поэтому клиенты должны порождать от них подклассы для создания специфичных для приложения реализаций. Например, чтобы создать приложение для рисования, мы определим классы DrawingApplication и DrawingDocument. Класс Application отвечает за управление документами и создает их по мере необходимости, допустим, когда пользователь выбирает из меню пункт Open (открыть) или New (создать).

Поскольку решение о том, какой подкласс класса Document инстанцировать, зависит от приложения, то Application не может предсказать, что именно понадобится. Этому классу известно лишь, когда нужно инстанцировать новый документ, а не какой документ создать. Возникает дилемма: каркас должен инстанцировать классы, но знает он лишь об абстрактных классах, которые инстанцировать нельзя.

Решение предлагает паттерн фабричный метод. В нем инкапсулируется информация о том, какой подкласс класса Document создать, и это знание выводится за пределы каркаса. Подклассы класса Application переопределяют абстрактную операцию CreateDocument таким образом, чтобы она возвращала подходящий подкласс класса Document. Как только подкласс Application инстанцирован, он может инстанцировать специфические для приложения документы, ничего не зная об их классах. Операцию CreateDocument мы называем фабричным методом, поскольку она отвечает за изготовление объекта.

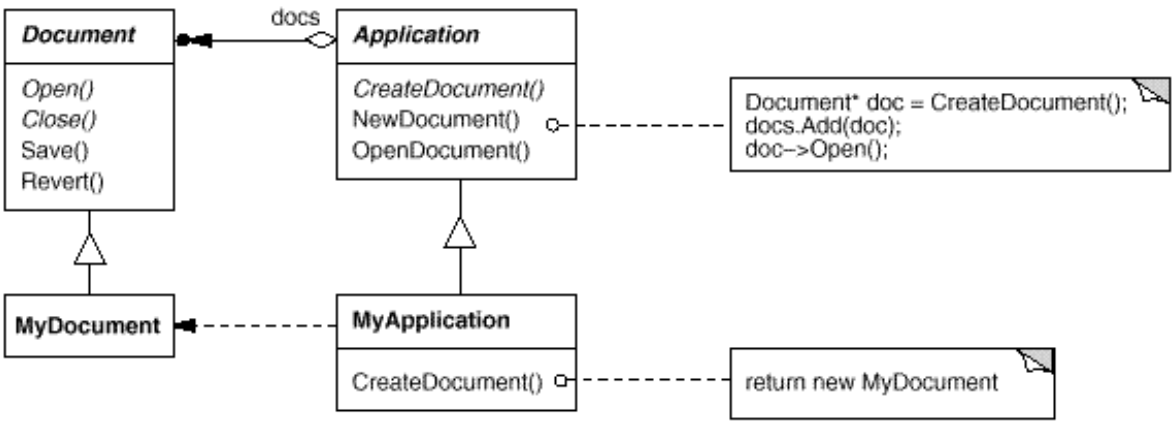

Рисунок 5

Factory Method: применимость

Используйте образец фабричный метод, когда:

- Классу заранее неизвестно, объекты каких классов ему нужно создавать;
- Класс спроектирован так, чтобы объекты, которые он создает, специфицировались подклассами;
- Класс делегирует свои обязанности одному из нескольких вспомогательных подклассов, и вы планируете локализовать знание о том, какой класс принимает эти обязанности на себя.

Factory Method: структура

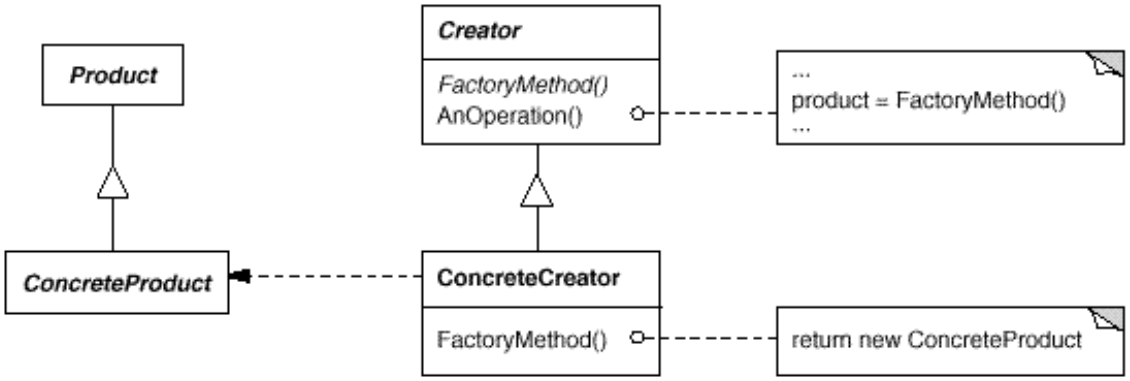

Рисунок 6

Участники

- **Product (Document)** – продукт - определяет интерфейс объектов, создаваемых фабричным методом;
- **ConcreteProduct (MyDocument**) - конкретный продукт - реализует интерфейс Product;
- **Creator (Application)** – создатель - объявляет фабричный метод, возвращающий объект типа Product. Creator может также определять реализацию по умолчанию фабричного метода, который возвращает объект ConcreteProduct; может вызывать фабричный метод для создания объекта Product;

- **ConcreteCreator (MyApplication)** - конкретный создатель - замещает фабричный метод, возвращающий объект ConcreteProduct.

Реализация

Создатель «полагается» на свои подклассы в определении фабричного метода, который будет возвращать экземпляр подходящего конкретного продукта.

Factory Method: особенности

- **Две основных разновидности: класс Creator – абстрактный** и не содержит реализации объявленного в нем фабричного метода. **Creator – конкретный класс,** в котором по умолчанию есть реализация фабричного метода. В первом случае для определения реализации необходимы подклассы, поскольку никакого разумного умолчания не существует. При этом обходится проблема, связанная с необходимостью инстанцировать заранее неизвестные классы. Во втором случае конкретный класс Creator использует фабричный метод, главным образом ради повышения гибкости. Выполняется правило: “Создавай объекты в отдельной операции, чтобы подклассы могли подменить способ их создания”. Соблюдение этого правила гарантирует, что авторы подклассов смогут при необходимости изменить класс объектов, инстанцируемых их родителем.

- **Параметризованные фабричные методы.** Это еще один вариант паттерна, который позволяет фабричному методу создавать разные виды продуктов. Фабричному методу передается параметр, который идентифицирует вид создаваемого объекта. Все объекты, получающиеся с помощью фабричного метода, разделяют общий интерфейс Product. В примере с документами класс Application может поддерживать разные виды документов. Вы передаете методу CreateDocument лишний параметр, который и определяет, документ какого вида нужно создать.

Применение в Java API

Каждый объект URL имеет связанный с ним объект URLConnection. Объекты URLConnection имеют метод getContent, который возвращает содержимое URL, упакованное в соответствующем объекте. Например, если URL содержит файл gif,

то метод getContent объекта URLConnection вернет объект Image. Объекты URLConnection играют роль инициатора запроса создания в паттерне Фабричный метод. Они делегируют функции метода getContent объекту класса ContentHandler. Это абстрактный класс, который играет роль Product и имеет информацию об управлении содержимым определенного типа. Объект URLConnection получает объект класса ContentHandler через объект класса ContentHandlerFactory. Это абстрактный класс, который участвует в паттерне Фабричный метод в качестве интерфейса класса-фабрики. Класс URLConnection имеет также метод setContentHandlerFactory.

Паттерн Prototype.

Название

Прототип – паттерн, порождающий объекты.

Назначение

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

Prototype: мотивация

Построить музыкальный редактор удалось бы путем адаптации общего каркаса графических редакторов и добавления новых объектов, представляющих ноты, паузы и нотный стан. В каркасе редактора может присутствовать палитра инструментов для добавления в партитуру этих музыкальных объектов. Палитра может также содержать инструменты для выбора, перемещения и иных манипуляций с объектами. Так, пользователь, щелкнув, например, по значку четверти поместил бы ее тем самым в партитуру. Или, применив инструмент перемещения, двигал бы ноту на стане вверх или вниз, чтобы изменить ее высоту.

Предположим, что каркас предоставляет абстрактный класс Graphic для графических компонентов вроде нот и нотных станов, а также абстрактный класс Tool для определения инструментов в палитре. Кроме того, в каркасе имеется

предопределенный подкласс GraphicTool для инструментов, которые создают графические объекты и добавляют их в документ.

Однако класс GraphicTool создает некую проблему для проектировщика каркаса. Классы нот и нотных станов специфичны для нашего приложения, а класс GraphicTool принадлежит каркасу. Этому классу ничего неизвестно о том, как создавать экземпляры наших музыкальных классов и добавлять их в партитуру. Можно было бы породить от GraphicTool подклассы для каждого вида музыкальных объектов, но тогда оказалось бы слишком много классов, отличающихся только тем, какой музыкальный объект они инстанцируют. Мы знаем, что гибкой альтернативой порождению подклассов является композиция. Вопрос в том, как каркас мог бы воспользоваться ею для параметризации экземпляров GraphicTool классом того объекта Graphic, который предполагается создать.

Решение - заставить GraphicTool создавать новый графический объект, копируя или «клонируя» экземпляр подкласса класса Graphic. Этот экземпляр мы будем называть прототипом. GraphicTool параметризуется прототипом, который он должен клонировать и добавить в документ. Если все подклассы Graphic поддерживают операцию clone, то GraphicTool может клонировать любой вид графических объектов.

Итак, в нашем музыкальном редакторе каждый инструмент для создания музыкального объекта - это экземпляр класса GraphicTool, инициализированный тем или иным прототипом. Любой экземпляр GraphicTool будет создавать музыкальный объект, клонируя его прототип и добавляя клон в партитуру.

Можно воспользоваться паттерном прототип, чтобы еще больше сократить число классов. Для целых и половинных нот у нас есть отдельные классы, но, быть может, это излишне. Вместо этого они могли бы быть экземплярами одного и того же класса, инициализированного разными растровыми изображениями и длительностями звучания. Инструмент для создания целых нот становится просто объектом класса GraphicTool, в котором прототип MusicalNote инициализирован

целой нотой. Это может значительно уменьшить число классов в системе. Заодно упрощается добавление нового вида нот в музыкальный редактор.

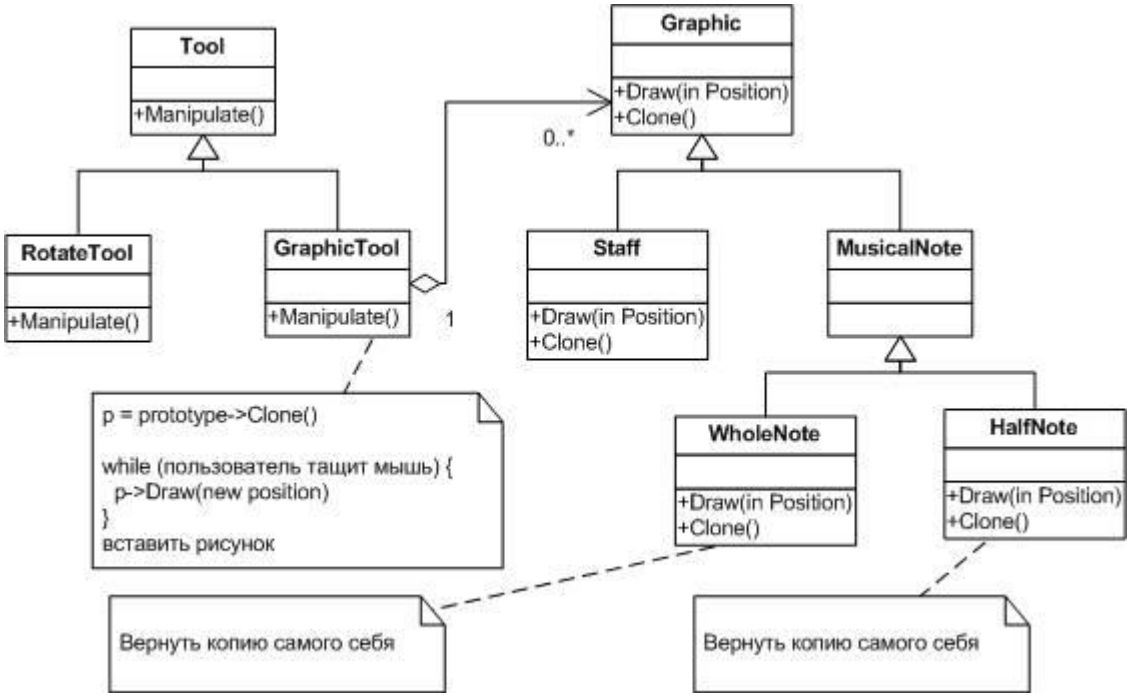

Рисунок 7

Prototype: применимость

Используйте паттерн прототип, когда система не должна зависеть от того, как в ней создаются, компонуются и представляются продукты:

- Инстанцируемые классы определяются во время выполнения, например с помощью динамической загрузки;
- Для того чтобы избежать построения иерархий классов или фабрик, параллельных иерархии классов продуктов;
- Экземпляры класса могут находиться в одном из не очень большого числа различных состояний. Может оказаться удобнее установить соответствующее число прототипов и клонировать их, а не инстанцировать каждый раз класс вручную в подходящем состоянии.

Prototype: структура

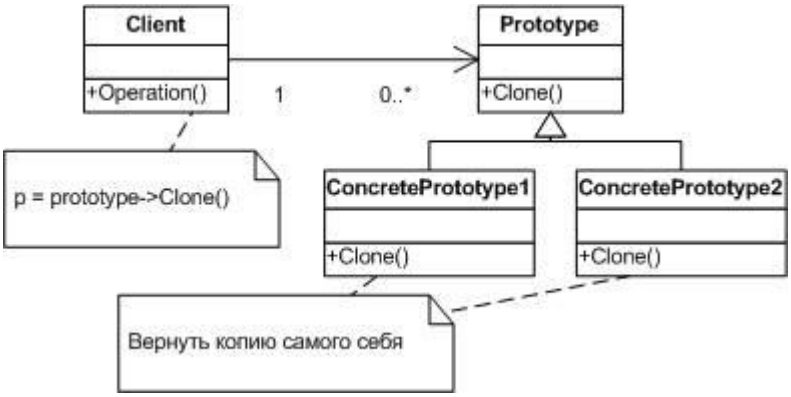

Рисунок 8

Участники

- **Prototype (Graphic)** – прототип - объявляет интерфейс для клонирования самого себя;
- **ConcretePrototype (Staff - нотный стан, WholeNote - целая нота, HalfNote - половинная нота)** - конкретный прототип - реализует операцию клонирования себя;
- **Client (GraphicTool)** – клиент - создает новый объект, обращаясь к прототипу с запросом клонировать себя.

Отношения

Клиент обращается к прототипу, чтобы тот создал свою копию.

Реализация

1. **Использование диспетчера прототипов.** Если число прототипов в системе не фиксировано (то есть они могут создаваться и уничтожаться динамически), ведите реестр доступных прототипов. Клиенты должны не управлять прототипами самостоятельно, а сохранять и извлекать их из реестра. Клиент запрашивает прототип из реестра перед его клонированием. Такой реестр мы будем называть диспетчером прототипов. Диспетчер прототипов - это ассоциативное хранилище, которое возвращает прототип, соответствующий заданному ключу. В нем есть операции для регистрации прототипа с указанным ключом и отмены регистрации. Клиенты могут изменять и даже

«просматривать» реестр во время выполнения, а значит, расширять систему и вести контроль над ее состоянием без написания кода;

2. **Реализация операции clone.** Самая трудная часть паттерна прототип - правильная реализация операции clone. Особенно сложно это в случае, когда в структуре объекта есть круговые ссылки. В большинстве языков имеется некоторая поддержка для клонирования объектов. Но эти средства не решают проблему «глубокого и поверхностного копирования». Суть ее в следующем: должны ли при клонировании объекта клонироваться также и его переменные экземпляра или клон просто разделяет с оригиналом эти переменные? Поверхностное копирование просто, и часто его бывает достаточно. Но для клонирования прототипов со сложной структурой обычно необходимо глубокое копирование, поскольку клон должен быть независим от оригинала. Поэтому нужно гарантировать, что компоненты клона являются клонами компонентов прототипа. При клонировании вам приходится решать, что именно может разделяться и может ли вообще;
3. **Инициализация клонов.** Хотя некоторым клиентам вполне достаточно клона как такового, другим нужно инициализировать его внутреннее состояние полностью или частично. Обычно передать начальные значения операции clone невозможно, поскольку их число различно для разных классов прототипов. Для некоторых прототипов нужно много параметров инициализации, другие вообще ничего не требуют. Передача clone параметров мешает построению единообразного интерфейса клонирования. Может оказаться, что в ваших классах прототипов уже определяются операции для установки и очистки некоторых важных элементов состояния. Если так, то этими операциями можно воспользоваться сразу после клонирования. В противном случае, возможно, понадобится ввести операцию Initialize, которая принимает начальные значения в качестве аргументов и соответственно устанавливает внутреннее состояние клона. Будьте осторожны, если операция

clone реализует глубокое копирование: копии может понадобиться удалять (явно или внутри Initialize) перед повторной инициализацией.

Применение в Java API

Шаблон Prototype очень важен для JavaBeans – это экземпляры классов, которые удовлетворяют определённым соглашениям об именах. Соглашения об именах позволяют программе создания компонентов (Beans) знать, как их настраивать. После настройки объекта компонента с целью использования его в приложении, объект сохраняется в файле, который загружается приложением на стадии выполнения.

Паттерн Singleton.

Название

Одиночка – паттерн, порождающий объекты.

Назначение

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

Singleton: применимость

- Должен существовать по крайней мере один экземпляр некоторого класса. Даже если методы класса используют только статические данные, может понадобится экземпляр такого класса, например, для передачи в качестве параметра методу другого класса или когда необходимо иметь косвенный доступ к классу, через интерфейс;
- Должен быть ровно один экземпляр некоторого класса, легко доступный всем клиентам. Это может объясняться тем, что нужно иметь только один источник некоторой информации. Например, чтобы был единственный объект, отвечающий за генерирование последовательности порядковых номеров;

- Единственный экземпляр должен расширяться путем порождения подклассов, и клиентам нужно иметь возможность работать с расширенным экземпляром без модификации своего кода;
- Создание объекта не требует больших затрат, но он занимает большой объем памяти или постоянно на протяжении всей жизни использует другие ресурсы.

Singleton: структура

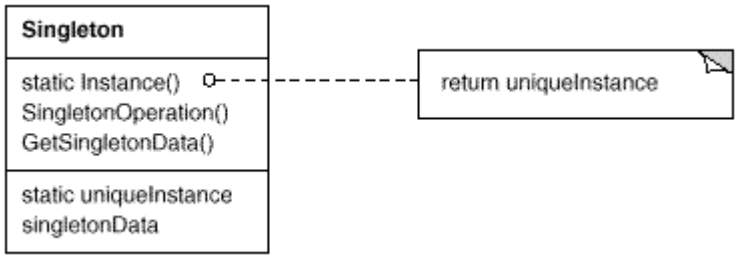

Рисунок 9

Участники

Singleton – одиночка. Определяет метод **Instance**, который позволяет клиентам получать доступ к единственному экземпляру. Может нести ответственность за создание собственного уникального экземпляра.

Реализация

Клиенты получают доступ к экземпляру класса Singleton только через его метод Instance. Для запрета другим классам прямым образом создавать экземпляры Одиночки необходимо сделать все конструкторы приватными. Обязательно нужно объявить хотя бы один приватный конструктор, иначе Java автоматически создаст для Одиночки общедоступный конструктор по умолчанию. Широко распространен вариант «ленивого инстанцирования», когда создание экземпляра Одиночки откладывается до первого обращения к методу Instance, так как экземпляр Одиночки может не понадобится. Класс Одиночки не должен реализовывать интерфейс Cloneable, т.е. класс не должен иметь возможность делать копирование, вызывая метод clone объекта Одиночки. Если существует вероятность того, что несколько

потоков смогут одновременно вызывать метод Instance, следует объявить метод как синхронизированный.

Применение в Java API

java.lang.Runtime – это класс-Одиночка. Он имеет строго один экземпляр, не имеет доступных конструкторов, чтобы получить ссылку на единственный экземпляр нужно вызвать статический метод getRuntime.

Паттерн Adapter.

Название и классификация образца

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).

Adapter: применимость

Применяйте **адаптер классов**, когда:

- хотите использовать существующий класс, но его интерфейс не соответствует вашим потребностям;
- собираетесь создать повторно используемый класс, который должен взаимодействовать с заранее неизвестными или не связанными с ним классами, имеющими несовместимые интерфейсы.

Применяйте **адаптер объектов**, когда

- нужно использовать несколько существующих подклассов, но непрактично адаптировать их интерфейсы путем порождения новых подклассов от каждого. В этом случае адаптер объектов может приспосабливать интерфейс их общего родительского класса.

Adapter (класса): структура

Адаптер класса использует множественное наследование для адаптации одного интерфейса к другому.

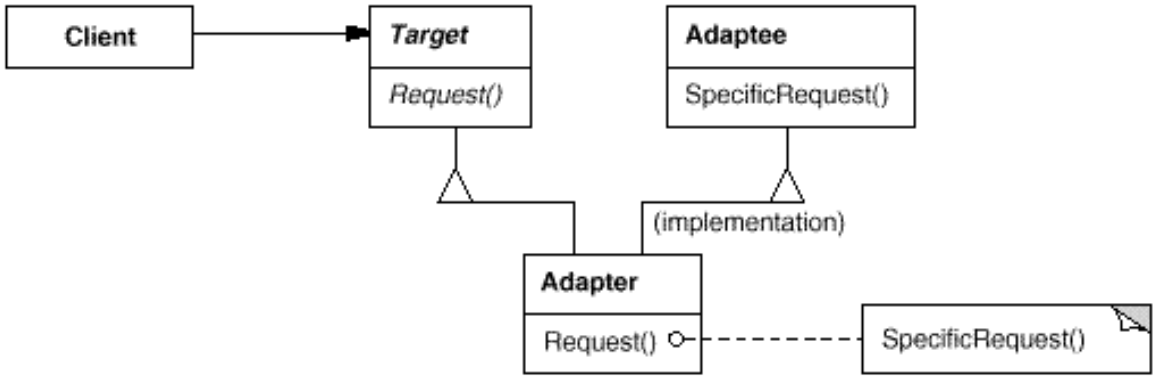

Рисунок 10

Adapter (класса): особенности

- 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.

Adapter (объекта): структура

Адаптер объекта применяет композицию объектов.

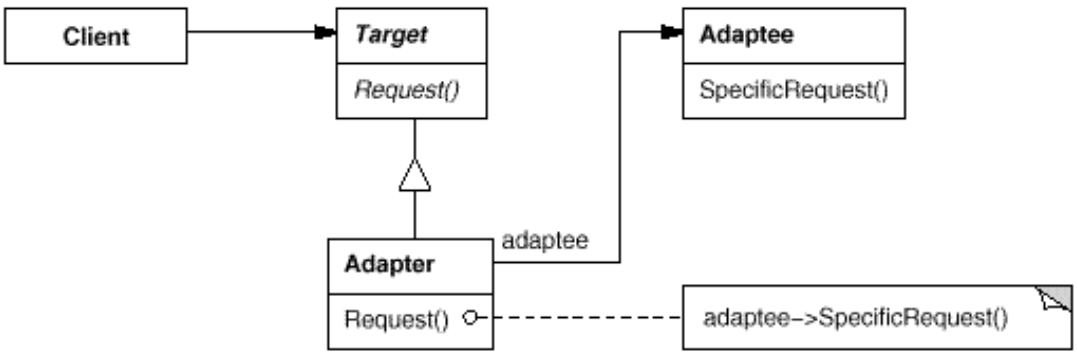

Рисунок 11

Adapter (объекта): особенности

- Позволяет одному адаптеру Adapter работать со многими 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.

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 (TextShape) –** адаптер - адаптирует интерфейс Adaptee к интерфейсу Target.

Реализация

Клиенты вызывают операции экземпляра адаптера Adapter. В свою очередь адаптер вызывает операции адаптируемого объекта или класса Adaptee, который и выполняет запрос. При реализации Адаптера возникает вопрос: откуда объекты-адаптеры узнают, какой экземпляр класса Adaptee вызвать. Существует два подхода к решению:

1) Передать ссылку на объект Adaptee в качестве параметра для конструктора объекта-адаптера или для одного из его методов. Это позволяет использовать объект-адаптер с любым экземпляром или с любым возможным количеством экземпляров класса Adaptee.

2) Сделать класс-адаптер внутренним классом класса Adaptee. При этом упрощается связь между объектом-адаптером и адаптируемым объектом. Кроме того, связь при этом становится жёсткой.

Применение в Java API

Java API не содержит каких-либо открытых классов адаптеров, готовых к использованию. Он имеет классы, например java.awt.event.WindowAdapter, предназначенные не для прямого использования, а для создания на их основе подклассов. Идея состоит в том, что некоторые интерфейсы слушателей событий, например WindowListener, объявляют множество методов. Как правило, не все эти методы должны быть реализованы, интерес представляют только несколько из них. Класс WindowAdapter реализует интерфейс WindowListener и реализует все его методы как пустые. Класс адаптера, являющийся подклассом класса WindowAdapter должен реализовывать только методы, соответствующие представляющим интерес событиям. Для всех остальных методов он наследует пустые варианты реализации.

Паттерн Bridge.

Название и классификация паттерна

Мост - паттерн, структурирующий объекты.

Назначение

Отделить абстракцию от ее реализации так, чтобы то и другое можно было изменять независимо.

Известен также под именем

Handle/Body (описатель/тело).

Bridge: мотивация

Если для некоторой абстракции возможно несколько реализаций, то обычно применяют наследование. Абстрактный класс определяет интерфейс абстракции, а его конкретные подклассы по-разному реализуют его. Но такой подход не всегда обладает достаточной гибкостью. Наследование жестко привязывает реализацию к

абстракции, что затрудняет независимую модификацию, расширение и повторное использование абстракции и ее реализации.

