В нём [Соборе], как и в любой Сущности, есть нечто такое, что не могут объяснить составляющие её элементы. Собор, есть нечто совсем иное, нежели просто нагромождение камней. Собор – это геометрия и архитектура. Не камни определяют собор, а, напротив, собор обогащает камни своим особым смыслом. Его камни облагорожены тем, что они камни собора. Самые разнообразные камни служат его единству.

Антуан де Сент-Экзюпери

Агрегация

Инкапсуляция, полиморфизм и наследование не решают одну из наиболее важных проблем: проблему объединения функциональных свойств нескольких классов в одном классе. Ситуация, когда необходимо объединить несколько классов в единое целое – новый класс, встречается очень часто. Сторонники одиночного наследования могут сколь угодно долго высмеивать нелепость множественного наследования, но ирония – это не решение проблемы. Однако и множественное наследование не столько решает данную задачу, сколько запутывает решение. Допустим, что нам надо получить класс «автомобиль». Предположим, что у нас под рукой есть все необходимые классы: «кузов», «двигатель», «трансмиссия» и «шасси». Можно ли путём наследования от этих классов получить новый класс: «автомобиль»? Конечно, нет. Предком автомобиля должно быть «транспортное средство» (конный экипаж, к примеру), но никак ни кузов, ни двигатель, ни трансмиссия и ни шасси. Перечисленные классы являются составными частями автомобиля, но не его суперклассами. Правильнее будет утверждение, что автомобиль включает в себя кузов, двигатель, трансмиссию и шасси, но он не может являться их наследником. Термин «включает» служит ключом к пониманию сути. Иными словами, автомобиль является контейнером для перечисленных классов. Из этого примера можно сделать и другой важный вывод, что контейнер – это не механическая смесь классов, а принципиально иная сущность!

От формальных параметров до ролей объектов

Ранее уже говорилось о том, что механизм формальных параметров имел большое значение в структурном программировании, поскольку делал подпрограммы независимыми от глобальных переменных. Не менее важным является и механизм декларации интерфейса, который берёт своё начало в модульном программировании. Суть его состоит в том, что декларация интерфейса оторвана от реализации. Как следствие, можно менять реализации при условии сохранения интерфейса как у одного и того же модуля или класса, так и у различных модулей или классов, например, в случае полиморфных свойств (виртуальных методов). Таким образом, виртуализация, как с помощью формальных параметров, так и с помощью интерфейса, позволяет получать очень гибкие конструкции, которые можно легко настраивать под конкретные задачи и модифицировать в случае необходимости. Но исчерпываются ли на этом возможности виртуализации?

Рассматривая возможность агрегации нескольких объектов в новый класс – контейнер, необходимо определить основы агрегации. Каждый объект обладает некоторым декларированным интерфейсом, с помощью которого с ним могут взаимодействовать другие объекты. Контейнер, имея в своём составе некоторое множество объектов, может предоставлять вовне весь интерфейс вложенных объектов или только его часть. Совокупность интерфейсов, которые использует контейнер у вложенного объекта, назовём ролью вложенного объекта. Несколько вложенных объектов могут играть одинаковые или различные роли в одном и том же контейнере.

Copyright © Alexus, 2001. http://www.alexus.ru

Понятие роли объекта не менее важно для агрегации, чем формальные параметры подпрограмм в структурном программировании. Это положение исходит из того, что понятие роли вводит виртуальность объектов в контейнере. Действительно, можно с полным правом сказать, что контейнер агрегирует объекты, способные выступать в таких-то ролях, то есть, объектов, имеющих некоторую определённую совокупность интерфейсов. Продолжая рассматривать пример с автомобилем, можно отметить, что один и в один и тот же автомобиль можно установить несколько различных моделей (классов) двигателей. При таких заменах автомобиль не перестаёт быть автомобилем, а двигатели – двигателями, даже в случае, если один двигатель карбюраторный, а другой – дизельный. Аналогично в компьютере можно заменить, например, один «жёсткий» диск на другой, или в операционной системе сменить старые драйвера на более новые.

