них зависит, насколько удачной или неудачной окажется структура во время выполнения.

Многие паттерны проектирования (особенно уровня объектов) явно подчеркивают различие между структурами на этапах компиляции и выполнения. Паттерны компоновщик и декоратор полезны для построения сложных структур времени выполнения. Наблюдатель порождает структуры времени выполнения, которые часто трудно понять, не зная паттерна. Паттерн цепочка обязанностей также приводит к таким схемам взаимодействия, в которых наследование неочевидно. В общем можно утверждать, что разобраться в структурах времени выполнения невозможно, если не понимаешь специфики паттернов.

Проектирование с учетом будущих изменений

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.

Благодаря паттернам систему всегда можно модифицировать определенным образом. Каждый паттерн позволяет изменять некоторый аспект системы независимо от всех прочих, таким образом, она менее подвержена влиянию изменений конкретного вида.

Вот некоторые типичные причины перепроектирования, а также паттерны, которые позволяют этого избежать:

- *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.
  Паттерны проектирования: абстрактная фабрика, фабричный метод, прототип;
- *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.
  Паттерны проектирования: цепочка обязанностей, команда;
- *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.

Паттерны проектирования: абстрактная фабрика, мост;

- *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.

  Паттерны проектирования: абстрактная фабрика, мост, хранитель, заместитель;

- *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.

  Паттерны проектирования: мост, итератор, стратегия, шаблонный метод, посетитель;

- *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. Для поддержки слабо связанных систем в паттернах проектирования применяются такие методы, как абстрактные связи и разбиение на слои.

  Паттерны проектирования: абстрактная фабрика, мост, цепочка обязанностей, команда, фасад, посредник, наблюдатель;

- *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.

  Композиция объектов и делегирование – гибкие альтернативы наследованию для комбинирования поведений. Приложению можно добавить новую функциональность, меняя способ композиции объектов, а не определяя новые подклассы уже имеющихся классов. С другой стороны, при интенсивном использовании композиции объектов проект может оказаться трудным для понимания. С помощью многих паттернов проектирования удается построить такое

решение, где специализация достигается за счет определения одного подкласса и комбинирования его экземпляров с уже существующими.
Паттерны проектирования: мост, цепочка обязанностей, компоновщик, декоратор, наблюдатель, стратегия;

- *неудобства при изменении классов.* Иногда нужно модифицировать класс, но делать это неудобно. Допустим, вам нужен исходный код, а его нет (так обстоит дело с коммерческими библиотеками классов). Или любое изменение тянет за собой модификации множества существующих подклассов. Благодаря паттернам проектирования можно модифицировать классы и при таких условиях.
Паттерны проектирования: адаптер, декоратор, посетитель.

Приведенные примеры демонстрируют ту гибкость работы, которой можно добиться, используя паттерны при проектировании приложений. Насколько эта гибкость необходима, зависит, конечно, от особенностей вашей программы. Давайте посмотрим, какую роль играют паттерны при разработке прикладных программ, инструментальных библиотек и каркасов приложений.

Прикладные программы

Если вы проектируете прикладную программу, например редактор документов или электронную таблицу, то наивысший приоритет имеют *внутреннее* повторное использование, удобство сопровождения и расширяемость. Первое подразумевает, что вы не проектируете и не реализуете больше, чем необходимо. Повысить степень внутреннего повторного использования помогут паттерны, уменьшающие число зависимостей. Ослабление связанности увеличивает вероятность того, что некоторый класс объектов сможет совместно работать с другими. Например, устраняя зависимости от конкретных операций путем изолирования и инкапсуляции каждой операции, вы упрощаете задачу повторного использования любой операции в другом контексте. К тому же результату приводит устранение зависимостей от алгоритма и представления.

Паттерны проектирования также позволяют упростить сопровождение приложения, если использовать их для ограничения платформенных зависимостей и разбиения системы на отдельные слои. Они способствуют и наращиванию функций системы, показывая, как расширять иерархии классов и когда применять композицию объектов. Уменьшение степени связанности также увеличивает возможность развития системы. Расширение класса становится проще, если он не зависит от множества других.

