1 Введение

Шаблоны проектирования — это один из важнейших компонентов объектно- ориентированной технологии разработки программного обеспечения Они широко применяются в инструментах анализа, подробно описываются в книгах и часто обсуждаются на семинарах по объектно-ориентированному проектированию.

1.1 Предыстория

Много лет назад архитектор по имени Кристофер Александер задумался над вопросом: «Является ли качество объективной категорией?». Следует ли считать представление о красоте сугубо индивидуальным, или люди могут прийти к общему соглашению, согласно которому некоторые вещи будут считаться красивыми, а другие нет?

Александер размышлял о красоте с точки зрения архитектуры. Его интересовало, по каким показателям мы оцениваем архитектурные проекты. Например, если некто вознамерился спроектировать крыльцо дома, то как он может получить гарантии, что созданный им проект будет хорош? Можем ли мы знать заранее, что проект будет действительно хорош? Имеются ли объективные основания для вынесения такого суждения?

Александер принял как постулат, что в области архитектуры такое объективное основание существует. Суждение о том, что некоторое здание является красивым,— это не просто вопрос вкуса. Красоту можно описать с помощью объективных критериев, которые могут быть измерены.

К похожим выводам пришли и исследователи в области в культурологии. В пределах одной культуры большинство индивидуумов имеют схожие представления о том, что является сделанным хорошо и что является красивым. В основе их суждений есть нечто более общее, чем сугубо индивидуальные представления о красоте.

Исходная посылка создания шаблонов проектирования также состояла в необходимости объективной оценки качества программного обеспечения. Александер сформулировал для себя следующие вопросы:

1. Что есть такого в проекте хорошего качества, что отличает его от плохого проекта?
2. Что именно отличает проект низкого качества от проекта высокого качества?

Эти вопросы навели Александера на мысль о том, что если качество проекта является объективной категорией, то мы можем явно определить, что именно делает проекты хорошими, а что — плохими.

Александер изучал эту проблему, обследуя множество зданий, городов, улиц и всего прочего, что люди построили для своего проживания. В результате он обнаружил, что все, что было построено хорошо, имело между собой нечто общее. Архитектурные структуры отличаются друг от друга, даже если они относятся к одному и тому же типу. Однако, не взирая на имеющиеся различия, они могут оставаться высококачественными.

1.2 От архитектурных шаблонов к шаблонам проектирования программного обеспечения

Но какое отношение может иметь весь этот архитектурный материал к специали стам в области программного обеспечения, коими мы являемся? В начале 1990х некоторые из опытных разработчиков программного обеспе чения Gangs of Four (GOF) ознакомились с упоминавшейся выше работой Александера об архитектур ных шаблонах. Они задались вопросом, возможно ли применение идеи архитектурных шаблонов при реализации проектов в области создания программного обеспечения. Сформулируем те вопросы, на которые требовалось получить ответ.

- Существуют ли в области программного обеспечения проблемы, возникающие снова и снова, и могут ли они быть решены тем же способом?
- Возможно ли проектирование программного обеспечения в терминах шабло нов — т.е. создание конкретных решений на основе тех шаблонов, которые будут выявлены в поставленных задачах?

Интуиция подсказывала исследователям, что ответы на оба эти вопроса опреде ленно будут положительными. Следующим шагом необходимо было идентифициро вать несколько подобных шаблонов и разработать стандартные методы каталогиза ции новых. Хотя в начале 1990х над шаблонами проектирования работали многие исследова тели, наибольшее влияние на это сообщество оказала книга Гаммы, Хелма, Джонсона и Влиссайдеса Шаблоны проектирования: элементы многократного использования кода в объектноориентированном программировании. Эта работа приобрела очень широкую из вестность. Свидетельством ее популярности служит тот факт, что четыре автора кни ги получили шутливое прозвище "банда четырех". Важно понять, что авторы сами не создавали тех шаблонов, которые описаны в их книге. Скорее, они идентифицировали эти шаблоны как уже существующие в разра ботках, выполненных сообществом создателей программного обеспечения. Поэтому у некоторых разработчиков возникает вопрос, зачем изучать шаблоны?

