Разнорукое программирование

Оглавление

Лирические ассоциации

Мне нравится объектно-ориентированное программирование (ООП). Доставляет удовольствие процесс формирования каркаса приложения на основе базовых классов и дальнейшая раскрутка его, с применением механизма наследования. Это не отступные, вызванные саркастическим тоном предыдущего материала. Это действительно так. Истинно объектные механизмы обеспечивают, при определенных условиях, эффективное конструирование.

Но мне нравится и процедурное программирование (ПП): своим четким разделением программных объектов на данные и процедуры, ясными и понятными концептуальными моделями, возможностью независимой функциональной декомпозицией и такой же декомпозицией данных, сочетаемых и совмещаемых на каждом шаге проектирования (чего обычно не хотят замечать поверхностные апологеты ООП [King], акцентируя критику только на функциональной декомпозиции).

Поэтому, я программирую на C++, который позволяет гармонично сочетать оба подхода. Этим давно пользуются специалисты, например Скотт Мейерс [Мейерс2000-1, Мейерс2000-2]. Он описал много эффективных приемов, обеспечивающих использование как чисто ОО стиля, так и его сочетаний с процедурным подходом, правда, явно не выделяя использование последнего [Meyers]. Думаю, что C++ будет еще долго оставаться одним из основных инструментов нашей рабочей группы (по крайней мере, до того момента, пока не будут созданы языки и трансляторы, более полно удовлетворяющие предъявляемым нами требованиям:).

Помимо этого я прекрасно отношусь к функциональному стилю. До сих пор считаю это направление одним из перспективных, особенно в области параллельного программирования.

Во многом это происходит оттого, что я рассматриваю программирование как искусство, являющееся предметом анализа. Стили программирования интересны мне сами по себе, а не конечным результатом, определяемым разработанной программой. А изучение техники программирования, во многом, определяемой парадигмами, - мое хобби. Исходя из этих рассуждений я и пытаюсь играть роль "искусствоведа", "восхваляя" то, что мне нравится, и критикуя то, что "не воспринимаю". Естественно, что получаемые оценки носят субъективный характер, не смотря на попытки объективно разобраться в сложившейся ситуации. Опираясь на эту позицию, я пытаюсь далее проанализировать основные черты парадигм программирования.

Подобное восприятие предметной области относится не только к парадигмам. В меньшей степени (хотя, и не такой уж и малой:) меня интересуют методологические и архитектурные аспекты программирования, между которыми находятся парадигмы. Именно такое промежуточное положение и определяет основные задачи стилей: служить средством преодоления семантического разрыва между архитектурами вычислительных систем и разработчиками программного обеспечения. А здесь, на мой взгляд, только одного ООП или ПП сейчас недостаточно.

Проведем ассоциацию программирования с человеком. Пусть ООП будет его правой рукой, а ПП - левой (можно и наоборот, но многие почему-то ассоциируют ООП с мышью, которая чаще располагается под правой рукой:). Отрежьте человеку одну руку, и он сможет со временем научиться выполнять функции, осуществляемые ранее отсутствующей рукой. Только это будет менее эффективно. Вот потому я с предубеждением отношусь к чисто объектным средствам и языкам программирования (Java, C#), предпочитая им "громоздкий и избыточный" C++. Нельзя нормально программировать, не используя при этом всех имеющихся знаний. Работа Скотта Мейерса [Meyers], перевод которой лежит на моем сайте, лишний раз подтверждает это. Поэтому, когда знатоки Java и C# говорят, что эти языки являются подмножеством C++, из которых убрали все лишнее, я судорожно начинаю соображать: какую руку мне отрезать, а какую дополнительно обучить (я, от природы, праворукий левша: правой рукой пишу, а левой - рисую).

Меня также не убеждают аргументы, что "урезанные" языки легче для изучения, а "однорукое программирование" порождает более надежный код. Во-первых, я достаточно долго изучал программирование вообще и C++ в частности, чтобы верить, что надежность определяется только языковыми средствами. Кроме чистого кодирования необходимо еще и формирование соответствующей культуры, которая складывается годами. Посмотрите, сколько книг и статей посвящено именно культуре программирования на C++. Почитайте "Дизайн и эволюцию C++" [Страуструп2000], в которой четко прослеживается идея создания языка, обеспечивающего поддержку "правильного" и "неправильного" программирования. А чему учат книги, посвященные "самым современным" языкам? По-моему, искусству манипуляции с мышью (еще одна однорукая ассоциация:).

Во-вторых, культура программирования во многом не зависит от того языка, на котором приходится писать программы. Кроме чистого кодирования и рисования форм существует еще множество других

технологических пластов, которые необходимо взрыть, осознать и применить. И никакой, даже самый современный язык программирования, ориентированный на использование любых компонент, сам по себе не обеспечит создание надежного и эффективного приложения.

Поверьте, я ничего не имею против визуального конструирования, компонент, волшебников, экспертов, мастеров, библиотек классов и прочих средств, обеспечивающих быстрое наращивание дополнительной функциональности приложений. Конечно, можно сразу сесть за компьютер и слепить из готовых компонент программу. Однако, это совершенно другая тема, требующая отдельного разговора. Она мало касается техники кодирования и практически не соприкасается с рассматриваемыми ниже вопросами. Любое из названных выше средств может быть разработано с использованием как объектно-ориентированного, так и любого другого стиля. Они могут применяться при разработке объектно-ориентированных и процедурных программ. Не я написал критику библиотеки классов MFC за отсутствие объектной ориентации. Не я разработал модель компонентных объектов (COM) таким образом, чтобы компоненты прекрасно создавались с использованием как процедурных, так и ОО языков [Роджерсон]. Популярность этих инструментов просто лишний раз показывает, что повышать эффективность программирования можно различными способами, в том числе, и расширением функциональных возможностей программных объектов, используемых в качестве базовых строительных блоков.

Хотелось бы также подчеркнуть, что представленный материал посвящен технике программирования (кодирования), а не сравнительному анализу языков. Поэтому, не затрагиваются особенности синтаксиса и семантики, наличия или отсутствия различных бантиков. Над любым из языков можно глумиться, но почти на любом из них можно писать прекрасные программы.

Но "вернемся к нашим баранам". "Противостояние" между сторонниками ОО и прочих стилей напоминает мне стычки между Линуксоидами и Виндовозниками. Но не содержанием перепалок, а качественным составом "противоборствующих" сторон.

ОО программисты видятся как чернокнижники-интеллектуалы, познавшие таинства магических заклинаний собранных по крупицам из сотен книг и сайтов. Выстраивая вокруг себя непонятные диаграммы классов и оживляя их магическими словами (класс, паттерн, UML...), они постепенно строят огромное чудище путем эволюционного добавления к нему все новых и новых кусков.

Процедурные программисты - это сторонники быстрых решений, которым ничего не стоит слепить программу, тут же разломать ее и перелепить заново, если что-то не понравилось. Им не нужна крепость, так как они не ждут волка. Достаточно иметь соломенную хижину Ниф-Нифа. И неважно, какой язык программирования при этом используется: процедурный или объектно-ориентированный.

Если же говорить серьезно, то коренным же отличием этих двух групп является ориентация на разработку приложений различного объема и сложности, что ведет к разным методологическим и техническим решениям. Именно использование технических решений, обеспечивающих эволюционное расширение уже написанного кода, следовать которым необходимо от начала и до конца проекта, определяет всю таинственность и громоздкость объектно-ориентированного программирования. Недостаточно просто выучить соответствующий язык. Необходимо еще овладеть множеством методов и приемов, определяющих правильное использование изученных конструкций. Вот почему объектно-ориентированное проектирование постоянно сопоставляется с архитектурой, а методы Кристофера Александра нашли широкое понимание и поддержку [Appleton, Гамма].

Вместе с тем, большинство приложений не требуют тщательного проектирования, а также повторного использования. Поэтому и существуют программисты, которым безразличны ритуальные танцы вокруг разрабатываемой программы. Зачастую здесь нет ничего плохого: зачем дополнительно выкладываться в обучение проектированию при написании серии слабо взаимосвязанных скриптов, манипулирующих высокоуровневыми конструкциями и доступными компонентами? Не важна также и принадлежность языка программирования к определенной парадигме.

Окончательно зафиксировать высказанные предположения мне хочется еще одной простенькой ассоциацией. Предположим, что надо написать программу (возможно, небольшую), осуществляющую перевозку груза из пункта A в пункт B. Обычный программист обратит все свое внимание на достижение окончательного результата, не задумываясь особо над вариациями, возможными при повторном решении этой же задачи. Скорее всего, его программа будет написана быстро и успешно справится с поставленной проблемой. ОО программист начнет выстраивать "дорогу жизни", изначально предусматривая объездные пути возможных преград и ловушек, которых в действительности может и не оказаться. Естественно, что такое программирование займет больше времени. Однако, если вслед за этой задачей, появится новая, определяющая перевозку из C в D, но при других условиях, ОО

программист начнет пожинать плоды со своей предусмотрительности. У Скотта Мейерса, по этому поводу, даже есть правило под номером 32: "Программируйте, заглядывая в будущее" [Мейерс2000-2].
Таким образом, объектно-ориентированное программирование - это не только соответствующий язык. Это еще и образ мышления, направленный на создание эволюционирующих программ.