К сожалению, понятие роли, как некоторой совокупности свойств (интерфейсов) объекта не только недооценивается, но и, что ещё хуже, интерпретируется неверно. Так, например, Тимоти Бадд в своей книге «Объектно-ориентированное программирование в действии» приводит иерархию объектов, где люди различных профессий являются подклассами класса «human» (люди) [ТБ стр. 32]. Однако такой взгляд не выдерживает никакой критики. Правильнее говорить о том, что люди могут играть различные роли (иметь различные профессии) благодаря наличию тех или иных свойств. В подходе, предложенном Т. Баддом, неизбежно придётся постоянно перекраивать иерархию, поскольку люди могут не только иметь несколько профессий и качеств, но и приобретать их в течение жизни. Гончар может быть прекрасным садовником, неплохим водителем, мужем, любящим отцом, заботливым дедом и т.п. И это множество профессий и качеств он может разделять со всем остальным человечеством. Можно, конечно, снова вернуться к множественному наследованию, но во что превратится стройная и логичная иерархия? К этому стоит ещё добавить, что часто можно столкнуться с ситуацией, когда в одной и той же роли могут выступать принципиально разные сущности. Например, регулировать движение транспорта может светофор и сотрудник службы безопасности движения. Означает ли это, что они являются «родственниками»? Гораздо логичнее предположить, что они оба имеют необходимый для регулирования движения транспорта набор свойств (качеств, интерфейсов).

Понятие роли динамично, поскольку оно позволяет непосредственно при работе системы объявить ролью некоторый набор интерфейсов и потребовать от системы список классов, объекты которых способны выступать в данной роли. При этом каждый класс может выступать в произвольном количестве ролей. Для примера можно рассмотреть класс, который имеет интерфейсы A, B, C, D и F. В роли «альфа» объединим интерфейсы A и B, в роли «бета» - интерфейсы B, C и F, в роли «гамма» - A, C и D и т.д. Понятно, что подклассы всегда могут выступать в тех ролях, в которых выступают их суперклассы. Однако, благодаря тому, что полиморфизм является понятием независимым от наследования (обратное неверно), то справедливо будет и утверждение, что классы, не находящиеся в наследственной связи, тоже способны играть одну и ту же роль.

Простота и удобство механизма виртуальности объектов в контейнере, предоставляемые агрегацией, позволяют существенно облегчить разработку сложных систем за счёт перехода от упрощенных опытных моделей к промышленным образцам. С другой стороны, на основе данного вида виртуализации можно моделировать и реальную эволюцию сложных систем, например, биологических, технических, социальных. Но не следует забывать, что этот процесс возможен только при соблюдении неизменности интерфейсов. Как следствие, требования тщательности разработки интерфейсов очень высоки. Однако, как будет отмечено ниже, такие требования не являются непреодолимыми, поскольку, с одной стороны, контейнеры существенно упрощают вложенные классы, в том числе и спецификации интерфейсов, а, с другой стороны, наследование контейнеров, как обычных классов,

приводит и к наследованию функционала, который представим в виде ролей вложенных объектов. И, наконец, у контейнеров, как и любых других классов можно будет «наращивать» функциональность по мере необходимости, повышая возможности системы. Поэтому можно установить порядок разработки контейнера, определяя тем самым приоритеты реализации ролей, то есть интерфейсов вложенных классов.

Контейнеры

Существует два основных вида контейнеров: статический и динамический контейнера, хотя вполне допустима и их комбинация. Отличие видов состоит в том, как контейнеры работают с вложенными объектами. Если обращения контейнера к вложенным объектам происходит только через общий интерфейс (все вложенные объекты выступают в одной роли), то такой контейнер является динамическим. Статический контейнер может обращаться к любым свойствам вложенных объектов.