1.3 Зачем нужно изучать шаблоны

Теперь, когда мы знаем, что такое шаблоны проектирования, можно попытаться ответить на вопрос, зачем нужно их изучать. На то имеется несколько причин, часть из которых вполне очевидна, тогда как об остальных этого не скажешь. Чаще всего причины, по которым следует изучать шаблоны проектирования, формулируют следующим образом.

- Возможность многократного использования. Повторное использование решений из уже завершенных успешных проектов позволяет быстро приступить к решению новых проблем и избежать типичных ошибок. Разработчик получает прямую выгоду от использования опыта других разработчиков, избежав необходимости вновь и вновь изобретать велосипед.

- Применение единой терминологии. Профессиональное общение и работа в группе (команде разработчиков) требует наличия единого базового словаря и единой точки зрения на проблему. Шаблоны проектирования предоставляют подобную общую точку зрения как на этапе анализа, так и при реализации проекта.

- Шаблоны проектирования предоставляют нам абстрактный высокоуровневый взгляд как на проблему, так и на весь процесс объектноориентированной разра ботки. Это помогает избежать излишней детализации на ранних стадиях проекти рования.

1.4 Другие преимущества

В результате применения шаблонов проектирования повышается эффективность труда отдельных исполнителей и всей группы в целом. Это происходит из-за того, что начинающие члены группы видят на примере более опытных разработчиков, как шаблоны проектирования могут применяться и какую пользу они приносят. Совместная работа дает новичкам стимул и реальную возможность быстрее изучить и освоить эти новые концепции. Применение многих шаблонов проектирования позволяет также создавать более модифицируемое и гибкое программное обеспечение. Причина состоит в том, что эти решения уже испытаны временем. Поэтому использование шаблонов позволяет создавать структуры, допускающие их модификацию в большей степени, чем это воз можно в случае решения, первым пришедшего на ум. Шаблоны проектирования, изученные должным образом, существенно помогают общему пониманию основных принципов объектноориентированного проектирова ния.

«Бандой четырех» было предложено несколько стратегий создания хорошего объ ектноориентированного проекта. В частности, они предложили следующее:

- Проектирование согласно интерфейсам.

- Предпочтение синтеза наследованию.

- Выявление изменяющихся величин и их инкапсуляция.

Эти стратегии использовались в большинстве шаблонов проектирования, обсуж даемых в данной книге. Для оценки полезности указанных стратегий вовсе не обяза тельно изучить большое количество шаблонов — достаточно всего нескольких. При обретенный опыт позволит применять новые концепции и к задачам собственных проектов, даже без непосредственного использования шаблонов проектирования.

Еще одно преимущество состоит в том, что шаблоны проектирования позволяют разработчику или группе разработчиков находить проектные решения для сложных проблем, не создавая громоздкой иерархии наследования классов.

2 UML-шпаргалка

Напомним о способах взаимосвязи различных классов. Если два класса как-то связаны друг с другом, то в UML диаграмме это отображается через ассоциацию. Существуют два типа ассоциаций: однонаправленная и двунаправленная. В этой связи каждый из объект класса имеет свой жизненный цикл, и не существует отношения владения. Реализация ассоциации может быть разной. Например, есть две сущности банк и пользователь банка. Банк может хранить деьги пользователя, а тот, в свою очередь, забирать деньги из банка. Это пример двунаправленной ассоциации, которая может быть реализована через импорт в каждый класс ассоциированного класса и использование его в качестве типа параметра некоторого метода.

Пример однонавправленной ассоциации: факультатив и студенты. С каждым факультативом может быть ассоциировано несколько студентов, с свою очередь студент может быть ассоциирован только с одним факультативом. Ассоциация может быть реализована через список ссылок на студентов.