Конкретные ограничения

После столь длительного лирического вступления, мне бы хотелось отметить специфику собственных интересов. Меня интересует техника эволюционной разработки программ и ее зависимость от избранной парадигмы программирования. Дело в том, что разрабатывать такие программы можно и без использования объектно-ориентированного подхода, что может нести свои плюсы и минусы. ООП, в данной сфере, тоже обладает не только достоинствами, но и (на мой взгляд) недостатками. Скорее всего, эволюционное программирование тоже должно быть двуруким. Однако для этого необходимо более четко определить: чем отличаются друг от друга ПП и ООП. Поиском ответа на поставленный вопрос я и пытаюсь заняться в представленных заметках. Несмотря на попытку объективного анализа, я, все-таки, высказываю свою, чисто субъективную точку зрения, которая, по некоторым аспектам, вполне может оказаться ошибочной. Поэтому, с благодарностью приму любые конструктивные замечания и предложения, обеспечивающие согласованное общее восприятие "Мира программирования". В этот раз я никого не собираюсь провоцировать (зачем дважды подставлять одни и те же грабли:), а просто выношу на всеобщее обозрение ряд актуальных, на мой взгляд, вопросов. Сразу хочу сказать, что этот материал является далеко не последним в общей серии, посвященной парадигмам программирования.
В представленных заметках я попытался отделить методы кодирования, определяемые различными парадигмами, от методологических аспектов, особенностей реализации и функционального наполнения. Такое обособление позволяет, на мой взгляд, провести более четкую границу между техникой, используемой при формировании кода, и концептуальными умозаключениями, построенными на основе комплексного восприятия решаемой задачи, реального мира, мира моделей и аппаратных ресурсов.

Использование терминов и определений

Я давно перешел в разряд отцов, и, по определению Тургенева, устарел. Проявлением консерватизма является и использование мною определений терминов, построенных еще в незапамятные времена [АРНФС]. В настоящее время многие из них слегка изменили окраску. Поэтому под программированием я понимаю, то, что ныне воспринимается как кодирование. Сейчас, довольно часто, программирование ассоциируется с элементами проектирования.
Более широкую трактовку приобрел термин "парадигма программирования". Он иногда воспринимается не только как стиль программирования, навязанный языком (техника кодирования), но метод, определяющий технику разработки программ в конкретной предметной области. Не вдаваясь в дискуссию по поводу терминологических метаморфоз, буду в дальнейшем считать близкими по смыслу понятия: программирование и кодирование, парадигма программирования и техника кодирования.
Стоит отметить и термин "эволюционное программирование". Он прямо не связан с эволюционной разработкой и спиральной моделью. Косвенная связь заключается в следующем. Закончив очередной виток эволюционного цикла разработки версии продукта, мы получаем на выходе код, написанный на некотором языке программирования. Новый виток спирали требует расширения и модификации этого кода, то есть его эволюции. Он не создан волшебником, его невозможно автоматически сгенерировать при повторном проектировании. При этом желательно как можно меньше изменять уже написанный код, используя методы, позволяющие его наращивать. То есть (учитывая метаморфозы терминологии), эволюционное программирование можно ассоциировать с эволюционным кодированием. В целом же эта задача была поставлена до меня. Поэтому не буду останавливаться на ее особенностях. Ниже рассмотрена возможность использования различной техники кодирования для эволюционного расширения уже написанной программы.
Использование этих терминов, в какой-то мере объясняет стремление очиститься и от ряда наслоений на программирование (кодирование).

Очищение от методологий

Нельзя отрицать ту важную роль, которую играют методологии в процессе разработки программного обеспечения. Но они больше связаны с философским восприятием и отображением окружающего нас мира, чем с техникой кодирования. Для методологий важнее то, в какой последовательности они

используют различные модели, а не то, в какой код они при этом отображаются. Фиксация последовательности шагов ведет к однобокому восприятию окружающего нас мира (что вполне соответствует и различным философским учениям). Например, независимо от того, обладает реальный объект поведением или нет, ОО подход навязывает ему внутренние методы. И хотя, на конечной реализации такое навязывание может и не сказываться, я не считаю правильным использовать одинаковые модели для театра людей и театра марионеток. Вряд ли здесь стоит говорить об одинаково адекватном отражении в общей объектной модели реалий окружающих нас миров. Полагаю, что по этому вопросу могут возникнуть альтернативные суждения. Поэтому сразу отказываюсь от дальнейших бесплодных (философских:) дискуссий.
Методологии могут одинаково манипулировать ортогональными понятиями рассматриваемой предметной области. Используя одни и те же методы, в той ж самой последовательности, но в "нестандартном" контексте можно получать аналогичные результаты. Об этом, в несколько утрированной форме я и пытался сказать в заметках, посвященных процессо-ориентированному программированию.
Именно из этих соображений я попытался отсечь методологический аспект, чтобы более конкретно рассмотреть реальную технику создания кода, поддерживаемую на уровне языков.

Очищение от аппаратных ресурсов

Точно таким же образом я попытался абстрагироваться и от отображения языковых конструкций на ресурсы вычислительных систем, хотя, важность этой составляющей весьма очевидна. Например, один из принципов, лежащий в основе языка программирования C++, заключается в том, что не стоит платить за вещи, которые не используются [Мейерс2000-2]. Существуют языки и инструментальные средства, для которых время выполнения программ зависит от выбранного стиля. Если не использовать подобных знаний, то трудно написать эффективную программу. Кроме того, трансляторы могут по-разному отображать на память один и тот же программный объект.

Очищение от функционального наполнения

На мой взгляд, визуальное проектирование, крупноблочное программирование, использование каркасов и языков с развитой идиоматической составляющей больше затрагивает не технику кодирования, а функциональные аспекты написания программ. Вместо мелких кирпичей используются крупные блоки и панели. Но и эти элементы надо объединять в конструкции. И вот здесь вновь начинает использоваться базовая техника, являющаяся предметом дальнейшего рассмотрения.
Эта базовая техника также не зависит от особенностей среды исполнения. Большинство компилируемых языков отличаются от языков сценариев функциональным составом. Это обусловлено тем, что интерпретация программы должна компенсироваться наличием более мощных встроенных операций. Однако техника написания скриптов мало чем отличается от техники кодирования, используемой в традиционных языках. А языки сценариев тоже могут поддерживать как одну, так и несколько парадигм программирования.

Сведения о демонстрационном примере

Особенности различных парадигм программирования, по ходу изложения, будем рассматривать на примере очень простой программы, осуществляющей различные манипуляции с заданным набором геометрических фигур, хранимых в едином контейнере. Исходными данными **Задачи 1** будут следующие:

- Первоначальный комплект геометрических фигур состоит из треугольников и прямоугольников.
- Заданные фигуры можно создавать, определять их площадь, выводить имеющиеся значения в стандартный поток.
- В дальнейшем задача может развиваться по самым различным направлениям, каждое из которых будет нумероваться как отдельная подзадача путем добавления к исходному номеру задачи номера ответвления (как обычно, через точку).

Естественно, что простой пример не позволяет рассмотреть все особенности проектирования сложных программных систем. Однако он обеспечивает сравнение различных парадигм программирования при решении рассматриваемых задач проектирования. Кроме того, следует отметить, что приведенный код

не является образцом для подражания, так как в нем отсутствуют многие функции и проверки, необходимые в реально эксплуатируемой программе. Такие допущения объясняются учебным характером примера.

Код предполагается писать с использованием различных языков программирования, являющихся наиболее типичными при реализации рассматриваемых приемов. Там, где это специально не оговаривается, используется C++. Кроме реально существующих языков, по ходу изложения, будит вводиться гипотетические конструкции, которые, по моему мнению, могли бы использоваться в языках программирования. Однако, это ни в коей мере не предполагает, что я собираюсь тут же внедрять эти конструкции в уже существующие или вновь создаваемые языки.

Роль парадигм программирования

Парадигмы (стили) программирования занимают важное место в технологии разработки программного обеспечения. Именно вокруг них начинают выстраиваться и развиваться методологические концепции. Такая роль обуславливается тем, что возникающие новые идеи по созданию программ первоначально реализуются в простых инструментах, поддерживающих исследование и экспериментальную проверку выдвигаемого стиля. Чаще всего в качестве инструментов выступают языки программирования. Упомянутые исследования начинаются с написания простых программ. Лишь после обобщения первоначального опыта приходит понимание достоинств и недостатков, позволяющих перейти к формированию методологий, обеспечивающих использование парадигмы при разработке больших программных систем. Если разработанная парадигма не способна служить основой промышленной методологии, она отвергается или применяется в ограниченных масштабах.

В своей книге Гради Буч [Буч98], ссылаясь на Боброва и Стетика [Bobrow], приводит пять основных стилей программирования (табл. 1).

Таблица 1

Основные стили программирования

Название стиля	Основополагающие абстракции
Логико-ориентированный	Цели, часто выраженные в терминах исчисления предикатов
Ориентированный на правила	Правила "если - то"
Ориентированный на ограничения	Инвариантные отношения
Процедурно-ориентированный	Алгоритмы, абстрактные типы данных
Объектно-ориентированный	Классы и объекты

Для некоторых из представленных стилей можно провести дополнительную градацию. В частности, процедурно-ориентированный стиль содержит императивную и функциональную парадигмы программирования.