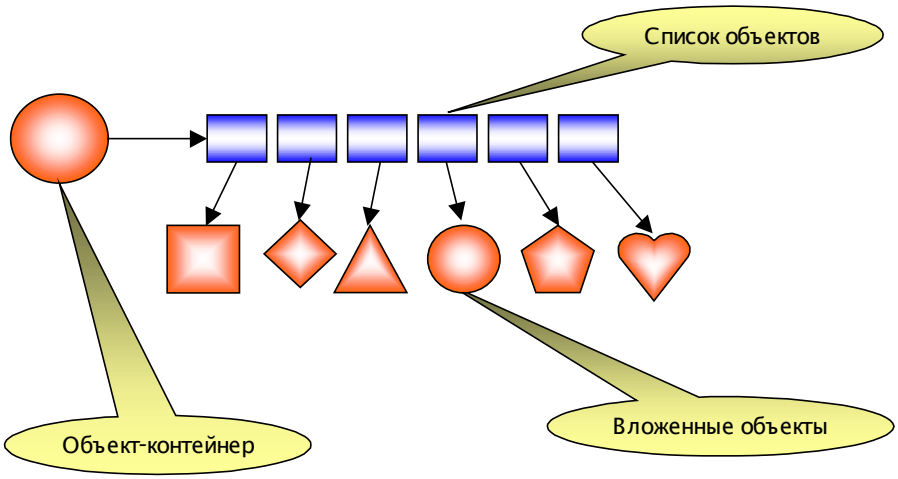

Рис. 03.01. Структура статического контейнера

Статический контейнер характеризуется уникальностью роли каждого вложенного объекта. В классе «автомобиль» роль каждого вложенного объекта «кузов», «двигатель», «трансмиссия» и «шасси» является уникальной. Следовательно, автомобиль – это статический контейнер. В статическом контейнере нельзя произвольно добавлять новые или удалять существующие объекты. Это связано с тем, что уникальность роли объекта предполагает его жёсткую привязку к схемам обработки сообщений, приходящих к контейнеру. При изменении количества или типов вложенных объектов необходимо переопределить свойства контейнера, поскольку они реализуются через интерфейсы вложенных объектов. Возвращаясь к примеру с автомобилем, можно сказать, что изменение числа вложенных классов в статическом контейнере равносильно удалению колёс во время движения автомобиля. Сказанное не означает, что единожды созданный статический контейнер не может быть изменён впоследствии. Изменения возможны и допустимы, но не в рамках исполняемой задачи.

Динамический контейнер состоит из объектов, имеющих одинаковое подмножество полиморфных свойств, при этом обращение к каждому из вложенных классов происходит унифицировано, через этот общий набор свойств, то есть, вложенные в динамический контейнер объекты не обладают уникальными ролями, они все выступают в одной и той же роли.

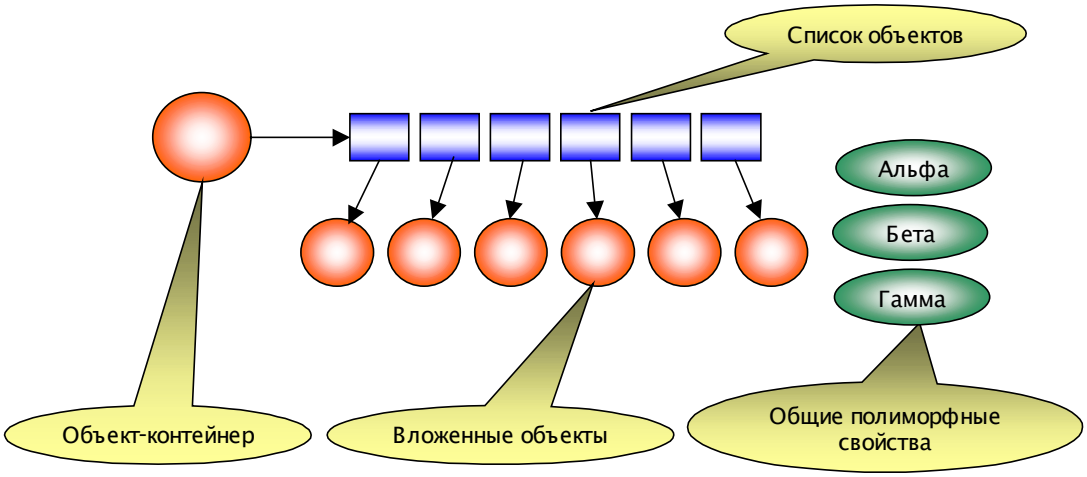

Рис. 03.02. Ст рукт ура динамического конт ейнера

Обычно основу динамического контейнера составляют объекты одного класса, это строгое, но не обязательное требование. В качестве примера динамического контейнера можно рассмотреть массив, стек, очередь, дек или список. Особенностью динамического контейнера является возможность произвольного изменения количества вложенных объектов. Это обусловлено тем, что обращение к объектам строится исключительно на основе роли, единой для всех объектов. Например, если нам необходимо сравнивать объекты с некоторым значением определённого типа и копировать их в заданную область памяти, то этими свойствами должны обладать все вложенные в динамический контейнер объекты, независимо от того, какому классу они принадлежат. Именно эта особенность позволяет произвольно подключать к контейнеру новые объекты. Схема работы динамического контейнера задаётся на стадии его проектирования. Допустимы, например, схемы, когда сообщение, поступившее к контейнеру, просто перенаправляется на один или более объектов, которые в данный момент являются активными. Но возможно, что сообщение проецируется на все объекты или часть объектов динамического контейнера.