Инструментальные библиотеки

Часто приложение включает классы из одной или нескольких библиотек предопределенных классов. Такие библиотеки называются *инструментальными.* Инструментальная библиотека – это набор взаимосвязанных, повторно используемых классов, спроектированный с целью предоставления полезных функций общего назначения. Примеры инструментальной библиотеки – набор контейнерных классов для списков, ассоциативных массивов, стеков и т.д., библиотека потокового ввода/вывода в C++. Инструментальные библиотеки не определяют

какой-то конкретный дизайн приложения, а просто предоставляют средства, благодаря которым в приложениях проще решать поставленные задачи, позволяют разработчику не изобретать заново повсеместно применяемые функции. Таким образом, в инструментальных библиотеках упор сделан на повторном использовании кода. Это объектно-ориентированные эквиваленты библиотек подпрограмм.

Можно утверждать, что проектирование инструментальной библиотеки сложнее, чем проектирование приложения, поскольку библиотеки должны использоваться во многих приложениях, иначе они бесполезны. К тому же автор библиотеки не знает заранее, какие специфические требования будут предъявляться конкретными приложениями. Поэтому ему необходимо избегать любых предположений и зависимостей, способных ограничить гибкость библиотеки, следовательно, сферу ее применения и эффективность.

Каркасы приложений

Каркас – это набор взаимодействующих классов, составляющих повторно используемый дизайн для конкретного класса программ [Deu89, JF88]. Например, можно создать каркас для разработки графических редакторов в разных областях: рисовании, сочинении музыки или САПР [VL90, Joh92]. Другим каркасом рекомендуется пользоваться при создании компиляторов для разных языков программирования и целевых машин [JML92]. Третий упростит построение приложений для финансового моделирования [BE93]. Каркас можно подстроить под конкретное приложение путем порождения специализированных подклассов от входящих в него абстрактных классов.

Каркас диктует определенную архитектуру приложения. Он определяет общую структуру, ее разделение на классы и объекты, основные функции тех и других, методы взаимодействия объектов и классов и потоки управления. Данные параметры проектирования задаются каркасом, а прикладные проектировщики или разработчики могут сконцентрироваться на специфике приложения. В каркасе аккумулированы проектные решения, общие для данной предметной области. Акцент в каркасе делается на повторном использовании дизайна, а не кода, хотя обычно он включает и конкретные подклассы, которые можно применять непосредственно.

Повторное использование на данном уровне меняет направление связей между приложением и программным обеспечением, лежащим в его основе, на противоположное. При использовании инструментальной библиотеки (или, если хотите, обычной библиотеки подпрограмм) вы пишете тело приложения и вызываете из него код, который планируете использовать повторно. При работе с каркасом вы, наоборот, повторно используете тело и пишете код, который оно вызывает. Вам приходится кодировать операции с предопределенными именами и параметрами вызова, но зато число принимаемых вами проектных решений сокращается.

В результате приложение создается быстрее. Более того, все приложения имеют схожую структуру. Их проще сопровождать, и пользователям они представляются более знакомыми. С другой стороны, вы в какой-то мере жертвуете свободой творчества, поскольку многие проектные решения уже приняты за вас.

Если проектировать приложения нелегко, инструментальные библиотеки – еще сложнее, то проектирование каркасов – задача самая трудная. Проектировщик каркаса рассчитывает, что единая архитектура будет пригодна для всех приложений в данной предметной области. Любое независимое изменение дизайна каркаса приведет к утрате его преимуществ, поскольку основной «вклад» каркаса в приложение – это определяемая им архитектура. Поэтому каркас должен быть максимально гибким и расширяемым.

Поскольку приложения так сильно зависят от каркаса, они особенно чувствительны к изменениям его интерфейсов. По мере усложнения каркаса приложения должны эволюционировать вместе с ним. В результате существенно возрастает значение слабой связанности, в противном случае малейшее изменение каркаса приведет к целой волне модификаций.