Императивное программирование [Хендерсон] базируется на основе автоматной модели вычислителя, разделяющей абстракции состояния и поведения. При этом программа рассматривается как процесс изменения состояния путем выполнения отдельных команд. Примерами таких вычислителей являются машина Тьюринга, фон-неймановская архитектура. Различные направления императивного программирования получили широкое развитие. В частности, из него выросло структурное программирование [Дал75]. Функциональное программирование [Хендерсон, Бердж] опирается на теорию рекурсивных функций [Барендрегт, Катленд]. Акцент делается на зависимость между функциями по данным. Модель состояний при этом практически игнорируется.

Перерастание парадигмы в методологию определяется различными факторами, среди которых можно выделить:

- эффективность реализации инструментальных средств, поддерживающих исследуемую парадигму;
- удобство в использовании на этапе проектирования;
- эффективна поддержка процесса разработки больших программных систем;
- генерация эффективного выходного представления;
- эффективное выполнение полученной программы.

Из стилей, представленных в таблице, только процедурно-ориентированный и объектно-ориентированный оказались пригодными для разработки больших программных систем, послужив стартовой площадкой для разработки соответствующих методологий. Такая ситуация возникла из-за ряда особенностей, присущих различным парадигмам. Большинство их, в конечном итоге, не смогли удовлетворить требованиям, предъявляемым к промышленным системам. Поэтому, использование многих стилей в настоящий момент ограничено научными исследованиями, быстрой разработкой прототипов, учебными задачами.

Общее в процедурном и объектно-ориентированном подходах

Близость процедурного и объектно-ориентированного подходов позволяют определить причины роста популярности первого по отношению ко второму. Различная распространенность парадигм программирования во многом обуславливается их способностями поддерживать современные методологии разработки программного обеспечения. Основной критерий в оценке программных продуктов - сложность [Буч98], а основными требованиями к методологиям разработки являются: удобство сопровождения, возможность безболезненного наращивания уже существующей программы, способность разработанных программных объектов к повторному использованию. При этом на второй план отступает такое требование, как быстрое проектирование первоначальной версии программы, потому что его воплощение обычно не позволяет соблюсти все остальные условия. Дело в том, что процесс разработки программного обеспечения не заканчивается выпуском одного релиза. Он сводится к итеративному расширению предыдущих версий, что, в некоторой степени, и помогает решать проблему сложности. Техника эволюционного развития реализована в возвратном проектировании [Буч98]. Она же используется при экстремальном программировании (X-programming), набирающем популярность в настоящее время [Beck, Бек].

В борьбе с проблемами, определяемыми сложностью программ, дальше всех продвинулась объектно-ориентированная методология (ООМ), которая и получила наибольшее распространение. В настоящее время она успешно развивается по самым разным направлениям, затрагивая как анализ и проектирование программных систем, так и написание самих программ. Последнее определяется как объектно-ориентированное программирование и связано с использованием соответствующих объектно-ориентированных языков. Развитие ООП практически вытеснило процедурное программирование из разработки сложных программных систем.

Такое достижение в первую очередь связано с методами группировки элементарных программных объектов в более крупные конструкции. Понятия, используемые в ООП, несколько отличаются от тех, что применяются в процедурном программировании. Это позволило по-новому подойти к процессу разработки и явилось ключом к повышению эффективности сопровождения и модификации программ.

Основные виды отношений между программными объектами

Отличие парадигм программирования заключается в реализациях моделей состояния и поведения, а также отношений между этими понятиями, осуществляемых через такие элементарные программные объекты как данные и операции. Абстрагирование от конкретных экземпляров достигается за счет введения понятий ***"абстрактный тип данных"*** и ***"процедура"*** (понятие ***"функция"*** используется как синоним процедуры). Элементарные понятия используются для построения составных программных объектов путем объединения в агрегаты и разделения по категориям. Категорию Г. Буч [Буч98] называет иерархией типа "is-a". Она также трактуется как ***обобщение*** программных объектов [Цикритзис]. ***Агрегаты*** и ***обобщения*** используются при конструировании композиций данных и процедур. В каждой из существующих парадигм программирования вопросы такого конструирования композиций решаются по-своему, что и вносит определенные отличительные черты.

Конструирование агрегатов

Агрегативные ассоциации

Агрегация (агрегирование) - это абстрагирование, посредством которого один объект конструируется из других [Цикритзис]. В связи с тем, что на самом нижнем уровне в программировании нами выделены две базовые абстракции (данные и процедуры), построение агрегатов может происходить следующими путями:

- агрегирование данных;
- агрегирование процедур;
- комбинированное агрегирование, обусловленное различными сочетаниями процедур и данных

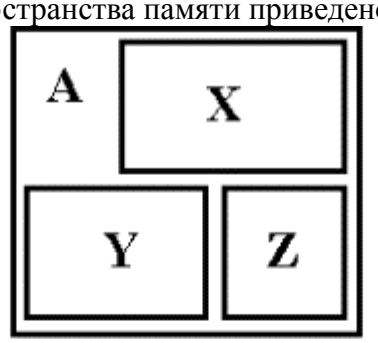
Необходимо отметить, что в данном случае под агрегированием подразумевается не просто размещение (вложение) одних абстракций в других. Оно рассматривается как конструирование новых программных объектов (данных, процедур и смешанных конструкций), из существующих. Следует отметить, что тело процедуры также можно рассматривать как агрегат, состоящий из команд и вызовов процедур. Но в данной работе этот вопрос обсуждаться не будет.

Следует понимать, что понятие "агрегат" является более узким, чем такие распространенные понятия, как модуль, пространство имен, класс. Оно определяет конструкцию, в которой основной акцент делается на композицию объектов, имеющих ресурсную привязку. Именно к таким объектам и относятся переменные с процедурами. Переменные размещаются в отведенном адресном пространстве и используются исключительно для хранения данных. Процедуры также должны храниться в памяти, хотя их использование более многогранно. Может осуществляться многократное тиражирование отдельных частей процедуры, таких как область локальных данных и тело. Это тиражирование производится во время вызовов процедур, а его характер определяется как архитектурой вычислительной системы, выполняющей программу, так и особенностями языка программирования высокого уровня. Например, вызов процедур в некоторых параллельных системах организован иначе, чем в последовательных компьютерах. Наряду с данными и процедурами общепринятые модульные конструкции используются для хранения АТД и других программных объектов, не занимающих ресурсов и используемых только во время компиляции программы. Тем самым обеспечивается локализация области видимости имен, что используется для борьбы со сложностью и сокрытия информации. Вместе с тем, размещение в традиционных модульных конструкциях объектов, располагаемых в памяти, позволяет говорить о них как об агрегатах, игнорируя при этом прочие понятия.

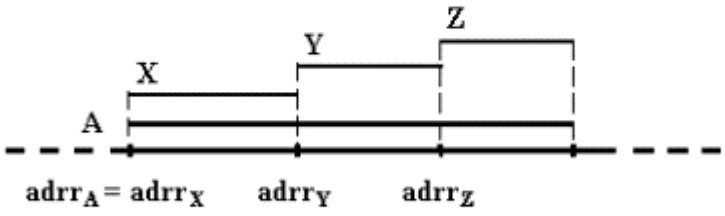
Методы агрегирования

Агрегирование обеспечивает формирование программных объектов одним из способов: непосредственным включением, косвенным (ссылочным) связыванием, с применением наследования (расширения) и образного восприятия.

Наиболее типичным является восприятие агрегата как единой абстракции, сформированной **непосредственным включением** используемых в нем программных объектов. В нашем сознании он видится как единый, монолитный ресурс, занимающий некоторое неразрывное пространство (почему-то в моем сознании чаще всего возникает плоскость:). На рис. 1а агрегат, сформированный непосредственным включением элементов отображен в двухмерном пространстве. Отображение такого агрегата на фрагмент одномерного пространства памяти приведено на рис. 1б.