Человек может пользоваться несколькими банками, тогда как студент может быть ассоциирован только с одним факультетом, то есть ассоциации могут выражать различные типы связей таких, как один-к-одному, один-ко-многим, много-ко-многим.

2.1 Аггрегация

Агрегация — это специализированная форма Ассоциации, где все объекты имеют собственный жизненный цикл, но собственность и дочерние объекты не могут принадлежать другому родительскому объекту.

Например, сотрудники EPAM не могут быть сотрудниками другой фирмы. Дочерний класс сотрудника также не может принадлежать другой фирме. Однако объекты классов могут существовать отдельно друг от друга, так, например, в случае увольнения объект сотрудника не уничтожается, а продолжает свое дальнейшее существование.

2.2 Композиция

Композиция — специльный тип аггрегации, когда ассоциированный объект не имеет своего жизненного цикла вне объекта владельца. Например, солнечной системе принадлежат ращличные планеты. При уничтожении солнечной системы, планеты будут уничтожены также.

2.3 Асоциация, аггрегация, композиция

Можно сказать, что ассоциация является базовым классом для аггрегации и композиции, а аггрегация — базовым для композиции.

3 Сруктурные паттерны

Структурные шаблоны связаны со способами формирования более крупных структур. Структурные шаблоны классов используют наследование или композицию для создания интерфейсов или реализаций.

3.1 Фасад

3.1.1 Назначение

В книге «банды четырех» назначение шаблона Facade (фасад) определяется следующим образом:

> Предоставление единого интерфейса для набора различных интерфейсов в системе. Шаблон Facade определяет интерфейс более высокого уровня, что упрощает работу с системой.

3.1.2 Мотивация

В основном этот шаблон используется в тех случаях, когда необходим новый способ взаимодействия с системой — более простой в сравнении с уже существующим. Кроме того, он может применяться, когда требуется использовать систему некоторым специфическим образом — например, обращаться к программе трехмерной графики для построения двухмерных изображений. В этом случае нам потребуется специальный метод взаимодействия с системой, поскольку будет использоваться лишь часть ее функциональных возможностей.

Известным примером фасада является JDBC интерфейс в Java, так как его пользователи создают подключение через интерфейс java.sql.Connection, реализается которого их не касается.

3.1.3 Пример

Рассмотрим сложную систему старта автомобиля. В некотором примерном и упрощенном виде, предположим, эта система состоит из подсистем двигатель, топливный инжектор, контроллер температуры, радитор, топливный насос, датчики температуры.

Старт автомобиля потребует знаний каждой подсистемы и поочердный вызов их методов, причем в необходимом порядке. Для пользователя системы, которому нужно лишь стартовать автомобиль такие знания являются избыточными и усложняют использование системы. Мы можем реализовать класс фасад, который будет предоставлять пользователю простой интерфейс и инкапсулировать в себя сложную логику запуска системы автомобиля.

3.1.4 Использование

- Когда нет необходимости использовать все функциональные возможности сложной системы и можно создать новый класс, который будет содержать все необходимые средства доступа к базовой системе. Если предполагается работа лишь с ограниченным набором функций исходной системы, как это

обычно и бывает, интерфейс (API), описанный в новом классе, будет намного проще, чем стандартный интерфейс, разработанный создателями основной системы.

- Используйте фасад для отделения подсистем от клиентов, способствуя тем самым снижению зависимостей и повышению переносимости
- Если подсистемы системы между собой зависимы, то, используя фасад, можно упростить зависимости между подсистемами.

3.2 Адаптер

3.2.1 Назначение

В книге «банды четырех» назначение шаблона определяется следующим образом:

Преобразование интерфейса класса в интерфейс, нужный пользователю.

3.2.2 Мотивация

Пусть есть некоторая система, и планируется расширить ее функциональность путем добавления некоторого нового класса. Причем новый класс не может быть использован, так как его интерфейс не совместим с интерфейсами уже использующихся классов. В таком случае применяется класс-адаптер, который преобразует интерфейс желаемого класса к виду уже существующх интерфейсов классов.