Наконец, контейнер может представлять собой комбинацию статического и динамического контейнеров. Предположим, что мы рассматриваем работу сотрудников отдела. Вполне возможно, что роль каждого служащего уникальна. При поступлении некоторого задания в отдел, оно распределяется по сотрудникам с учётом их специализации. Но, тем не менее, каждый сотрудник является служащим и к нему применимы операции допустимые для всех служащих: приём на работу и увольнение, начисление зарплаты и отпускных, оплаты больничного листа и т.п. Таким образом, отдел представляет собой комбинацию статического и динамического контейнеров. В отличие от статического контейнера комбинированный контейнер позволяет одинаково взаимодействовать с вложенными объектами, выступающими в одной роли. В свою очередь, разница с динамическим контейнером заключается в том, что в комбинированном контейнере нельзя произвольно изменять количество вложенных объектов, поскольку роль каждого объекта уникальна.

Как отмечалось в начале раздела, контейнер представляет собой класс, отличный от классов вложенных в него объектов. То есть, контейнеры образуют собственную иерархию, или ветвь иерархии, если в проекте предполагается иметь общий суперкласс для всех классов. При наследовании реализации контейнеров подкласс получает от суперкласса не только его свойства, но вложенные в него классы. Например, можно создать подклассы класса «автомобиль»: «грузовой автомобиль», «легковой автомобиль» и «специальный автотранспорт». Все эти подклассы наследуют от класса «автомобиль» вложенные классы: «кузов», «двигатель», «трансмиссию» и «шасси». Как и всегда при наследовании, подкласс может переопределять реализацию свойств, полученных от родителя.

Кроме иерархии наследования, контейнеры образуют иерархию вложений. Контейнеры могут иметь неограниченную, по крайней мере, теоретически, глубину вложенности. Вкладывать можно любой тип контейнера в любой тип контейнера. Например, можно вложить динамический и комбинированный контейнер в статический, но можно статический контейнер вложить в динамический или комбинированный. Наконец, контейнеры имеют собственную логику развития и собственную иерархию классов-контейнеров.

Контейнеры обладают двумя качествами, о которых следует поговорить подробно. Первое из них заключается в уменьшении числа элементарных классов. Под элементарными классами здесь и далее понимаются классы, которые вкладываются в контейнеры, но при этом сами контейнерами не являются. Ограниченное количество элементарных классов, в свою очередь, приводит к простым и компактным иерархиям. Если Вы уже работали с иерархиями классов, поставляемыми вместе с компиляторами объектно-процедурных языков, то, возможно, Вам будет интересно оценить это качество. Уменьшение числа элементарных классов объясняется существующим ограниченным числом основных примитивов, характерных практически для любой предметной области. Объединяя в контейнере эти примитивы, можно добиться любой нужной комбинации (композиции). С этой точки зрения переоценить возможности, предоставляемые контейнерами, очень сложно. В свою очередь простота элементарных классов позволяет предположить и определённую лёгкость их разработки и, соответственно, весьма низкие требования к используемым при создании инструментальным средствам. Использование низкоуровневых инструментов программирования не затруднит кодирование, в силу простоты алгоритмов, но позволит получить компактный и эффективный код свойств элементарных классов.

Второе качество контейнера, заслуживающее отдельного рассмотрения, заключается в том, что его можно конструировать, а не кодировать. Свойства контейнера реализуются посредством схем. Схемы представляет собой граф разложения входного сообщения на сообщения к вложенным объектам. Таким образом, любой контейнер представляет собой менеджер запросов к вложенным объектам. В свою очередь, конструирование предполагает если не полное отсутствие кодирования, то, по крайней мере, высокий уровень инструментальных средств и простоту операций. Можно констатировать, что создание кода свойств элементарных классов – это практически единственная область программирования в том виде, как мы привыкли его видеть.

Контейнеры представляют собой более мощный и гибкий вид агрегации, по сравнению с множественным наследованием. Увеличение мощности связано с тем, что количество вложенных объектов можно наращивать, в том числе и непосредственно в процессе работы контейнера. Большая гибкость контейнеров обусловлена возможностью замены вложенных объектов, что позволяет отразить процессы, происходящие в предметной области, наиболее естественным образом. Очень часто можно встретиться с ситуацией, когда те или иные объекты появляются в контейнере только на некоторое время. Например, можно рассмотреть обычный кухонный комбайн, меняя насадки на основном агрегате, мы меняем его возможности и функциональность. Насадки, снятые с агрегата, перестают быть составной частью контейнера и на их место могут быть установлены другие насадки. В качестве другого примера можно рассмотреть седельный тягач. Меняя тип прицепа, можно получить и рефрижератор, и бетономешалку, и автокран, и многие другие виды специального автотранспорта. Вряд ли множественное наследование сможет помочь в создании подобных сущностей.