Рассмотрим реализацию переносимой абстракции окна в библиотеке для разработки пользовательских интерфейсов. Написанные с ее помощью приложения должны работать в разных средах, например под X Window System и Presentation Manager (PM) от компании IBM. С помощью наследования мы могли бы определить абстрактный класс Window и его подклассы XWindow и PMWindow, реализующие интерфейс окна для разных платформ. Но у такого решения есть два недостатка:

1) неудобно распространять абстракцию Window на другие виды окон или новые платформы. Представьте себе подкласс IconWindow, который специализирует абстракцию окна для пиктограмм. Чтобы поддержать пиктограммы на обеих платформах, нам придется реализовать два новых подкласса XlconWindow и PMIconWindow. Более того, по два подкласса необходимо определять для каждого вида окон. А для поддержки третьей платформы придется определять для всех видов окон новый подкласс Window;
2) клиентский код становится платформенно-зависимым. При создании окна клиент инстанцирует конкретный класс, имеющий вполне определенную реализацию. Например, создавая объект XWindow, мы привязываем абстракцию окна к ее реализации для системы X Window и, следовательно, делаем код клиента ориентированным именно на эту оконную систему. Таким образом, усложняется перенос клиента на другие платформы. Клиенты должны иметь возможность создавать окно, не привязываясь к конкретной реализации. Только сама реализация окна должна зависеть от платформы, на которой работает приложение. Поэтому в клиентском коде не может быть никаких упоминаний о платформах.

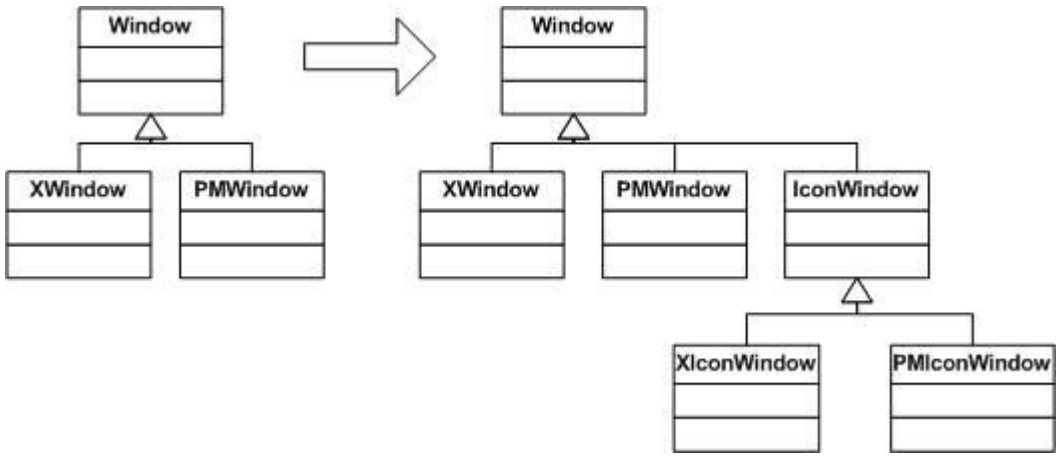

Рисунок 12

С помощью паттерна мост эти проблемы решаются. Абстракция окна и ее реализация помещаются в раздельные иерархии классов. Таким образом, существует одна иерархия для интерфейсов окон (Window, IconWindow, TransientWindow) и другая (с корнем Windowimp) - для платформенно-зависимых реализаций. Так, подкласс XWindowImp предоставляет реализацию в системе X Window System.

Все операции подклассов Window реализованы в терминах абстрактных операций из интерфейса Windowimp. Это отделяет абстракцию окна от различных ее платформенно-зависимых реализаций. Отношение между классами Window и Windowimp мы будем называть **мостом,** поскольку между абстракцией и реализацией строится мост, и они могут изменяться независимо.

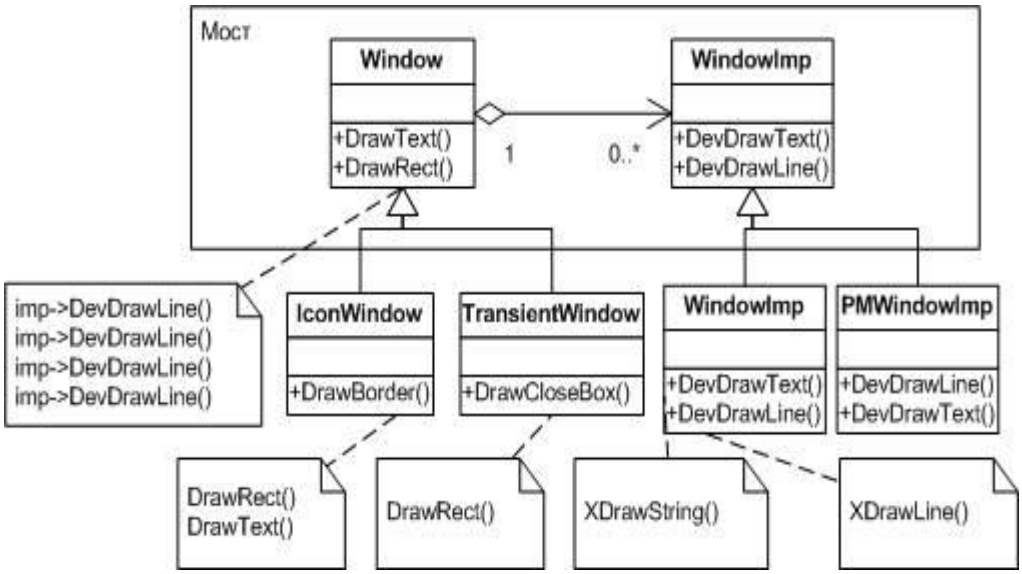

Рисунок 13

Bridge: применимость

Используйте паттерн мост, когда:

1. хотите избежать постоянной привязки абстракции к реализации. Так, например, бывает, когда реализацию необходимо выбирать во время выполнения программы;
2. и абстракции, и реализации должны расширяться новыми подклассами. В таком случае паттерн мост позволяет комбинировать разные абстракции и реализации и изменять их независимо;
3. изменения в реализации абстракции не должны сказываться на клиентах, то есть клиентский код не должен перекомпилироваться;
4. только для C++! Вы хотите полностью скрыть от клиентов реализацию абстракции. В C++ представление класса видимо через его интерфейс;
5. число классов начинает быстро расти, как мы видели на первой диаграмме из раздела «Мотивация». Это признак того, что иерархию следует разделить на две части;
6. вы хотите разделить одну реализацию между несколькими объектами (быть может, применяя подсчет ссылок), и этот факт необходимо скрыть от клиента.

Bridge: структура

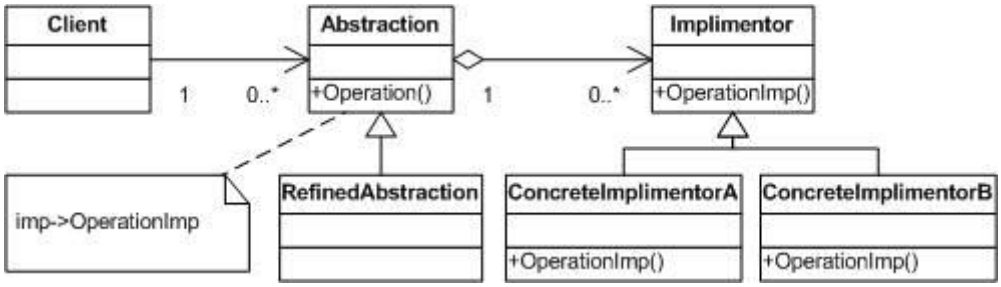

Рисунок 14

Участники

- **Abstraction (Window)** – абстракция - определяет интерфейс абстракции и хранит ссылку на объект типа Implementor;

- **RefinedAbstraction (iconWindow)** – уточненная абстракция - расширяет интерфейс, определенный абстракцией Abstraction;
- **Implementor (Windowlmp)** – реализатор - определяет интерфейс для классов реализации. Он не обязан точно соответствовать интерфейсу класса Abstraction. На самом деле оба интерфейса могут быть совершенно различны. Обычно интерфейс класса Implementor предоставляет только примитивные операции, а класс Abstraction определяет операции более высокого уровня, базирующиеся на этих примитивах;
- **Concretelmplementor (XWindowlmp, PMWindowlmp)** – конкретный реализатор - содержит конкретную реализацию интерфейса класса Implementor.

Отношения

Объект Abstraction перенаправляет своему объекту Implementor запросы клиента.

Реализация

1. **Только один класс Implementor.** В ситуациях, когда есть только одна реализация, создавать абстрактный класс Implementor необязательно. Это вырожденный случай паттерна мост - между классами Abstraction и Implementor существует взаимно-однозначное соответствие. Тем не менее, разделение все же полезно, если нужно, чтобы изменение реализации класса не отражалось на существующих клиентах (должно быть достаточно заново скомпоновать программу, не перекомпилируя клиентский код). Например, в C++ интерфейс класса Implementor можно определить в закрытом заголовочном файле, который не передается клиентам. Это позволяет полностью скрыть реализацию класса от клиентов.
2. **Создание правильного объекта Implementor.** Как, когда и где принимается решение о том, какой из нескольких классов Implementor инстанцировать? Если у класса Abstraction есть информация о конкретных классах Concretelmplementor, то он может инстанцировать один из них в своем

конструкторе; какой именно - зависит от переданных конструктору параметров. Так, если класс коллекции поддерживает несколько реализаций, то решение можно принять в зависимости от размера коллекции. Для небольших коллекций применяется реализация в виде связанного списка, для больших - в виде хэшированных таблиц. Другой подход - заранее выбрать реализацию по умолчанию, а позже изменять ее в соответствии с тем, как она используется. Например, если число элементов в коллекции становится больше некоторой условной величины, то мы переключаемся с одной реализации на другую, более эффективную. Можно также делегировать решение другому объекту. В примере с иерархиями Window/Windowlmp уместно было бы ввести фабричный объект, единственная задача которого - инкапсулировать платформенную специфику. Фабрика обладает информацией, объекты Windowlmp какого вида надо создавать для данной платформы, а объект Window просто обращается к ней с запросом о предоставлении какого-нибудь объекта Windowlmp, при этом понятно, что объект получит то, что нужно. Преимущество описанного подхода: класс Abstraction напрямую не привязан ни к одному из классов Implementor;

3. **Разделение реализаторов.** В C++ можно применить идиому описатель/тело, чтобы несколькими объектами могла совместно использоваться одна и та же реализация. В теле хранится счетчик ссылок, который увеличивается и уменьшается в классе описателя.

Паттерн Composite.

Название и классификация образца

Компоновщик - паттерн, структурирующий объекты.

Назначение

Компонует объекты в древовидные структуры для представления иерархий часть-целое. Позволяет клиентам единообразно трактовать индивидуальные и составные объекты.

Composite: мотивация

Такие приложения, как графические редакторы и редакторы электрических схем, позволяют пользователям строить сложные диаграммы из более простых компонентов. Проектировщик может сгруппировать мелкие компоненты для формирования более крупных, которые, в свою очередь, могут стать основой для создания еще более крупных. В простой реализации допустимо было бы определить классы графических примитивов, например текста и линий, а также классы, выступающие в роли контейнеров для этих примитивов.

Но у такого решения есть существенный недостаток. Программа, в которой эти классы используются, должна по-разному обращаться с примитивами и контейнерами, хотя пользователь чаще всего работает с ними единообразно. Необходимость различать эти объекты усложняет приложение. Паттерн компоновщик описывает, как можно применить рекурсивную композицию таким образом, что клиенту не придется проводить различие между простыми и составными объектами.

Ключом к паттерну компоновщик является абстрактный класс, который представляет одновременно и примитивы, и контейнеры. В графической системе этот класс может называться Graphic. В нем объявлены операции, специфичные для каждого вида графического объекта (такие как Draw) и общие для всех составных объектов, например операции для доступа и управления потомками.

Подклассы Line, Rectangle и Text (см. диаграмму выше) определяют примитивные графические объекты. В них операция Draw реализована соответственно для рисования прямых, прямоугольников и текста. Поскольку у примитивных объектов нет потомков, то ни один из этих подклассов не реализует операции, относящиеся к управлению потомками.

Класс Picture определяет агрегат, состоящий из объектов Graphic. Реализованная в нем операция Draw вызывает одноименную функцию для каждого потомка, а операции для работы с потомками уже не пусты. Поскольку

интерфейс класса Picture соответствует интерфейсу Graphic, то в состав объекта Picture могут входить и другие такие же объекты.

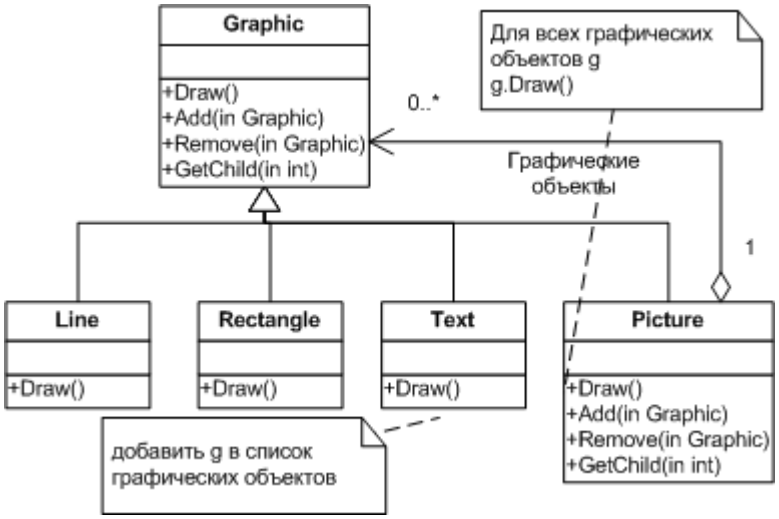

Рисунок 15

Composite: применимость

Используйте паттерн компоновщик, когда:

1. нужно представить иерархию объектов вида часть-целое;
2. хотите, чтобы клиенты единообразно трактовали составные и индивидуальные объекты.

Composite: структура

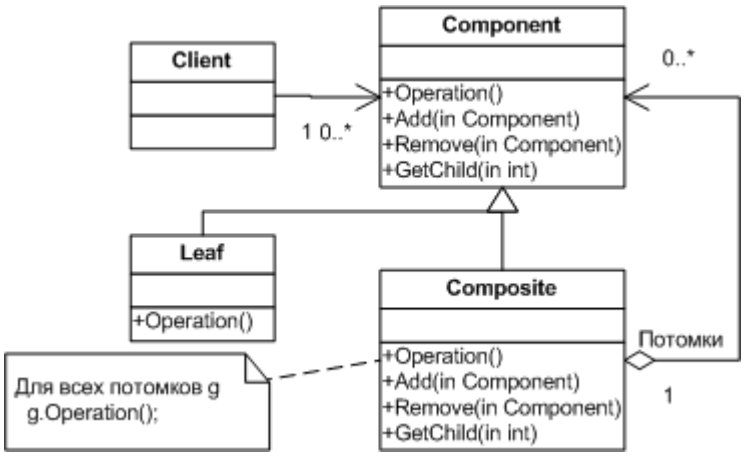

Рисунок 16

Участники

- **Component (Graphic)** - компонент: - объявляет интерфейс для компонуемых объектов;

- предоставляет подходящую реализацию операций по умолчанию, общую для всех классов;
- объявляет интерфейс для доступа к потомкам и управления ими;
- определяет интерфейс для доступа к родителю компонента в рекурсивной структуре и при необходимости реализует его. Описанная возможность необязательна;

• **Leaf (Rectangle, Line, Text, и т.п.) –** лист - представляет листовые узлы композиции и не имеет потомков;

- определяет поведение примитивных объектов в композиции;

• **Composite (Picture) -** составной объект: - определяет поведение компонентов, у которых есть потомки;
- хранит компоненты-потомки;
- реализует относящиеся к управлению потомками операции в интерфейсе класса Component;

• **Client –** клиент - манипулирует объектами композиции через интерфейс Component.

Отношения

Клиенты используют интерфейс класса Component для взаимодействия с объектами в составной структуре. Если получателем запроса является листовый объект Leaf, то он и обрабатывает запрос. Когда же получателем является составной объект Composite, то обычно он перенаправляет запрос своим потомкам, возможно, выполняя некоторые дополнительные операции до или после перенаправления.

Реализация

1. **Явные ссылки на родителей.** Хранение в компоненте ссылки на своего родителя может упростить обход структуры и управление ею. Наличие такой ссылки облегчает передвижение вверх по структуре и удаление компонента. Кроме того, ссылки на родителей помогают поддержать паттерн цепочка обязанностей. Обычно ссылку на родителя определяют в классе Component. Классы Leaf и Composite могут унаследовать саму ссылку и операции с ней.

При наличии ссылки на родителя важно поддерживать следующий инвариант: если некоторый объект в составной структуре ссылается на другой составной объект как на своего родителя, то для последнего первый является потомком. Простейший способ гарантировать соблюдение этого условия - изменять родителя компонента только тогда, когда он добавляется или удаляется из составного объекта. Если это удается один раз реализовать в операциях Add и Remove, то реализация будет унаследована всеми подклассами и, значит, инвариант будет поддерживаться автоматически.

2. **Разделение компонентов.** Часто бывает полезно разделять компоненты, например для уменьшения объема занимаемой памяти. Но если у компонента может быть более одного родителя, то разделение становится проблемой. Возможное решение - позволить компонентам хранить ссылки на нескольких родителей. Однако в таком случае при распространении запроса по структуре могут возникнуть неоднозначности. Паттерн приспособленец показывает, как следует изменить дизайн, чтобы вовсе отказаться от хранения родителей. Работает он в тех случаях, когда потомки могут не посылать сообщений своим родителям, вынеся за свои границы часть внутреннего состояния.
3. **Максимизация интерфейса класса Component.** Одна из целей паттерна компоновщик - избавить клиентов от необходимости знать, работают ли они с листовым или составным объектом. Для достижения этой цели класс Component должен сделать как можно больше операций общими для классов Composite и Leaf. Обычно класс Component предоставляет для этих операций реализации по умолчанию, а подклассы Composite и Leaf замещают их. Однако иногда эта цель вступает в конфликт с принципом проектирования иерархии классов, согласно которому класс должен определять только логичные для всех его подклассах операции. Класс Component поддерживает много операций, не имеющих смысла для класса Leaf. Как же тогда предоставить для них реализацию по умолчанию? Иногда удается перенести в класс Component операцию, которая, на первый взгляд, имеет смысл только

для составных объектов. Например, интерфейс для доступа к потомкам является фундаментальной частью класса Composite, но вовсе не обязательно класса Leaf. Однако если рассматривать Leaf как Component, у которого никогда не бывает потомков, то мы можем определить в классе Component операцию доступа к потомкам как никогда не возвращающую потомков. Тогда подклассы Leaf могут использовать эту реализацию по умолчанию, а в подклассах Composite она будет переопределена, чтобы возвращать потомков.

4. **Объявление операций для управления потомками.** Хотя в классе Composite реализованы операции Add и Remove для добавления и удаления потомков, но для паттерна компоновщик важно, в каких классах эти операции объявлены. Надо ли объявлять их в классе Component и тем самым делать доступными в Leaf, или их следует объявить и определить только в классе Composite и его подклассах? Решая этот вопрос, мы должны выбирать между безопасностью и прозрачностью:

   - если определить интерфейс для управления потомками в корне иерархии классов, то мы добиваемся прозрачности, так как все компоненты удается трактовать единообразно. Однако расплачиваться приходится безопасностью, поскольку клиент может попытаться выполнить бессмысленное действие, например добавить или удалить объект из листового узла;
   - если управление потомками сделать частью класса Composite, то безопасность удастся обеспечить, ведь любая попытка добавить или удалить объекты из листьев в статически типизированном языке вроде C++ будет перехвачена на этапе компиляции. Но прозрачность мы утрачиваем, ибо у листовых и составных объектов оказываются разные интерфейсы.

5. **Должен ли Component реализовывать список компонентов**. Может возникнуть желание определить множество потомков в виде переменной экземпляра класса Component, в котором объявлены операции доступа и управления потомками. Но размещение указателя на потомков в базовом классе приводит к непроизводительному расходу памяти во всех листовых

узлах, хотя у листа потомков быть не может. Такой прием можно применить, только если в структуре не слишком много потомков.

6. **Упорядочение потомков.** Во многих случаях порядок следования потомков составного объекта важен. В составных объектах, описывающих деревья синтаксического разбора, составные операторы могут быть экземплярами класса Composite, порядок следования потомков которых отражает семантику программы. Если порядок следования потомков важен, необходимо учитывать его при проектировании интерфейсов доступа и управления потомками. В этом может помочь паттерн итератор.
7. **Кэширование для повышения производительности.** Если приходится часто обходить композицию или производить в ней поиск, то класс Composite может кэшировать информацию об обходе и поиске. Кэшировать разрешается либо полученные результаты, либо только информацию, достаточную для ускорения обхода или поиска. Например, класс Picture из примера, приведенного в разделе «Мотивация», мог бы кэшировать охватывающие прямоугольники своих потомков. При рисовании или выборе эта информация позволила бы пропускать тех потомков, которые не видимы в текущем окне. Любое изменение компонента должно делать кэши всех его родителей недействительными. Наиболее эффективен такой подход в случае, когда компонентам известно об их родителях. Поэтому, если вы решите воспользоваться кэшированием, необходимо определить интерфейс, позволяющий уведомить составные объекты о недействительности их кэшей.
8. **Кто должен удалять компоненты.** В языках, где нет сборщика мусора, лучше всего поручить классу Composite удалять своих потомков в момент уничтожения. Исключением из этого правила является случай, когда листовые объекты постоянны и; следовательно, могут разделяться.
9. **Какая структура данных лучше всего подходит для хранения компонентов.** Составные объекты могут хранить своих потомков в самых разных структурах данных, включая связанные списки, деревья, массивы и

хэш-таблицы. Выбор структуры данных определяется, как всегда, эффективностью. Собственно говоря, вовсе не обязательно пользоваться какой-либо из универсальных структур. Иногда в составных объектах каждый потомок представляется отдельной переменной. Правда, для этого каждый подкласс Composite должен реализовывать свой собственный интерфейс управления памятью.

Применение в Java API

В пакете java.awt. содержится удачный пример шаблона Composite: класс Component (наследники Label, Button) и класс Container (наследники Panel, Frame).

Паттерн Decorator.

Название и классификация образца

Декоратор – образец, структурирующий объекты.

Назначение

Динамически добавляет объекту новые обязанности. Является гибкой альтернативой порождению подклассов с целью расширения функциональности.

Известен также под именем

Wrapper (обертка).

Decorator: мотивация

Иногда бывает нужно возложить дополнительные обязанности на отдельный объект, а не на класс в целом. Так, библиотека для построения графических интерфейсов пользователя должна уметь добавлять новое свойство, скажем, рамку или новое поведение (например, возможность прокрутки к любому элементу интерфейса).

Добавить новые обязанности допустимо с помощью наследования. При наследовании класса с рамкой вокруг каждого экземпляра подкласса будет рисоваться рамка. Однако это решение статическое, а значит, недостаточно гибкое. Клиент не может управлять оформлением компонента рамкой.

Более гибким является другой подход: поместить компонент в другой объект, называемый декоратором, который как раз и добавляет рамку. Декоратор следует интерфейсу декорируемого объекта, поэтому его присутствие прозрачно для клиентов компонента. Декоратор переадресует запросы внутреннему компоненту, но может выполнять и дополнительные действия (например, рисовать рамку) до или после переадресации. Поскольку декораторы прозрачны, они могут вкладываться друг в друга, добавляя тем самым любое число новых обязанностей.

Предположим, что имеется объект класса TextView, который отображает текст в окне. По умолчанию TextView не имеет полос прокрутки, поскольку они не всегда нужны. Но при необходимости их удастся добавить с помощью декоратора ScrollDecorator. Допустим, что еще мы хотим добавить жирную сплошную рамку вокруг объекта TextView. Здесь может помочь декоратор BorderDecorator. Классы ScrollDecorator и BorderDecorator являются подклассами Decorator - абстрактного класса, который представляет визуальные компоненты, применяемые для оформления других визуальных компонентов. VisualComponent - это абстрактный класс для представления визуальных объектов. В нем определен интерфейс для рисования и обработки событий. Отметим, что класс Decorator просто переадресует запросы на рисование своему компоненту, а его подклассы могут расширять эту операцию.

Подклассы Decorator могут добавлять любые операции для обеспечения необходимой функциональности. Так, операция ScrollTo объекта ScrollDecorator позволяет другим объектам выполнять прокрутку, если им известно о присутствии объекта ScrollDecorator. Важная особенность этого паттерна состоит в том, что декораторы могут употребляться везде, где возможно появление самого объекта VisualComponent. Поэтому клиент не может отличить декорированный объект от недекорированного, а значит, и никоим образом не зависит от наличия или отсутствия оформлений.

