Программирование в соответствии с интерфейсом, а не с реализацией

Наследование классов – это не что иное, как механизм расширения функциональности приложения путем повторного использования функциональности родительских классов. Оно позволяет быстро определить новый вид объектов в терминах уже имеющегося. Новую реализацию вы можете получить посредством наследования большей части необходимого кода из ранее написанных классов.

Однако не менее важно, что наследование позволяет определять семейства объектов с *идентичными* интерфейсами (обычно за счет наследования от абстрактных классов). Почему? Потому что от этого зависит полиморфизм.

Если пользоваться наследованием осторожно (некоторые сказали бы *правильно*), то все классы, производные от некоторого абстрактного класса, будут обладать его интерфейсом. Отсюда следует, что подкласс добавляет новые или замещает старые операции и не скрывает операций, определенных в родительском классе. *Все* подклассы могут отвечать на запросы, соответствующие интерфейсу абстрактного класса, поэтому они являются подтипами этого абстрактного класса.

У манипулирования объектами строго через интерфейс абстрактного класса есть два преимущества:

- клиенту не нужно иметь информации о конкретных типах объектов, которыми он пользуется, при условии, что все они имеют ожидаемый клиентом интерфейс;
- клиенту необязательно «знать» о классах, с помощью которых реализованы объекты. Клиенту известно только об абстрактном классе (или классах), определяющих интерфейс.

Данные преимущества настолько существенно уменьшают число зависимостей между подсистемами, что можно даже сформулировать принцип объектно-ориентированного проектирования для повторного использования: *программируйте в соответствии с интерфейсом, а не с реализацией.*

Не объявляйте переменные как экземпляры конкретных классов. Вместо этого придерживайтесь интерфейса, определенного абстрактным классом. Это одна из наших ключевых идей.

Конечно, где-то в системе вам придется инстанцировать конкретные классы, то есть определить конкретную реализацию. Как раз это и позволяют сделать порождающие паттерны: абстрактная фабрика, строитель, фабричный метод, прототип и одиночка. Абстрагируя процесс создания объекта, эти паттерны предоставляют вам разные способы прозрачно ассоциировать интерфейс с его реализацией в момент инстанцирования. Использование порождающих паттернов гарантирует, что система написана в терминах интерфейсов, а не реализации.

Механизмы повторного использования

Большинству проектировщиков известны концепции объектов, интерфейсов, классов и наследования. Трудность в том, чтобы применить эти знания для построения гибких, повторно используемых программ. С помощью паттернов проектирования вы сможете сделать это проще.

Наследование и композиция

Два наиболее распространенных приема повторного использования функциональности в объектно-ориентированных системах – это наследование класса и *композиция объектов*. Как мы уже объясняли, наследование класса позволяет определить реализацию одного класса в терминах другого. Повторное использование за счет порождения подкласса называют еще *прозрачным ящиком* (white-box reuse). Такой термин подчеркивает, что внутреннее устройство родительских классов видимо подклассам.

Композиция объектов – это альтернатива наследованию класса. В этом случае новую, более сложную функциональность мы получаем путем объединения или композиции объектов. Для композиции требуется, чтобы объединяемые объекты имели четко определенные интерфейсы. Такой способ повторного использования называют *черным ящиком* (black-box reuse), поскольку детали внутреннего устройства объектов остаются скрытыми.

И у наследования, и у композиции есть достоинства и недостатки. Наследование класса определяется статически на этапе компиляции, его проще использовать, поскольку оно напрямую поддержано языком программирования. В случае наследования классов упрощается также задача модификации существующей реализации. Если подкласс замещает лишь некоторые операции, то могут оказаться затронутыми и остальные унаследованные операции, поскольку не исключено, что они вызывают замещенные.

Но у наследования класса есть и минусы. Во-первых, нельзя изменить унаследованную от родителя реализацию во время выполнения программы, поскольку само наследование фиксировано на этапе компиляции. Во-вторых, родительский класс нередко хотя бы частично определяет физическое представление своих подклассов. Поскольку подклассу доступны детали реализации родительского класса, то часто говорят, что *наследование нарушает инкапсуляцию* [Sny86]. Реализации подкласса и родительского класса настолько тесно связаны, что любые изменения последней требуют изменять и реализацию подкласса.

