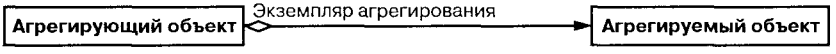
Сравнение структур времени выполнения и времени компиляции

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 *агрегированием* и *осведомленностью* (acquaintance) объектов и его проявления на этапах компиляции и выполнения. Агрегирование 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. Агрегирование означает, что агрегат и его составляющие имеют одинаковое время жизни.

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. Линия со стрелкой и ромбиком вначале обозначает агрегирование.



Агрегирование и осведомленность легко спутать, поскольку они часто реализуются одинаково. В языке Smalltalk всё переменные являются ссылками на объекты, здесь нет различия между агрегированием и осведомленностью. В C++ агрегирование можно реализовать путем определения переменных-членов, которые являются экземплярами, но чаще их определяют как указатели или ссылки. Осведомленность также реализуется с помощью указателей и ссылок.

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 длятся лишь во время исполнения некоторой операции. Отношения осведомленности, кроме того, более динамичны, что затрудняет их выявление в исходном тексте программы.

Коль 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.

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.