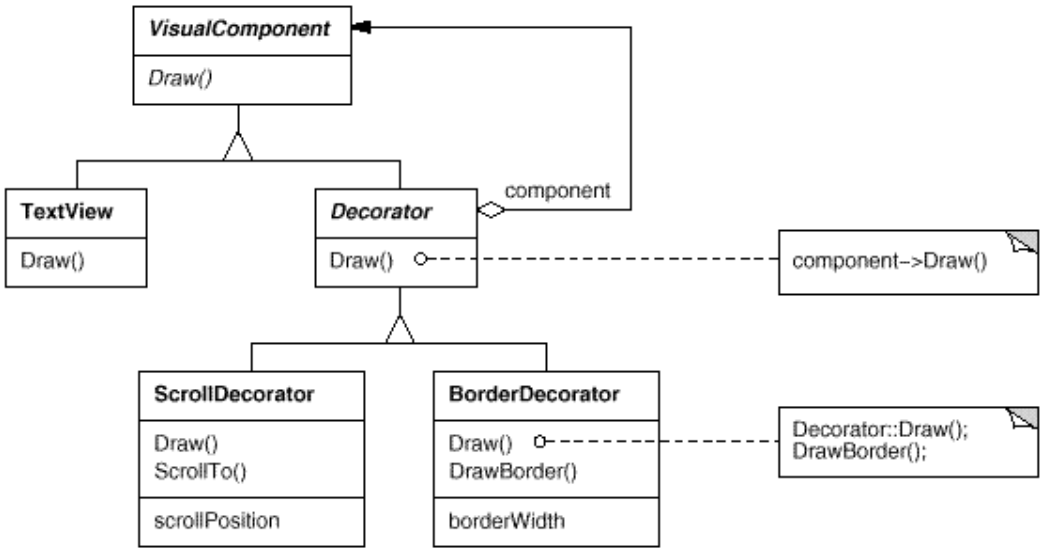

Рисунок 17

Decorator: применимость

Используйте образец декоратор:

- для динамического, прозрачного для клиентов добавления обязанностей объектам;
- для реализации обязанностей, которые могут быть сняты с объекта;
- когда расширение путем порождения подклассов по каким-то причинам неудобно или невозможно. Иногда приходится реализовывать много независимых расширений, так что порождение подклассов для поддержки всех возможных комбинаций приведет к комбинаторному росту их числа. В других случаях определение класса может быть скрыто или почему-либо еще недоступно, так что породить от него подкласс нельзя.

Decorator: структура

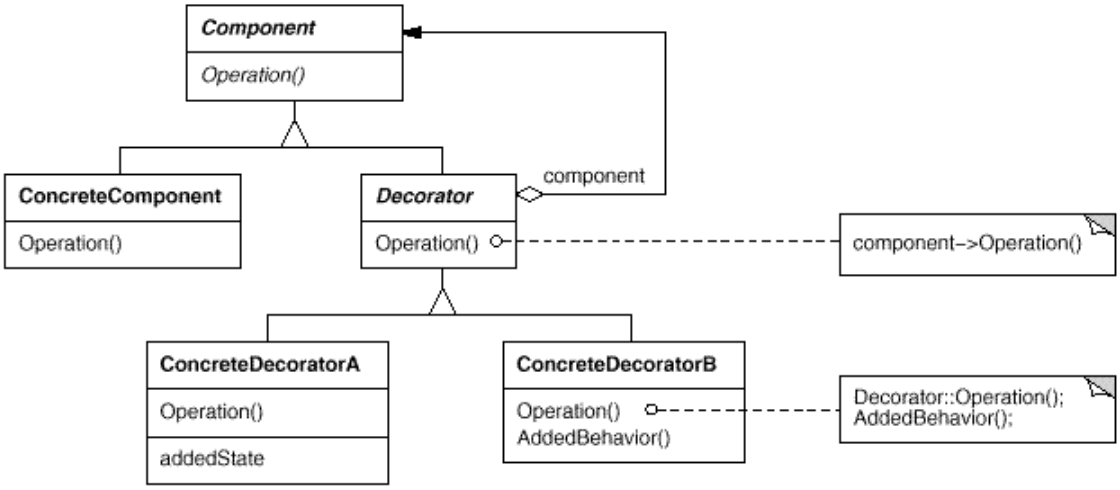

Рисунок 18

Участники

- **Component (VisualComponent)** - компонент - определяет интерфейс для объектов, на которые могут быть динамически возложены дополнительные обязанности;
- **ConcreteComponent (TextView)** - конкретный компонент: - определяет объект, на который возлагаются дополнительные обязанности;
- **Decorator – декоратор** - хранит ссылку на объект Component и определяет интерфейс, соответствующий интерфейсу Component;
- **ConcreteDecorator (BorderDecorator, ScrollDecorator)** - конкретный декоратор - возлагает дополнительные обязанности на компонент.

Отношения

Decorator переадресует запросы объекту Component. Может выполнять и дополнительные операции до и после переадресации.

Decorator: особенности

- **Большая гибкость, чем у статического наследования.** Паттерн декоратор позволяет более гибко добавлять объекту новые обязанности, чем было бы возможно в случае статического (множественного) наследования. Декоратор может добавлять и удалять обязанности во время выполнения программы. При использовании же наследования требуется создавать новый

класс для каждой дополнительной обязанности (например, BorderedScrollableTextView, BorderedTextView), что ведет к увеличению числа классов и, как следствие, к возрастанию сложности системы. Кроме того, применение нескольких декораторов к одному компоненту позволяет произвольным образом сочетать обязанности. Декораторы позволяют легко добавить одно и то же свойство дважды. Например, чтобы окружить объект Text View двойной рамкой, нужно просто добавить два декоратора BorderDecorators. Двойное наследование классу Border в лучшем случае чревато ошибками;

- **Создание цепочек декораторов, в том числе из одних и тех же в одной цепочке.**
- **Позволяет избежать перегруженных функциями классов на верхних уровнях иерархии.** Декоратор разрешает добавлять новые обязанности по мере необходимости. Вместо того чтобы пытаться поддержать все мыслимые возможности в одном сложном, допускающем разностороннюю настройку классе, вы можете определить простой класс и постепенно наращивать его функциональность с помощью декораторов. В результате приложение уже не платит за неиспользуемые функции. Нетрудно также определять новые виды декораторов независимо от классов, которые они расширяют, даже если первоначально такие расширения не планировались. При расширении же сложного класса обычно приходится вникать в детали, не имеющие отношения к добавляемой функции.
- **Декоратор и его компонент неидентичны.** Декоратор действует как прозрачное обрамление. Но декорированный компонент все же неидентичен исходному. При использовании декораторов это следует иметь в виду.
- **Множество мелких объектов**. При использовании в проекте паттерна декоратор нередко получается система, составленная из большого числа мелких объектов, которые похожи друг на друга и различаются только

способом взаимосвязи, а не классом и не значениями своих внутренних переменных. Хотя проектировщик, разбирающийся в устройстве такой системы, может легко настроить ее, но изучать и отлаживать ее очень тяжело.

- **Соответствие интерфейсов декоратора и декорируемого объекта.** Поэтому классы ConcreteDecorator должны наследовать общему классу
- **Возможное отсутствие абстрактного класса декоратора.** Нет необходимости определять абстрактный класс Decorator, если планируется добавить всего одну обязанность. Так часто происходит, когда вы работаете с уже существующей иерархией классов, а не проектируете новую. В таком случае ответственность за переадресацию запросов, которую обычно несет класс Decorator, можно возложить непосредственно на ConcreteDecorator.
- **Облегчение, по возможности, декорируемого класса Component.** Чтобы можно было гарантировать соответствие интерфейсов, компоненты и декораторы должны наследовать общему классу Component. Важно, чтобы этот класс был настолько легким, насколько возможно. Иными словами, он должен определять интерфейс, а не хранить данные. В противном случае декораторы могут стать весьма тяжеловесными, и применять их в большом количестве будет накладно. Включение большого числа функций в класс Component также увеличивает вероятность, что конкретным подклассам придется платить за то, что им не нужно.
- **Изменяется «облик», а не внутренне устройство объекта.** Декоратор можно рассматривать как появившуюся у объекта оболочку, которая изменяет его поведение. Альтернатива - изменение внутреннего устройства объекта, хорошим примером чего может служить паттерн стратегия. Стратегии лучше подходят в ситуациях, когда класс Component уже достаточно тяжел, так что применение паттерна декоратор обходится слишком дорого. В паттерне стратегия компоненты передают часть своей функциональности отдельному объекту-стратегии, поэтому изменить или

расширить поведение компонента допустимо, заменив этот объект. Например, мы можем поддержать разные стили рамок, поручив рисование рамки специальному объекту Border. Объект Border является примером объекта-стратегии: в данном случае он инкапсулирует стратегию рисования рамки. Число стратегий может быть любым, поэтому эффект такой же, как от рекурсивной вложенности декораторов.

Паттерн Facade.

Название и классификация образца

Фасад - паттерн, структурирующий объекты.

Назначение

Предоставляет унифицированный интерфейс вместо набора интерфейсов некоторой подсистемы. Фасад определяет интерфейс более высокого уровня, который упрощает использование подсистемы.

Facade: мотивация

Разбиение на подсистемы облегчает проектирование сложной системы в целом. Общая цель всякого проектирования - свести к минимуму зависимость подсистем друг от друга и обмен информацией между ними. Один из способов решения этой задачи - введение объекта фасад, предоставляющий единый упрощенный интерфейс к более сложным системным средствам.

Рассмотрим, например, среду программирования, которая дает приложениям доступ к подсистеме компиляции. В этой подсистеме имеются такие классы, как Scanner (лексический анализатор), Parser (синтаксический анализатор), ProgramNode (узел программы), ByteCodeStream (поток байтовых кодов) и ProgramNodeBuilder (строитель узла программы). Все вместе они составляют компилятор. Некоторым специализированным приложениям, возможно, понадобится прямой доступ к этим классам. Но для большинства клиентов компилятора такие детали, как синтаксический разбор и генерация кода, обычно не нужны; им просто требуется откомпилировать некоторую

программу. Для таких клиентов применение мощного, но низкоуровневого интерфейса подсистемы компиляции только усложняет задачу.

Чтобы предоставить интерфейс более высокого уровня, изолирующий клиента от этих классов, в подсистему компиляции включен также класс Compiler. (компилятор). Он определяет унифицированный интерфейс ко всем возможностям компилятора. Класс Compiler выступает в роли фасада: предлагает простой интерфейс к более сложной подсистеме. Он “склеивает” классы, реализующие функциональность компилятора, но не скрывает их полностью. Благодаря фасаду компилятора работа большинства программистов облегчается. При этом те, кому нужен доступ к средствам низкого уровня, не лишаются его.

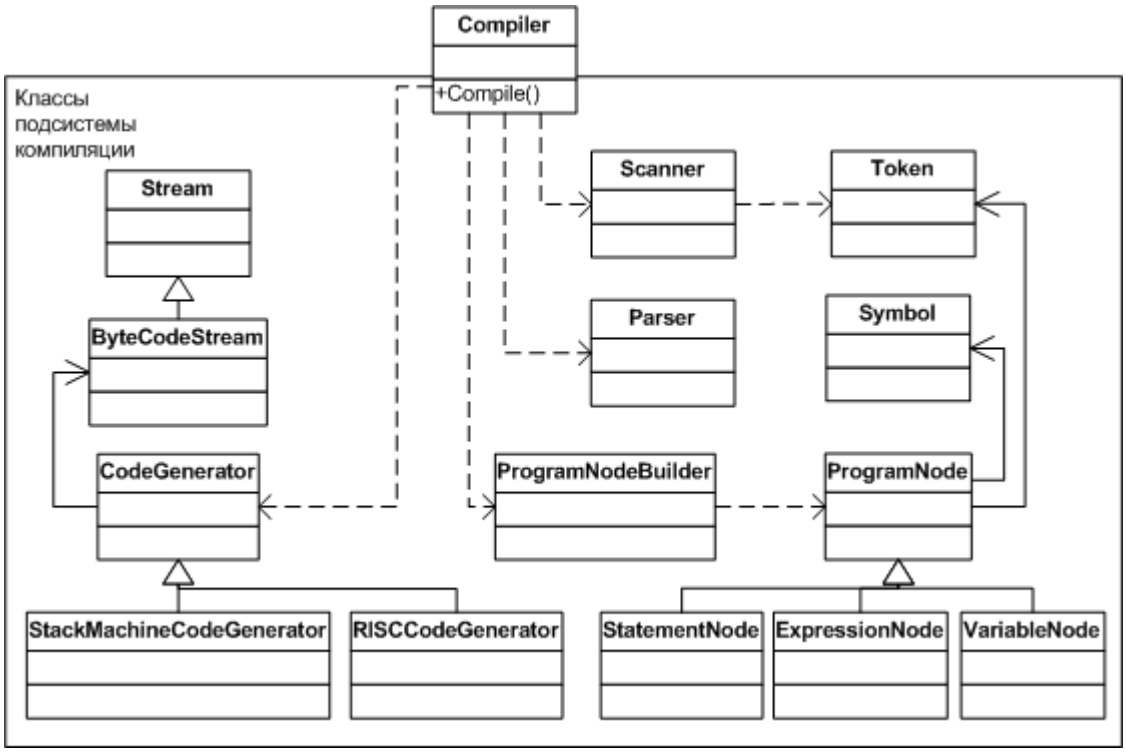

Рисунок 19

Facade: Применимость

Используйте паттерн фасад, когда:

- хотите предоставить простой интерфейс к сложной подсистеме. Часто подсистемы усложняются по мере развития. Применение большинства паттернов приводит к появлению меньших классов, но в большем количестве. Такую подсистему проще повторно использовать и настраивать под

конкретные нужды, но вместе с тем применять подсистему без настройки становится труднее. Фасад предлагает некоторый вид системы по умолчанию, устраивающий большинство клиентов. И лишь те объекты, которым нужны более широкие возможности настройки, могут обратиться напрямую к тому, что находится за фасадом;

- между клиентами и классами реализации абстракции существует много зависимостей. Фасад позволит отделить подсистему как от клиентов, так и от других подсистем, что, в свою очередь, способствует повышению степени независимости и переносимости;
- вы хотите разложить подсистему на отдельные слои. Используйте фасад для определения точки входа на каждый уровень подсистемы. Если подсистемы зависят друг от друга, то зависимость можно упростить, разрешив подсистемам обмениваться информацией только через фасады.

Facade: структура

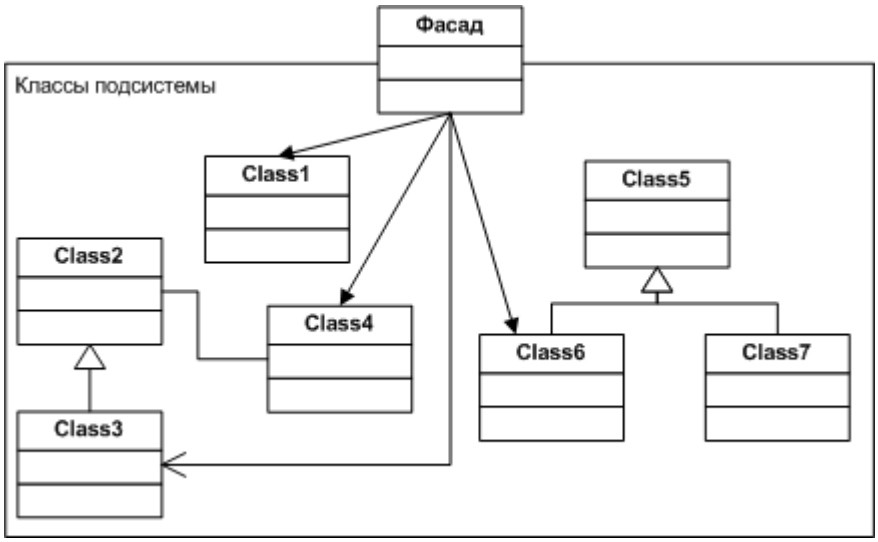

Рисунок 20

Участники

- **Facade (Compiler)** – фасад - «знает», каким классам подсистемы адресовать запрос и делегирует запросы клиентов подходящим объектам внутри подсистемы;
- **Классы подсистемы (Scanner, Parser, ProgramNode и т.д.):**

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

Отношения

Клиенты общаются с подсистемой, посылая запросы фасаду. Он переадресует их подходящим объектам внутри подсистемы. Хотя основную работу выполняют именно объекты подсистемы, фасаду, возможно, придется преобразовать свой интерфейс в интерфейсы подсистемы. Клиенты, пользующиеся фасадом, не имеют прямого доступа к объектам подсистемы.

Реализация

1) **Уменьшение степени связанности клиента с подсистемой.** Степень связанности можно значительно уменьшить, если сделать класс Facade абстрактным. Его конкретные подклассы будут соответствовать различным реализациям подсистемы. Тогда клиенты смогут взаимодействовать с подсистемой через интерфейс абстрактного класса Facade. Это изолирует клиентов от информации о том, какая реализация подсистемы используется. Вместо порождения подклассов можно сконфигурировать объект Facade различными объектами подсистем. Для настройки фасада достаточно заменить один или несколько таких объектов;
2) **Открытые и закрытые классы подсистем.** Подсистема похожа на класс в том отношении, что у обоих есть интерфейсы и оба что-то инкапсулируют. Класс инкапсулирует состояние и операции, а подсистема - классы. Открытый интерфейс подсистемы состоит из классов, к которым имеют доступ все клиенты; закрытый интерфейс доступен только для расширения подсистемы. Класс Facade, конечно же, является частью открытого интерфейса, но это не единственная часть. Другие классы подсистемы также могут быть открытыми. Например, в

системе компиляции классы Parser и Scanner - часть открытого интерфейса.

Применение в Java API

В java.net есть класс URL, который может служить примером шаблона Facade. Он обеспечивает доступ к содержимому ресурсов, на которые указывают URL. Класс может быть клиентом класса URL и использовать его для получения содержимого URL, не будучи осведомленным о множестве классов, находящихся за фасадом, который предоставляется классом URL. С другой стороны, чтобы послать данные на URL, клиент объекта URL может вызвать его метод openConnection, который возвращает объект URLConnection, используемый объектом URL.

Паттерн Flyweight

Название и классификация образца

Приспособленец – паттерн, структурирующий объекты.

Назначение

Использует разделение для эффективной поддержки множества мелких объектов.

Flyweight: мотивация

В некоторых приложениях использование объектов могло бы быть очень полезным, но прямолинейная реализация оказывается недопустимо расточительной. Например, в большинстве редакторов документов имеются средства форматирования и редактирования текстов, в той или иной степени модульные. Объектно-ориентированные редакторы обычно применяют объекты для представления таких встроенных элементов, как таблицы и рисунки. Но они не используют объекты для представления каждого символа, несмотря на то что это увеличило бы гибкость на самых нижних уровнях приложения. Ведь тогда к рисованию и форматированию символов и встроенных элементов можно было бы

применить единообразный подход. И для поддержки новых наборов символов не пришлось бы как-либо затрагивать остальные функции редактора. Да и общая структура приложения отражала бы физическую структуру документа.

У такого дизайна есть один недостаток - стоимость. Даже в документе скромных размеров было бы несколько сотен тысяч объектов-символов, а это привело бы к расходованию огромного объема памяти и неприемлемым затратам во время выполнения. Паттерн приспособленец показывает, как разделять очень мелкие объекты без недопустимо высоких издержек.

Приспособленец - это разделяемый объект, который можно использовать одновременно в нескольких контекстах. В каждом контексте он выглядит как независимый объект, то есть неотличим от экземпляра, который не разделяется. Приспособленцы не могут делать предположений о контексте, в котором работают.

Ключевая идея здесь - различие между внутренним и внешним состояниями.

Внутреннее состояние хранится в самом приспособленце и состоит из информации, не зависящей от его контекста. Именно поэтому он может разделяться. Внешнее состояние зависит от контекста и изменяется вместе с ним, поэтому не подлежит разделению. Объекты-клиенты отвечают за передачу внешнего состояния приспособленцу, когда в этом возникает необходимость.

Приспособленцы моделируют концепции или сущности, число которых слишком велико для представления объектами. Например, редактор документов мог бы создать по одному приспособленцу для каждой буквы алфавита. Каждый приспособленец хранит код символа, но координаты положения символа в документе и стиль его начертания определяются алгоритмами размещения текста и командами форматирования, действующими в том месте, где символ появляется. Код символа - это внутреннее состояние, а все остальное - внешнее.

Логически для каждого вхождения данного символа в документ существует объект. Физически, однако, есть лишь по одному объекту-приспособленцу для каждого символа, который появляется в различных контекстах в структуре

документа. Каждое вхождение данного объекта-символа ссылается на один и тот же экземпляр в разделяемом пуле объектов-приспособленцев.

Ниже изображена структура класса для этих объектов. Glyph - это абстрактный класс для представления графических объектов (некоторые из них могут быть приспособленцами). Операции, которые могут зависеть от внешнего состояния, передают его в качестве параметра. Например, операциям Draw (рисование) и Intersects (пересечение) должно быть известно, в каком контексте встречается глиф, иначе они не смогут выполнить то, что от них требуется.

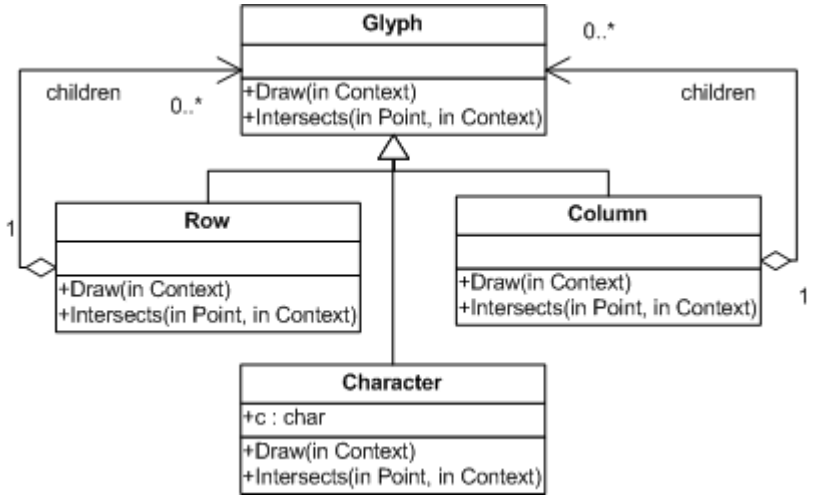

Рисунок 21

Приспособленец, представляющий букву «а», содержит только соответствующий ей код; ни положение, ни шрифт буквы ему хранить не надо. Клиенты передают приспособленцу всю зависящую от контекста информацию, которая нужна, чтобы он мог изобразить себя. Например, глифу Row известно, где его потомки должны себя показать, чтобы это выглядело как горизонтальная строка. Поэтому вместе с запросом на рисование он может передавать каждому потомку координаты.

Поскольку число различных объектов-символов гораздо меньше, чем число символов в документе, то и общее количество объектов существенно меньше, чем было бы при простой реализации. Документ, в котором все символы изображаются одним шрифтом и цветом, создаст порядка 100 объектов-символов (это примерно равно числу кодов в таблице ASCII) независимо от своего размера. А поскольку в

большинстве документов применяется не более десятка различных комбинаций шрифта и цвета, то на практике эта величина возрастет несущественно. Поэтому абстракция объекта становится применимой и к отдельным символам.

Flyweight: применимость

Эффективность паттерна приспособленец во многом зависит от того, как и где он используется. Применяйте этот паттерн, когда выполнены все нижеперечисленные условия:

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

Flyweight: структура

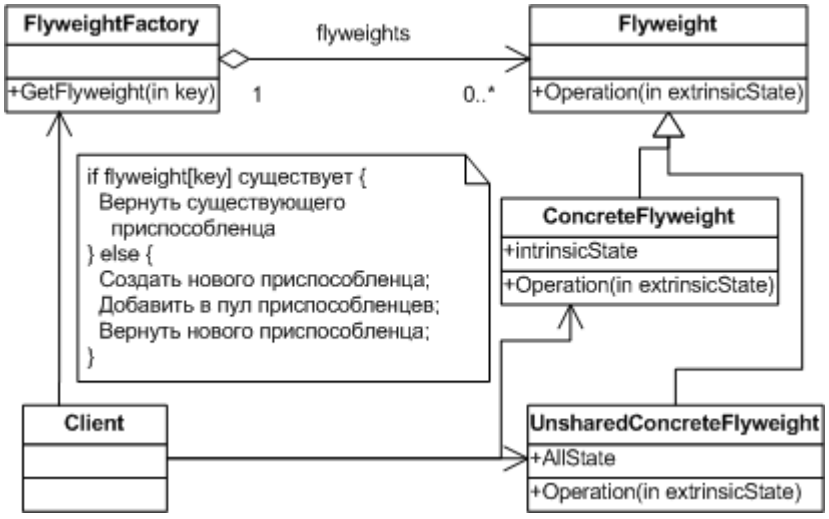

Рисунок 22

Участники

- **Flyweight (Glyph)** – приспособленец - объявляет интерфейс, с помощью которого приспособленцы могут получать внешнее состояние или как-то воздействовать на него;
- **ConcreteFlyweight (Character)** - конкретный приспособленец: - реализует интерфейс класса Flyweight и добавляет при необходимости внутреннее состояние. Объект класса ConcreteFlyweight должен быть разделяемым. Любое сохраняемое им состояние должно быть внутренним, то есть не зависящим от контекста;
- **UnsharedConcreteFlyweight (Row, Column)** - неразделяемый конкретный приспособленец - не все подклассы Flyweight обязательно должны быть разделяемыми. Интерфейс Flyweight допускает разделение, но не навязывает его. Часто у объектов UnsharedConcreteFlyweight на некотором уровне структуры приспособленца есть потомки в виде объектов класса ConcreteFlyweight, как, например, у объектов классов Row и Column;
- **FlyweightFactory** - фабрика приспособленцев - создает объекты-приспособленцы и управляет ими, а также обеспечивает должное разделение приспособленцев. Когда клиент запрашивает приспособленца, объект FlyweightFactory предоставляет существующий экземпляр или создает новый, если готового еще нет;
- **Client** – клиент - хранит ссылки на одного или нескольких приспособленцев и вычисляет или хранит внешнее состояние приспособленцев.

Отношения

Состояние, необходимое приспособленцу для нормальной работы, можно охарактеризовать как внутреннее или внешнее. Первое хранится в самом объекте ConcreteFlyweight. Внешнее состояние хранится или вычисляется клиентами. Клиент передает его приспособленцу при вызове операций. Клиенты не должны создавать экземпляры класса ConcreteFlyweight напрямую, а могут получать их

только от объекта FlyweightFactory. Это позволит гарантировать корректное разделение.