а) условное изображение на плоскости

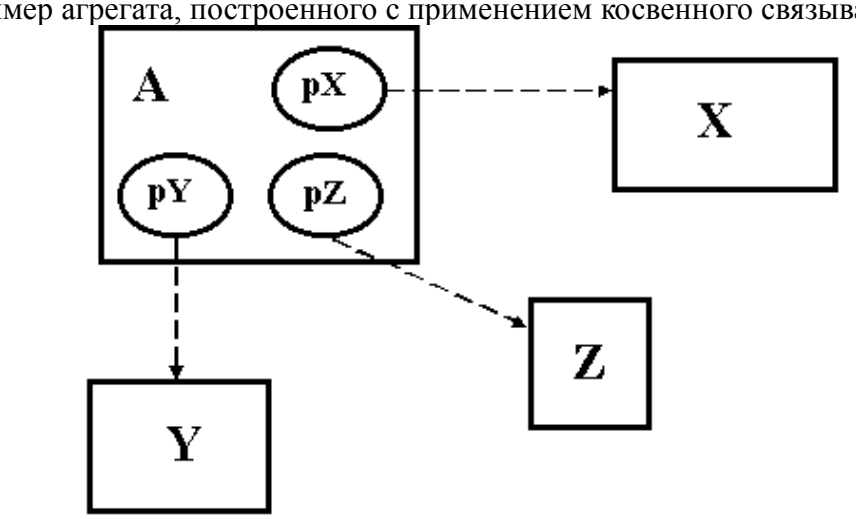


б) отображение на одномерное адресное пространство

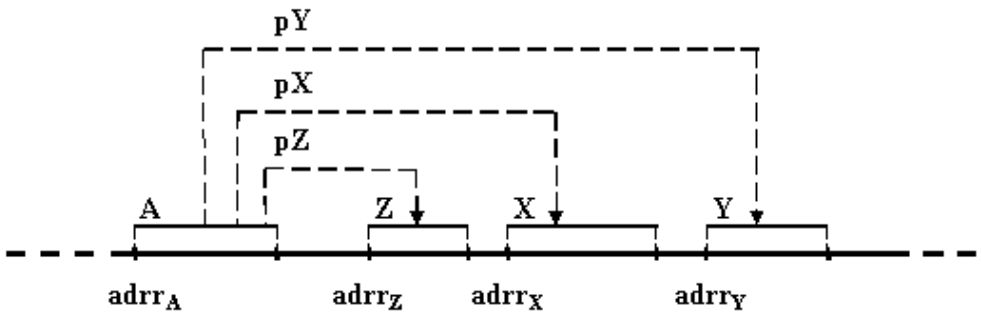
Рис. 1. Агрегат A из элементов X, Y, Z, построенный с использованием непосредственного включения.

Цельность и законченность данного объекта не требует выполнения дополнительных алгоритмов, связанных с формированием его структуры. Можно сразу приступать к алгоритмическому использованию объекта, например, его инициализации.

Косвенное связывание позволяет формировать агрегаты из отдельных элементов, уже располагаемых в пространстве. Взаимосвязь осуществляется алгоритмическим заданием значений ссылок. Спецификой является необходимость выполнения кода обеспечивающего конструирование объекта из разрозненных экземпляров абстракций. Однако этот код выполняется только один раз, после чего работа с агрегатом осуществляется точно так же, как и в предыдущем случае. В ряде случаев, когда элементы и агрегат создаются статически, инициализация связей может быть проведена во время трансляции программы. Образный пример агрегата, построенного с применением косвенного связывания, представлен на рис. 2.



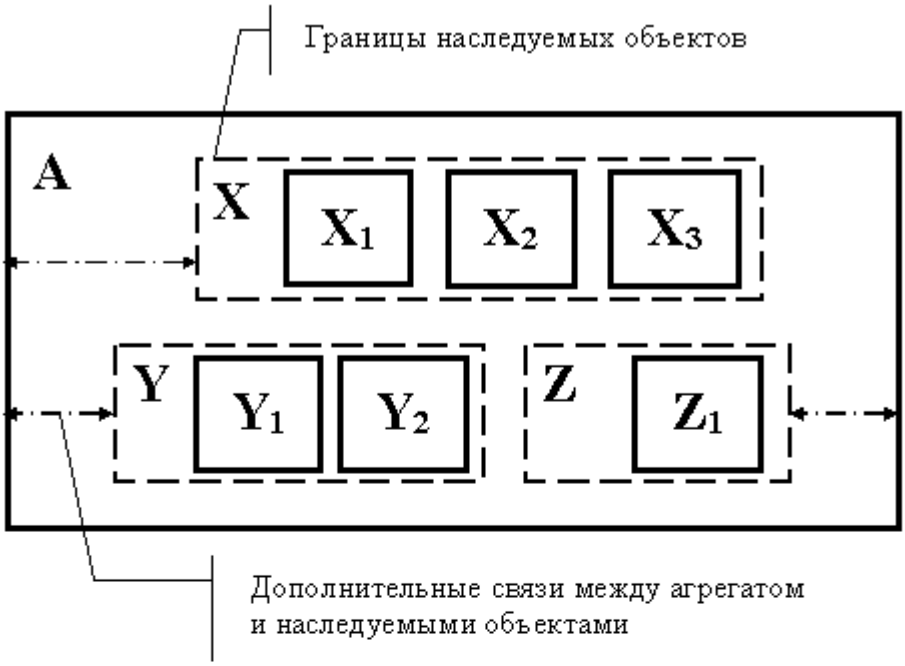
а) условное изображение на плоскости



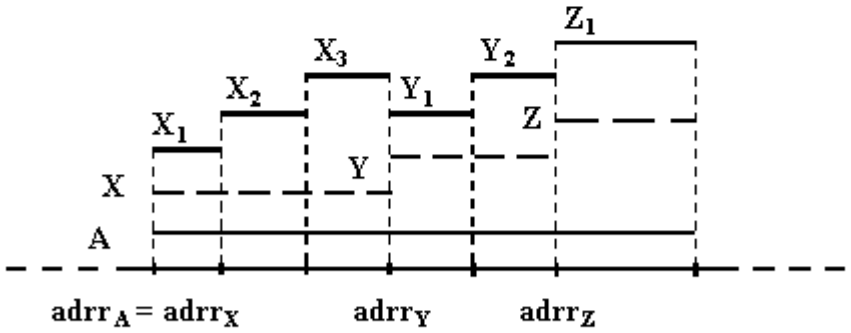
б) отображение на одномерное адресное пространство

Рис. 2. Агрегат A из элементов X, Y, Z, построенный с использованием косвенного связывания.

Применение **наследования** позволяет создавать структуру объекта, эквивалентную той, которая формируется непосредственным включением. Однако наследование дополнительно подерживает свойства конкатенации (слияния), в результате чего обращение к элементам подключаемых абстракций осуществляется напрямую, не затрагивая имена добавляемых абстракций. При этом, в отличие от конкатенации, сохраняется информация о базовых абстракциях, которая может использоваться для разрешения проблем неоднозначности доступа к элементам сформированного агрегата при совпадении их имен. Кроме того, в отличие от непосредственного включения, каждому наследуемому элементу "предоставляется" дополнительная информация о формируемом агрегате. Эта информация может быть "задана" специальными методами построения агрегата или за счет введения дополнительных внутренних переменных, инициализируемых во время конструирования объекта. Использование дополнительной информации позволяет правильно манипулировать агрегатом, используя лишь сведения об одном из его базовых (включаемых) элементов. Например, через базовый элемент можно определить тип агрегата, его размер, выполнить операцию удаления. Упрощенное графическое представление агрегата, построенного с использованием наследования, приведено на рис. 3.



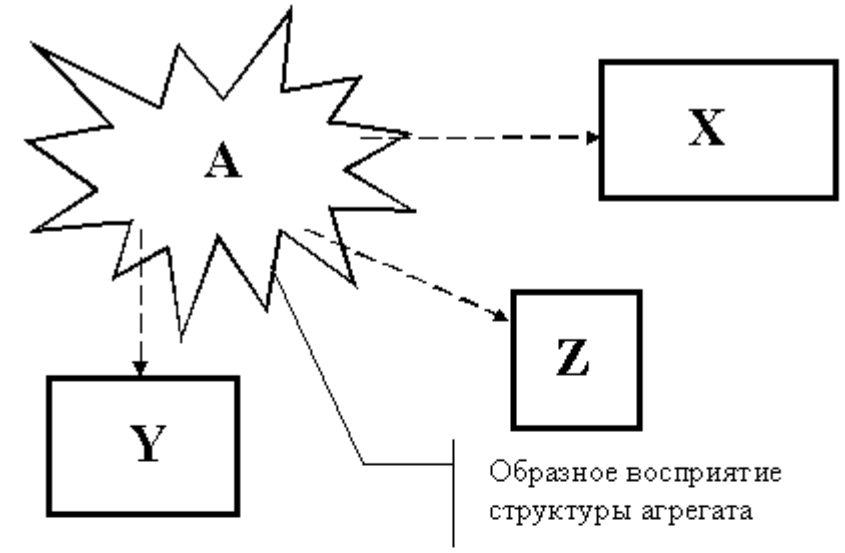
а) условное изображение на плоскости



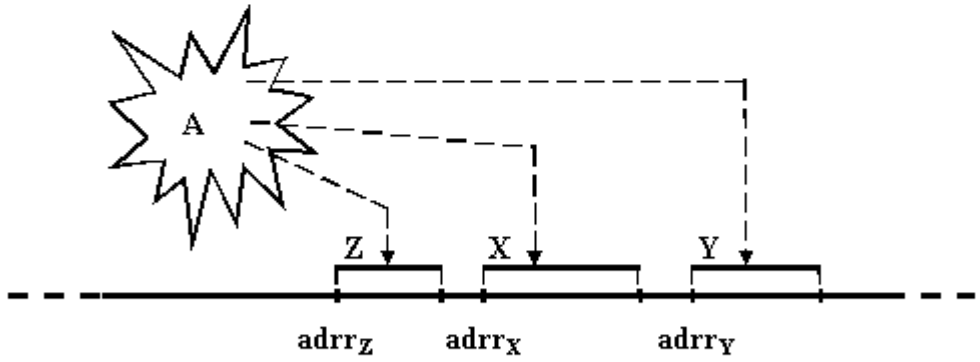
б) отображение на одномерное адресное пространство

Рис. 3. Агрегат A из элементов X, Y, Z, построенный с использованием наследования.