3.2.3 Пример

Предположим, что есть некоторое устройство-хост, которое занимается сбором показаний температуры с различных датчиков разных фирм. Датчики подключены к хосту по COM порту. В системе есть базовый класс датчик температуры, который имеет публичный метод «считать данные по ком-порту».

Заказчик приобрел очень точный и надежный датчик, который работает по USB порту. Необходимо считывать показания с USB-датчика. С новым датчиком распространяется класс, с помощью объекта которого можно запросить данные с датчика. Однако данный класс реализует интерфейс, который отличается от интерфейса базового класса системы.

Если изменить базовый класс путем добавления в него еще одного метода для считывания данных по USB порту для решения задачи, то тогда придется менять реализации дочернх классов. Такое решение является трудозатратным или его вообще невозможно реализовать, если нет доступа к коду класса какого-то из датчиков.

В такой ситуации можно применить адаптер, который преобразует интерфейс нового класса в необходимый. Эта задача имеет проекцию на реальный мир, когда используются физические адаптеры, решающие ту же задачу.

Адаптер можно реализовать двумя способами:

1. путем создания экземпляра адаптируемого класса;

2. путем наследования интерфейса адаптируемого класса.

3.2.4 Использование

- Вы хотите использовать существующий класс, и его интерфейс не соответствует тому, который вам нужен

- Вы хотите создать повторно используемый класс, который взаимодействует с несвязанными или непредвиденными классами, то есть классами, которые не обязательно имеют совместимые интерфейсы.

- Вам нужно использовать несколько существующих подклассов, не непрактично адаптировать их интерфейсы путем наследования каждого. Адаптер может адаптировать интерфейс их родителя.

3.3 Мост

3.3.1 Назначение

Отделить абстракцию от реализации, что позволит независмо вносить изменения в асбтракци и в реализацию.

3.3.2 Мотивация

Понятно каждое слово в этом предложении, но непонятен смысл. Ясно следующее:

- отделение означает обеспечение независимого поведения элементов, или, по крайней мере, явное указание на то, что между ними существуют некоторые отношения;

- абстракция — это концептуальное представление о том, как различные элементы связаны друг с другом.

также понятно, что реализация — это конкретный способ представления абстракции, поэтому смущает предложение отделить абстракцию от конкретного способа ее реализации. Замешательство связано с недопониманием того, что здесь в действительности представляет собой реализация. В данном случае под реализацией понимались те объекты, которые абстрактный и производные от него классы использовали для реализации своих функций (а не те классы, которые называются конкретными и являются производными от абстрактного класса).

Когда абстракция может иметь одну из нескольких возможных реализаций, стандартным способом их размещения является использование наследования. Абстрактный класс определяет интерфейс к абстракции, а конкретные подклассы реализуют его по-разному. Но этот подход не всегда достаточно гибок.

Наследование связывает постоянная реализация абстракции, что затрудняет самостоятельное изменение, расширение и повторное использование абстракций и реализаций.

Чтобы лучше понять идею построения шаблона Bridge и принципы его работы, рас смотрим конкретный пример поэтапно. Сначала обсудим установленные требования, а затем проанализируем вывод основной идеи шаблона и способы его применения. Возможно, данный пример может показаться слишком простым. Однако присмот ритесь к обсуждаемым в нем концепциям, а затем попытайтесь вспомнить аналогич ные ситуации, с которыми нам приходилось сталкиваться ранее. Обратите особое внимание на следующее.

- Наличие вариаций в абстрактном представлении концепций.
- Наличие вариаций в том, как эти концепции реализуются.

Bridge — один из самых трудных для понимания шаблонов. До некоторой степени это вызвано тем, что он является очень мощным и часто применяется в самых раз личных ситуациях. Кроме того, он противоречит распространенной практике при менения механизма наследования для реализации специальных случаев. Тем не ме нее, этот шаблон служит превосходным примером следования двум основным лозун гам технологии шаблонов проектирования: «Найди то, что изменяется, и инкапсулируй это» и «Компоновка объектов в структуру предпочтительней обращения к наследованию классов» (в дальнейшем мы убедимся в этом).