Реализация

1) **Вынесение внешнего состояния.** Применимость паттерна в значительной степени зависит от того, насколько легко идентифицировать внешнее состояние и вынести его за пределы разделяемых объектов. Вынесение внешнего состояния не уменьшает стоимости хранения, если различных внешних состояний так же много, как и объектов до разделения. Лучший вариант - внешнее состояние вычисляется по объектам с другой структурой, требующей значительно меньшей памяти. Например, в нашем редакторе документов мы можем поместить карту с типографской информацией в отдельную структуру, а не хранить шрифт и начертание вместе с каждым символом. Данная карта будет отслеживать непрерывные серии символов с одинаковыми типографскими атрибутами. Когда объект-символ изображает себя, он получает типографские атрибуты от алгоритма обхода. Поскольку обычно в документах используется немного разных шрифтов и начертаний, то хранить эту информацию отдельно от объекта-символа гораздо эффективнее, чем непосредственно в нем;
2) **Управление разделяемыми объектами.** Так как объекты разделяются, клиенты не должны инстанцировать их напрямую. Фабрика FlyweightFactory позволяет клиентам найти подходящего приспособленца. В объектах этого класса часто есть хранилище, организованное в виде ассоциативного массива, с помощью которого можно быстро находить приспособленца, нужного клиенту. Так, в примере редактора документов фабрика приспособленцев может содержать внутри себя таблицу, индексированную кодом символа, и возвращать нужного приспособленца по его коду. А если требуемый приспособленец отсутствует, он тут же создается. Разделяемость

подразумевает также, что имеется некоторая форма подсчета ссылок или сбора мусора для освобождения занимаемой приспособленцем памяти, когда необходимость в нем отпадает. Однако ни то, ни другое необязательно, если число приспособленцев фиксировано и невелико (например, если речь идет о представлении набора символов кода ASCII). В таком случае имеет смысл хранить приспособленцев постоянно.

Применение в Java API

Java использует шаблон Flyweight для управления объектами String, которые применяются для представления строковых литералов. Если в программе имеется более одного строкового литерала и эти литералы содержат одну и ту же последовательность символов, то Java использует один и тот же объект String для представления всех этих литералов. Метод intern класса String отвечает за управление объектами String, которые используются для представления строковых литералов.

Паттерн Proxy.

Название и классификация образца

Заместитель – паттерн, структурирующий объекты.

Назначение

Является суррогатом другого объекта и контролирует доступ к нему.

Известен также под именем

Surrogate (суррогат)

Proxy: мотивация

Разумно управлять доступом к объекту, поскольку тогда можно отложить расходы на создание и инициализацию до момента, когда объект действительно понадобится. Рассмотрим редактор документов, который допускает встраивание в документ графических объектов. Затраты на создание некоторых таких объектов, например больших растровых изображений, могут быть весьма значительны. Но

документ должен открываться быстро, поэтому следует избегать создания всех «тяжелых» объектов на стадии открытия (да и вообще это излишне, поскольку не все они будут видны одновременно).

В связи с такими ограничениями кажется разумным создавать «тяжелые» объекты по требованию. Это означает «когда изображение становится видимым». Но что поместить в документ вместо изображения? И как, не усложняя реализации редактора, скрыть то, что изображение создается по требованию? Например, оптимизация не должна отражаться на коде, отвечающем за рисование и форматирование.

Решение состоит в том, чтобы использовать другой объект – заместитель изображения, который временно подставляется вместо реального изображения. Заместитель ведет себя точно так же, как само изображение, и выполняет при необходимости его инстанцирование.

Заместитель создает настоящее изображение, только если редактор документа вызовет операцию Draw. Все последующие запросы заместитель переадресует непосредственно изображению. Поэтому после создания изображения он должен сохранить ссылку на него.

Предположим, что изображения хранятся в отдельных файлах. В таком случае мы можем использовать имя файла как ссылку на реальный объект. Заместитель хранит также размер изображения, то есть длину и ширину. «Зная» ее, заместитель может отвечать на запросы форматера о своем размере, не инстанцируя изображение.

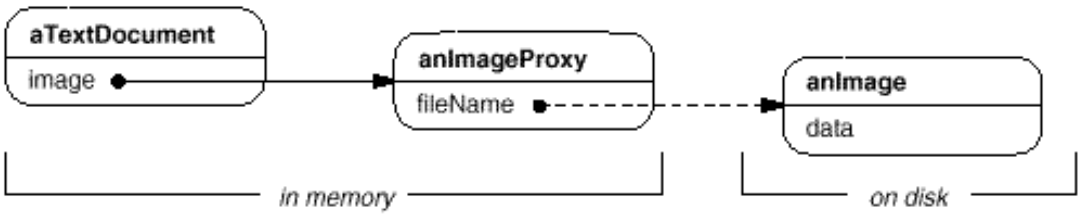

Рисунок 23

Proxy: Применимость

Паттерн заместитель применим во всех случаях, когда возникает необходимость сослаться на объект более изощренно, чем это возможно, если использовать простой указатель.

- Удаленный заместитель предоставляет локального представителя вместо объекта, находящегося в другом адресном пространстве (на другом компьютере). Такой заместитель используют RMI (удаленный вызов метода), CORBA (технология построения распределенных приложений).
- Виртуальный заместитель создает тяжелые объекты по требованию. Это выгодно в том случае, если создание объекта, предоставляющего сервис, требует больших затрат, а сервис может не понадобиться.
- Защищающий заместитель контролирует доступ к исходному объекту.

«Умная» ссылка – это замена обычного указателя. Она позволяет выполнить дополнительные действия при доступе к объекту. К типичным применениям такой ссылки можно отнести:

- подсчет числа ссылок на реальный объект;
- загрузку объекта в память при первом обращении к нему;
- проверку и установку блокировки на реальный объект при обращении к нему, чтобы никакой другой объект не смог в это время изменить его.

Proxy: структура

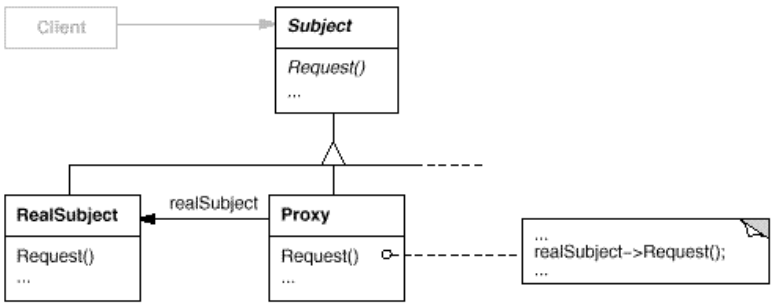

Рисунок 24

Диаграмма объектов

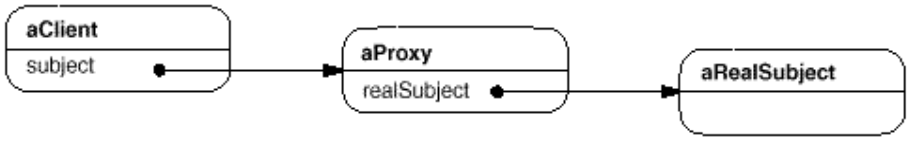

Рисунок 25

Участники

•**Proxy (imageProxy)** – заместитель - хранит ссылку, которая позволяет заместителю обратиться к реальному субъекту. Объект класса Proxy может обращаться к объекту класса Subject, если интерфейсы классов RealSubject и Subject одинаковы;
- предоставляет интерфейс, идентичный интерфейсу Subject, так что заместитель всегда может быть подставлен вместо реального субъекта;
- контролирует доступ к реальному субъекту и может отвечать за его создание и удаление;
- прочие обязанности зависят от вида заместителя:
- удаленный заместитель отвечает за кодирование запроса и его аргументов и отправление закодированного запроса реальному субъекту в другом адресном пространстве;
- виртуальный заместитель может кэшировать дополнительную информацию о реальном субъекте, чтобы отложить его создание.
- защищающий заместитель проверяет, имеет ли вызывающий объект необходимые для выполнения запроса права;

•**Subject (Graphic)** – субъект - определяет общий для RealSubject и Proxy интерфейс, так что класс Proxy можно использовать везде, где ожидается RealSubject.

•**RealSubject (Image)** - реальный субъект - определяет реальный объект, представленный заместителем.

Отношения

Proxy при необходимости переадресует запросы объекту RealSubject. Детали зависят от вида заместителя.

Паттерн Chain of Responsibility.

Название и классификация образца

Цепочка обязанностей - паттерн поведения объектов.

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.

Chain of Responsibility: 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. Нам 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. Мы говорим, что у запроса есть анонимный получатель.

Предположим, что пользователь запрашивает справку по кнопке Print (печать). Она находится в диалоговом окне PrintDialog, содержащем информацию об объекте приложения, которому принадлежит. На представленной диаграмме взаимодействий показано, как запрос на получение справки перемещается по цепочке.

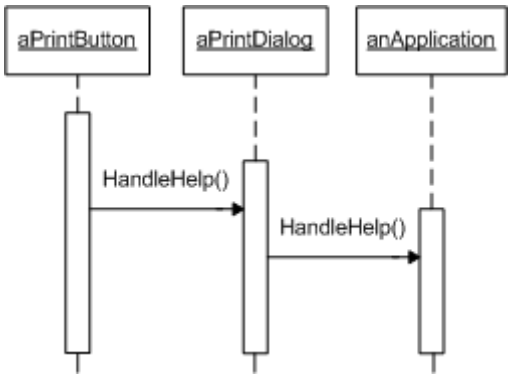

Рисунок 26

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 выполнит.

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.

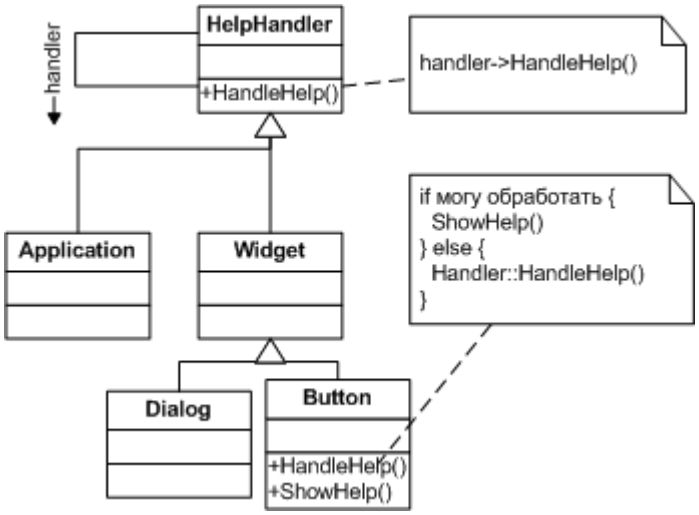

Рисунок 27

Chain of Responsibility: применимость

Используйте цепочку обязанностей, когда:

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

Chain of Responsibility: структура

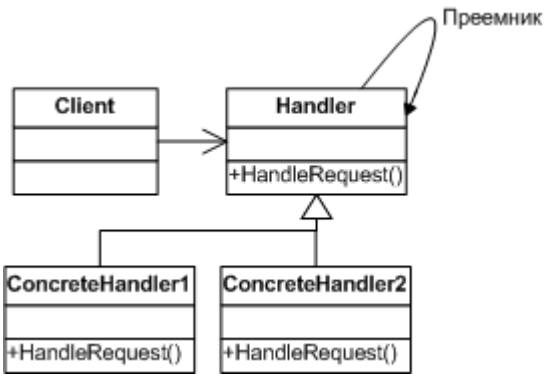

Рисунок 28

Участники

- **Handler (HelpHandler)** – обработчик - определяет интерфейс для обработки запросов и (необязательно) реализует связь с преемником;
- **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 - 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.

Реализация

1) **Реализация цепочки преемников.** Есть два способа реализовать такую цепочку: определить новые связи (обычно это делается в классе Handler, но можно и в ConcreteHandler) или использовать существующие связи. До сих пор в наших примерах определялись новые связи, однако можно воспользоваться уже имеющимися ссылками на объекты для формирования цепочки преемников. Например, ссылка на родителя в иерархии «часть-целое» может заодно определять и преемника «части». В структуре виджетов такие связи тоже могут существовать.

Существующие связи можно использовать, когда они уже поддерживают нужную цепочку. Тогда мы избежим явного определения новых связей и сэкономим память. Но если структура не отражает устройства цепочки обязанностей, то уйти от определения избыточных связей не удастся;

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.
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 поддержать 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.

Применение в Java API

В версии 1.0 языка Java использовался шаблон Chain of Responsibility для обработки событий пользовательского интерфейса. При этом в качестве цепочки применялась иерархия контейнеров пользовательского интерфейса. Если событие отправлялось кнопке или другому элементу GUI, он должен был или обработать событие, или передать его своему контейнеру. Хотя эта схема была работоспособной, существовало достаточно проблем, поэтому создатели языка Java предприняли решительные шаги по изменению модели событий в языке Java.

Паттерн Command.

Название и классификация образца

Команда – паттерн поведения объектов.

Назначение

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

Известен также под именем

Action (действие), Transaction (транзакция).

Command: мотивация

Иногда необходимо посылать объектам запросы, ничего не зная о том, выполнение какой операции запрошено и кто является получателем. Например, в библиотеках для построения пользовательских интерфейсов встречаются такие объекты, как кнопки и меню, которые посылают запрос в ответ на действие пользователя. Но в саму библиотеку не заложена возможность обрабатывать этот запрос, так как только приложение, использующее ее, располагает информацией о том, что следует сделать. Проектировщик библиотеки не владеет никакой информацией о получателе запроса и о том, какие операции тот должен выполнить.

Паттерн команда позволяет библиотечным объектам отправлять запросы неизвестным объектам приложения, преобразовав сам запрос в объект. Этот объект можно хранить и передавать, как и любой другой. В основе списываемого паттерна лежит абстрактный класс Command, в котором объявлен интерфейс для выполнения операций. В простейшей своей форме этот интерфейс состоит из одной абстрактной операции Execute. Конкретные подклассы Command определяют пару «получатель-действие», сохраняя получателя в переменной экземпляра, и реализуют операцию Execute, так чтобы она посылала запрос. У получателя есть информация, необходимая для выполнения запроса.

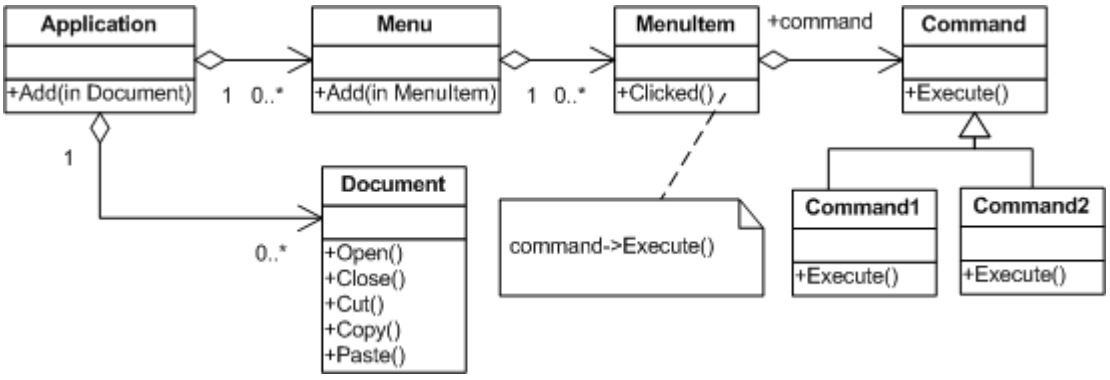

Рисунок 29

С помощью объектов Command легко реализуются меню. Каждый пункт меню - это экземпляр класса Menultem. Сами меню и все их пункты создает класс Application наряду со всеми остальными элементами пользовательского

интерфейса. Класс Application отслеживает также открытые пользователем документы.

Приложение конфигурирует каждый объект MenuItem экземпляром конкретного подкласса Command. Когда пользователь выбирает некоторый пункт меню, ассоциированный с ним объект MenuItem вызывает Execute для своего объекта-команды, а Execute выполняет операцию. Объекты MenuItem не имеют информации, какой подкласс класса Command они используют. Подклассы Command хранят информацию о получателе запроса и вызывают одну или несколько операций этого получателя.

Например, подкласс PasteCommand поддерживает вставку текста из буфера обмена в документ. Получателем для PasteCommand является Document, который был передан при создании объекта. Операция Execute вызывает операцию Paste документа-получателя.

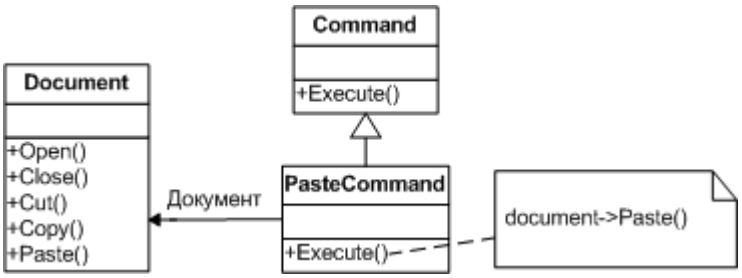

Рисунок 30

Иногда объект MenuItem должен выполнить последовательность команд. Например, пункт меню для центрирования страницы стандартного размера можно было бы сконструировать сразу из двух объектов: CenterDocumentCommand и NormalsizeCommand. Поскольку такое комбинирование команд – явление обычное, то мы можем определить класс MacroCommand, позволяющий объекту MenuItem выполнять произвольное число команд. MacroCommand - это конкретный подкласс класса Command, который просто выполняет последовательность команд. У него нет явного получателя, поскольку для каждой команды определен свой собственный.

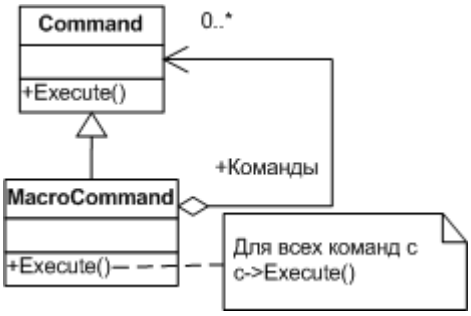

Рисунок 31

Обратите внимание, что в каждом из приведенных примеров паттерн команда отделяет объект, инициирующий операцию, от объекта, который «знает», как ее выполнить. Это позволяет добиться высокой гибкости при проектировании пользовательского интерфейса. Пункт меню и кнопка одновременно могут быть ассоциированы в приложении с некоторой функцией, для этого достаточно приписать обоим элементам один и тот же экземпляр конкретного подкласса класса Command. Мы можем динамически подменять команды, что очень полезно для реализации контекстно-зависимых меню. Можно также поддержать сценарии, если компоновать простые команды в более сложные. Все это выполнимо потому, что объект, инициирующий запрос, должен располагать информацией лишь о том, как его отправить, а не о том, как его выполнить.

Command: применимость

Используйте паттерн команда, когда хотите:

- Параметризовать объекты выполняемым действием, как в случае с пунктами меню MenuItem. В процедурном языке такую параметризацию можно выразить с помощью функции обратного вызова, то есть такой функции, которая регистрируется, чтобы быть вызванной позднее. Команды представляют собой объектно-ориентированную альтернативу функциям обратного вызова.
- Определять, ставить в очередь и выполнять запросы в разное время. Время жизни объекта Command необязательно должно зависеть от времени жизни исходного запроса. Если получателя запроса удается

реализовать так, чтобы он не зависел от адресного пространства, то объект-команду можно передать другому процессу, который займется его выполнением.

- Поддержать отмену операций. Операция Execute объекта Command может сохранить состояние, необходимое для отката действий, выполненных командой. В этом случае в интерфейсе класса Command должна быть дополнительная операция Unexecute, которая отменяет действия, выполненные предшествующим обращением к Execute. Выполненные команды хранятся в списке истории. Для реализации произвольного числа уровней отмены и повтора команд нужно обходить этот список соответственно в обратном и прямом направлениях, вызывая при посещении каждого элемента команду Unexecute или Execute.
- Поддержать протоколирование изменений, чтобы их можно было выполнить повторно после аварийной остановки системы. Дополнив интерфейс класса Command операциями сохранения и загрузки, вы сможете вести протокол изменений во внешней памяти. Для восстановления после сбоя нужно будет загрузить сохраненные команды с диска и повторно выполнить их с помощью операции Execute.
- Структурировать систему на основе высокоуровневых операций, построенных из примитивных. Такая структура типична для информационных систем, поддерживающих транзакции. Транзакция инкапсулирует набор изменений данных. Паттерн команда позволяет моделировать транзакции. У всех команд есть общий интерфейс, что дает возможность работать одинаково с любыми транзакциями. С помощью этого паттерна можно легко добавлять в систему новые виды транзакций.

Command: структура

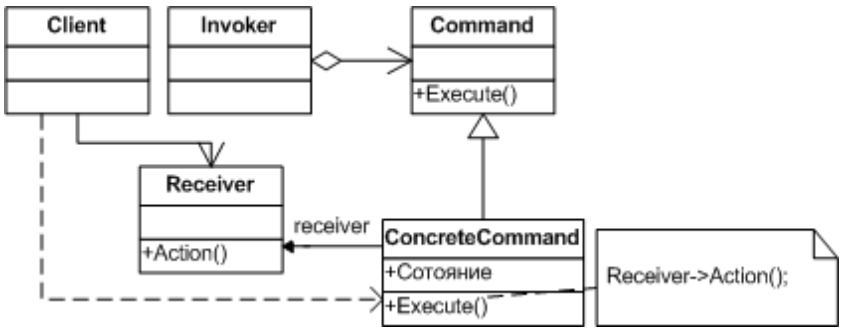

Рисунок 32

Участники

- **Command** - команда- объявляет интерфейс для выполнения операции;
- **ConcreteCommand (PasteCommand, OpenCommand)** - конкретная команда - определяет связь между объектом-получателем Receiver и действием и реализует операцию Execute путем вызова соответствующих операций объекта Receiver;
- **Client (Application)** – клиент - создает объект класса ConcreteCommand и устанавливает его получателя;
- **Invoker (Menultem)** – инициатор - обращается к команде для выполнения запроса;
- **Receiver (Document, Application)** – получатель - располагает информацией о способах выполнения операций, необходимых для удовлетворения запроса. В роли получателя может выступать любой класс.

Отношения

Клиент создает объект ConcreteCommand и устанавливает для него получателя. Инициатор Invoker сохраняет объект ConcreteCommand. Инициатор отправляет запрос, вызывая операцию команды Execute. Если поддерживается отмена выполненных действий, то ConcreteCommand перед вызовом Execute сохраняет информацию о состоянии, достаточную для выполнения отката. Объект ConcreteCommand вызывает операции получателя для выполнения запроса.

Реализация

1) **Насколько «умной» должна быть команда.** У команды может быть широкий круг обязанностей. На одном полюсе стоит простое определение связи между получателем и действиями, которые нужно выполнить для удовлетворения запроса. На другом - реализация всего самостоятельно, без обращения за помощью к получателю. Последний вариант полезен, когда вы хотите определить команды, не зависящие от существующих классов, когда подходящего получателя не существует или когда получатель команде точно не известен. Например, команда, создающая новое окно приложения, может не понимать, что именно она создает, а трактовать окно, как любой другой объект. Где-то посередине между двумя крайностями находятся команды, обладающие достаточной информацией для динамического обнаружения своего получателя;
2) **Поддержка отмены и повтора операций.** Команды могут поддерживать отмену и повтор операций, если имеется возможность отменить результаты выполнения (например, операцию Unexecute или Undo). В классе ConcreteCommand может сохраняться необходимая для этого дополнительная информация, в том числе: 1) объект-получатель Receiver, который выполняет операции в ответ на запрос; 2) аргументы операции, выполненной получателем; 3) исходные значения различных атрибутов получателя, которые могли измениться в результате обработки запроса. Получатель должен предоставить операции, позволяющие команде вернуться в исходное состояние. Для поддержки всего одного уровня отмены приложению достаточно сохранять только последнюю выполненную команду. Если же нужны многоуровневые отмена и повтор операций, то придется вести список истории выполненных команд. Максимальная длина этого списка и определяет число уровней отмены и повтора. Проход по списку в обратном направлении и откат результатов всех встретившихся по пути

команд отменяет их действие; проход в прямом направлении и выполнение встретившихся команд приводит к повтору действий. Команду, допускающую отмену, возможно, придется скопировать перед помещением в список истории. Дело в том, что объект команды, использованный для доставки запроса, скажем от пункта меню MenuIt em, позже мог быть использован для других запросов. Поэтому копирование необходимо, чтобы определить разные вызовы одной и той же команды, если ее состояние при любом вызове может изменяться. Например, команда DeleteCommand, которая удаляет выбранные объекты, при каждом вызове должна сохранять разные наборы объектов. Поэтому объект DeleteCommand необходимо скопировать после выполнения, а копию поместить в список истории. Если в результате выполнения состояние команды никогда не изменяется, то копировать не нужно - в список достаточно поместить лишь ссылку на команду. Команды, которые обязательно нужно копировать перед помещением в список истории, ведут себя подобно прототипам;

3) **Как избежать накопления ошибок в процессе отмены.** При обеспечении надежного, сохраняющего семантику механизма отмены и повтора может возникнуть проблема гистерезиса. При выполнении, отмене и повторе команд иногда накапливаются ошибки, в результате чего состояние приложения оказывается отличным от первоначального. Поэтому порой необходимо сохранять в команде больше информации, дабы гарантировать, что объекты будут целиком восстановлены. Чтобы предоставить команде доступ к этой информации, не раскрывая внутреннего устройства объектов, можно воспользоваться паттерном хранитель;
4) **Применение шаблонов в C++.** Для команд, которые не допускают отмену и не имеют аргументов, в языке C++ можно воспользоваться

шаблонами, чтобы не создавать подкласс класса Command для каждой пары действие-получатель.

Применение в Java API

Классы кнопок или пунктов меню имеют методы getActionCommand и setActionCommand, которые можно использовать для получения и задания имени команды, связанной с кнопкой или пунктом меню.