Контейнер, в отличие от элементарного класса, обладает ещё одной особенностью. Как и любой другой класс, контейнер поддерживает инкапсуляцию. Но инкапсуляция распространяется только на сам контейнер, на его структуру данных и его методы. Несколько иначе обстоит дело с вложенными объектами.

Поскольку контейнер является сборной конструкцией, то это означает, что мы можем различать его составные части. То есть, вложенные объекты не инкапсулируются в контейнер автоматически. Реализация контейнера, как составной сущности, может быть как скрытой (инкапсулированной), так и открытой. Но даже в том случае, если структура вложенных объектов известна внешнему программному обеспечению, доступ к вложенным объектам должен происходить через интерфейс, предоставляемый контейнером. Прямой доступ к вложенным объектам, минуя контейнер, должен быть закрыт, в противном случае трудно гарантировать правильную работу контейнера. Как отмечалось выше, динамический контейнер позволяет добавлять вложенные объекты и удалять их непосредственно во время работы. Статический контейнер тоже может изменять количество вложенных объектов или заменять одни объекты на другие, способные играть те же роли. Однако изменение структуры вложенных объектов у статического контейнера нельзя производить во время работы, поскольку это может привести к сбою в работе контейнера. Более правильным подходом в данном случае будет следующая последовательность действий:

- выведение статического контейнера из рабочего режима;
- проведение изменений;
- повторный запуск в работу.

Однако это не более чем рекомендации и в каждой конкретной ситуации надо поступать так, как этого требует логика взаимодействия с контейнером.

К сожалению, иногда исследователи объектной технологии противопоставляют контейнеры самой объектной технологии. Об этом достаточно много и подробно говорится в книге Тимоти Бадда. Например, он пишет, что «Одна из наиболее необычных идей в STL [Standard Template Library] – обобщённые алгоритмы. Она заслуживает отдельного рассмотрения, поскольку выглядит вызовом объектно-ориентированным принципам, которые мы обсуждали до сих пор…». Т. Бадд исходит из того, что контейнеры обобщают алгоритмы реализации, абстрагируясь от типов объектов, которые они обслуживают. То есть, например, можно создать класс-контейнер, как список или очередь произвольных объектов. Алгоритмы работы со списком или очередью не зависят от того, объекты каких классов вложены в данный контейнер. На самом деле и список, и очередь в данном случае представляют собой частную разновидность динамического контейнера. По всей видимости, есть определённая недооценка или непонимание того факта, что контейнер представляет собой иную сущность, отличную от вложенных в него объектов, даже в том случае, если и контейнер и вложенные объекты имеют один и тот же набор интерфейсов. Организация способа хранения вложенных объектов и взаимодействия с ними менее существенны для прикладной задачи, нежели логика работы контейнера. Поэтому вопросы организации контейнера, способов хранения и взаимодействия с вложенными объектами правильнее оставить за системой, в то время как вопросы описания логики контейнера должны решаться программистом. При работе с динамическими контейнерами, наверное, можно допустить опциональную возможность для того, чтобы программист указал желательный способ организации контейнера, исходя из объёма информации, обслуживаемой контейнером, и/или специфики решаемой задачи.

Схемы обработки сообщений

Важным требованием к включающим сущностям, будь то класс, плод множественного наследования, или контейнер, является возможность проецирования свойств вложенных объектов. То есть, эти сущности могут обладать некоторыми свойствами тех классов, которые в них включены. В контейнере данная проблема решается за счёт использования полиморфности свойств. Вполне достаточно указать, что данный контейнер умеет обрабатывать необходимый набор сообщений.

Обработка сообщений, в том числе спроецированных от вложенных классов, может быть простой или сложной, но в любом случае она реализуется посредством схем.

Схема – это реакция контейнера на определённое входное сообщение. В динамическом контейнере схема обработки сообщений, как правило, достаточно тривиальна. Входное сообщение обычно либо проецируется на все вложенные объекты, либо передаётся активному в данный момент объекту. Иначе обстоит дело со статическими контейнерами. Здесь обработка сообщений является важным инструментом реализации логики контейнера. При разложении исходного сообщения на множество сообщений к вложенным объектам возможны две ситуации: некоторая последовательность сообщений может быть обработана вложенными объектами параллельно или последовательно. То есть, при конструировании схем ясно видны сообщения, которые допускают параллельную обработку и те, которые её не допускают. Параллельные участки могут в рамках одной схемы сменяться последовательными участками и наоборот. Точку, в которой необходимо дождаться завершения обработки нескольких сообщений, исполняемых параллельно во вложенных классах, будем называть точкой синхронизации. Сообщения, обрабатываемые последовательно, разделяются точкой синхронизации по умолчанию. Завершение схемы обработки сообщения в контейнере тоже сопровождается точкой синхронизации.