3.3.3 Пример

Предположим, что нам нужно написать программу, которая будет выводить изо бражения прямоугольников с помощью одной из двух имеющихся графических про грамм. Кроме того, допустим, что указания о выборе первой (Drawing Program 1 — DP1) или второй (DP2) графической программы будут предоставляться непосредственно при инициализации прямоугольника. Прямоугольники определяются двумя парами точек, как это показано на рис. Различия между графическими программами описаны в табл.

Заказчик поставил условие, что объект коллекции (клиент, использующий программу отображения прямоугольников) не должен иметь никакого отношения к тому, какая именно графическая программа будет использоваться в каждом случае. Исходя из этого я пришел к заключению, что, поскольку при инициализации прямоугольника указывается, какая именно программа должна использоваться, можно создать два различных типа объекта прямоугольника. Один из них будет вызывать для отображения программу DP1, а другой — программу DP2. В каждом типе объекта будет присутствовать метод отображения прямоугольника, но реализованы они будут по-разному — как показано на рис.

Необходимо включить в программу поддержку еще одного вида геометрических фигур — окружностей. Однако при этом уточняется, что объект коллекции (клиент) не должен ничего знать о различиях, существующих между объектами, представляющими прямоугольники (Rectangle) и окружности (Circle).

Логично будет сделать вывод, что можно применить выбранный ранее подход и просто добавить еще один уровень в иерархию классов программы. Потребуется только добавить в проект новый абстрактный

класс (назовем его Shape (фигура)), а классы Rectangle и Circle будут производными от него. В этом случае объект Client сможет просто обращаться к объекту класса Shape, совершенно не заботясь о том, какая именно геометрическая фигура им представлена.

Проблемы подхода:

-
- Классы в этом ряду представляют те четыре конкретных подтипа класса Shape , с которыми мы имеем дело.
- Что произойдет, если появится третий тип графической программы, т.е. еще один возможный вариант реализации графических функций? В этом случае потребуется создать шесть различных подтипов класса Shape (для представления двух видов фигур и трех графических программ).
- Теперь предположим, что требуется реализовать поддержку нового, третьего, типа фигур. В этом случае нам понадобятся уже девять различных подтипов класса Shape (для представления трех видов фигур и трех графических программ).

Быстрый рост количества требуемых производных классов вызван тем, что в dalном решении абстракция (виды фигур) и реализация (графические программы) жестко связаны между собой. Каждый класс, представляющий конкретный тип фигуры, должен точно знать, какую из существующих графических программ он использует.

Для решения данной проблемы необходимо отделить изменения в абстракции от изменений в реализации таким образом, чтобы количество требуемых классов возрастало линейно — как показано на рис

3.3.4 Пример

Пусть необходимо реализовать систему, которая имеет возможность работать с различными базами данных. На UML диграмме видно, что решение через наследование заставляет разработчика релизововывать большое число наследников. Это связано с тем, что одновременно мы пытаемся решить две задачи: предоставить интерфейс и реализацию, то есть смешиваем две функциональности: "что сделать"и "как сделать". Выделим "что сделать"в интерфейс, "как сделать— в реализацию. Интерфейс аггрегирует реализацию.

По определению всякий шаблон характеризуется повторяемостью — чтобы считаться шаблоном, некоторое решение должно быть применено, по крайней мере, в трех независимых случаях. Под словом "вывести"здесь подразумевается, что в процессе проектирования самостоятельно будет найдено решение, соответствующее идее шаблона, как если бы до этого момента мы не имели о нем никакого представления. Именно этот подход позволит нам выявить ключевые принципы и полезные стратегии использования данного шаблона.

Наша задача состоит в том, чтобы определить, где возможны изменения (анализ общности), а затем установить, как это изменение происходит (анализ изменчивости).