Паттерн Interpreter.

Название и классификация образца

Интерпретатор – паттерн поведения классов.

Назначение

Для заданного языка определяет представление его грамматики, а также интерпретатор предложений этого языка.

Interpreter: мотивация

Если некоторая задача возникает часто, то имеет смысл представить ее конкретные проявления в виде предложений на простом языке. Затем можно будет создать интерпретатор, который решает задачу, анализируя предложения этого языка. Например, поиск строк по образцу - весьма распространенная задача. Регулярные выражения - это стандартный язык для задания образцов поиска. Вместо того чтобы программировать специализированные алгоритмы для сопоставления строк с каждым образцом, не проще ли построить алгоритм поиска так, чтобы он мог интерпретировать регулярное выражение, описывающее множество строк-образцов? Паттерн интерпретатор определяет грамматику простого языка, представляет предложения на этом языке и интерпретирует их. Для приведенного примера паттерн описывает определение грамматики и интерпретации языка регулярных выражений.

Паттерн интерпретатор использует класс для представления каждого правила грамматики. Символы в правой части правила - это переменные экземпляров таких классов.

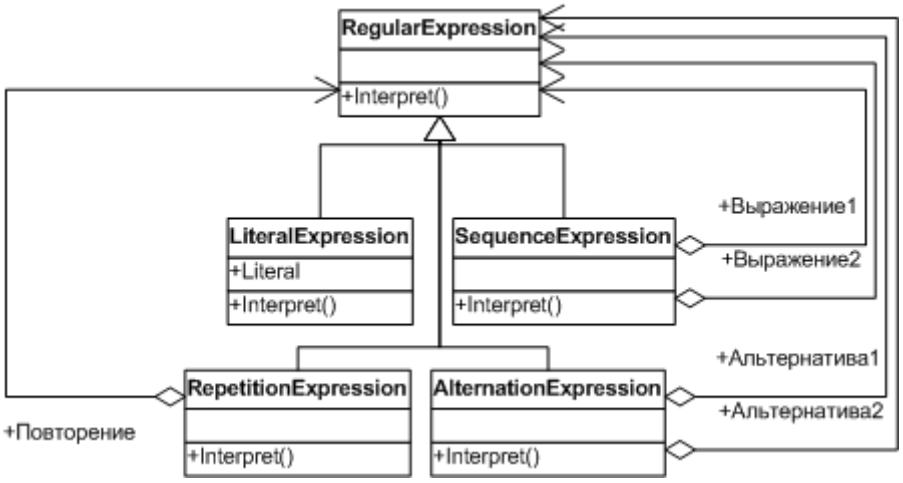

Рисунок 33

Каждое регулярное выражение, описываемое этой грамматикой, представляется в виде абстрактного синтаксического дерева, в узлах которого находятся экземпляры этих классов. Мы можем создать интерпретатор регулярных выражений, определив в каждом подклассе RegularExpression операцию Interpret, принимающую в качестве аргумента контекст, где нужно интерпретировать выражение. Контекст состоит из входной строки и информации о том, как далеко по ней мы уже продвинулись. В каждом подклассе RegularExpression операция Interpret производит сопоставление с оставшейся частью входной строки. Например:

LiteralExpression проверяет, соответствует ли входная строка литералу, который хранится в объекте подкласса;

AlternationExpression проверяет, соответствует ли строка одной из альтернатив;

RepetitionExpression проверяет, если в строке повторяющиеся вхождения выражения, совпадающего с тем, что хранится в объекте.

И так далее.

Interpreter: применимость

Используйте паттерн интерпретатор, когда есть язык для интерпретации, предложения которого можно представить в виде абстрактных синтаксических деревьев. Лучше всего этот паттерн работает, когда:

• Грамматика проста. Для сложных грамматик иерархия классов становится слишком громоздкой и неуправляемой. В таких случаях лучше применять генераторы синтаксических анализаторов, поскольку они могут интерпретировать выражения, не строя абстрактных синтаксических деревьев, что экономит память, а возможно, и время.

• Эффективность не является главным критерием. Наиболее эффективные интерпретаторы обычно не работают непосредственно с деревьями, а сначала транслируют их в другую форму. Так, регулярное выражение часто преобразуют в конечный автомат. Но даже в этом случае сам транслятор можно реализовать с помощью паттерна интерпретатор.

Interpreter: структура

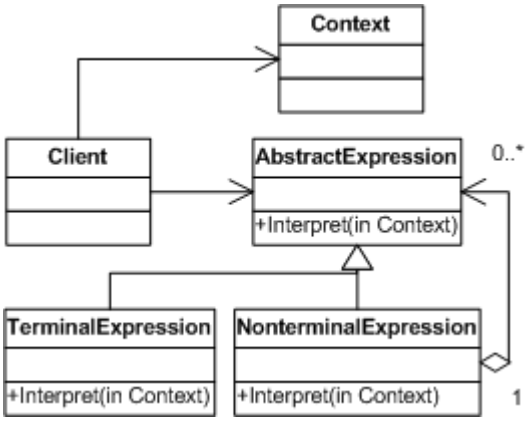

Рисунок 34

Участники

• **AbstractExpression (RegularExpression)** - абстрактное выражение - объявляет абстрактную операцию Interpret, общую для всех узлов в абстрактном синтаксическом дереве;

• **TerminalExpression (LiteralExpression)** - терминальное выражение - реализует операцию Interpret для терминальных символов грамматики;

- необходим отдельный экземпляр для каждого терминального символа в предложении;

- **NonterminalExpression(AlternationExpression, RepetitionExpression, SequenceExpressions)** - нетерминальное выражение - по одному такому классу требуется для каждого грамматического правила R :: = R_1 R_2... R_n;
- хранит переменные экземпляра типа AbstractExpression для каждого символа от R_1 до R_n;
- реализует операцию Interpret для нетерминальных символов грамматики.

- **Context** – контекст - содержит информацию, глобальную по отношению к интерпретатору;
- **Client** – клиент - строит (или получает в готовом виде) абстрактное синтаксическое дерево, представляющее отдельное предложение на языке с данной грамматикой. Дерево составлено из экземпляров классов Nonterminal-Expression и Terminal-Expression;
- вызывает операцию Interpret.

Отношения

Клиент строит (или получает в готовом виде) предложение в виде абстрактного синтаксического дерева, в узлах которого находятся объекты классов NonterminalExpression и Terminal-Expression. Затем клиент инициализирует контекст и вызывает операцию Interpret. В каждом узле вида NonterminalExpression через операции Interpret определяется операция Interpret для каждого подвыражения. Для класса TerminalExpression операция Interpret определяет базу рекурсии. Операции Interpret в каждом узле используют контекст для сохранения и доступа к состоянию интерпретатора.

Interpreter: особенности

- **Грамматику легко изменять и расширять.** Поскольку для представления грамматических правил в паттерне используются классы, то для изменения - или расширения грамматики можно применять наследование. Существующие выражения можно модифицировать постепенно, а новые определять как вариации старых.

- **Простая реализация грамматики.** Реализации классов, описывающих узлы абстрактного синтаксического дерева, похожи. Такие классы легко кодировать, а зачастую их может автоматически сгенерировать компилятор или генератор синтаксических анализаторов.

- **Сложные грамматики трудно сопровождать.** В паттерне интерпретатор определяется по меньшей мере один класс для каждого правила грамматики. Поэтому сопровождение грамматики с большим числом правил иногда оказывается трудной задачей. Для ее решения могут быть применены другие паттерны. Но если грамматика очень сложна, лучше прибегнуть к другим методам, например воспользоваться генератором компиляторов или синтаксических анализаторов.

- **Добавление новых способов интерпретации выражений.** Паттерн интерпретатор позволяет легко изменить способ вычисления выражений. Например, реализовать красивую печать выражения вместо проверки входящих в него типов можно, просто определив новую операцию в классах выражений. Если вам приходится часто создавать новые способы интерпретации выражений, подумайте о применении паттерна посетитель. Это поможет избежать изменения классов, описывающих грамматику.

Паттерн Iterator.

Название и классификация образца

Итератор – паттерн поведения объектов.

Назначение

Предоставляет способ последовательного доступа ко всем элементам составного объекта, не раскрывая его внутреннего представления.

Известен также под именем

Cursor (курсор)

Iterator: мотивация

Составной объект, скажем список, должен предоставлять способ доступа к своим элементам, не раскрывая их внутреннюю структуру. Более того, иногда требуется обходить список по-разному, в зависимости от решаемой задачи. Но вряд ли вы захотите засорять интерфейс класса List операциями для различных вариантов обхода, даже если все их можно предвидеть заранее. Кроме того, иногда нужно, чтобы в один и тот же момент было определено несколько активных обходов списка.

Все это позволяет сделать паттерн итератор. Основная его идея в том, чтобы за доступ к элементам и способ обхода отвечал не сам список, а отдельный объект итератор. В классе Iterator определен интерфейс для доступа к элементам списка. Объект этого класса отслеживает текущий элемент, то есть он располагает информацией, какие элементы уже посещались.

Например, класс List мог бы предусмотреть класс ListIterator.

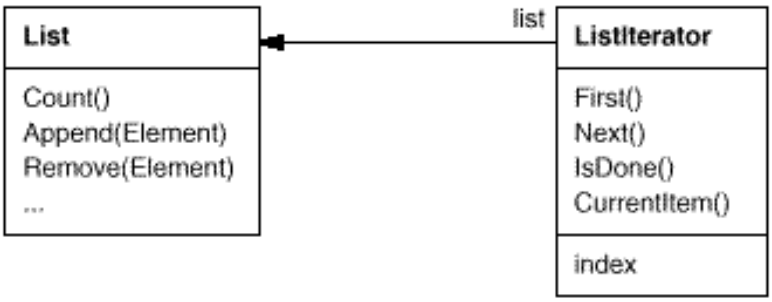

Рисунок 35

Прежде чем создавать экземпляр класса ListIterator, необходимо иметь список, подлежащий обходу. С объектом ListIterator вы можете последовательно посетить все элементы списка. Операция CurrentItem возвращает текущий элемент списка, операция First инициализирует текущий элемент первым элементом списка, Next делает текущим следующий элемент, а IsDone проверяет, не оказались ли мы за последним элементом, если да, то обход завершен.

Отделение механизма обхода от объекта List позволяет определять итераторы, реализующие различные стратегии обхода, не перечисляя их в интерфейсе класса

List. Например, FilteringListIterator мог бы предоставлять доступ только к тем элементам, которые удовлетворяют условиям фильтрации.

Заметим: между итератором и списком имеется тесная связь, клиент должен иметь информацию, что он обходит именно список, а не какую-то другую агрегированную структуру. Поэтому клиент привязан к конкретному способу агрегирования. Было бы лучше, если бы мы могли изменять класс агрегата, не трогая код клиента. Это можно сделать, обобщив концепцию итератора и рассмотрев полиморфную итерацию.

Например, предположим, что у нас есть еще класс SkipList, реализующий список. Список с пропусками (skiplist) - это вероятностная структура данных, по характеристикам напоминающая сбалансированное дерево. Нам нужно научиться писать код, способный работать с объектами как класса List, так и класса SkipList.

Определим класс AbstractList, в котором объявлен общий интерфейс для манипулирования списками. Еще нам понадобится абстрактный класс Iterator, определяющий общий интерфейс итерации. Затем мы смогли бы определить конкретные подклассы класса Iterator для различных реализаций списка. В результате механизм итерации оказывается не зависящим от конкретных агрегированных классов

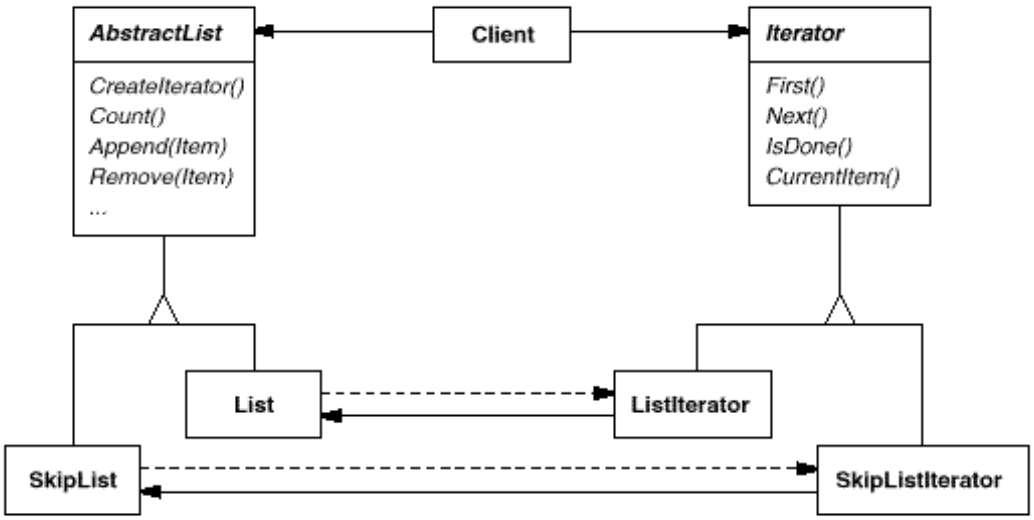

Рисунок 36

Остается понять, как создается итератор. Поскольку мы хотим написать код, независящий от конкретных подклассов List, то нельзя просто инстанцировать конкретный класс. Вместо этого мы поручим самим объектам-спискам создавать для себя подходящие итераторы, вот почему потребуется операция CreateIterator, посредством которой клиенты смогут запрашивать объект-итератор.

CreateIterator - это пример использования паттерна фабричный метод. В данном случае он служит для того, чтобы клиент мог запросить у объекта-списка подходящий итератор. Применение фабричного метода приводит к появлению двух иерархий классов - одной для списков, другой для итераторов. Фабричный метод CreateIterator «связывает» эти две иерархии.

Iterator: применимость

Используйте итератор:

- Для доступа к содержимому агрегированных объектов без раскрытия их внутреннего представления.
- Для поддержки нескольких активных обходов одного и того же агрегированного объекта.
- Для предоставления единообразного интерфейса с целью обхода различных агрегированных структур (для поддержки полиморфной итерации).

Iterator: структура

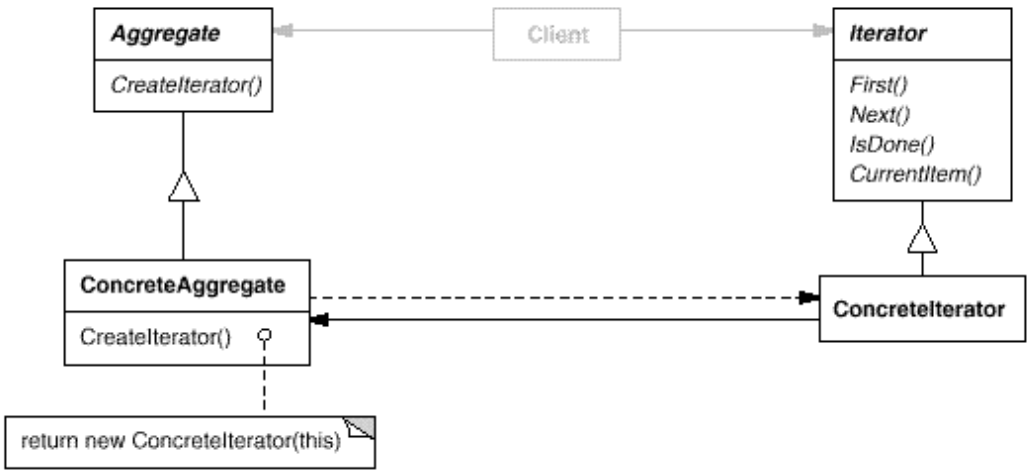

Рисунок 37

Участники

- **Iterator – итератор -** определяет интерфейс для доступа и обхода элементов;
- **Concretelterator - конкретный итератор:**

- реализует интерфейс класса Iterator;
- следит за текущей позицией при обходе агрегата;

- **Aggregate – агрегат -** определяет интерфейс для создания объекта-итератора;
- **ConcreteAggregate - конкретный агрегат** - реализует интерфейс создания итератора и возвращает экземпляр подходящего класса ConcreteIterator.

Отношения

ConcreteIterator отслеживает текущий объект в агрегате и может вычислить идущий за ним.

Iterator: особенности

- **Поддерживает различные виды обхода агрегата.** Сложные агрегаты можно обходить по-разному. Итераторы упрощают изменение алгоритма обхода - достаточно просто заменить один экземпляр итератора другим. Для поддержки новых видов обхода можно определить и подклассы класса Iterator.
- **Итераторы упрощают интерфейс класса-агрегата.** Наличие интерфейса для обхода в классе Iterator делает излишним дублирование этого интерфейса в классе Aggregate. Тем самым интерфейс агрегата упрощается;
- **Одновременно для данного агрегата может быть активно несколько обходов.** Итератор следит за инкапсулированным в нем самом состоянием обхода. Поэтому одновременно разрешается осуществлять несколько обходов агрегата.

Iterator: реализация

Существует множество вариантов реализации итератора. Ниже перечислены наиболее употребительные.

- **Какой участник управляет итерацией?** Важнейший вопрос состоит в том, что управляет итерацией: сам итератор или клиент, который им пользуется. Если итерацией управляет клиент, то итератор называется внешним, в противном случае - внутренним. Клиенты, применяющие внешний итератор должны явно запрашивать у итератора следующий элемент, чтобы двигаться дальше по агрегату. Напротив, в случае внутреннего итератора клиент передает итератору некоторую операцию, а итератор уже сам применяет эту операцию к каждому посещенному во время обхода элементу агрегата. Внешние итераторы обладают большей гибкостью, чем внутренние. Например, сравнить две коллекции на равенство с помощью внешнего итератора очень легко, а с помощью внутреннего - практически невозможно. Но, с другой стороны, внутренние итераторы проще в использовании, поскольку они вместо вас определяют логику обхода.
- **Насколько итератор устойчив?** Модификация агрегата в то время, как совершается его обход, может оказаться опасной. Если при этом добавляются или удаляются элементы, то не исключено, что некоторый элемент будет посещен дважды или вообще ни разу. Простое решение - скопировать агрегат и обходить копию, но обычно это слишком дорого. Устойчивый итератор (robust) гарантирует, что ни вставки, на удаления не помешают обходу, причем достигается это без копирования агрегата. Есть много способов реализации устойчивых итераторов. В большинстве из них итератор регистрируется в агрегате. При вставке или удалении агрегат либо подправляет внутреннее состояние всех созданных им итераторов, либо организует внутреннюю информацию так, чтобы обход выполнялся правильно.
- **Дополнительные операции итератора.** Минимальный интерфейс класса Iterator состоит из операций First, Next, IsDone и Currentltem. Но могут

оказаться полезными и некоторые дополнительные операции. Например, упорядоченные агрегаты могут предоставлять операцию Previous, позиционирующую итератор на предыдущий элемент. Для отсортированных или индексированных коллекций интерес представляет операция SkipTo, которая позиционирует итератор на объект, удовлетворяющий некоторому критерию.

- **Пустой итератор.** Это итератор, который не возвращает никаких объектов, его метод hasNext всегда возвращает false. Пустые итераторы обычно представляют собой простой класс, который реализует соответствующий интерфейс итератора. Использование пустых итераторов может упростить реализацию классов коллекции и других классов итераторов, так как в этом случае нет необходимости в коде, предназначенном для обработки специального случая нулевого обхода.

Применение в Java API

Классы коллекций в пакете java.util созданы в соответствии с шаблоном Iterator. Классы этого пакета, реализующие java.util.Collection, определяют внутренние закрытые классы, которые реализуют java.util.Iterator.

Паттерн Mediator.

Название и классификация образца

Посредник – паттерн поведения объектов.

Назначение

Определяет объект, инкапсулирующий способ взаимодействия множества объектов. Посредник обеспечивает слабую связанность системы, избавляя объекты от необходимости явно ссылаться друг на друга и позволяя тем самым независимо изменять взаимодействия между ними.

Mediator: мотивация

Объектно-ориентированное проектирование способствует распределению некоторого поведения между объектами. Но при этом в получившейся структуре

объектов может возникнуть много связей или (в худшем случае) каждому объекту придется иметь информацию обо всех остальных.

Несмотря на то что разбиение системы на множество объектов в общем случае повышает степень повторного использования, однако изобилие взаимосвязей приводит к обратному эффекту. Если взаимосвязей слишком много, тогда система подобна монолиту и маловероятно, что объект сможет работать без поддержки других объектов. Более того, существенно изменить поведение системы практически невозможно, поскольку оно распределено между многими объектами. Если вы предпримете подобную попытку, то для настройки поведения системы вам придется определять множество подклассов.

Рассмотрим реализацию диалоговых окон в графическом интерфейсе пользователя. Здесь располагается ряд виджетов: кнопки, меню, поля ввода и т.д.

Часто между разными виджетами в диалоговом окне существуют зависимости. Например, если одно из полей ввода пустое, то определенная кнопка недоступна. При выборе из списка может измениться содержимое поля ввода. И наоборот, ввод текста в некоторое поле может автоматически привести к выбору одного или нескольких элементов списка. Если в поле ввода присутствует какой-то текст, то могут быть активизированы кнопки, позволяющие произвести определенное действие над этим текстом, например изменить либо удалить его.

В разных диалоговых окнах зависимости между виджетами могут быть различными. Поэтому, несмотря на то что во всех окнах встречаются однотипные виджеты, просто взять и повторно использовать готовые классы виджетов не удастся, придется производить настройку с целью учета зависимостей. Индивидуальная настройка каждого виджета - утомительное занятие, ибо участвующих классов слишком много.

Всех этих проблем можно избежать, если инкапсулировать коллективное поведение в отдельном объекте-посреднике. Посредник отвечает за координацию взаимодействий между группой объектов. Он избавляет входящие в группу объекты

от необходимости явно ссылаться друг на друга. Все объекты располагают информацией только о посреднике, поэтому количество взаимосвязей сокращается.

Так, класс FontDialogDirector может служить посредником между виджетами в диалоговом окне. Объект этого класса «знает» обо всех виджетах в окне и координирует взаимодействие между ними, то есть выполняет функции центра коммуникаций.

Последовательность событий, в результате которых информация о выбранном элемента списка передается в поле ввода, следующая:

1. Список информирует распорядителя о происшедших в нем изменениях.

2. Распорядитель получает от списка выбранный элемент.

3. Распорядитель передает выбранный элемент полю ввода.

4. Теперь, когда поле ввода содержит какую-то информацию, распорядитель активизирует кнопки, позволяющие выполнить определенное действие (например, изменить шрифт на полужирный или курсив).

Обратите внимание на то, как распорядитель осуществляет посредничество между списком и полем ввода. Виджеты общаются друг с другом не напрямую, а через распорядитель. Им вообще не нужно владеть информацией друг о друге, они осведомлены лишь о существовании распорядителя. А коль скоро поведение локализовано в одном классе, то его несложно модифицировать или сделать совершенно другим путем расширения или замены этого класса.

Абстракцию FontDialogDirector можно было бы интегрировать в библиотеку классов так, как показано на рисунке.

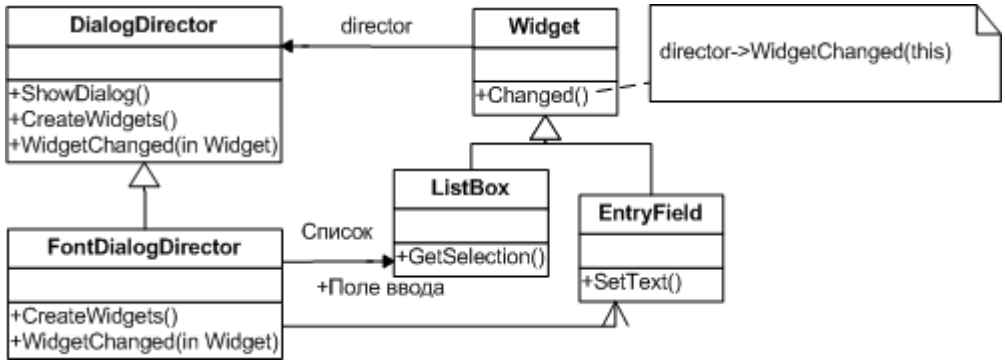

Рисунок 38

DialogDirector - это абстрактный класс, который определяет поведение диалогового окна в целом. Клиенты вызывают его операцию ShowDialog для отображения окна на экране. CreateWidgets - это абстрактная операция для создания виджетов в диалоговом окне. WidgetChanged - еще одна абстрактная операция; с ее помощью виджеты сообщают распорядителю об изменениях. Подклассы DialogDirector замещают операции CreateWidgets (для создания нужных виджетов) и WidgetChanged (для обработки извещений об изменениях).

Mediator: применимость

Используйте паттерн посредник, когда

- Имеются объекты, связи между которыми сложны и четко определены. Получающиеся при этом взаимозависимости не структурированы и трудны для понимания.
- Нельзя повторно использовать объект, поскольку он обменивается информацией со многими другими объектами.
- Поведение, распределенное между несколькими классами, должно поддаваться настройке без порождения множества подклассов.

Mediator: структура

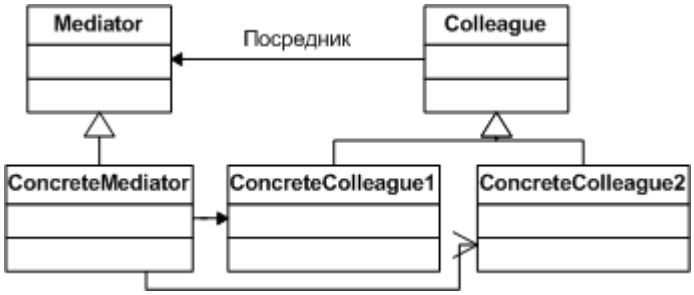

Рисунок 39

Участники

- **Mediator (DialogDirector)** – посредник - определяет интерфейс для обмена информацией с объектами Colleague;
- **ConcreteMediator (FontDialogDirector) -** конкретный посредник — реализует кооперативное поведение, координируя действия объектов Colleague и владеет информацией о коллегах и подсчитывает их;

- **Классы Colleague (ListBox, EntryField) –** коллеги — каждый класс Colleague «знает» о своем объекте Mediator и все коллеги обмениваются информацией только с посредником, так как при его отсутствии им пришлось бы общаться между собой напрямую.