Основное значение схем контейнеров состоит в том, что они позволяют отделить логику работы вложенных объектов от логики управления этими объектами или, что то же самое, от логики взаимосвязи между вложенными объектами. Такое разделение логики приводит к двум эффектам: независимость логических слоёв и значительное упрощение иерархий элементарных классов. Независимость логических слоёв основывается на том, что при разработке конкретных классов не надо решать вопросы взаимодействия этого класса с другими классами, в том числе и контейнерами, в которые он может быть вложен. Любой класс должен соответствовать своему назначению и поддерживать специфицированный интерфейс. Как следствие данного положения, можно отметить и тот факт, что в данном случае всегда остаётся возможность разделения текущего уровня на подуровни, без переделки и перекомпиляции смежных уровней. В результате можно вначале строить «грубую» модель, а затем постепенно уточнять её, детализируя подуровни. Это и есть правило локализации для логических уровней. Реализация каждого логического уровня является независимой от других уровней или, говоря другими словами, каждый логический уровень инкапсулирует собственную реализацию. Например, линия не должна «знать» о том, какие графические примитивы будут созданы на её основе; атому нет дела до того, в образовании каких молекул он примет участие; структурным группам Ассура нет дело до того, какие механизмы будут создаваться на их основе.

При обработке сообщения, возможно, возникнет такая цепочка сообщений к вложенным объектам, когда результат обработки сообщения одним объектом должен быть передан для последующей обработки на другой вложенный объект. Возникает соблазн связать выход одного объекта с входом другого, минуя контейнер. Однако так делать не следует. Обмен сообщениями должен происходить только через контейнер. Это обусловлено тем, что состояние контейнера или контекста могло измениться за то время, пока первый объект обрабатывал сообщение. Изменение состояния контейнера может повлиять на порядок обработки исходного сообщения. Более подробно о состояниях контейнера и контексте будет сказано ниже.

Схема может обладать локальными данными. Эти данные обеспечивают сохранение промежуточных результатов, полученных в результате обращений к свойствам вложенных объектов. Локальные данные схем существуют только при работе схемы и не сохраняются между вызовами схемы. Можно провести аналогию между локальными данными схемы и локальными данными подпрограммы. Суть их аналогична. Точно также как и подпрограмма, схема может иметь локальные данные,

сохраняемые между её вызовами. Эти данные расширяют структуру данных контейнера. Наконец, схема может взаимодействовать с глобальными данными контейнера, которые доступны и другим схемам.

При проектировании контейнера нам известны те сообщения, которые он должен уметь обрабатывать. Очевидно, что и логика обработки сообщений тоже известны. Но тогда нужно допустить, что последовательность сообщений к вложенным объектам, нам также известна. Следовательно, схема на физическом уровне представляет собой ничто иное, как последовательность сообщений к вложенным объектам и свойствам самого контейнера разделённая точками синхронизации. Отсюда можно заключить, что приход сообщения к контейнеру производит запуск (проигрывание) соответствующей последовательности сообщений к вложенным объектам в заданном порядке. Но поскольку схемы являются единым механизмом реализации свойств контейнеров, то и проигрывание последовательности сообщений к вложенным объектам происходит единообразно, и не зависит ни от типа сообщений, ни от класса контейнера. Что, в свою очередь, позволяет говорить не о программировании, а об общих принципах конструирования схем.

При конструировании схем весьма удобно пользоваться графическими средствами, которые наглядно отображают последовательность обработки входного сообщения посредством обращения к свойствам вложенных объектов. Конечной целью проектирования схемы является получение последовательности сообщений к вложенным объектам и свойствам самого контейнера.

В целом ряде задач схемы могут создаваться динамически, например, на основе запроса пользователя. В качестве такого примера можно рассмотреть запросы пользователя к базе данных. Запрос может быть оформлен на языке SQL, а затем переведён в набор схем, которые поступают на обработку.