Образное агрегирование связано с отсутствием специально созданной абстракции, соответствующей формируемому агрегату. Вместо этого агрегат воссоздается только в мысленном восприятии программиста, а на уровне программы имеются его отдельные элементы, обрабатываемых как единое целое. Например, точку на плоскости можно представить как две независимые целочисленные переменные x и y. Конечно, такое агрегирование уходит корнями в далекое прошлое (эпоху Фортрана и Алгола-60), но и сейчас встречаются программисты, которым "лень" вписать лишнюю абстракцию. Это приводит к определенным проблемам, связанным с мобильностью и повторным использованием кода, но иногда так хочется поскорее написать программу, что не остается времени на раздумья о стиле! Графическая интерпретация образного агрегата приведена на рис. 4.



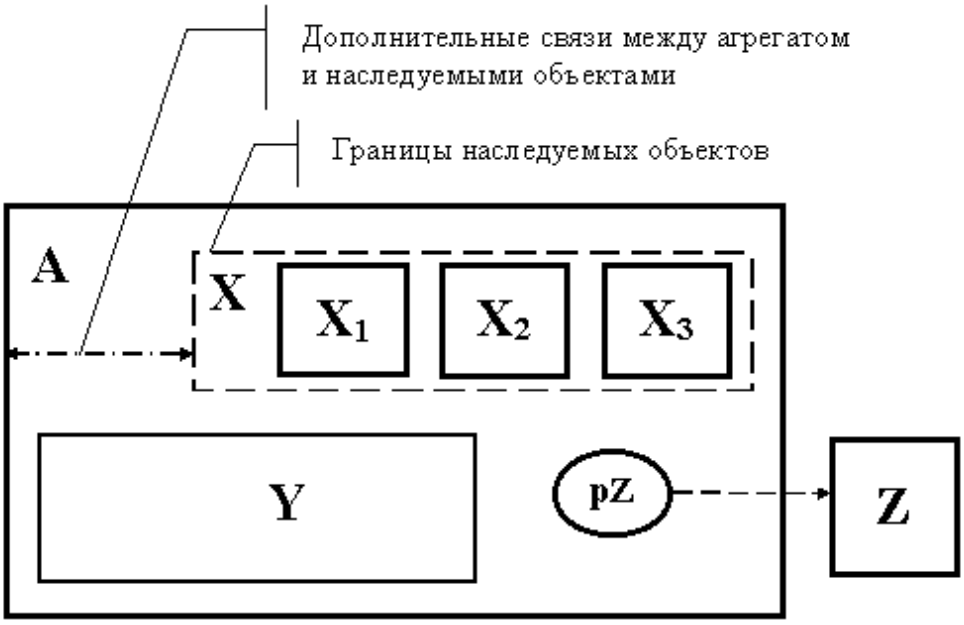
а) условное изображение на плоскости



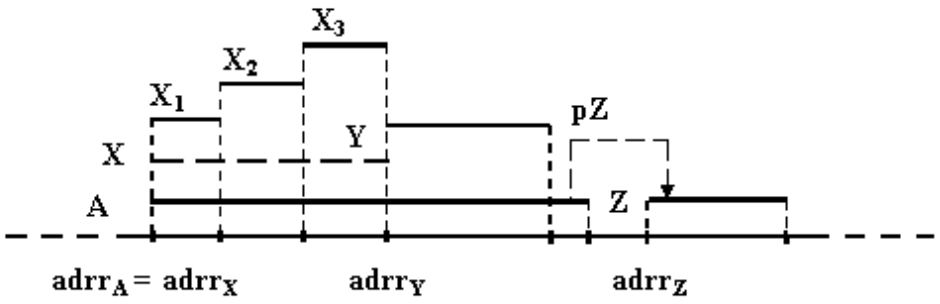
б) отображение на одномерное адресное пространство

Рис. 4. Агрегат A из элементов X, Y, Z, построенный на основе образного восприятия.

Вполне очевидно, что, наряду с "чистыми" методами, агрегаты могут строиться по смешанному принципу, когда одновременно применяются различные комбинации методов агрегирования. В качестве примера на рис. 5 представлен агрегат, выстроенный с применением включения, косвенного связывания и наследования.



а) условное изображение на плоскости



б) отображение на одномерное адресное пространство

Рис. 5. Агрегат A из элементов X, Y, Z, построенный с использованием наследования, непосредственного включения и косвенного связывания.

Именно такой подход и является наиболее популярным в настоящее время, так как позволяет рационально использовать различную технику в зависимости от организации и назначения агрегируемых элементов. Можно, конечно, подключать и образное восприятие, но мне кажется, что в подавляющем большинстве случаев проще сформировать еще одну абстракцию, зафиксировав, тем самым, используемое решение "документально".

Возможно, у Вас возник вполне закономерный вопрос: от чего я впал в маразм и подробно привел прописные истины? А чтобы подчеркнуть, что различные способы организации программных объектов отображаются в виде определенных стереотипов в наших затуманенных мозгах, что во многом и определяет приверженность выбранному стилю программирования.

К специфике различных парадигм программирования можно отнести способы построения агрегатов и их использования. В частности, при процедурном программировании осуществляется разделение на агрегаты данных и процедур. Объектно-ориентированный подход использует комбинированное агрегирование. Ниже рассмотрены особенности построения и использования агрегатов для этих парадигм.

Процедурное агрегирование

Построение агрегатов при процедурном подходе осуществляется с использованием следующих понятий:

- ***Агрегатам данных*** соответствуют абстрактные типы данных, представляемые, например, структурами в языках С, С++ и обычными записями в языках Паскаль и Модула. Они состоят из объектов данных, подключаемых непосредственно или с использованием ссылок (указателей). В дальнейшем, при обсуждении агрегатов данных, используемых при процедурном подходе, будем использовать понятие "***запись***". Понятие "***структура***", для обозначения агрегата данных не используется в связи с его перегруженностью другими смысловыми значениями.
- Процедурный подход предполагает независимость записей (**R**) от процедур (**P**). Запись **R** состоит из элементов данных: $\mathbf{R=(D_1, D_2, ..., D_k)}$, называемых ***полями записи***. При этом поле $\mathbf{D_i}$ может, в свою очередь, состоять из других абстракций.
- ***Агрегатам процедур***, осуществляющим обработку агрегатов данных, соответствуют вложения в тела процедур различных по иерархии объектов действия: операций, операторов, вызовов процедур, осуществляющих обработку отдельных элементов (полей) записи. Обозначим данную абстракцию понятием "***независимая процедура***". Доступ к различным экземплярам записи, определяющим агрегируемую абстракцию, осуществляется через один из элементов списка формальных параметров.

Процедура используют запись в качестве элемента списка параметров, обеспечивая тем самым связь с различными экземплярами и их обработку. Элементы записи, в свою очередь, могут относиться как к базовым данным, так и иметь более сложную организацию, то есть быть агрегатами или обобщениями. Обработка записей независимыми процедурами осуществляется после передачи входных параметров через сигнатуру в тело процедуры. Тело независимой процедуры представляет агрегат, в котором запись расщепляется на отдельные элементы, каждый из которых обрабатывается своей независимой процедурой.

Пример использования процедурного агрегирования

Используем агрегирование при создании простого контейнера геометрических фигур для процедурной версии программы. Абстрактный тип данных, определяющий элементы контейнера можно задать следующим (не единственным) образом:

```
//-----------------------------------------------------
// Простейший контейнер на основе одномерного массива
//-----------------------------------------------------
// Контейнер должен знать о фигуре
#include "shape_atd.h"
//-----------------------------------------------------
// Данные контейнера
struct container
{
   enum {max_len = 100}; // максимальная длина
   int len; // текущая длина
   shape *cont[max_len];
};
//-----------------------------------------------------
```

В контейнере хранятся различные геометрические фигуры, определяемые обобщением ***shape***, которое должно быть предварительно определено. Внутренняя организация АТД ***shape*** роли не играет, так как она доступна через сигнатуры функций, обрабатывающих обобщение. Этот прием демонстрирует возможность сокрытия данных от пользователя и в процедурном подходе. К независимым процедурам, осуществляющим полную обработку элементов представленного агрегата данных, можно отнести:

- процедуру инициализации контейнера ***void Init(container &c)***, заключающуюся в установке его начального состояния (просто обнуляется количество элементов);
- процедуру утилизации ***void Clear(container &c)***, осуществляющую удаление фигур, размещенных в контейнере и установку его в начальное состояние;

- процедуру ввода геометрических фигур в контейнер из входного потока ***void In(container &c)***;
- процедуру вывода содержимого контейнера ***void Out(container &c)***, предоставляющую полную информацию об его текущем состоянии;
- процедуру вычисления суммарной площади геометрических фигур ***double Area(container &c)***, хранимых в контейнере.