Отношения

Коллеги посылают запросы посреднику и получают запросы от него. Посредник реализует кооперативное поведение путем переадресации каждого запроса подходящему коллеге (или нескольким коллегам).

Mediator: реализация

- **Избавление от абстрактного класса Mediator.** Если коллеги работают только с одним посредником, то нет необходимости определять абстрактный класс Mediator. Обеспечиваемая классом Mediator абстракция позволяет коллегам работать с разными подклассами класса Mediator и наоборот;
- **Обмен информацией между коллегами и посредником.** Коллеги должны обмениваться информацией со своим посредником только тогда, когда возникает представляющее интерес событие. Одним из подходов к реализации посредника является применение паттерна наблюдатель. Тогда классы коллег действуют как субъекты, посылающие извещения посреднику о любом изменении своего состояния. Посредник реагирует на них, сообщая об этом другим коллегам. Другой подход: в классе Mediator определяется специализированный интерфейс уведомления, который позволяет коллегам обмениваться информацией более свободно.

Применение в Java API

GUI на языке Java может быть создан, главным образом, из объектов, которые являются экземплярами подклассов класса java.awt.swing.JComponent. Объекты JComponent используют экземпляр подкласса класса java.awt.swing.FocusManager в качестве посредника.

Паттерн Memento.

Название и классификация образца

Хранитель – паттерн поведения объектов

Назначение

Не нарушая инкапсуляции, фиксирует и выносит за пределы объекта его внутреннее состояние так, чтобы позднее можно было восстановить в нем объект.

Известен также под именем

Token (лексема).

Memento: Мотивация

Иногда необходимо тем или иным способом зафиксировать внутреннее состояние объекта. Такая потребность возникает, например, при реализации контрольных точек и механизмов отката, позволяющих пользователю отменить пробную операцию или восстановить состояние после ошибки. Его необходимо где-то сохранить, чтобы позднее восстановить в нем объект. Но обычно объекты инкапсулируют всесвое состояние или хотя бы его часть, делая его недоступным для других объектов, так что сохранить состояние извне невозможно. Раскрытие же состояния явилось бы нарушением принципа инкапсуляции и поставило бы под угрозу надежность и расширяемость приложения.

Рассмотрим, например, графический редактор, который поддерживает связанность объектов. Пользователь может соединить два прямоугольника линией, и они останутся в таком положении при любых перемещениях. Редактор сам перерисовывает линию, сохраняя связанность конфигурации.

Система разрешения ограничений - хорошо известный способ поддержания связанности между объектами. Ее функции могут выполняться объектом класса ConstraintSolver, который регистрирует вновь создаваемые соединения и генерирует описывающие их математические уравнения. А когда пользователь каким-то образом модифицирует диаграмму, объект решает эти уравнения. Результаты вычислений объект ConstraintSolver использует для перерисовки графики так, чтобы были сохранены все соединения.

Поддержка отката операций в приложениях не так проста, как может показаться на первый взгляд. Очевидный способ откатить операцию перемещения - это сохранить расстояние между старым и новым положением, а затем переместить объект на такое же расстояние назад. Однако при этом не гарантируется, что все объекты окажутся там же, где находились. Предположим, что в способе расположения соединительной линии есть некоторая свобода. Тогда, переместив прямоугольник на прежнее место, мы можем не добиться желаемого эффекта.

Открытого интерфейса ConstraintSolver иногда не хватает для точного отката всех изменений смежных объектов. Механизм отката должен работать в тесном взаимодействии с ConstraintSolver для восстановления предыдущего состояния, но необходимо также позаботиться о том, чтобы внутренние детали ConstraintSolver не были доступны этому механизму.

Паттерн хранитель поможет решить данную проблему. Хранитель — это объект, в котором сохраняется внутреннее состояния другого объекта - хозяина хранителя. Для работы механизма отката нужно, чтобы хозяин предоставил хранитель, когда возникнет необходимость записать контрольную точку состояния хозяина. Только хозяину разрешено помещать в хранитель информацию и извлекать ее оттуда, для других объектов хранитель непрозрачен.

В примере графического редактора, который обсуждался выше, в роли хозяина может выступать объект ConstraintSolver. Процесс отката характеризуется такой последовательностью событий:

1. Редактор запрашивает хранитель у объекта ConstraintSolver в процессе выполнения операции перемещения.

2. ConstraintSolver создает и возвращает хранитель, в данном случае экземпляр класса SolverState..Хранитель SolverState содержит структурыданных, описывающие текущее состояние внутренних уравнений и переменных ConstraintSolver.

3. Позже, когда пользователь отменяет операцию перемещения, редактор возвращает SolverState объекту ConstraintSolver.

4. Основываясь на информации, которая хранится в объекте SolverState, ConstraintSolver изменяет свои внутренние структуры, возвращая уравнения и переменные в первоначальное состояние.

Такая организация позволяет объекту ConstraintSolver «знакомить» другие объекты с информацией, которая ему необходима для возврата в предыдущее состояние, не раскрывая в то же время свою структуру и представление.

Memento: применимость

Используйте паттерн хранитель, когда:

- Необходимо сохранить мгновенный снимок состояния объекта (или его части), чтобы впоследствии объект можно было восстановить в том же состоянии.
- Прямое получение этого состояния раскрывает детали реализации и нарушает инкапсуляцию объекта.

Memento: структура

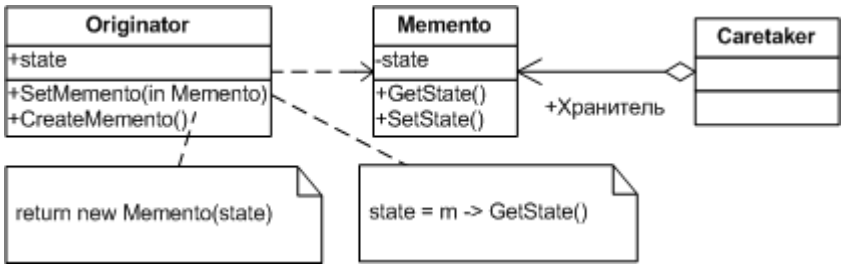

Рисунок 40

Участники

- **Memento (SolverState)** – хранитель - сохраняет внутреннее состояние объекта Originator. Объем сохраняемой информации может быть различным и определяется потребностями хозяина;
- запрещает доступ всем другим объектам, кроме хозяина. По существу, у хранителей есть два нтерфейса. «Посыльный» Caretaker «видит» лишь «узкий» интерфейс хранителя - он может только передавать хранителя другим объектам. Напротив, хозяину доступен «широкий» интерфейс, который обеспечивает доступ ко всем данным, необходимым для восстановления в

прежнем состоянии. Идеальный вариант - когда только хозяину, создавшему хранитель, открыт доступ к внутреннему состоянию последнего.

- **Originator (ConstraintSolver)** – хозяин - создает хранителя, содержащий снимок текущего внутреннего состояния и использует хранителя для восстановления внутреннего состояния.
- **Caretaker (механизм отката)** – посыльный - отвечает за сохранение хранителя и не производит никаких операций над хранителем и не исследует его внутреннее содержимое.

Отношения

Посыльный запрашивает хранитель у хозяина, некоторое время держит его у себя, а затем возвращает хозяину. Иногда этого не происходит, так как последнему не нужно восстанавливать прежнее состояние. Хранители пассивны. Только хозяин, создавший хранителя, имеет доступ к информации о состоянии.

Memento: особенности

- **Сохранение границ инкапсуляции.** Хранитель позволяет избежать раскрытия информации, которой должен распоряжаться только хозяин, но которую тем не менее необходимо хранить вне последнего. Этот паттерн экранирует объекты от потенциально сложного внутреннего устройства хозяина, не изменяя границы инкапсуляции.
- **Упрощение структуры хозяина.** При других вариантах дизайна, направленного на сохранение границ инкапсуляции, хозяин хранит внутри себя версии внутреннего состояния, которое запрашивали клиенты. Таким образом, вся ответственность за управление памятью лежит на хозяине. При перекладывании заботы о запрошенном состоянии на клиентов упрощается структура хозяина, а клиентам дается возможность не информировать хозяина о том, что они закончили работу.
- **Значительные издержки при использовании хранителей.** С хранителями могут быть связаны заметные издержки, если хозяин должен копировать большой объем информации для занесения в память хранителя

или если клиенты создают и возвращают хранителей достаточно часто. Если плата за инкапсуляцию и восстановление состояния хозяина велика, то этот паттерн не всегда подходит (см. также обсуждение инкрементности в разделе «Реализация»).

- **Определение «узкого» и «широкого» интерфейсов.** В некоторых языках сложно гарантировать, что только хозяин имеет доступ к состоянию хранителя.
- **Скрытая плата за содержание хранителя.** Посыльный отвечает за удаление хранителя, однако не располагает информацией о том, какой объем информации о состоянии скрыт в нем. Поэтому нетребовательный к ресурсам посыльный может расходовать очень много памяти при работе с хранителем.

Паттерн Observer.

Название и классификация образца

Наблюдатель – паттерн поведения объектов

Назначение

Определяет зависимость типа “один ко многим” между объектами таким образом, что при изменении состояния одного объекта все зависящие от него оповещаются об этом и автоматически обновляются

Известен также под именем

Dependents (подчинённые), Publish-Subscribe (издатель-подписчик)

Observer: применимость

- Когда у абстракции есть два аспекта, один из которых зависит от другого. Инкапсуляции этих аспектов в разные объекты позволяют изменять и повторно использовать их независимо.
- Когда при модификации одного объекта требуется изменить другие и вы не знаете, сколько именно объектов нужно изменить.

• Когда один объект должен оповещать других, не делая предположений об уведомляемых объектах. Другими словами, вы не хотите, чтобы объекты были тесно связаны между собой.

Observer: структура

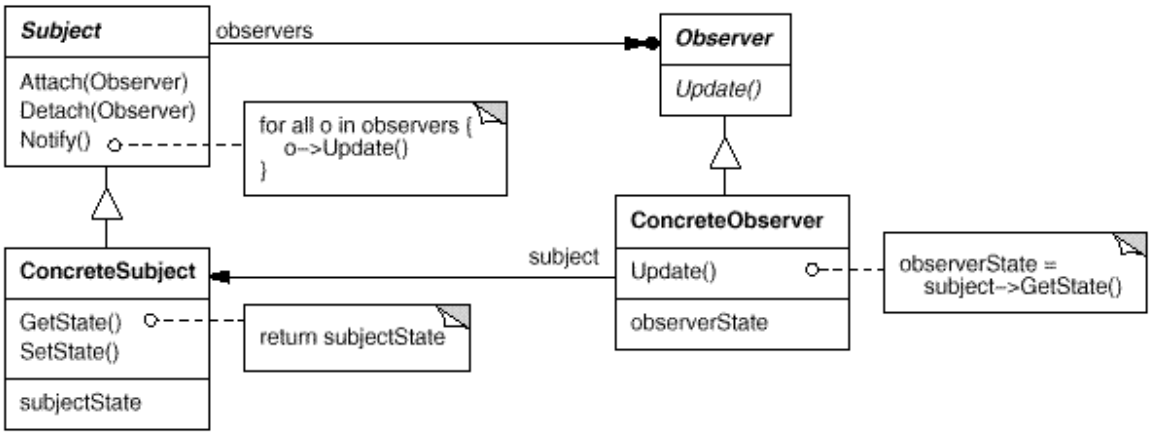

Рисунок 41

Участники

• **Subject – субъект** - располагает информацией о своих наблюдателях. За субъектом может «следить» любое число наблюдателей;
- предоставляет интерфейс для присоединения и отделения наблюдателей.

• **Observer – наблюдатель** - определяет интерфейс обновления для объектов, которые должны быть уведомлены об изменении субъекта.

• **ConcreteSubject - конкретный субъект** - сохраняет состояние, представляющее интерес для конкретного наблюдателя ConcreteObserver
- посылает информацию своим наблюдателям, когда происходит изменение.

• **a ConcreteObserver - конкретный наблюдатель -** хранит ссылку на объект класса ConcreteSubject;
- сохраняет данные, которые должны быть согласованы с данными субъекта;
- реализует интерфейс обновления, определенный в классе Observer, чтобы поддерживать согласованность с субъектом.

Отношения

- Объект ConcreteSubject уведомляет своих наблюдателей о любом изменении, которое могло бы привести к рассогласованности состояний наблюдателя и субъекта;
- После получения от конкретного субъекта уведомления об изменении объект ConcreteObserver может запросить у субъекта дополнительную информацию, которую использует для того, чтобы оказаться в состоянии, согласованном с состоянием субъекта.

На диаграмме взаимодействий показаны отношения между субъектом и двумя наблюдателями.

Observer: отношения

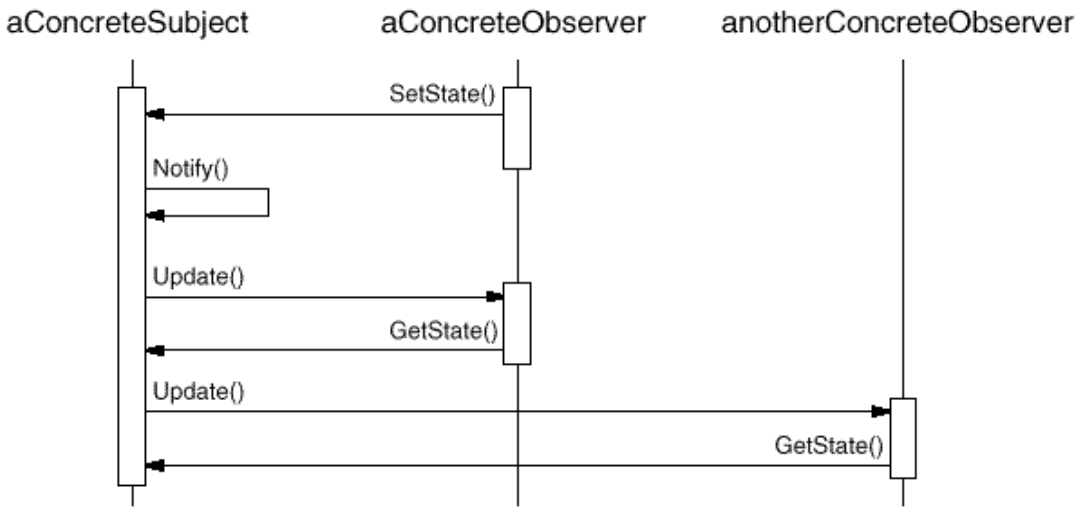

Рисунок 42

Отметим, что объект Observer, который инициирует запрос на изменение, откладывает свое обновление до получения уведомления от субъекта. Операция Notify не всегда вызывается субъектом. Ее может вызвать и наблюдатель, и посторонний объект.

Observer: результаты

- **Абстрактная связанность субъекта и наблюдателя.** Субъект имеет информацию лишь о том, что у него есть ряд наблюдателей, каждый из которых подчиняется простому интерфейсу абстрактного класса Observer. Субъекту неизвестны конкретные классы наблюдателей. Таким образом, связи

между субъектами и наблюдателями носят абстрактный характер и сведены к минимуму. Поскольку субъект и наблюдатель не являются тесно связанными, то они могут находиться на разных уровнях абстракции системы. Субъект более низкого уровня может уведомлять наблюдателей, находящихся на верхних уровнях, не нарушая иерархии системы. Если бы субъект и наблюдатель представляли собой единое целое, то получающийся объект либо пересекал бы границы уровней (нарушая принцип их формирования), либо должен был находиться на каком-то одном уровне (компрометируя абстракцию уровня).

- **Поддержка широковещательных коммуникаций.** В отличие от обычного запроса для уведомления, посылаемого субъектом, не нужно задавать определенного получателя. Уведомление автоматически поступает всем подписавшимся на него объектам. Субъекту не нужна информация о количестве таких объектов, от него требуется всего лишь уведомить своих наблюдателей. Поэтому мы можем в любое время добавлять и удалять наблюдателей. Наблюдатель сам решает, обработать полученное уведомление или игнорировать его.

- **Неожиданные обновления.** Поскольку наблюдатели не располагают информацией друг о друге, им неизвестно и о том, во что обходится изменение субъекта. Безобидная, на первый взгляд, операция над субъектом может вызвать целый ряд обновлений наблюдателей и зависящих от них объектов. Более того, нечетко определенные или плохо поддерживаемые критерии зависимости могут стать причиной непредвиденных обновлений, отследить которые очень сложно. Эта проблема усугубляется еще и тем, что простой протокол обновления не содержит никаких сведений о том, что именно изменилось в субъекте. Без дополнительного протокола, помогающего выяснить характер изменений, наблюдатели будут вынуждены проделать сложную работу для косвенного получения такой информации.

Observer: реализация

- **Отображение субъектов на наблюдателей.** С помощью этого простейшего способа субъект может отследить всех наблюдателей, которым он должен посылать уведомления, то есть хранить на них явные ссылки. Однако при наличии большого числа субъектов и всего нескольких наблюдателей это может оказаться накладно. Один из возможных компромиссов в пользу экономии памяти за счет времени состоит в том, чтобы использовать ассоциативный массив (например, хэш-таблицу) для хранения отображения между субъектами и наблюдателями. Тогда субъект, у которого нет наблюдателей, не будет зря расходовать память. С другой стороны, при таком подходе увеличивается время поиска наблюдателей.
- **Наблюдение более чем за одним субъектом.** Иногда наблюдатель может зависеть более чем от одного субъекта. Например, у электронной таблицы бывает более одного источника данных. В таких случаях необходимо расширить интерфейс Update, чтобы наблюдатель мог «узнать», какой субъект прислал уведомление. Субъект может просто передать себя в качестве параметра операции Update, тем самым сообщая наблюдателю, что именно нужно обследовать.
- **Кто инициатор обновления.** Чтобы сохранить согласованность, субъект и его наблюдатели полагаются на механизм уведомлений. Но какой именно объект вызывает операцию-Notify для инициирования обновления? Есть два варианта:

- операции класса Subject, изменившие состояние, вызывают Notify для уведомления об этом изменении. Преимущество такого подхода в том, что клиентам не надо помнить о необходимости вызывать операцию Notify субъекта. Недостаток же заключается в следующем: при выполнении каждой из нескольких последовательных операций будут производиться обновления, что может стать причиной неэффективной работы программы;
- ответственность за своевременный вызов Notify возлагается на клиента.

Преимущество: клиент может отложить инициирование обновления до завершения серии изменений, исключив тем самым ненужные промежуточные обновления. Недостаток: у клиентов появляется дополнительная обязанность. Это увеличивает вероятность ошибок, поскольку клиент может забыть вызвать Notify.

- **Как избежать зависимости протокола обновления от наблюдателя: модели вытягивания и проталкивания.** В реализациях паттерна наблюдатель субъект довольно часто транслирует всем подписчикам дополнительную информацию о характере изменения. Она передается в виде аргумента операции Update, и объем ее меняется в широких диапазонах. На одном полюсе находится так называемая модель проталкивания (push model), когда субъект посылает наблюдателям детальную информацию об изменении независимо от того, нужно ли им это. На другом - модель вытягивания (pull model), когда субъект не посылает ничего, кроме минимального уведомления, а наблюдатели запрашивают детали позднее. В модели вытягивания подчеркивается неинформированность субъекта о своих наблюдателях, а в модели проталкивания предполагается, что субъект владеет определенной информацией о потребностях наблюдателей. В случае применения модели проталкивания степень повторного их использования может снизиться, так как классы Subject предполагают о классах Observer, которые не всегда могут быть верны. С другой стороны, модель вытягивания может оказаться неэффективной, ибо наблюдателям без помощи субъекта необходимо выяснять, что изменилось.

- **Явное специфицирование представляющих интерес модификаций.** Эффективность обновления можно повысить, расширив интерфейс регистрации субъекта, то есть предоставив возможность при регистрации наблюдателя указать, какие события его интересуют. Когда событие происходит, субъект информирует лишь тех наблюдателей, которые проявили к нему интерес.

Применение в Java API

Модель делегирования событий в языке Java – это специальная форма шаблона Observer.

Паттерн State.

Название и классификация образца

Состояние - паттерн поведения объектов.

Назначение

Позволяет объекту варьировать свое поведение в зависимости от внутреннего состояния. Извне создается впечатление, что изменился класс объекта.

State: мотивация

Рассмотрим класс TCPConnection, с помощью которого представлено сетевое соединение. Объект этого класса может находиться в одном из нескольких состояний: Established (установлено), Listening (прослушивание), Closed (закрыто). Когда объект TCPConnection получает запросы от других объектов, то в зависимости от текущего состояния он отвечает по-разному. Например, ответ на запрос Open (открыть) зависит от того, находится ли соединение в состоянии Closed или Established. Паттерн состояние описывает, каким образом объект TCPConnect ion может вести себя по-разному, находясь в различных состояниях.

Основная идея этого паттерна заключается в том, чтобы ввести абстрактный класс TCPState для представления различных состояний соединения. Этот класс объявляет интерфейс, общий для всех классов, описывающих различные рабочие состояния. В подклассах TCPState реализовано поведение, специфичное для конкретного состояния. Например, в классах TCPEstabli shed и TCPClosed реализовано поведение, характерное для состояний Established и Closed соответственно.

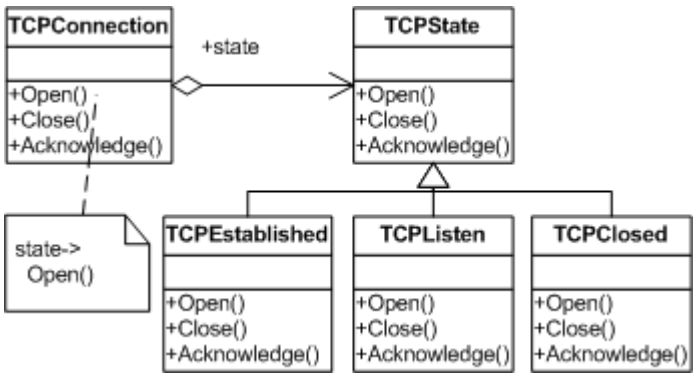

Рисунок 43

Класс TCPConnect ion хранит у себя объект состояния (экземпляр некоторого подкласса TCPState), представляющий текущее состояние соединения, и делегирует все зависящие от состояния запросы этому объекту. TCPConnection использует свой экземпляр подкласса TCPState для выполнения операций, свойственных только данному состоянию соединения.

При каждом изменении состояния соединения TCPConnection изменяет свой объект-состояние. Например, когда установленное соединение закрывается, TCPConnection заменяет экземпляр класса TCPEstablished экземпляром TCPCIosed.

State: применимость

Используйте паттерн состояние в следующих случаях:

- Когда поведение объекта зависит от его состояния и должно изменяться во время выполнения.
- Когда в коде операций встречаются состоящие из многих ветвей условные операторы, в которых выбор ветви зависит от состояния. Обычно в таком случае состояние представлено перечисляемыми константами. Часто одна и та же структура условного оператора повторяется в нескольких операциях. Паттерн состояние предлагает поместить каждую ветвь в отдельный класс. Это позволяет трактовать состояние объекта как самостоятельный объект, который может изменяться независимо от других.

State: структура

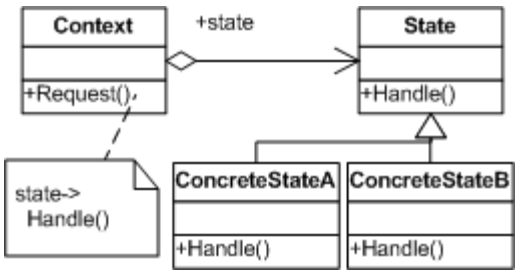

Рисунок 44

Участники

- **Context (TCPConnection)** – контекст - определяет интерфейс, представляющий интерес для клиентов и - хранит экземпляр подкласса ConcreteState, которым определяется текущее состояние;
- **State (TCPState)** – состояние - определяет интерфейс для инкапсуляции поведения, ассоциированного с конкретным состоянием контекста Context;
- **Подклассы ConcreteState (TCPEstablished, TCPListen, TCPClosed)** - конкретное состояние - каждый подкласс реализует поведение, ассоциированное с некоторым состоянием контекста Context.

Отношения

Класс Context делегирует зависящие от состояния запросы текущему объекту ConcreteState. Контекст может передать себя в качестве аргумента объекту State, который будет обрабатывать запрос. Это дает возможность объекту-состоянию при необходимости получить доступ к контексту. Context - это основной интерфейс для клиентов. Клиенты могут конфигурировать контекст объектами состояния State. Один раз сконфигурировав контекст, Клиенты уже не должны напрямую связываться с объектами состояния. Либо Context, либо подклассы ConcreteState могут решить, при каких условиях и в каком порядке происходит смена состояний.

State: реализация

- **Что определяет переходы между состояниями.** Паттерн состояние ничего не сообщает о том, какой участник определяет критерий перехода между состояниями. Если критерии зафиксированы, то их можно

реализовать непосредственно в классе Context. Однако в общем случае более гибкий и правильный подход заключается в том, чтобы позволить самим подклассам класса State определять следующее состояние и момент перехода. Для этого в класс Context надо добавить интерфейс, позволяющий объектам State установить состояние контекста. Такую децентрализованную логику переходов проще модифицировать и расширять - нужно лишь определить новые подклассы State. Недостаток децентрализации в том, что каждый подкласс State должен «знать» еще хотя бы об одном подклассе, что вносит реализационные зависимости между подклассами.

