

## **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. Поэтому говорят, что композиция описывает отношение “часть-целое”. Композиция – это еще один способ представить иерархичность реального мира. 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. 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. По 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.

Более общим понятием по сравнению с композицией является агрегация. Отличие в том, что в случае композиции время жизни контейнера и внутреннего объекта одинаково. Композицию поэтому еще называют агрегацией по значению. Композиция не может быть цикличной: один объект – это целое, другой – часть, т.е. физическое вхождение одного в другое нельзя "зациклить", а вот указатели или ссылки, с помощью которых создается агрегация – можно (каждый из двух объектов может содержать указатель или ссылку на другой). Таким образом, композиция используется, когда у некоторого объекта есть неотъемлимые составные части. Например, Самолет имеет крылья, двигатель, хвост и т.д. Иногда композиция соответствует физическому объекту, иногда это более абстрактное понятие (например, классы Клуб или Пользователь).

В случае агрегации уничтожение контейнера не уничтожает его содержимое. Агрегация по ссылке позволяет части быть независимой от целого. Пример: курс и студенты.

**Композиция:** автомобиль имеет двигатель

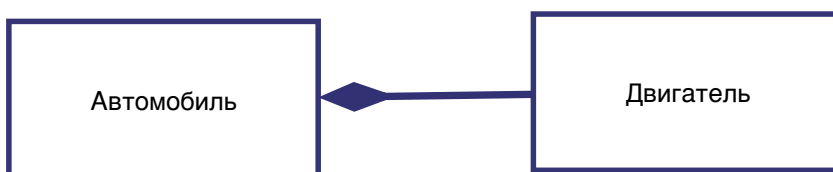


Рис. 10.1

### Агрегация: студент факультета

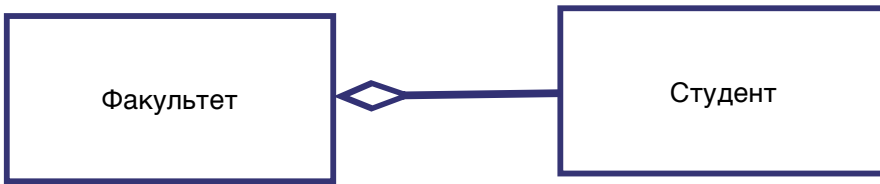
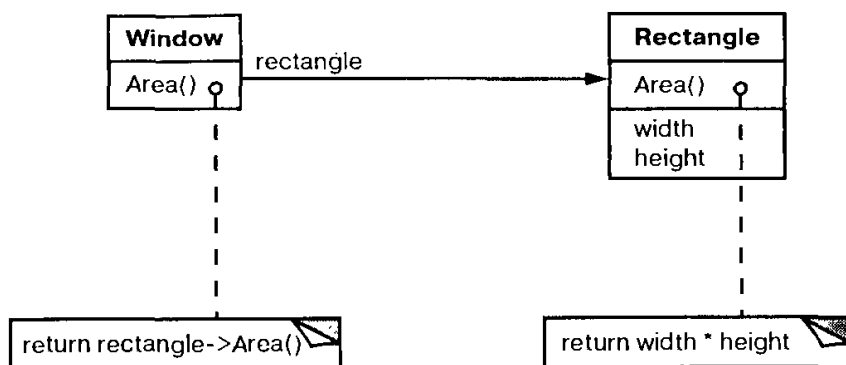


Рис. 10.2

### Делегирование

С помощью делегирования композицию можно сделать столь же мощным инструментом повторного использования, сколь и наследование. 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 `this` (в C#). 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.

### **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 или шаблонные классы. 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 обобщенного типа, 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.

### **Сравнение структур времени выполнения и времени компиляции**

Структура объектно-ориентированной программы на этапе выполнения часто имеет мало общего со структурой ее исходного кода. Последняя фиксируется на этапе

компиляции; код состоит из классов, отношения наследования между которыми неизменны. На этапе же выполнения структура программы – быстро изменяющаяся сеть из взаимодействующих объектов. Две эти структуры почти независимы.

Рассмотрим различие между композицией и осведомленностью (acquaintance) (синоним “использование”) объектов и его проявления на этапах компиляции и выполнения. Композиция подразумевает, что один объект владеет другим или несет за него ответственность.

В общем случае мы говорим, что объект содержит другой объект или является его частью. В этом случае мы говорим о композиции. Композиция означает, что агрегат и его составляющие имеют одинаковое время жизни. Если агрегат и его составляющие могут иметь разное время жизни, то используем более общее понятие – агрегацию.

Говоря же об осведомленности, мы имеем в виду, что объекту известно о другом объекте. Иногда осведомленность называют ассоциацией или отношением «использует». Осведомленные объекты могут запрашивать друг у друга операции, но они не несут никакой ответственности друг за друга. Осведомленность – это более слабое отношение, чем агрегирование; оно предполагает гораздо менее тесную связь между объектами.