Вполне естественно, что часть процедур осуществляет обработку всех элементов контейнера. Однако, существуют и процедуры, обрабатывающие лишь отдельные элементы, что обусловлено решаемой ими задачей. Но и в этом случае изменение одного элемента контейнера трактуется как изменение агрегата в целом. Ниже приведена реализация указанных независимых процедур.

```
//--------------------------------------------------------------------
// Процедуры должны знать о контейнере и фигуре,
// доступной через модуль, описывающий контейнер
#include "container_atd.h"
//--------------------------------------------------------------------
// Инициализация контейнера
void Init(container &c)
{
   c.len = 0;
}
//--------------------------------------------------------------------
// Очистка контейнера от элементов (освобождение памяти)
void Clear(container &c)
{
   for(int i = 0; i < c.len; i++)
   {
    delete c.cont[i];
   }
   c.len = 0;
}
//--------------------------------------------------------------------
// Необходим прототип функции, формирующей фигуру при вводе
shape *In();
//--------------------------------------------------------------------
// Ввод содержимого контейнера
void In(container &c) {
   cout
    << "Do you want to input next shape"
       " (yes=\'y\', no=other character)? "
    << endl;
   char k;
   cin >> k;
   while(k == 'y')
   {
    cout << c.len << ": ";
    if((c.cont[c.len] = In()) != 0)
    {
     c.len++;
    }
    cout
     << "Do you want to input next shape"
        " (yes=\'y\', no=other character)? "
     << endl;
    cin >> k;
   }
}
//--------------------------------------------------------------------
```

```
// Необходим прототип функции вывода отдельной фигуры
void Out(shape &s);
//---------------------------------------------------------------------
// Вывод содержимого контейнера
void Out(container &c)
{
   cout << "Container contents " << c.len << " elements." << endl;
   for(int i = 0; i < c.len; i++)
   {
     cout << i << ": ";
     Out(*(c.cont[i]));
   }
}
//---------------------------------------------------------------------
// Необходим прототип функции, вычисляющей площадь отдельной фигуры
double Area(shape &s);
//---------------------------------------------------------------------
// Вычисление суммарной площади для фигур, размещенных в контейнере
double Area(container &c)
{
   double a = 0;
   for(int i = 0; i < c.len; i++)
   {
     a += Area(*(c.cont[i]));
   }
   return a;
}
//---------------------------------------------------------------------
```

Указанные функции использованы в примере, расположенном в архиве pp_examp1.zip.

Объектно-ориентированное агрегирование

Объектно-ориентированное программирование предлагает следующие варианты композиции для создания агрегатов:

- Основной агрегирующей единицей является виртуальная или реальная ***оболочка*** **(С)**, Реально существующая оболочка, вместе с размещенными в ней программными объектами, чаще всего называется ***классом***. В нем могут быть размещены как данные, так и процедуры. Оболочка в явном виде может не присутствовать в языке и проявляться через связывание процедур с типами данных, как в языке программирования Оберон-2 [MoessenboeckWirth]. Авторы, для обозначения программных объектов, даже используют терминологию, принятую в процедурном программировании. Но при этом подчеркивается объектно-ориентированная направленность механизма связывания процедур. В языке С++ термин "класс" может быть заменен "структурой" [Страуструп], в Паскале и Delphi - "объектом" [Гофман]. Однако, понятие "***класс***" является наиболее распространенным. Поэтому, в дальнейшем будем использовать именно его.

- Обычные процедуры, размещаемые в классе и используемые для изменения своего внутреннего состояния, часто называются методами, функциями-членами класса. Будем использовать термин "***процедура класса***". Виртуальные процедуры, переопределяемые в производных классах, будут рассмотрены ниже как составляющие обобщений.
- Данные, определяющие внутреннее состояние класса, обычно называются ***переменными класса***.
- Термин "***оболочка класса***" (или просто "***оболочка***", если понятен контекст) будем использовать для обозначения класса в том случае, когда хотим исключить из рассмотрения его переменные и процедуры. Оболочка, при доступе извне, выступает в роли посредника к программным объектам, расположенным внутри класса.

Следует также отметить, что разная терминология в ОО подходе усугубляется также различными способами организации классов. В языке программирования C++ тело обычной и виртуальной процедуры может располагаться как внутри оболочки, содержащей также переменные класса (внутреннее описание процедуры класса), так и вне нее. В последнем случае она представляет внешнее описание процедуры, которому должно предшествовать ее объявление в оболочке.
В языке программирования Java, а также в ряде других языков, процедура всегда располагается внутри оболочки. В ней же размещаются и переменные класса.
В языке программирования Оберон-2 виртуальная процедура класса всегда размещается вне физической оболочки, составляя, однако, со связанным типом данных единое целое, подчиняющееся законам полиморфизма и наследования. При этом сохраняется терминология присущая процедурному подходу. Такая относительность в целом ведет к одинаковому набору связей между объектами класса, но разным образом сказывается на их размещении в тексте программы. Это позволяет изменять связанную процедуру, не изменяя тип данных, к которому она привязана. Но изменения затрагивают модуль, в котором находятся АТД и связанные с ним процедуры. А так как типы данных и связанные процедуры должны находиться в одном модуле, то его можно считать общей физической оболочкой класса для всех размещаемых в нем АТД и связанных с ними процедур. В других же языках программирования изменения, связанные с модификацией данных или процедур класса, происходят внутри своей собственной оболочки. В целом же можно считать, что изменения в классе, как совокупности оболочки, переменных и процедур, происходят в любом случае.
Следует отметить, что процедуры класса осуществляют непосредственный доступ к его данным (через тела процедур), что ведет к фиксации используемого адресного пространства. Однако каждый из порождаемых экземпляров класса имеет дополнительно свою адресную привязку, относительно которой и располагаются адреса используемые в телах процедур. Создается иллюзия наличия для каждого экземпляра класса своего экземпляра внутренних процедур, хотя реальная реализация обеспечивает обслуживания всех экземпляров одним комплектом процедур [Голуб] через использование на системном уровне механизма параметризации доступа к членам класса. Независимо от трактовки данного способа доступа, он, по своим параметрам, эквивалентен доступу к телу процедуры через сигнатуру.

Пример использования объектно-ориентированного агрегирования

Контейнер для хранения различных геометрических фигур, реализованный с применением объектно-ориентированного подхода, содержит не только данные, но и процедуры их обработки. Описание, определяющее интерфейс класса ***container***, доступный пользователям, выглядит следующим образом:

```
//-------------------------------------------------------------------
// Простейший контейнер на основе одномерного массива
//-------------------------------------------------------------------
// Контейнер должен знать о фигуре
#include "shape_atd.h"
//-------------------------------------------------------------------
// Описание контейнера
class container
{
   enum {max_len = 100}; // максимальная длина
   int len; // текущая длина
   shape *cont[max_len];
public:
   void In();     // ввод фигур в контейнер из входного потока
   void Out();    // вывод фигур в выходного потока
   double Area(); // подсчет суммарной площади
   void Clear();  // очистка контейнера от фигур
   container();   // инициализация контейнера
   ~container() {Clear();} // утилизация контейнера перед уничтожением
};
//-------------------------------------------------------------------
```

Как и в случае с процедурной программой, интерфейс с внешним миром осуществляется через аналогичные же процедуры, но уже являющиеся методами класса:

- инициализация класса поддерживается конструктором ***container()***;
- для очистки контейнер от содержимого без уничтожения используется процедура ***void Clear()***;
- при уничтожении контейнера в деструкторе ***~container()***, тело которой размещено непосредственно в классе, процедура ***Clear()*** используется для предварительного освобождения памяти, занимаемой элементами контейнера;
- ввод данных из входного потока обеспечивается процедурой ***void In()***;
- вывод информации о фигурах, расположенных в контейнер производится методом ***void Out()***;
- вычисление суммарной площади геометрических фигур, хранимых в контейнере, осуществляется процедурой класса ***double Area()***.

Большинство методов класса отличаются от независимых процедур только отсутствием в сигнатуре входного параметра, обеспечивающего доступ к данным в процедурном подходе. Однако дополнительные языковые возможности C++ позволяют более гибко реализовать инициализацию и утилизацию с применением конструктора и деструктора, что позволяет убрать дополнительные вызовы процедур в самой программе.

При разработке больших программных проектов, содержащих объемные процедуры, тела процедур - членов класса, обычно размещаются (если язык предоставляет такую возможность) в других единицах компиляции. Несмотря на то, что наш проект мал, да и тела процедур ростом не вышли, будем использовать этот же принцип. Это позволит в дальнейшем более гибко имитировать эволюционное развитие программы. Реализация процедур контейнера выполнена следующим образом:

```
//---------------------------------------------------------------------
// Необходимо знать описание контейнера и методов фигуры,
// доступных через container_atd.h
#include "container_atd.h"
//---------------------------------------------------------------------
// Прототип обычной внешней функции, формирующей фигуру при вводе
shape *In();
//---------------------------------------------------------------------
// Инициализация контейнера
container::container(): len(0) { }
//---------------------------------------------------------------------
// Очистка контейнера от элементов (освобождение памяти)
void container::Clear()
{
   for(int i = 0; i < len; i++)
   {
    delete cont[i];
   }
   len = 0;
}
//---------------------------------------------------------------------
// Ввод содержимого контейнера
void container::In()
{
   cout
    << "Do you want to input next shape"
       " (yes=\'y\', no=other character)? "
    << endl;
   char k;
   cin >> k;
   while(k == 'y')
   {
    cout << len << ": ";
    if((cont[len] = simple_shapes::In()) != 0)
    {
     len++;
```

```
      }
      cout
        << "Do you want to input next shape"
           " (yes=\'y\', no=other character)? "
        << endl;
      cin >> k;
    }
}
//---------------------------------------------------------------------
// Вывод содержимого контейнера
void container::Out()
{
   cout << "Container contents " << len << " elements." << endl;
   for(int i = 0; i < len; i++) {
     cout << i << ": ";
     cont[i]->Out();
   }
}
//---------------------------------------------------------------------
// Вычисление суммарной площади для фигур, размещенных в контейнере
double container::Area()
{
   double a = 0;
   for(int i = 0; i < len; i++) {
     a += cont[i]->Area();
   }
   return a;
}
//---------------------------------------------------------------------
```

Указанные функции использованы в примере, расположенном в архиве oop_examp1.zip.