- **Табличная альтернатива.** Том Каргилл в книге C++Programming Style описывает другой способ структурирования кода, управляемого сменой состояний. Он использует таблицу для отображения входных данных на переходы между состояниями. С ее помощью можно определить, в какое состояние нужно перейти при поступлении некоторых входных данных. По существу, тем самым мы заменяем условный код поиском в таблице. Основное преимущество таблиц - в их регулярности: для изменения критериев перехода достаточно модифицировать только данные, а не код. Но есть и недостатки:

- поиск в таблице часто менее эффективен, чем вызов функции (виртуальной);
- представление логики переходов в однородном табличном формате делает критерии менее явными и, стало быть, более сложными для понимания;
- обычно трудно добавить действия, которыми сопровождаются переходы между состояниями. Табличный метод учитывает состояния и переходы между ними, но его необходимо дополнить, чтобы при каждом изменении состоянии можно было выполнять произвольные вычисления. Главное различие между конечными автоматами на базе таблиц и паттерном состояние можно сформулировать так: паттерн состояние моделирует поведение, зависящее от состояния, а табличный метод акцентирует внимание на определении переходов между состояниями.

- **Создание и уничтожение объектов состояния**. В процессе разработки обычно приходится выбирать между:
  - созданием объектов состояния, когда в них возникает необходимость, и уничтожением сразу после использования;
  - созданием их заранее и навсегда.

  Первый вариант предпочтителен, когда заранее неизвестно, в какие состояния будет попадать система, и контекст изменяет состояние сравнительно редко. При этом мы не создаем объектов, которые никогда не будут использованы, что существенно, если в объектах состояния хранится много информации. Когда изменения состояния происходят часто, поэтому не хотелось бы уничтожать представляющие их объекты (ибо они могут очень скоро понадобиться вновь), следует воспользоваться вторым подходом. Время на создание объектов затрачивается только один раз, в самом начале, а на уничтожение - не затрачивается вовсе. Правда, этот подход может оказаться неудобным, так как в контексте должны храниться ссылки на все состояния, в которые система теоретически может попасть.

Паттерн Strategy.

Название и классификация образца

Стратегия - паттерн поведения объектов.

Назначение

Определяет семейство алгоритмов, инкапсулирует каждый из них и делает их взаимозаменяемыми. Стратегия позволяет изменять алгоритмы независимо от клиентов, которые ими пользуются.

Известен также под именем

Policy (политика)

Strategy: мотивация

Существует много алгоритмов для разбиения текста на строки. Жестко «зашивать» все подобные алгоритмы в классы, которые в лих нуждаются,

нежелательно по нескольким причинам:

- клиент, которому требуется алгоритм разбиения на строки, усложняется при включении в него соответствующего кода. Таким образом, клиенты становятся более громоздкими, а сопровождать их труднее, особенно если нужно поддержать сразу несколько алгоритмов;
- в зависимости от обстоятельств стоит применять тот или иной алгоритм. Не хотелось бы поддерживать несколько алгоритмов разбиения на строки, если мы не будем ими пользоваться;
- если разбиение на строки - неотъемлемая часть клиента, то задача добавления новых и модификации существующих алгоритмов усложняется.

Всех этих проблем можно избежать, если определить классы, инкапсулирующие различные алгоритмы разбиения на строки. Инкапсулированный таким образом алгоритм называется стратегией.

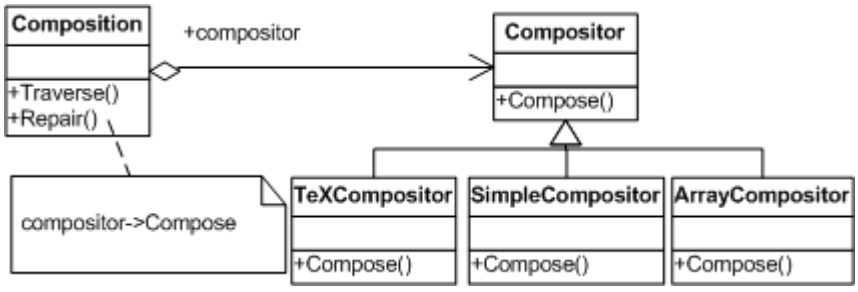

Рисунок 45

Предположим, что класс Composition отвечает за разбиение на строки текста, отображаемого в окне программы просмотра, и его своевременное обновление. Стратегии разбиения на строки определяются не в классе Composition, а в подклассах абстрактного класса Compositor. Это могут быть, например, такие стратегии:

- SimpleCompositor реализует простую стратегию, выделяющую по одной строке за раз;
- TeXCompositor реализует алгоритм поиска точек разбиения на строки, принятый в редакторе TJX. Эта стратегия пытается выполнить глобальную оптимизацию разбиения на строки, рассматривая сразу целый параграф;

- ArrayCompositor реализует стратегию расстановки переходов на новую строку таким образом, что в каждой строке оказывается одно и то же число элементов. Это полезно, например, при построчном отображении набора пиктограмм.

Объект Composition хранит ссылку на объект Compositor. Всякий раз, когда объекту Composition требуется переформатировать текст, он делегирует данную обязанность своему объекту Compositor. Клиент указывает, какой объект Compositor следует использовать, параметризуя им объект Composition.

Strategy: применимость

Используйте паттерн стратегия, когда:

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

Strategy: структура

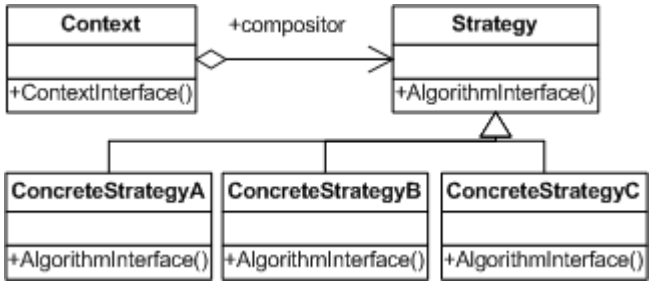

Рисунок 46

Участники

- **Strategy (Compositor)** – стратегия - объявляет общий для всех поддерживаемых алгоритмов интерфейс. Класс Context пользуется этим интерфейсом для вызова конкретного алгоритма, определенного в классе ConcreteStrategy;
- **ConcreteStrategy (SimpleCompositor, TeXCompositor, ArrayCompositor)** - конкретная стратегия - реализует алгоритм, использующий интерфейс, объявленный в классе Strategy;
- **Context (Composition)** – контекст - конфигурируется объектом класса ConcreteStrategy;

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

Отношения

Классы Strategy и Context взаимодействуют для реализации выбранного алгоритма. Контекст может передать стратегии все необходимые алгоритму данные в момент его вызова. Вместо этого контекст может позволить обращаться к своим операциям в нужные моменты, передав ссылку на самого себя операциям класса Strategy. Контекст переадресует запросы своих клиентов объекту-стратегии. Обычно клиент создает объект ConcreteStrategy и передает его контексту, после чего клиент «общается» исключительно с контекстом. Часто в распоряжении клиента находится несколько классов ConcreteStrategy, которые он может выбирать.

Strategy: реализация

- **Определение интерфейсов классов Strategy и Context.** Интерфейсы классов Strategy и Context могут обеспечить объекту класса ConcreteStrategy эффективный доступ к любым данным контекста, и наоборот. Например, Context передает данные в виде параметров операциям класса Strategy. Это разрывает тесную связь между контекстом и стратегией. При

этом не исключено, что контекст будет передавать данные, которые стратегии не нужны. Другой метод - передать контекст в качестве аргумента, в таком случае стратегия будет запрашивать у него данные, или, например, сохранить ссылку на свой контекст, так что передавать вообще ничего не придется. И в том, и в другом случаях стратегия может запрашивать только ту информацию, которая реально необходима. Но тогда в контексте должен быть определен более развитый интерфейс к своим данным, что несколько усиливает связанность классов Strategy и Context. Какой подход лучше, зависит от конкретного алгоритма и требований, которые он предъявляет к данным.

- **Стратегии как параметры шаблона.** В C++ для конфигурирования класса стратегией можно использовать шаблоны. Этот способ хорош, только если стратегия определяется на этапе компиляции и ее не нужно менять во время выполнения. Тогда конфигурируемый класс (например, Context) определяется в виде шаблона, для которого класс Strategy является параметром. Затем этот класс конфигурируется классом Strategy в момент инстанцирования. При использовании шаблонов отпадает необходимость в абстрактном классе для определения интерфейса Strategy. Кроме того, передача стратегии в виде параметра шаблона позволяет статически связать стратегию с контекстом, вследствие чего повышается эффективность программы.

- **Объекты-стратегии можно не задавать.** Класс Context разрешается упростить, если для него отсутствие какой бы то ни было стратегии является нормой. Прежде чем обращаться к объекту Strategy, объект Context проверяет наличие стратегии. Если да, то работа продолжается как обычно, в противном случае контекст реализует некое поведение по умолчанию. Достоинство такого подхода в том, что клиентам вообще не нужно иметь дело со стратегиями, если их устраивает поведение по умолчанию.

Применение в Java API

Пакет java.util.zip содержит некоторые классы, которые используют шаблон Strategy. Оба класса – CheckedInputStream и CheckedOutputStream – используют шаблон Strategy для вычисления контрольных сумм для байтовых потоков.

Паттерн Template Method

Название и классификация образца

Шаблонный метод — паттерн поведения классов.

Назначение

Шаблонный метод определяет основу алгоритма и позволяет подклассам переопределить некоторые шаги алгоритма, не изменяя его структуру в целом.

Template Method: мотивация

Рассмотрим каркас приложения, в котором имеются классы Application и Document. Класс Application отвечает за открытие существующих документов, хранящихся во внешнем формате, например в виде файла. Объект класса Document представляет информацию документа после его прочтения из файла.

Приложения, построенные на базе этого каркаса, могут порождать подклассы от классов Application и Document, отвечающие конкретным потребностям. Например, графический редактор определит подклассы DrawApplication и DrawDocument, а электронная таблица — подклассы Spreadsheet Application и SpreadsheetDocument.

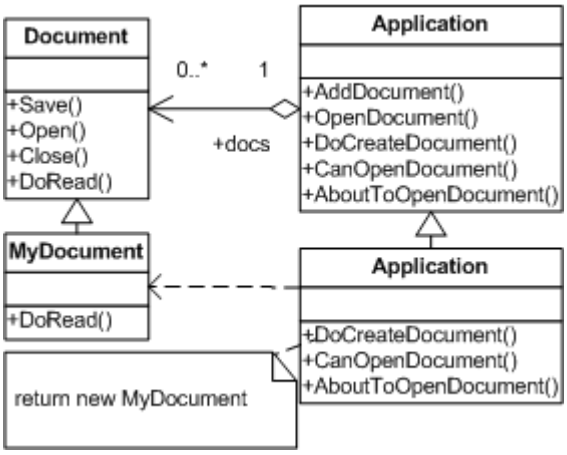

Рисунок 47

В абстрактном классе Application определен алгоритм открытия и считывания документа в операции OpenDocument.

Операция OpenDocument определяет все шаги открытия документа. Она проверяет, можно ли открыть документ, создает объект класса Document, добавляет его к набору документов и считывает документ из файла.

Операцию вида OpenDocument мы будем называть шаблонным методом, описывающим алгоритм в терминах абстрактных операций, которые замещены в подклассах для получения нужного поведения. Подклассы класса Appl icat ion выполняют проверку возможности открытия (CanOpenDocument) и создания документа (DoCreateDocument). Подклассы класса Document считывают документ (DoRead). Шаблонный метод определяет также операцию, которая позволяет подклассам Application получить информацию о том, что документ вот-вот будет открыт (AboutToOpenDocument). Определяя некоторые шаги алгоритма с помощью абстрактных операций, шаблонный метод фиксирует их последовательность, но позволяет реализовать их в подклассах классов Application и Document.

Template Method: применимость

Паттерн шаблонный метод следует использовать:

- Чтобы однократно использовать инвариантные части алгоритма, оставляя реализацию изменяющегося поведения на усмотрение подклассов;
- Когда нужно вычленить и локализовать в одном классе поведение, общее для всех подклассов, дабы избежать дублирования кода. Сначала идентифицируются различия в существующем коде, а затем они выносятся в отдельные операции. В конечном итоге различающиеся фрагменты кода заменяются шаблонным методом, из которого вызываются новые операции;
- Для управления расширениями подклассов. Можно определить шаблонный метод так, что он будет вызывать операции-зацепки в определенных точках, разрешив тем самым расширение только в этих точках.

Template Method: структура

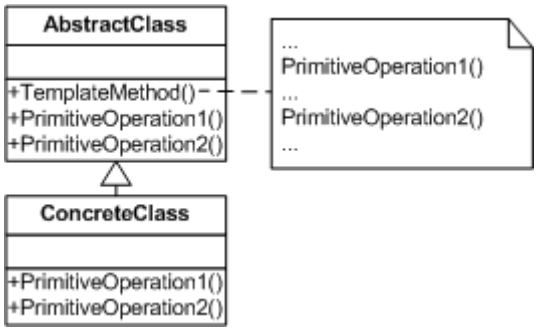

Рисунок 48

Участники

- **AbstractClass (Application)** - абстрактный класс - определяет абстрактные примитивные операции, замещаемые в конкретных подклассах для реализации шагов алгоритма и реализует шаблонный метод, определяющий скелет алгоритма. Шаблонный метод вызывает примитивные операции, а также операции, определенные в классе AbstractClass или в других объектах;
- **ConcreteClass (MyApplication)** - конкретный класс - реализует примитивные операции, выполняющие шаги алгоритма способом, который зависит от подкласса.

Отношения

ConcreteClass предполагает, что инвариантные шаги алгоритма будут выполнены в AbstractClass.

Template Method: реализация

- **Использование контроля доступа в C++.** В этом языке примитивные операции, которые вызывает шаблонный метод, можно объявить защищенными членами. Тогда гарантируется, что вызывать их сможет только сам шаблонный метод. Примитивные операции, которые обязательно нужно замещать, объявляются как чисто виртуальные функции. Сам шаблонный метод замещать не надо, так что его можно сделать невиртуальной функцией-членом;
- **Сокращение числа примитивных операций.** Важной целью при проектировании шаблонных методов является всемерное сокращение числа

примитивных операций, которые должны быть замещены в подклассах. Чем больше операций нужно замещать, тем утомительнее становится программирование клиента;

- **Соглашение об именах.** Выделить операции, которые необходимо заместить, можно путем добавления к их именам некоторого префикса. Например, в каркасе MacApp для приложений на платформе Macintosh имена шаблонных методов начинаются с префикса Do: DoCreateDocument, DoRead и т.д.

Паттерн Visitor

Название и классификация образца

Посетитель — паттерн поведения классов.

Назначение

Описывает операцию, выполняемую с каждым объектом из некоторой структуры. Паттерн посетитель позволяет определить новую операцию, не изменяя классы этих объектов.

Visitor: мотивация

Рассмотрим компилятор, который представляет программу в виде абстрактного синтаксического дерева. Над такими деревьями он должен выполнять операции «статического семантического» анализа, например проверять, что все переменные определены. Еще ему нужно генерировать код. Аналогично можно было бы определить операции контроля типов, оптимизации кода, анализа потока выполнения, проверки того, что каждой переменной было присвоено конкретное значение перед первым использованием, и т.д. Более того, абстрактные синтаксические деревья могли бы служить для красивой печати программы, реструктурирования кода и вычисления различных метрик программы. В большинстве таких операций узлы дерева, представляющие операторы присваивания, следует рассматривать иначе, чем узлы, представляющие переменные и арифметические выражения. Поэтому один класс будет создан для операторов.

присваивания, другой - для доступа к переменным, третий - для арифметических выражений и т.д. Набор классов узлов, конечно, зависит от компилируемого языка, но не очень сильно.

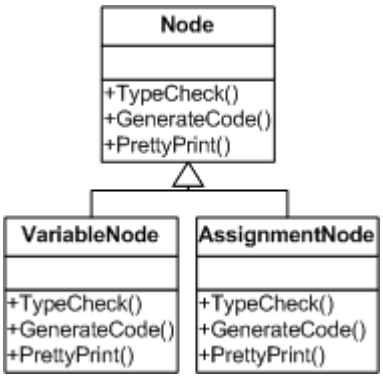

Рисунок 49

На представленной диаграмме показана часть иерархии классов Node. Проблема здесь в том, что если раскидать все операции по классам различных узлов, то получится система, которую трудно понять, сопровождать и изменять. Вряд ли кто-нибудь разберется в программе, если код, отвечающий за проверку типов, будет перемешан с кодом, реализующим красивую печать или анализ потока выполнения. Кроме того, добавление любой новой операции потребует перекомпиляции всех классов. Оптимальный вариант - наличие возможности добавлять операции по отдельности и отсутствие зависимости классов узлов от применяемых к ним операций.

И того, и другого можно добиться, если поместить взаимосвязанные операции из каждого класса в отдельный объект, называемый посетителем, и передавать его элементам абстрактного синтаксического дерева по мере обхода. ≪Принимая≫ посетителя, элемент посылает ему запрос, в котором содержится, в частности, класс элемента. Кроме того, в запросе присутствует в виде аргумента и сам элемент. Посетителю в данной ситуации предстоит выполнить операцию над элементом, ту самую, которая наверняка находилась бы в классе элемента.

Например, компилятор, который не использует посетителей, мог бы проверить тип процедуры, вызвав операцию TypeCheck для представляющего ее абстрактного синтаксического дерева. Каждый узел дерева должен был реализовать операцию

TypeCheck путем рекурсивного вызова ее же для своих компонентов (см. приведенную выше диаграмму классов). Если же компилятор проверяет тип процедуры посредством посетителей, то ему достаточно создать объект класса TypeCheckingVisitor и вызвать для дерева операцию Accept, передав ей этот объект в качестве аргумента. Каждый узел должен был реализовать Accept путем обращения к посетителю: узел, соответствующий оператору присваивания, вызывает операцию посетителя Visit Assignment, а узел, ссылающийся на переменную, - операцию VisitVariableRef erence. То, что раньше было операцией TypeCheck в классе AssignmentNode, стало операцией VisitAssignment в классе TypeCheckingVisitor.

Чтобы посетители могли заниматься не только проверкой типов, нам необходим абстрактный класс Nodevisitor, вляющийся родителем для всех посетителей синтаксического дерева. Приложение, которому нужно вычислять метрики программы, определило бы новые подклассы Nodevisitor, так что нам не пришлось бы добавлять зависящий от приложения код в классы узлов. Паттерн посетитель инкапсулирует операции, выполняемые на каждой фазе компиляции, в классе Visitor, ассоциированном с этой фазой.

Применяя паттерн посетитель, вы определяете две иерархии классов: одну для элементов, над которыми выполняется операция (иерархия Node), а другую – для посетителей, описывающих те операции, которые выполняются над элементами (иерархия NodeVisitor). Новая операция создается путем добавления подкласса в иерархию классов посетителей. До тех пор пока грамматика языка остается постоянной (то есть не добавляются новые подклассы Node), новую функциональность можно получить путем определения новых подклассов NodeVisitor.

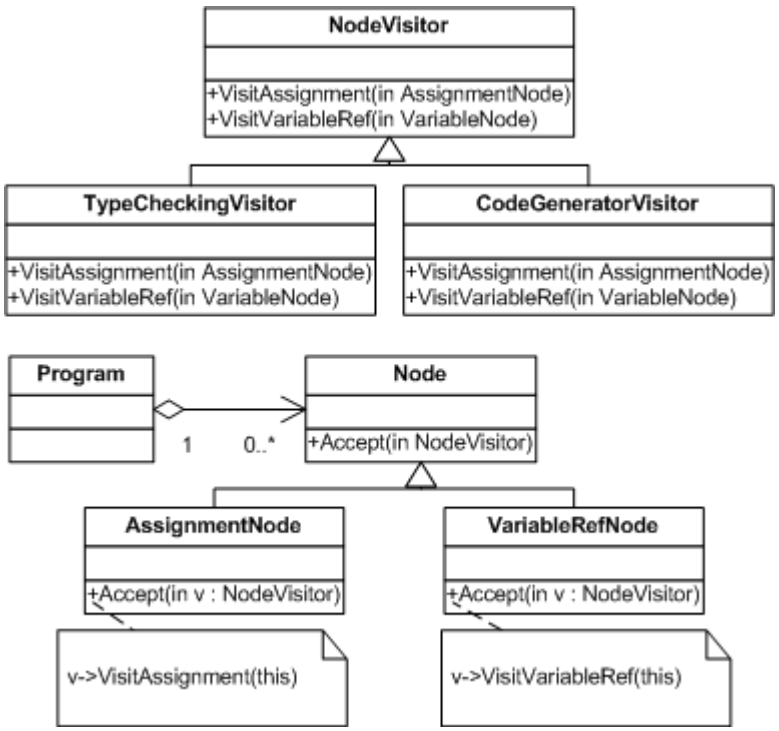

Рисунок 50

Visitor: применимость

Используйте паттерн посетитель, когда:

- В структуре присутствуют объекты многих классов с различными интерфейсами и вы хотите выполнять над ними операции, зависящие от конкретных классов;
- Над объектами, входящими в состав структуры, надо выполнять разнообразные, не связанные между собой операции и вы не хотите «засорять» классы такими операциями. Посетитель позволяет объединить родственные операции, поместив их в один класс. Если структура объектов является общей для нескольких приложений, то паттерн посетитель позволит в каждое приложение включить только относящиеся к нему операции;
- Классы, устанавливающие структуру объектов, изменяются редко, но новые операции над этой структурой добавляются часто. При изменении классов, представленных в структуре, нужно будет переопределить интерфейсы всех посетителей, а это может вызвать затруднения. Поэтому если классы меняются достаточно часто, то, вероятно, лучше определить операции прямо в них.

Visitor: структура

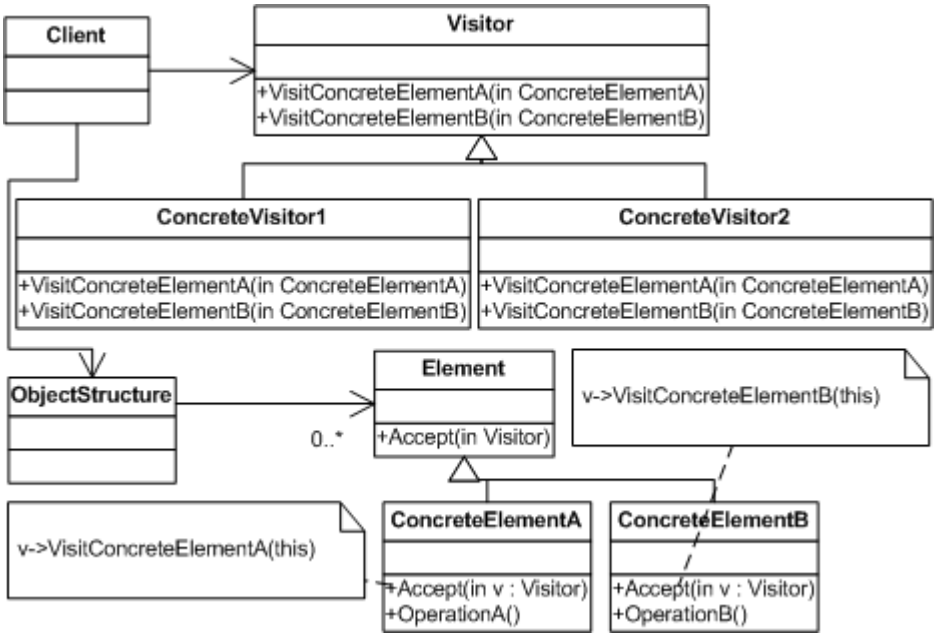

Рисунок 51

Участники

- **Visitor (NodeVisitor)** – посетитель - объявляет операцию Visit для каждого класса ConcreteElement в структуре объектов. Имя и сигнатура этой операции идентифицируют класс, который посылает посетителю запрос Visit. Это позволяет посетителю определить, элемент какого конкретного класса он посещает. Владея такой информацией, посетитель может обращаться к элементу напрямую через его интерфейс;
- **Concrete Visitor (TypeCheckingVisitor)** - конкретный посетитель - реализует все операции, объявленные в классе Visitor. Каждая операция реализует фрагмент алгоритма, определенного для класса соответствующего объекта в структуре. Класс ConcreteVisitor предоставляет контекст для этого алгоритма и сохраняет его локальное состояние. Часто в этом состоянии аккумулируются результаты, полученные в процессе обхода структуры;
- **Element (Node)** – элемент - определяет операцию Accept, которая принимает посетителя в качестве аргумента;

- **ConcreteElement (AssignmentNode, VariableRefNode)** – конкретный элемент - реализует операцию Accept, принимающую посетителя как аргумент;
- **ObjectStructure (Program)** - структура объектов - может перечислить свои элементы;
- может предоставить посетителю высокоуровневый интерфейс для посещения своих элементов;
- может быть как составным объектом (см. паттерн компоновщик), так и коллекцией, например списком или множеством.

Отношения

Клиент, использующий паттерн посетитель, должен создать объект класса ConcreteVisitor, а затем обойти всю структуру, посетив каждый ее элемент. При посещении элемента последний вызывает операцию посетителя, соответствующую своему классу. Элемент передает этой операции себя в качестве аргумента, чтобы посетитель мог при необходимости получить доступ к его состоянию.