Состояния контейнеров

Контейнеры представляют собой более высокий уровень проектирования по сравнению с проектированием элементарных классов. При вложении одного контейнера в другой происходит дальнейшее повышение уровня, что в конечном итоге позволяет получать очень сложные классы. Обычно повышение уровня контейнера сопровождается усложнением логики его работы. Например, контейнер в зависимости от своего состояния может различно отрабатывать одно и тоже сообщение. Контейнеры могут иметь произвольное число состояний и условий перехода из одного состояния в другое. Процессы определения состояния контейнера и выполнения условий перехода в новое состояние могут быть очень нетривиальны, и зависеть от многих факторов. Факторы, побуждающие к переходу контейнера в новое состояние, делятся на внутренние и внешние. Влияние внутренних факторов напрямую зависит от работы контейнера и его текущего состояния. Внешние факторы определяются тем контейнером или средой, в которые вложен рассматриваемый контейнер.

Образование логики изменений состояний может быть как очень простым, так и очень сложным с точки зрения формализации этого процесса. При построении сложных систем возникают ситуации, когда определение состояния контейнера и условий перехода производится на основании эмпирических знаний, так как может не иметь строгой формализации. В таких случаях весьма уместно использовать механизмы, применяемые в экспертных системах, и которые, как минимум, включают в себя фактологическую базу и машину вывода. Применение этих механизмов внутри контейнера можно представить следующим образом. Из всех сообщений, поступающих к контейнеру, отмечаются те, которые способны привести к изменению его состояния. При обработке данных сообщений контейнер должен контролировать своё состояние на основании заданных условий. Переход в новое состояние означает

переключение схем обработки сообщений. Дальнейшая обработка сообщений будет происходить при новом состоянии контейнера.

Для многих приложений весьма желательно регистрировать моменты изменения состояния контейнера. Кроме того, регистрация существенно упрощает процесс отладки логики работы контейнера, условий перехода и изменений фактологической базы. Необходимо отметить, что использование нескольких состояний контейнера встречается достаточно часто, но обычно логика переходов достаточно проста и не требует введения элементов систем искусственного интеллекта. Но, тем не менее, существует определённый круг предметных областей, где невозможно получить качественную формализацию. Здесь трудно обойтись без использования эмпирических знаний.

Говоря иными словами, наличие нескольких состояний контейнера фактически приводит к виртуализации схем обработки сообщений, поступающих к контейнеру. Виртуализация может касаться только части схемы, например, начало схемы может не зависеть от состояния контейнера. В такой ситуации удобно делить схему на несколько подсхем, то есть, проводить декомпозицию схемы.

Образование языков

Вложенные классы, их свойства, а также свойства контейнера-класса могут иметь синонимы. Область видимости синонимов ограничена контейнером. Одно и тоже полиморфное свойство может иметь различные синонимы как в рамках одного объекта, так и у различных объектов. Все синонимы должны быть уникальны в пределах одного объекта. Аналогично, синонимы всех вложенных объектов должны быть уникальны внутри контейнера. Синонимы не могут пересекаться с естественными названиями объектов и свойств. Наконец, синонимы должны поддерживать национальные кодировки.

Введение синонимов у свойств противоположно по действию полиморфизму. Так, одно и тоже полиморфное свойство может иметь различные синонимы даже у объектов, относящихся к одному классу. Механизм поддержки синонимов удобно использовать на различных логических уровнях (уровнях вложений контейнеров). Это позволяет пользователям косвенно указывать, к какому логическому уровню они обращаются. Как следствие, упрощается взаимодействие с системой. Для примера, можно сказать, что понятие «свойства класса» и «схемы контейнера» являются синонимами, поскольку контейнер тоже является классом. Однако, когда речь идёт о схеме, то становится понятно, что в контексте подразумевается именно контейнер, а не элементарный класс.

Наименования объектов и их свойств в рамках схем контейнера образуют основу декларативного языка схем. Этот язык необходим в случаях, если логика схемы сложна и её графическое представление перегружено, что затрудняет восприятие. Использование синонимов позволяет достичь очень хорошей выразительности языка.

При разработке сложных систем, имеющих несколько логических уровней, язык каждого уровня, возможно, будет отличаться по своим возможностям. Более низкие логические уровни используют и более низкоуровневые языки, в состав которых будут входить операторы, используемые в традиционных языках высокого уровня (3GL). Необходимость использования этих операторов на более высоких уровнях вызывает сомнения. Здесь вполне достаточно обращений к свойствам вложенных классов, которые и производят всю работу. Можно отметить тенденцию изменения языковых средств с повышением логических уровней контейнеров. На нижних уровнях описание схем удобно производить на языках написания скриптов, аналогичных Perl или Tcl. На верхних уровнях происходит смещение в область языков логического программирования, но здесь очень важна и удобна графическая нотация языка.