Рассмотренные выше проблемы проектирования актуальны именно для каркасов. Каркас, в котором они решены путем применения паттернов, может лучше обеспечить высокий уровень проектирования и повторного использования кода, чем тот, где паттерны не применялись. В отработанных каркасах обычно можно обнаружить несколько разных паттернов проектирования. Паттерны помогают адаптировать архитектуру каркаса ко многим приложениям без повторного проектирования.

Дополнительное преимущество появляется потому, что вместе с каркасом документируются те паттерны, которые в нем использованы [BJ94]. Тот, кто знает паттерны, способен быстрее разобраться в тонкостях каркаса. Но даже не работающие с паттернами увидят их преимущества, поскольку паттерны помогают удобно структурировать документацию по каркасу. Повышение качества документирования важно для всех типов программного обеспечения, но для каркасов этот аспект важен вдвойне. Для освоения работы с каркасами надо потратить немало усилий, и только после этого они начнут приносить реальную пользу. Паттерны могут существенно упростить задачу, явно выделив ключевые элементы дизайна каркаса.

Поскольку между паттернами и каркасами много общего, часто возникает вопрос, в чем же различия между ними и есть ли они вообще. Так вот, существует три основных различия:

- *паттерны проектирования 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, описанные в данной книге, необходимо реализовывать всякий раз, когда в них возникает необходимость. Паттерны объясняют намерения проектировщика, компромиссы и последствия выбранного дизайна;
- *как архитектурные элементы, паттерны проектирования мельче, чем каркасы.* Типичный каркас содержит несколько паттернов. Обратное утверждение неверно;

- *паттерны проектирования менее специализированы, чем каркасы.* Каркас всегда создается для конкретной предметной области. В принципе каркас графического редактора можно использовать для моделирования работы фабрики, но его никогда не спутаешь с каркасом, предназначенным специально для моделирования. Напротив, включенные в наш каталог 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.

Значение каркасов возрастает. Именно с их помощью объектно-ориентированные системы можно использовать повторно в максимальной степени. Крупные объектно-ориентированные приложения составляются из слоев взаимодействующих друг с другом каркасов. Дизайн и код приложения в значительной мере определяются теми каркасами, которые применялись при его создании.

1.7. Как выбирать паттерн проектирования

Если в распоряжение проектировщика предоставлен каталог из более чем 20 паттернов, трудно решать, какой паттерн лучше всего подходит для решения конкретной задачи проектирования. Ниже представлены разные подходы к выбору подходящего паттерна:

- *подумайте, как паттерны решают проблемы проектирования.* В разделе 1.6 обсуждается то, как с помощью паттернов можно найти подходящие объекты, определить нужную степень их детализации, специфицировать их интерфейсы. Здесь же говорится и о некоторых иных подходах к решению задач с помощью паттернов;
- *пролистайте разделы каталога, описывающие назначение паттернов.* В разделе 1.4 перечислены назначения всех представленных паттернов. Ознакомьтесь с целью каждого паттерна, когда будете искать тот, что в наибольшей степени относится к вашей проблеме. Чтобы сузить поиск, воспользуйтесь схемой в таблице 1.1;
- *изучите взаимосвязи паттернов.* На рис. 1.1 графически изображены соотношения между различными паттернами проектирования. Данная информация поможет вам найти нужный паттерн или группы паттернов;
- *проанализируйте паттерны со сходными целями.* Каталог состоит из трех частей: порождающие паттерны, структурные паттерны и паттерны поведения. Каждая часть начинается со вступительных замечаний о паттернах соответствующего вида и заканчивается разделом, где они сравниваются друг с другом;
- *разберитесь в причинах, вызывающих перепроектирование.* Взгляните на перечень причин, приведенный выше. Быть может, в нем упомянута ваша проблема? Затем обратитесь к изучению паттернов, помогающих устранить эту причину;