Visitor: результаты

- **Упрощает добавление новых операций.** С помощью посетителей легко добавлять операции, зависящие от компонентов сложных объектов. Для определения новой операции над структурой объектов достаточно просто ввести нового посетителя. Напротив, если функциональность распределена по нескольким классам, то для определения новой операции придется изменить каждый класс;
- **Объединяет родственные операции и отсекает те, которые не имеют к ним отношения.** Родственное поведение не разносится по всем классам, присутствующим в структуре объектов, оно локализовано в посетителе. Не связанные друг с другом функции распределяются по отдельным подклассам класса Visitor. Это способствует упрощению как классов, определяющих элементы, так и алгоритмов, инкапсулированных в

посетителях. Все относящиеся к алгоритму структуры данных можно скрыть в посетителе;

- **Добавление новых классов ConcreteElement затруднено.** Паттерн посетитель усложняет добавление новых подклассов класса Element. Каждый новый конкретный элемент требует объявления новой абстрактной операции в классе Visitor, которую нужно реализовать в каждом из существующих классов ConcreteVisitor. Иногда большинство конкретных посетителей могут унаследовать операцию по умолчанию, предоставляемую классом Visitor, что скорее исключение, чем правило. Поэтому при решении вопроса о том, стоит ли использовать паттерн посетитель, нужно прежде всего посмотреть, что будет изменяться чаще: алгоритм, применяемый к объектам структуры, или классы объектов, составляющих эту структуру. Вполне вероятно, что сопровождать иерархию классов Visitor будет нелегко, если новые классы ConcreteElement добавляются часто. В таких случаях проще определить операции прямо в классах, представленных в структуре. Если же иерархия классов Element стабильна, но постоянно расширяется набор операций или модифицируются алгоритмы, то паттерн посетитель поможет лучше управлять такими изменениями;
- **Посещение различных иерархий классов.** Итератор может посещать объекты структуры по мере ее обхода, вызывая операции объектов. Но итератор не способен работать со структурами, состоящими из объектов разных типов. У посетителя таких ограничений нет. Ему разрешено посещать объекты, не имеющие общего родительского класса. В интерфейс класса Visitor можно добавить операции для объектов любого типа.
- **Аккумулирование состояния.** Посетители могут аккумулировать информацию о состоянии при посещении объектов структуры. Если не использовать этот паттерн, состояние придется передавать в виде дополнительных аргументов операций, выполняющих обход, или хранить в глобальных переменных;

- **Нарушение инкапсуляции.** Применение посетителей подразумевает, что у класса ConcreteElement достаточно развитый интерфейс для того, чтобы посетители могли справиться со своей работой. Поэтому при использовании данного паттерна приходится предоставлять открытые операции для доступа к внутреннему состоянию элементов, что ставит под угрозу инкапсуляцию.

Системные паттерны

Паттерн Model-View-Controller(MVC).

Представьте MP3 проигрыватель, например iTunes. Интерфейс программы используется для добавления новых песен, управления списками воспроизведения и т.д. Проигрыватель ведет базу данных с названиями и информацией о песнях и воспроизводит их, причем в процессе воспроизведения пользовательский интерфейс постоянно обновляется: в нём выводится название текущей песни, позиция воспроизведения и т.д. В основе этой модели заложен паттерн Модель-Представление-Контроллер. Вы видите как обновляется информация (название и т.д.) и слышите, как она воспроизводится. Представление обновляется автоматически. Вы хотите воспроизвести песню и работаете с интерфейсом, а ваши действия передаются контроллеру. Контроллер выполняет операции с моделью, «приказывает» модели начать воспроизведение песни. Модель содержит всю информацию состояния, данные и логику приложения, необходимые для ведения базы данных и воспроизведения MP3 файлов. Модель оповещает представление об изменении состояния.

MVC: структура

Рисунок 52

Участники

Контроллер. Получает данные, вводимые пользователем и определяет их смысл для модели.

Модель. Хранит все данные, информацию состояния и логику приложения. Она не знает о существовании представления и контроллера, хотя и предоставляет интерфейс для получения/изменения состояния,а также может отправлять оповещения об изменениях состояния наблюдателям.

Представление. Определяет представление модели (пользовательский интерфейс). Как правило, представление получает состояние и данные для отображения непосредственно от модели.

1 – Пользователь взаимодействует с моделью. Представление – «окно», через которое пользователь воспринимает модель. Когда вы делаете что-то с представлением, например, щелкаете по кнопке вопроизведения, представление сообщает контроллеру, какая операция была выполнена. Контроллер должен обработать это действие.

2 – Контроллер обращается к модели с запросами об изменении состояния. Контроллер получает действия пользователя и интерпретирует их. Если вы щелкаете на кнопке, контроллер должен разобраться, что это значит и какие опреации с моделью должны быть выполнены при данном действии.

3 – Контроллер также может обратиться к представлению с запросом об изменении. Когда контролллер получает действие от представления, в результате его обработки он может обратиться к представлению с запросом на изменение, например, заблокировать некоторые кнопки.

4 – Модель оповещает представление об изменении состояния. Когда в модели что-то изменяется, вследствие действий пользователя или других внутренних изменений, например, переходу к следующей песне в списке, модель оповещает представление об изменении состояния.

5 – Представление запрашивает у модели информацию состояния. Представление получает отбражаемую информацию состояния непосредственно от модели. Например, когда модель оповещает представление о начале вопроизведения новой песни, представление запрашивает название песни и отображает его. Представление также может запросить у модели информацию состояния в результате запроса на изменение состояния со стороны контроллера.

Контроллер является наблюдателем для модели. В некоторых архитектурах контроллер регистрируется у модели и оповещается о её изменениях, например, если какие-либо аспекты модели напрямую влияют на пользовательский интерфейс (в некоторых состояних модели отдельные элементы интерфейса могут блокироваться).

Функции контроллера не сводятся к передаче данных модели. Контроллер отвечает за интерпретацию ввода и выполнение соответствующих операций с моделью. Почему нельзя сделать всё в коде представления? Можно, но нежелательно по двум причинам. Во-первых, это усложнит код представления – у него появятся две обязанности: управление пользовательским интерфейсом и логика управления моделью. Во-вторых, между представление и моделью формируется жёсткая привязка. О повторном использовании кода представления с другой моделью можно забыть. Логическая изоляция представления и контроллера способствуют формированию более гибкой и расширяемой архитектуры, которая лучше адаптируется к возможным изменениям.

Другие типы паттернов

Паттерн Data Access Object (DAO).

Способ доступа к данным бывает разным и зависит от источника данных. Способ доступа к персистентному хранилищу, например к базе данных, очень зависит от типа этого хранилища (реляционные базы данных, объектно-ориентированные базы данных, однородные или «плоские» файлы и т.д.) и от конкретной реализации.

Многие реальные приложения платформы Java 2 Platform, Enterprise Edition (J2EE) должны использовать на некотором этапе персистентные данные. Для этих приложений персистентное хранение реализуется различными механизмами и существуют значительные отличия в API, используемых для доступа к этим механизмам. Другим приложениям может понадобиться доступ к данным, расположенным на разных системах. Например, данные могут находиться на мэйнфреймах, LDAP-репозиториях (Lightweight Directory Access Protocol - облегченный протокол доступа к каталогам) и т.д. Другим примером является ситуация, когда данные предоставляются службами, выполняющимися на разных внешних системах, таких как системы business-to-business (B2B), системы обслуживания кредитных карт и др.

Обычно приложения совместно используют распределенные компоненты для представления персистентных данных, например, компоненты управления данными. Считается, что приложение использует управляемую компонентом персистенцию (BMP- bean-managed persistence) для своих компонентов управления данными, если эти компоненты явно обращаются к персистентным данным - то есть компонент содержит код прямого доступа к хранилищу данных. Приложение с более простыми требованиями может вместо компонентов управления данными использовать сессионные компоненты или сервлеты с прямым доступом к хранилищу данных для извлечения и изменения данных. Также, приложение могло бы использовать компоненты управления данными с управляемой контейнером персистенцией, передавая, таким образом, контейнеру функции управления транзакциями и деталями персистенции.

Для доступа к данным, расположенным в системе управления реляционными базами данных (RDBMS), приложения могут использовать JDBC API. JDBC API предоставляет стандартный механизм доступа и управления данными в персистентном хранилище, таком как реляционная база данных. JDBC API позволяет в J2EE-приложениях использовать SQL-команды, являющиеся стандартным средством доступа к RDBMS-таблицам. Однако, даже внутри среды

RDBMS фактический синтаксис и формат SQL-команд может сильно зависеть от конкретной базы данных.

Для различных типов персистентных хранилищ существует еще большее число вариантов. Механизмы доступа, поддерживаемые API и функции отличаются для различных типов персистентных хранилищ, таких как RDBMS, объектно-ориентированные базы данных, плоские файлы и т.д. Приложения, которым нужен доступ к данным, расположенным на традиционных или несовместимых системах (например, мэйнфреймы или B2B-службы), часто вынуждены использовать патентованные API. Такие источники данных представляют проблему для приложений и могут потенциально создавать прямую зависимость между кодом приложения и кодом доступа к данным. Когда бизнес-компонентам (компонентам управления данными, сессионным компонентам и даже презентационным компонентам, таким как сервлеты и вспомогательные объекты для JSP-страниц) необходим доступ к источнику данных, они могут использовать соответствующий API для получения соединения и управления этим источником данных. Но включение кода для установления соединения и доступа к данным в код этих компонентов создает тесную связь между компонентами и реализацией источника данных. Такая зависимость кода в компонентах может сделать миграцию приложения от одного типа источника данных к другому трудной и громоздкой. При изменениях источника данных компоненты необходимо изменить.

Используйте Data Access Object (DAO) для абстрагирования и инкапсулирования доступа к источнику данных. DAO управляет соединением с источником данных для получения и записи данных.

DAO реализует необходимый для работы с источником данных механизм доступа. Источником данных может быть персистентное хранилище (например, RDBMS), внешняя служба (например, B2B-биржа), репозиторий (LDAP-база данных), или бизнес-служба, обращение к которой осуществляется при помощи протокола CORBA Internet Inter-ORB Protocol (IIOP) или низкоуровневых сокетов. Использующие DAO бизнес-компоненты работают с более простым интерфейсом,

предоставляемым объектом DAO своим клиентам. DAO полностью скрывает детали реализации источника данных от клиентов. Поскольку при изменениях реализации источника данных представляемый DAO интерфейс не изменяется, этот паттерн дает возможность DAO принимать различные схемы хранилищ без влияния на клиенты или бизнес-компоненты. По существу, DAO выполняет функцию адаптера между компонентом и источником данных.

DAO: структура

На рисунке 53 показана диаграмма классов, представляющая взаимоотношения в паттерне DAO.

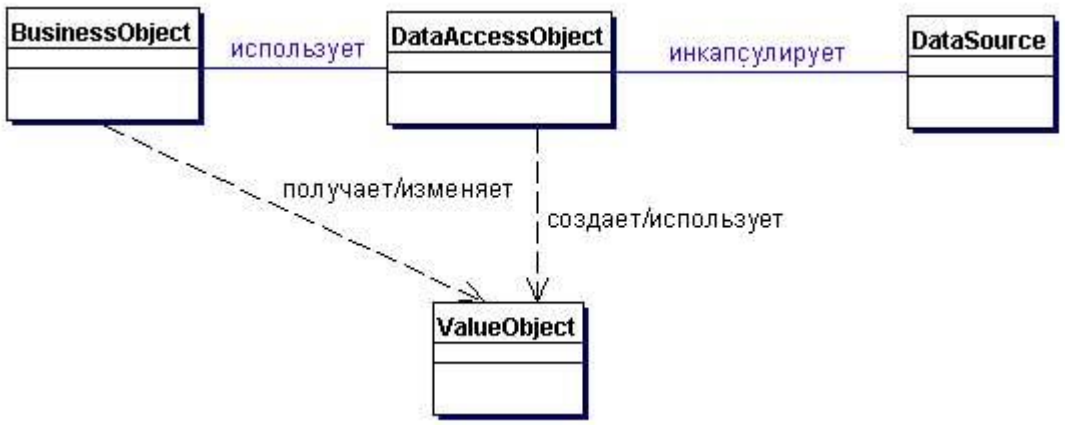

Рисунок 53

На рисунке 54 представлена диаграмма последовательности действий, показывающая взаимодействия между различными участниками в данном паттерне.

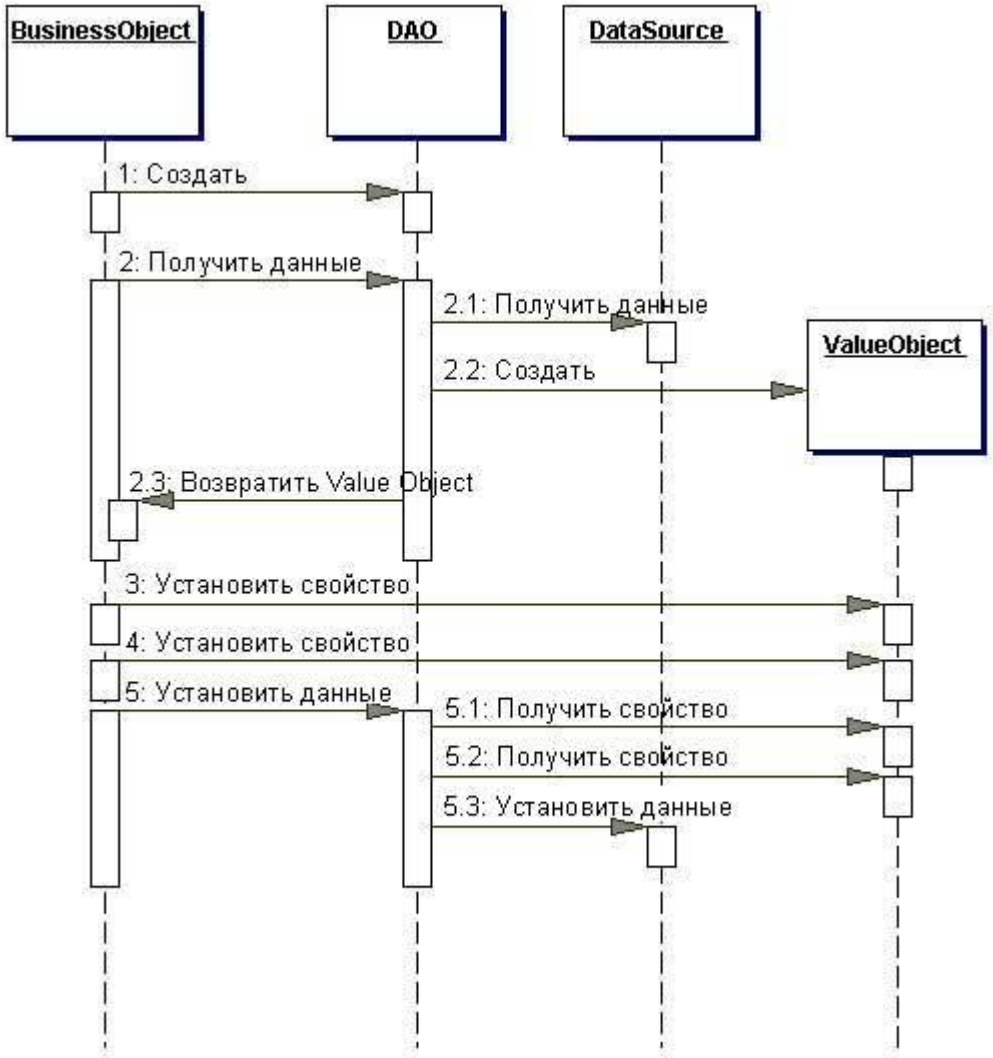

Рисунок 54

Участники

- **BusinessObject** представляет клиента данных. Это объект, который нуждается в доступе к источнику данных для получения и сохранения данных. BusinessObject может быть реализован как сессионный компонент, компонент управления данными или другой Java-объект, сервлет или вспомогательный компонент.
- **DataAccessObject** является первичным объектом данного паттерна. DataAccessObject абстрагирует используемую реализацию доступа к данным для BusinessObject, обеспечивая прозрачный доступ к источнику данных. BusinessObject передает также ответственность за выполнение операций загрузки и сохранения данных объекту DataAccessObject.

- **DataSource** представляет реализацию источника данных. Источником данных может быть база данных, например, RDBMS, OODBMS, XML-репозиторий, система плоских файлов и др. Источником данных может быть также другая система (традиционная/мэйнфрейм), служба (B2B-служба или система обслуживания кредитных карт), или какой-либо репозиторий (LDAP).
- **ValueObject** используется для передачи данных. DataAccessObject может использовать ValueObject для возврата данных клиенту. DataAccessObject может также принимать данные от клиента в объекте ValueObject для их обновления в источнике данных.

Реализация

Паттерн DAO может быть сделан очень гибким при использовании паттернов Abstract Factory и Factory Method.

Данная стратегия может быть реализована с использованием паттерна Factory Method для генерации нескольких объектов DAO, которые нужны приложению, в тех случаях, когда применяемое хранилище данных не изменяется при переходе от одной реализации к другой. Диаграмма классов этого случая приведена на рисунке 55.

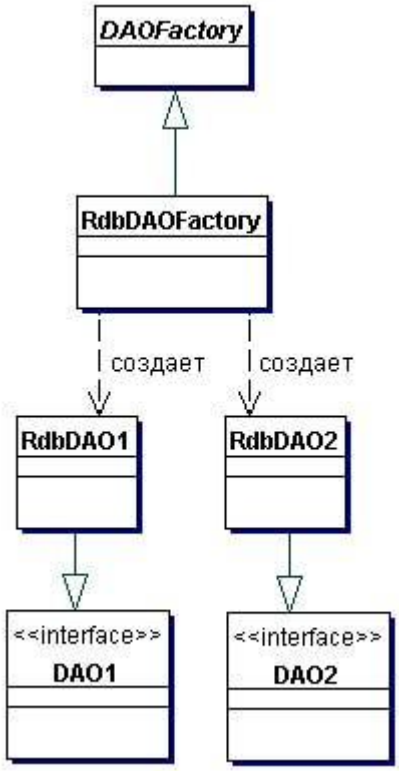

Рисунок 55

Когда используемое хранилище данных может измениться при переходе от одной реализации к другой, данная стратегия может быть реализована с применением паттерна Abstract Factory. Abstract Factory, в свою очередь, может создать и использовать реализацию Factory Method implementation. В этом случае данная стратегия предоставляет абстрактный объект генератора DAO (Abstract Factory), который может создавать конкретные генераторы DAO различного типа, причем каждый генератор может поддерживать различные типы реализаций персистентных хранилищ данных. После получения конкретного генератора для конкретной реализации вы можете использовать его для генерации объектов DAO, поддерживаемых и реализуемых в этой реализации.

Диаграмма классов этой стратегии представлена на рисунке 56. Эта диаграмма классов показывает базовый генератор DAO, являющийся абстрактным классом, который наследуется и реализуется различными конкретными генераторами DAO для поддержки доступа к специфической реализации хранилища данных. Клиент может получить реализацию конкретного генератора DAO, например

RdbDAOFactory, и использовать его для получения конкретных объектов DAO, работающих с этой конкретной реализацией хранилища данных. Например, клиент может получить RdbDAOFactory и использовать его для получения конкретных DAO, таких как RdbCustomerDAO, RdbAccountDAO и др. Объекты DAO могут расширять и реализовывать общий базовый класс (показанные как DAO1 и DAO2) и детально описывать требования к DAO для поддерживаемых бизнес-объектов. Каждый конкретный объект DAO отвечает за соединение с источником данных и за получение и управление данными для поддерживаемого им бизнес-объекта.

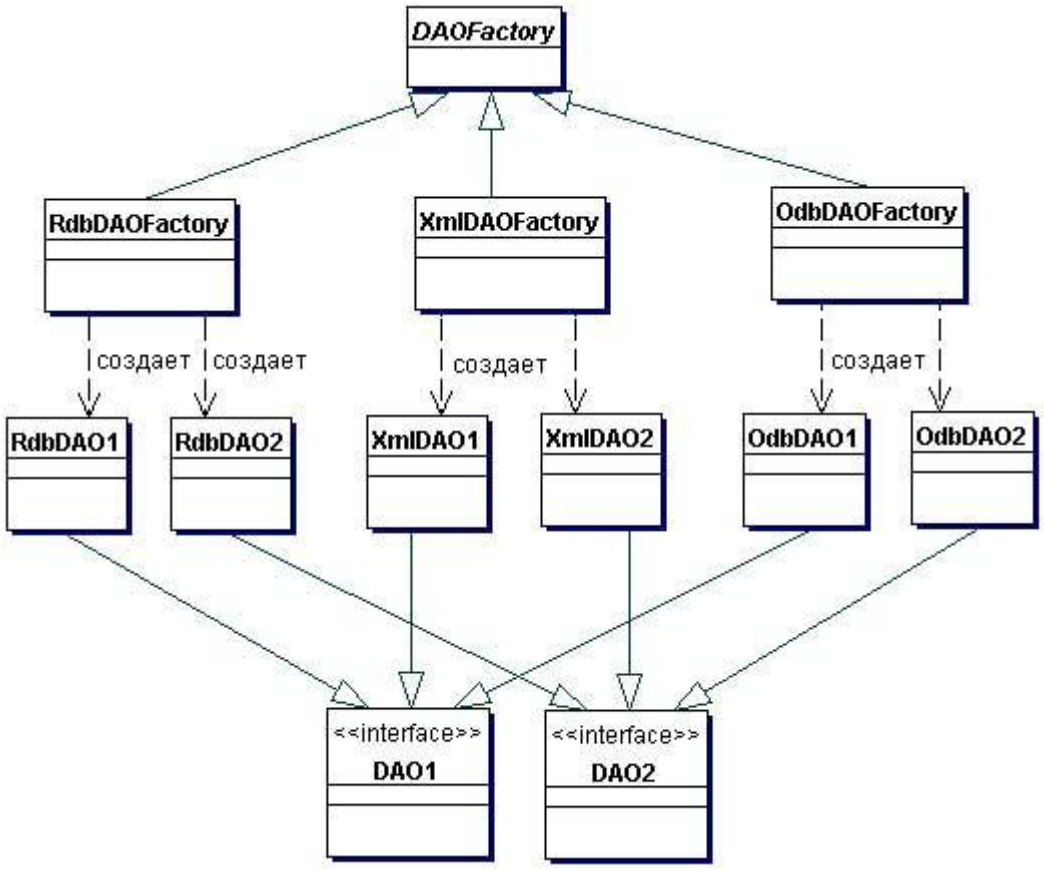

Рисунок 56

DAO: результаты

- **Разрешает прозрачность**

Бизнес-объекты могут использовать источник данных, не имея знаний о конкретных деталях его реализации. Доступ является прозрачным, поскольку детали реализации скрыты внутри DAO.

- **Облегчает миграцию**

Уровень объектов DAO облегчает приложению миграцию на другую реализацию базы данных. Бизнес-объекты не знают о деталях реализации используемых данных. Следовательно, процесс миграции требует изменений только в уровне DAO. Более того, при использовании стратегии генератора можно предоставить конкретную реализацию генератора для каждой реализации хранилища данных. В этом случае миграция на другую реализацию хранилища означает предоставление приложению новой реализации генератора.

- **Уменьшает сложность кода в бизнес-объектах**

Поскольку объекты DAO управляют всеми сложностями доступа к данным, упрощается код бизнес-компонентов и других клиентов данных, использующих DAO. Весь зависящий от реализации код (например, SQL-команды) содержится в DAO, а не в бизнес-объекте. Это улучшает читаемость кода и производительность разработки.

- **Централизует весь доступ к данным в отдельном уровне**

Поскольку все операции доступа к данным реализованы в объектах DAO, отдельный уровень доступа к данным может рассматриваться как уровень, изолирующий остальную часть приложения от реализации доступа к данным. Такая централизация облегчает поддержку и управление приложением.

- **Добавляет дополнительный уровень**

Объекты DAO создают дополнительный уровень объектов между клиентом данных и источником данных, который должен быть разработан и реализован для использования преимуществ, предлагаемых данным паттерном. Но за реализуемые при этом преимущества приходится платить дополнительными усилиями при разработке.

- **Требуется разработка иерархии классов**

При использовании стратегии генератора необходимо разработать и реализовать иерархию конкретных генераторов и иерархию конкретных объектов,

производимых генераторами. Эти дополнительные усилия необходимо принимать во внимание, если существует достаточно оснований для реализации такой гибкости. Это увеличивает сложность разработки. Однако, вы можете сначала реализовать эту стратегию с паттерном Factory Method, а затем, при необходимости, перейти к паттерну Abstract Factory.

Литература:

1. Гаврилов, А.В. Учебное пособие по языку Java. 10 задач с решением [Текст]. / Гаврилов А.В., Дегтярёва О.А., Лёзин И.А., Лёзина И.В. - Самара: Издательство СНЦ РАН, 2012. – 224 с. - ISBN 978-5-93424-619-9
2. Гамма, Э. Приемы объектно-ориентированного проектирования. Паттерны проектирования [Текст] : [пер. с англ.] / Э. Гамма, Р. Хелм, Р. Джонсон, Д. Влиссидес. - СПб.: Питер, 2013. - 368 с.: ил. - (Серия «Библиотека программиста»). - ISBN 978-5-496-00389-4
3. Гранд, Марк. Шаблоны проектирования в Java [Текст] : кат. популярных шаблонов проектирования, проиллюстрир. при помощи UML : [пер. с англ.] / Марк Гранд. - М. : Новое знание, 2004. - 558 с. - ISBN 5-94735-047-5
4. Хорстманн, Кей С. Java 2. Т. 1. Основы [Текст] / Кей С. Хорстманн, Гари Корнелл. - 8-е изд. -М.; СПб : Вильямс, 2012. - 813 с. - (Библиотека профессионала). - ISBN 978-5-8459-1378-4.- 50 экз.
5. Хорстманн, Кей С. Java 2. Т. 2. Тонкости программирования [Текст] / Кей С. Хорстманн, Гари Корнелл. - 8-е изд. - М.; СПб : Вильямс, 2012.– 983 с. – (Библиотека профессионала). - ISBN 978-5-8459-1482-8. - 50 экз.
6. Фримен, Э. Паттерны проектирования [Текст] / Эрик Фримен, Элизабет Фримен, Кэтти Сьерра, Берт Бейтс. – Спб.: Питер, 2011. – 656 с., ил.- ISBN 978-5-459-00435-9.
7. http://ooad.asf.ru/Patterns.aspx

8. http://javatutor.net/articles/j2ee-pattern-data-access-object