Сосуществование нескольких языков на соответствующих логических уровнях позволяет совместить эффективность и детальность кода низкоуровневых средств программирования с высокой скоростью разработки, которую обеспечивают языки высокого уровня. И это положение вещей вполне согласуется с тем, что низкие уровни разрабатываются системными программистами, в то время, как средние уровни создаются прикладными программистами, а высокие логические уровни могут разрабатываться подготовленными пользователями.

Заключение

Заканчивая обзор основных положений объектной технологии, хотелось бы ещё раз остановить внимание на ключевых моментах.

Объектная технология является логичным следствием развития идей структурного программирования. Структурное программирование за счёт декомпозиции кода позволило создавать удобные подпрограммы, которые можно было использовать во многих проектах. Декомпозиция кода посредством подпрограмм снизило сложность программы за счёт локализации связей между операторами и отделения интерфейса. Интерфейсом подпрограммы являлись: название, тип и количество передаваемых и возвращаемых параметров. Механизм формальных/фактических параметров являл собой механизм виртуализации данных.

Объектная технология развивает идею виртуализации и переносит её на сами подпрограммы. Объект представляет собой агрегат, объединяющий некоторый набор свойств, реализованных в виде данных и подпрограмм (методов). Виртуализация подпрограмм позволила абстрагировать интерфейс подпрограммы, поставив ему в соответствие не одну, а несколько реализаций. Каждый подкласс может иметь собственную реализацию интерфейса (метода), объявленного у его суперкласса. Последующее развитие идеи виртуализации логично приводит к виртуализации самих объектов в рамках контейнера. В основе виртуализации объектов лежит понятие роли. Роль понимается как произвольная совокупность интерфейсов. Говоря о том, что объект некоторого класса, может выступать в данной роли, подразумевают, что объект этого класса обладает необходимыми для данной роли интерфейсами. Для контейнера важна роль, а не конкретный объект и, поэтому вполне допустимо заменять в контейнере объект одного класса на объект другого класса, если оба этих объекта обладают необходимыми для данной роли интерфейсами. Точно так же, как происходит виртуализация элементарных объектов в контейнере, можно подвергать виртуализации и сами контейнеры, вкладывая их в контейнеры более высоких логических уровней.

Полиморфизм, рассматриваемый как самостоятельное понятие, позволяет строить иерархии классов более естественным образом, достигая их простоты и выразительности. Кроме того, самостоятельность полиморфизма является основой для введения понятия ролей. Посредством полиморфизма, в частности, становится возможным объявлять у контейнеров интерфейсы, аналогичные интерфейсам, которые имеют вложенные классы. А в схемах, которые реализуют эти интерфейсы контейнеров, определить способ передачи сообщения одному или более вложенным классам.

Инкапсуляция необходима, если в создаваемом проекте записаны требования надёжности и модернизируемости. Главным следствием инкапсуляции можно смело назвать необычайно высокую гибкость и настраиваемость системы. На основе этого возможен переход к иному подходу к разработке программного обеспечения: подходе, основанном на макетировании и моделировании. То есть, программное обеспечение создаётся в виде очень простого макета, обладающего минимальной функциональностью, а затем постепенно доводится до необходимого уровня готовности. Такой подход отражает тот факт, что реальный мир не является чем-то завершённым и находится в постоянном развитии.

Copyright © Alexus, 2001. http://www.alexus.ru

Наследование неотделимо от абстракции. Благодаря наследованию достигается абстрактная спецификация интерфейсов классов. Реализация этих интерфейсов может различаться в подклассах. Благодаря наследованию, подклассы могут получать от своего суперкласса не только интерфейсы, но и их реализацию, что увеличивает повторное использование кода и повышает его качество. На уровне суперкласса можно определять и поведение данного вида. Подклассы наследуют это поведение, обеспечивая тем самым, идентичность реакций на те или иные события. Абстракция при наследовании реализации наиболее полно раскрывает свои возможности и достоинства при агрегации. Подкласс-контейнер получает реализацию суперкласса-контейнера и может видоизменять её, увеличивая функциональные возможности. Этот процесс хорошо отражает явления, происходящие в реальной жизни.

Таким образом, объектная технология предоставляет превосходную основу для построения программных моделей. Однако обилие возможностей, заложенных в объектной технологии, позволяет применить иные подходы к проектированию, подходы, способные не только ускорить разработку программного обеспечения, но и дающие возможность реализовывать гораздо более сложные системы. Но перед тем, как перейти к вопросам проектирования, необходимо рассмотреть два важных положения: о формах связей между объектами и о параллелизме. Этим вопросам посвящено следующее письмо.