Зависимость от реализации может повлечь за собой проблемы при попытке повторного использования подкласса. Если хотя бы один аспект унаследованной реализации непригоден для новой предметной области, то приходится переписывать родительский класс или заменять его чем-то более подходящим. Такая зависимость ограничивает гибкость и возможности повторного использования. С проблемой можно справиться, если наследовать только абстрактным классам, поскольку в них обычно совсем нет реализации или она минимальна.

Композиция объектов определяется динамически во время выполнения за счет того, что объекты получают ссылки на другие объекты. Композицию можно применить, если объекты соблюдают интерфейсы друг друга. Для этого, в свою очередь, требуется тщательно проектировать интерфейсы, так чтобы один объект можно было использовать вместе с широким спектром других. Но и выигрыш велик. Поскольку доступ к объектам осуществляется только через их интерфейсы, мы не нарушаем инкапсуляцию. Во время выполнения программы любой объект можно заменить другим, лишь бы он имел тот же тип. Более того, поскольку при

реализации объекта кодируются прежде всего его интерфейсы, то зависимость от реализации резко снижается.

Композиция объектов влияет на дизайн системы и еще в одном аспекте. Отдавая предпочтение композиции объектов, а не наследованию классов, вы инкапсулируете каждый класс и даете ему возможность выполнять только свою задачу. Классы и их иерархии остаются небольшими, и вероятность их разрастания до неуправляемых размеров невелика. С другой стороны, дизайн, основанный на композиции, будет содержать больше объектов (хотя число классов, возможно, уменьшится), и поведение системы начнет зависеть от их взаимодействия, тогда как при другом подходе оно было бы определено в одном классе.

Это подводит нас ко второму правилу объектно-ориентированного проектирования: *предпочитайте композицию наследованию класса.*

В идеале, чтобы добиться повторного использования, вообще не следовало бы создавать новые компоненты. Хорошо бы, чтобы можно было получить всю нужную функциональность, просто собирая вместе уже существующие компоненты. На практике, однако, так получается редко, поскольку набор имеющихся компонентов все же недостаточно широк. Повторное использование за счет наследования упрощает создание новых компонентов, которые можно было бы применять со старыми. Поэтому наследование и композиция часто используются вместе.

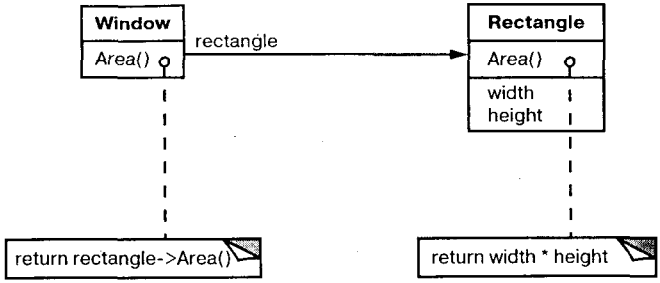
Тем не менее наш опыт показывает, что проектировщики злоупотребляют наследованием. Нередко дизайн мог бы стать лучше и проще, если бы автор больше полагался на композицию объектов.

Делегирование

С помощью *делегирования* композицию можно сделать столь же мощным инструментом повторного использования, сколь и наследование [Lie86, JZ91]. При делегировании в процесс обработки запроса вовлечено *два* объекта: получатель поручает выполнение операций другому объекту – *уполномоченному*. Примерно так же подкласс делегирует ответственность своему родительскому классу. Но унаследованная операция всегда может обратиться к объекту-получателю через переменную-член (в C++) или переменную `self` (в Smalltalk). Чтобы достичь того же эффекта для делегирования, получатель передает указатель на самого себя соответствующему объекту, дабы при выполнении делегированной операции последний мог обратиться к непосредственному адресату запроса.

Например, вместо того чтобы делать класс `Window` (окно) подклассом класса `Rectangle` (прямоугольник) – ведь окно является прямоугольником, – мы можем воспользоваться внутри `Window` поведением класса `Rectangle`, поместив в класс `Window` переменную экземпляра типа `Rectangle` и делегируя ей операции, специфичные для прямоугольников. Другими словами, окно не *является* прямоугольником, а *содержит* его. Теперь класс `Window` может явно перенаправлять запросы своему члену `Rectangle`, а не наследовать его операции.

На диаграмме ниже изображен класс `Window`, который делегирует операцию `Area()` над своей внутренней областью переменной экземпляра `Rectangle`.



Сплошная линия со стрелкой обозначает, что класс содержит ссылку на экземпляр другого класса. Эта ссылка может иметь необязательное имя, в данном случае прямоугольник.