Отличие методов агрегирования

Возникает вполне резонный вопрос: чем отличаются друг от друга процедурное и объектно-ориентированное агрегирование? Очень многим, если следовать традиционному восприятию стереотипов. Основным отличием является то, что ОО агрегирование выстраивает в голове программиста более устойчивую ассоциацию объекта, поддержанную непосредственным размещением процедур внутри классов. Дополнительный эффект проявляется в том, что методы класса имеют на один параметр меньше, так как абстрактный характер переменных класса проявляется через их размещение в абстрактной оболочке. Практически все приверженцы ООП утверждают адекватность моделей реального мира, используемых для отображения в программные объекты, структурам, описываемым классами. Однако, этот тезис является весьма спорным. Существует множество реальных систем, более эффективные модели которых не определяются через отношения объектов, а возможность их отображения с помощью объектов связана просто с гибкостью средств программирования, позволяющих с легкостью переделывать одни понятия в другие, сохраняя при этом тем же самым результат работы: программу, функционирующую должным образом. Программирование - это философия наоборот. Философы различным образом объясняют единый мир - программисты различным образом приходят к одному и тому же результату.

Однако, наряду с отображаемыми моделями, разработчики, занимающиеся созданием сложных эволюционирующих программ, держат в своей голове множество других ассоциаций, связанных, например, с возможностями дальнейшего расширения кода без изменения уже написанных фрагментов. А обеспечить нормальную реализацию таких ассоциаций невозможно, если воспринимать моделируемые объекты только физиологически. Разделение классов на данные и процедуры как раз и позволяет вернуть гибкость в реализации. Да и вряд ли у более-менее опытного программиста возникнут проблемы с восприятием объекта реализованного без помощи класса.

Приведу простой пример. Предположим, нами разработан класс:

```
class simple {
```

```
  int v;
public:
  simple(int val);
  void out();
};
```

Лично я не думаю, что его процедурная реализация менее понятна для восприятия:

```
struct simple {
  int v;
};
simple *create_simple (int val);
void destroy_simple(simple* s);
void out (simple &s);
```

Вместе с тем, отделение процедур от данных позволяет скрыть от клиента информацию о внутренней организации структуры данных и процедур ее обработки, не прибегая при этом к каким-либо специальным ухищрениям. А предоставляемое клиенту описание объекта взаимодействия определяет только интерфейс и не забивает его голову дополнительными деталями (о сокрытии информации я сейчас говорить не хочу, так как это, на мой взгляд, тема отдельного разговора):

```
struct simple;
simple *create_simple (int val);
void destroy_simple(simple* s);
void out (simple &s);
```

Как видите, Скотт Мейерс не зря говорит о гибкости внешних функций [Meyers] и об их способности улучшить инкапсуляцию. Следуя его же рекомендациям (весьма часто и без этого используемым в процедурном подходе, особенно тогда, когда не хочется к каждой единице компиляции подключать объемный заголовочный файл), мы можем создать несколько подмножеств аналогичных интерфейсов в различных модулях (единицах компиляции). Например:

```
struct simple;
simple *create_simple (int val);
void destroy_simple(simple* s);
```

Или:

```
struct simple;
void out (simple &s);
```

И так далее. Исходные тексты этих примеров находятся в архиве simple.zip.

Естественно, что возникает ряд возражений. Одно из них связано с тем, что приходится вводить дополнительные функции создания и разрушения объектов вместо использования конструктора, обеспечивающего более наглядную запись. Однако, это, на мой взгляд, связано с издержками существующих процедурных языков. Никто не пытался перепроектировать их таким образом, чтобы создание экземпляра абстрактного типа данных ассоциировалось с операциями преобразования типа и его разрушения. Мне кажется, что не составит особого труда найти соответствующее техническое решение. При программировании на C++ можно попытаться использовать перегрузку оператора приведения типа. В конце концов, можно использовать в структуре конструкторы и деструкторы, а все остальные процедуры вынести за класс.

Другим слабым аргументом в пользу внешних процедур является то, что и в объектных моделях применяют аналогичные приемы для отделения создаваемого класса от метода его создания. В качестве примера следует привести образцы "Абстрактная фабрика" и "Фабричный метод" [Гамма]. Для того, чтобы возвращать экземпляры различных классов, в них используется дополнительная обертка над конструкторами, выполненная в виде процедур создания объектов, являющихся, правда, виртуальными методами классов.

Итак, процедурное и ОО агрегирование практически ничем не отличаются между собой, если не считать за отличие размещение процедур. Между тем, процедурное агрегирование обеспечивает более гибкую реализацию ассоциаций, связанных с конструированием эволюционирующих программ.

Иллюзии агрегирования

Обсуждая долго и нудно альтернативные методы агрегирования, нельзя не отметить, что мы говорим о двух различных способах записи, которые можно применять к одной и той же конечной реализации. То есть, наши разные внешние ассоциации зачастую несут одну и ту же конечную смысловую

интерпретацию. Посмотрите на то, как, в конце концов, реализуется после трансляции невиртуальный метод класса [Голуб]. Это обычная внешняя процедура, использующая структуру данных, размещенных в классе, в качестве аргумента this! А раз все в мире так относительно, то мы можем немного еще поиграть с внешними ассоциациями, чтобы процедурное стало выглядеть как объектное.

Я стараюсь поддерживать переписку с рядом своих бывших студентов, разбросанных по всему Шару и продолжающих серьезно заниматься программированием, чтобы таким образом получать дополнительную информацию (надеюсь, что когда они встанут на ноги, то возьмут меня к себе сторожем или уборщицей:). В свое время Алексей Гуртовой, весьма серьезно занимающийся ООП и использующий его на практике в MetaCommunications, сообщил мне о статье Скотта Мейерса (которую я и перевел), а также следующей идее расширения C++, промелькнувшей в news:comp.lang.c++.moderated. При вызове внешних функций, принимающих ссылку на некоторый класс в качестве своего первого аргумента, предлагается указывать экземпляр класса не в виде параметра, а как префикс, предшествующий вызову функции. Вот дословный текст цитаты, присланной им.

Post comp.lang.c++.moderated Andrei Alexandrescu

What I think Scott's article should bring into discussion, is a future alternate syntax for uniformizing member and nonmember function calls. If Scott's article is well understood and its consequences taken seriously by the C++ community, maybe the language could allow member call syntax for nonmembers.

For instance, a nice rule would be that a nonmember that takes a reference to an object of type T as the first parameter could be invoked using member syntax, like this:

```
class A { ... };
void Fun(A& obj, int x, double y);
A a;
Fun(a, 6, 7.8); // classic call syntax
a.Fun(6, 7.8);  // alternate (new) syntax
```

Если обратиться к ранее написанному простому примеру, то этот механизм можно проиллюстрировать следующим образом:

```
struct simple {
  int v;
};
void out (simple& s) {
  cout << "value = " << s.v << endl;
}
simple s;
...
out(s);  // обычный вызов функции вывода
s.out(); // новый (альтернативный) вариант
```

Итак, все в мире относительно, что подтверждается и развитием ряда других языков. Этот же прием был использован в 1996 году для расширения языка Оберон. Так в нем появились связанные процедуры, расширившие механизм наследования виртуализацией и обеспечившие использование ООП в ранее процедурном языке. При этом Вирт и Мессенбек полностью сохранили при описании языка программирования Оберон-2 процедурную терминологию [MoessenboeckWirth]. Но и в языке, похожем на C++, можно пойти дальше того, чтобы использовать образную ассоциацию внешней функции для ее явного ОО вызова. Можно, вместо образной ассоциации, ввести конкретную синтаксическую форму, похожую на представление методов класса. Для этого достаточно "вытащить" соответствующий формальный параметр из скобок и прописать вместо него префикс класса перед именем функции. Вот, как это может выглядеть на предыдущем простом примере:

```
struct simple {
  int v;
};
extern void simple::out()
{
  cout << "value = " << v << endl;
}
simple s;
...
s.out(); // только новый вариант вызова
```

Получается, что любая внешняя функция может прикинуться методом класса, не являясь, на самом деле, таковой. То, что это внешняя функция, а не метод, определяется ключевым словом extern. Увидев его, транслятор не станет "кричать" о том, что Вы забыли указать в классе этот метод. Можно даже добавить ключевое слово static, которое локализует использование данной функции текущей единицей компиляции:

```
struct simple {
  int v;
};
// ограничение использования текущим модулем
extern static void simple::out()
{
  cout << "value = " << v << endl;
}
simple s;
...
s.out(); // только в текущем модуле
```

Хотелось бы еще раз напомнить, что я не занимаюсь ревизией C++. Приводимые примеры, скорее всего, являются намеками для тех, кто занимается разработкой собственных языков. Переделкой стандартов пусть занимаются соответствующие комитеты. Поэтому, вполне допустимо, что в предлагаемом синтаксисе существуют противоречия с C++. Основное, что хотелось бы показать в этой работе, это возможность использования других конструкций. Хотя, я думаю, что большинству C++ программистов все равно, как писать внешнюю функцию и с чем ее образно ассоциировать.