На наших диаграммах осведомленность будет обозначаться сплошной линией со стрелкой.

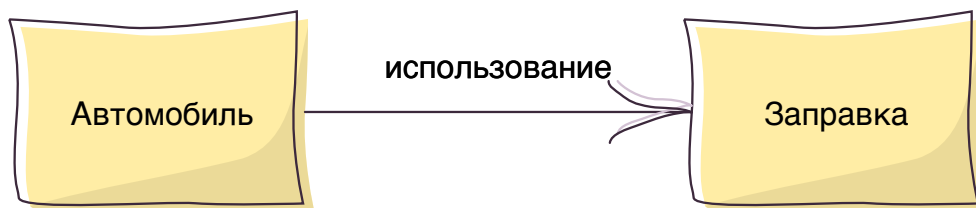


Рис. 10.3

Линия со стрелкой и незакрашенным ромбиком обозначает агрегирование.

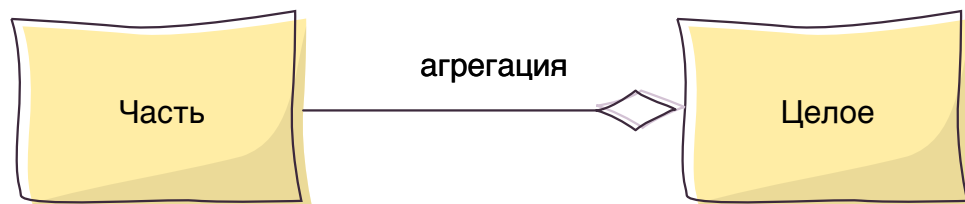


Рис. 10.4

Агрегирование и осведомленность легко спутать, поскольку они часто реализуются одинаково. Различие между осведомленностью и агрегированием определяется, скорее, предполагаемым использованием, а не языковыми механизмами. В структуре, существующей на этапе компиляции, увидеть различие нелегко, но тем не менее оно существенно. Обычно отношений агрегирования меньше, чем отношений осведомленности, и они более постоянны. Напротив, отношения осведомленности возникают и исчезают чаще и иногда делятся лишь во время исполнения некоторой операции. Отношения осведомленности, кроме того, более динамичны, что затрудняет их выявление в исходном тексте программы.

Когда в исходном коде в определении одного класса встречается упоминание другого класса как параметра метода или в некотором методе есть объявление локального объекта этого типа, то мы говорим, что один класс использует другой или осведомлен о его существовании.

Когда скоро несоответствие между структурой программы на этапах компиляции и выполнения столь велико, ясно, что изучение исходного кода может сказать о работе системы совсем немного. Поведение системы во время выполнения должно определяться проектировщиком, а не языком. Соотношения между объектами и их типами нужно проектировать очень аккуратно, поскольку именно от них зависит, насколько удачной или неудачной окажется структура во время выполнения.

Пример.

Рассмотрим сходства и различия между следующими понятиями (которые можно описать классами): цветы, маргаритки, красные розы, желтые розы, лепестки и божьи коровки. Мы можем заметить следующее:

- Маргаритка – цветок.
- Роза – (другой) цветок.
- Красная и желтая розы – розы.
- Лепесток является частью обоих видов цветов.
- Божьи коровки питаются вредителями, поражающими некоторые цветы.

Отношения между понятиями, а значит и между классами, могут означать одно из двух. Во-первых, у них может быть что-то общее. Например, и маргаритки, и розы – это разновидности цветов: и те, и другие имеют ярко окрашенные лепестки, испускают аромат и так далее. Во-вторых, может быть какая-то семантическая (смысловая) связь. Например, красные розы больше похожи на желтые розы, чем на маргаритки. Но между розами и маргаритками больше общего, чем между цветами и лепестками. Также существует симбиотическая связь между цветами и божьими коровками: божьи коровки защищают цветы от вредителей, которые, в свою очередь, служат пищей божьим коровкам.

Еще раз перечислим три основных типа отношений между классами. Во-первых, это отношение "обобщение/специализация" (общее и частное), известное как "is-a" (является разновидностью). Розы – это цветы, что значит: розы являются специализированным частным случаем, разновидностью, подклассом более общего класса "цветы". Во-вторых, это отношение "целое/часть" (композиция), известное как "part of". Так, лепестки являются частью цветов. В-третьих, это семантические, смысловые отношения, ассоциации. Например, божьи коровки ассоциируются с цветами – хотя, казалось бы, что у них общего. Или вот: розы и свечи – и то, и другое можно использовать для украшения стола.

Обычно аналитик констатирует наличие ассоциации и, постепенно уточняя проект, превращает ее в какую-то более специализированную связь.

## **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.

Вот некоторые типичные причины перепроектирования:

- **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.
- **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.
- **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.
- **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. Скрытие этой информации от клиентов поможет уберечься от каскада изменений.
- **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.
- **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 применяются такие методы, как абстрактные связи и разбиение на слои.
- **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.

– **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.