Анализ общности заключается в поиске общих элементов, что поможет понять, чем члены семейства похожи друг на друга. Таким образом, это процесс поиска общих черт во всех элементах, составляющих некоторое семейство (и, следовательно, их различий). Анализ изменчивости позволяет установить, каким образом члены семейства изменяются. Изменчивость имеет смысл только в пределах данной общности. В архитектурном смысле анализ общности дает архитектуре ее долговечность, а анализ изменчивости способствует достижению удобства в использовании.

Другими словами, если изменчивость — это особые случаи в рамках заданной предметной области, то общность устанавливает в ней концепции, объединяющие эти особые случаи между собой. Общие концепции будут представлены в системе абстрактными классами. Вариации, обнаруженные при анализе изменчивости, реализуются посредством создания конкретных классов, производных от этих абстрактных классов.

В объектно-ориентированном проектировании стала уже почти аксиомой практика, когда разработчик, анализируя описание проблемной области, выделяет в нем существительные и создает объекты, представляющие их. Затем он отыскивает глаголы, связанные с этими существительным (т.е. их действия), и реализует их, добавляя к объектам необходимые методы. Подобный процесс проявления повышенного внимания к существительным и глаголам в большинстве случаев приводит к созданию слишком громоздкой иерархии классов.

В практике проектирования для работы с изменяющимися элементами применяются две основные стратегии:

- Найти то, что изменяется, и инкапсулировать это.
- Преимущественно использовать композицию вместо наследования.

Ранее для координации изменяющихся элементов разработчики часто создавали обширные схемы наследования классов. Однако вторая из приведенных выше стратегий рекомендует везде, где только возможно, заменять наследование композицией. Идея состоит в том, чтобы инкапсулировать изменения в независимых классах, что позволит при обработке будущих изменений обойтись без модификации программного кода. Одним из способов достижения подобной цели является помещение каждого подверженного изменениям элемента в собственный абстрактный класс с последующим анализом, как эти абстрактные классы соотносятся друг с другом.

Рассмотрим данный процесс на примере задачи с вычерчиванием прямоугольника. Сначала идентифицируем то, что изменяется. В нашем случае это различные типы фигур (представленных абстрактным классом Shape) и различные версии графических программ. Следовательно, общими концепциями являются понятия типа фигуры и графической программы.

В данном случае предполагается, что абстрактный класс Shape инкапсулирует концепцию типов фигур, с которыми необходимо работать. Каждый тип фигуры должен знать, как себя нарисовать. В свою очередь, абстрактный класс Drawing (рисование) отвечает за вычерчивание линий и окружностей. На ри-

сунке указанные выше обязательства представлены посредством определения соответствующих методов в каждом из классов.

Имея перед собой два набора классов, очевидно, следует задаться вопросом, как они будут взаимодействовать друг с другом. На этот раз попытаемся обойтись без того, чтобы добавлять в систему еще один новый набор классов, построенный на углублении иерархии наследования, поскольку последствия этого нам уже известны (см. рис. 9.3 и 9.7). На этот раз мы подойдем с другой стороны и попробуем определить, как эти классы могут использовать друг друга (в полном соответствии с приведенным выше утверждением о предпочтительности композиции над наследованием). Главный вопрос здесь состоит в том, какой же из абстрактных классов будет использовать другой?

Рассмотрим две возможности: либо класс Shape использует классы графических программ, либо класс Drawing использует классы фигур. Начнем со второго варианта. Чтобы графические программы могли рисовать различные фигуры непосредственно, они должны знать некоторую общую информацию о фигурах: что они собой представляют и как выглядят. Однако это требование нарушает фундаментальный принцип объектной технологии: каждый объект должен нести ответственность только за себя.

Это требование также нарушает инкапсуляцию. Объекты класса Drawing должны были бы знать определенную информацию об объектах класса Shape , чтобы иметь возможность отобразить их (а именно — тип конкретной фигуры). В результате объекты класса Drawing фактически оказываются ответственными не только за свое собственное поведение.