Главное достоинство делегирования в том, что оно упрощает композицию поведений во время выполнения. При этом способ комбинирования поведений можно изменять. Внутреннюю область окна разрешается сделать круговой во время выполнения, просто подставив вместо экземпляра класса `Rectangle` экземпляр класса `Circle`; предполагается, конечно, что оба эти класса имеют одинаковый тип.

У делегирования есть и недостаток, свойственный и другим подходам, применяемым для повышения гибкости за счет композиции объектов. Заключается он в том, что динамическую, в высокой степени параметризованную программу труднее понять, нежели статическую. Есть, конечно, и некоторая потеря машинной производительности, но неэффективность работы проектировщика гораздо более существенна. Делегирование можно считать хорошим выбором только тогда, когда оно позволяет достичь упрощения, а не усложнения дизайна. Нелегко сформулировать правила, ясно говорящие, когда следует пользоваться делегированием, поскольку эффективность его зависит от контекста и вашего личного опыта. Лучше всего делегирование работает при использовании в составе привычных идиом, то есть в стандартных паттернах.

Делегирование используется в нескольких паттернах проектирования: **состояние**, **стратегия**, **посетитель**. В первом получатель делегирует запрос объекту, представляющему его текущее состояние. В паттерне **стратегия** обработка запроса делегируется объекту, который представляет стратегию его исполнения. У объекта может быть только одно состояние, но много стратегий для исполнения различных запросов. Назначение обоих паттернов – изменить поведение объекта за счет замены объектов, которым делегируются запросы. В паттерне **посетитель** операция, которая должна быть выполнена над каждым элементом составного объекта, всегда делегируется **посетителю**.

В других паттернах делегирование используется не так интенсивно. Паттерн **посредник** вводит объект, осуществляющий посредничество при взаимодействии других объектов. Иногда объект-посредник реализует операции, переадресуя их другим объектам; в других случаях он передает ссылку на самого себя, используя тем самым делегирование как таковое. Паттерн **цепочка обязанностей**

обрабатывает запросы, перенаправляя их от одного объекта другому по цепочке. Иногда вместе с запросом передается ссылка на исходный объект, получивший запрос, и в этом случае мы снова сталкиваемся с делегированием. Паттерн мост отделяет абстракцию от ее реализации. Если между абстракцией и конкретной реализацией имеется существенное сходство, то абстракция может просто делегировать операции своей реализации.

Делегирование показывает, что наследование как механизм повторного использования всегда можно заменить композицией.

Наследование и параметризованные типы

Еще один (хотя и не в точности объектно-ориентированный) метод повторного использования имеющейся функциональности – это применение *параметризованных* типов, известных также как обобщенные типы (Ada, Eiffel) или шаблоны (C++). Данная техника позволяет определить тип, не задавая типы, которые он использует. Неспецифицированные типы передаются в виде параметров в точке использования. Например, класс `List` (список) можно параметризовать типом помещаемых в список элементов. Чтобы объявить список целых чисел, вы передаете тип `integer` в качестве параметра параметризованному типу `List`. Если же надо объявить список строк, то в качестве параметра передается тип `String`. Для каждого типа элементов компилятор языка создаст отдельный вариант шаблона класса `List`.

Параметризованные типы дают в наше распоряжение третий (после наследования класса и композиции объектов) способ комбинировать поведение в объектно-ориентированных системах. Многие задачи можно решить с помощью любого из этих трех методов. Чтобы параметризовать процедуру сортировки операцией сравнения элементов, мы могли бы сделать сравнение:

- операцией, реализуемой подклассами (применение паттерна шаблонный метод);
- функцией объекта, передаваемого процедуре сортировки (стратегия);
- аргументом шаблона в C++ или обобщенного типа в Ada, который задает имя функции, вызываемой для сравнения элементов.

Но между тремя данными подходами есть важные различия. Композиция объектов позволяет изменять поведение во время выполнения, но для этого требуются косвенные вызовы, что снижает эффективность. Наследование разрешает предоставить реализацию по умолчанию, которую можно замещать в подклассах. С помощью параметризованных типов допустимо изменять типы, используемые классом. Но ни наследование, ни параметризованные типы не подлежат модификации во время выполнения. Выбор того или иного подхода зависит от проекта и ограничений на реализацию.

Ни в одном из паттернов, описанных в этой книге, параметризованные типы не используются, хотя изредка мы прибегаем к ним для реализации паттернов в C++. В языке вроде Smalltalk, где нет проверки типов во время компиляции, параметризованные типы не нужны вовсе.