Вернемся к первому варианту. Что если объекты класса Shape для отображения себя будут использовать объекты класса Drawing ? Объектам класса Shape не нужно знать, какой именно тип объекта класса Drawing будет использоваться, поэтому классу Shape можно разрешить ссылаться на класс Drawing . Дополнительно класс Shape в этом случае можно сделать ответственным за управление рисованием.

3.3.5 Использование

- избежание постоянной связи между абстракцией и ее реализацией. Например, когда реализация должна быть выбрана или переключена во время выполнения;
- и абстракции, и их реализации должны быть расширяемыми с помощью подклассов. В этом случае паттерн Bridge позволяет комбинировать различные абстракции и реализации и независимо расширять их;
- изменения в реализации абстракции не должны влиять на клиентов; то есть их код не должен быть перекомпилирован.

3.4 Декоратор (обертка)

3.4.1 Назначение

Динамическое расширение функциональности. Декораторы предоставляют гибкую альтернативу подклассам для расширения функциональности.

3.4.2 Мотивация

Добавление функциональных возможностей отдельному объекту, а не всему классу.

3.4.3 Пример

Пусть у нас есть плата, которая позволяет проводить замеры различных физических величин. Периодически к телеметрическому сервису поступет запрос на чтение того или иного датчика. Мы хотим "на лету"считывать показатели в зависимости от запроса. Интерфейс платы имеет метод read, который позволяет считывать данные с некоторого датчика. Создадим несколько классов, каждый из которых соответствует определенной физической величине и реализует интерфейс. Декоратор TelemetryService также реализует интерфейс платы и включает в себя конкретную реализацию датчика, через которую может рпоизводить считывание конкретной величины.

3.4.4 Использование

- динамически и прозрачно добавлять обязанности к отдельным объектам, не затрагивая другие объекты;
- для обязанностей, которые могут быть сняты. То есть можно проводить не только расширение, но и сужение;
- когда расширение подклассами нецелесообразно. Иногда возможно большое количество независимых расширений, что может привести к взрыву подклассов для поддержки каждой комбинации. Или определение класса может быть скрыто или иным образом недоступно для подклассов. .

3.5 Заместитель (proxy)

3.5.1 Назначение

Прокси предоставляет заместителя для другого объекта, чтобы контролировать доступ к нему, не изменяя при этом поведение клиента.

3.5.2 Мотивация

Существуют ситуации, в которых клиент не может или не может ссылаться на объект напрямую, но все же хочет взаимодействовать с объектом. Предоставить суррогат или заполнитель для другого объекта контролировать доступ к нему.

3.5.3 Пример

Представим редактор документов, который может встраивать графические объекты в документ. Некоторые графические объекты, такие как большие растровые изображения, могут быть дорогими в создании. Но открытие документа должно быть быстрым, поэтому мы должны избегать одновременного создания всех дорогих объектов при открытии документа. В любом случае это не обязательно, потому что не все эти объекты будут видны в документе одновременно.

Эти ограничения предполагают создание каждого дорогого объекта по запросу, что в этом случае происходит, когда изображение становится видимым. Но что мы помещаем в документ вместо изображения? И как мы можем скрыть тот факт, что изображение создается по требованию, чтобы не усложнять реализацию редактора? Например, эта оптимизация не должна влиять на код рендеринга и форматирования. Решение состоит в том, чтобы использовать другой объект, прокси изображения, который выступает в качестве замены для реального изображения. Прокси действует так же, как изображение и заботится о его создании, когда это необходимо.

3.5.4 Использование

- динамически и прозрачно добавлять обязанности к отдельным объектам, не затрагивая другие объекты;
- для обязанностей, которые могут быть сняты. То есть можно проводить не только расширение, но и сужение;
- когда расширение подклассами нецелесообразно. Иногда возможно большое количество независимых расширений, что может привести к взрыву подклассов для поддержки каждой комбинации. Или определение класса может быть скрыто или иным образом недоступно для подклассов. .