В заключении хочу еще раз отметить, что, по моему разумению, объектно-ориентированное агрегирование ничего не дает, по сравнению с обычным процедурным агрегированием. Более того, фиксация процедур в классах уменьшает гибкость при разработке эволюционирующей программы. Наличие же в языке различных методов создания программных объектов (с добавлением к уже существующим способам представления тех, которые были описаны выше) ведет к разбуханию языка программирования. И здесь я делаю первый намек на то, что от языков, отображающих одни и те же понятия разными способами, надо переходить к инструментам, позволяющим отображать множественные отношения и ассоциации между базовыми понятиями. Захотели, посмотрели на программные объекты как на процедуры и данные, захотели - посмотрели как на классы. А разработку таких понятий также можно вести с любых позиций и парадигм.

Конструирование обобщений

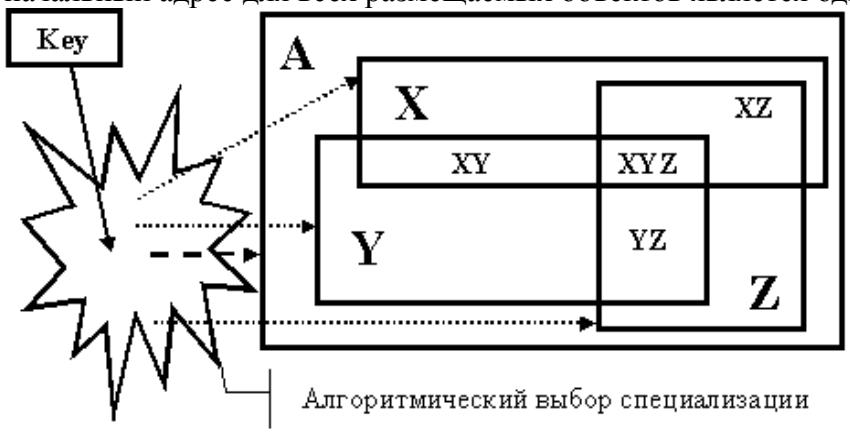
Обобщение - это композиция альтернативных по своим свойствам программных объектов, принадлежащих к единой категории в некоторой системе классификации. Оно включает ***обобщение данных*** и ***обобщение процедур (процедурное обобщение)***. Обобщение данных состоит из ***основы обобщения***, к которой присоединяются различные ***специализации***. ***Специализации обобщения*** - это отдельные альтернативные понятия, принадлежащие к единой категории. В общем случае, специализации могут тоже являться обобщениями, определяя, таким образом, ***многоуровневые обобщения***. Многоуровневые обобщения могут быть иерархическими и рекурсивными. Обобщение является ***одноуровневым***, если оно рассматривается как конструкция, состоящая из основы обобщения и его специализаций.

Обработка обобщений данных осуществляется соответствующими процедурами, включаемыми в состав процедурного обобщения. Процедура, связанная с обработкой основы обобщения называется ***обобщающей процедурой***. Процедура, осуществляющая обработку отдельной специализации обобщения, называется ***специализированной***.

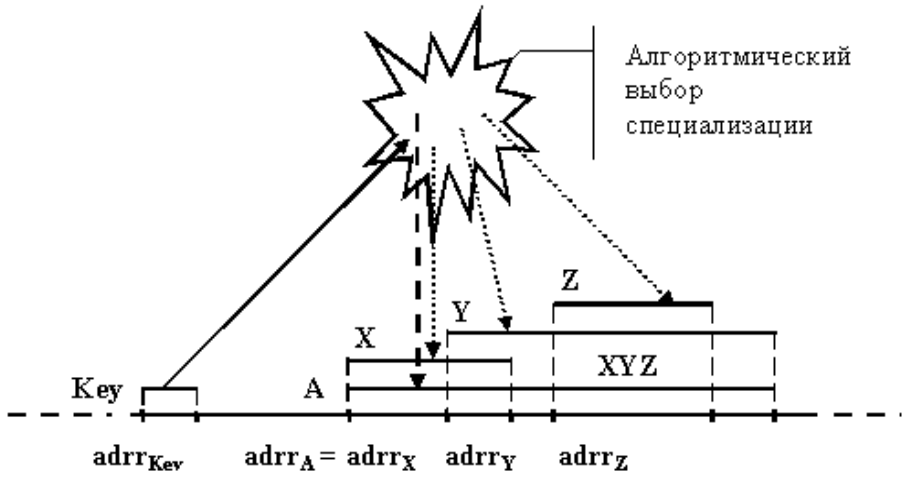
Методы формирования обобщений

Существуют различные методы построения обобщений: объединение абстракций на основе общего ресурса; использование альтернативного связывания; образное обобщение. Каждый из методов обобщения, как и агрегирования, имеет свои специфические особенности.

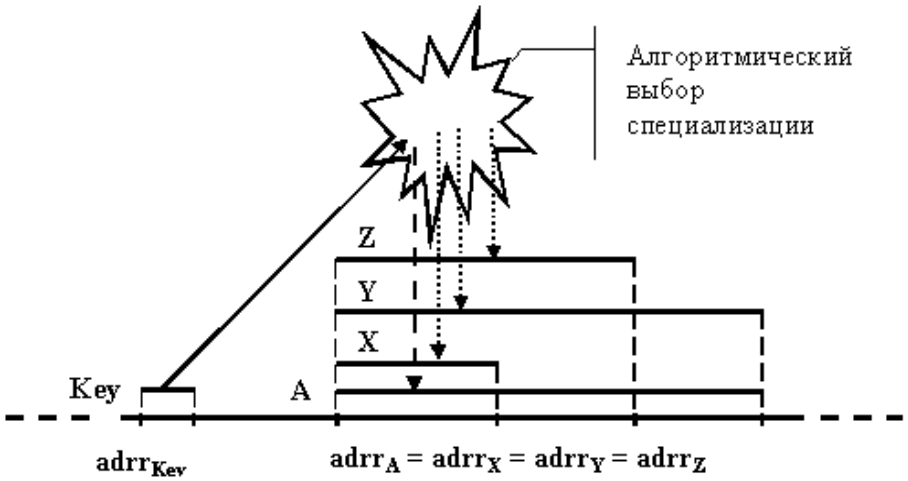
При объединении **на основе общего ресурса** происходит размещение специализаций в едином адресном пространстве. При этом обычно существует часть ресурса, одновременно перекрываемая всеми размещаемыми программными объектами. Общая схема такого перекрытия для плоскости приведена на рис. 6а. На рис. 6б показано аналогичное размещение с перекрытием для одномерного пространства адресов памяти. Однако, чаще всего, обобщения на основе общего ресурса строится таким образом, что начальный адрес для всех размещаемых объектов является одинаковым (рис. 6в).



а) условное изображение на плоскости



б) отображение на одномерное адресное пространство



в) отображение на одномерное адресное пространство при размещении специализаций по одному начальному адресу

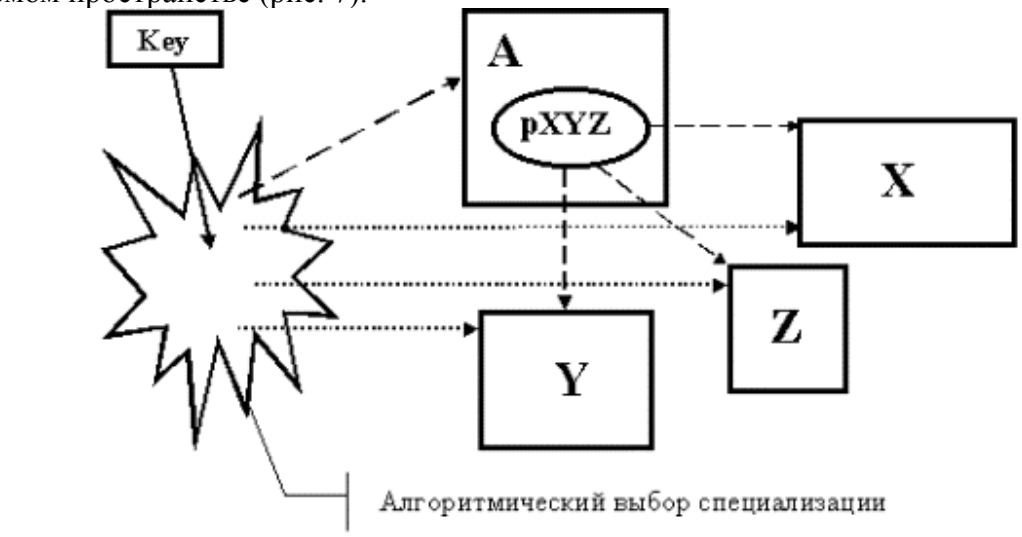
Рис. 6. Обобщение A из элементов X, Y, Z, построенное с использованием объединения на основе общего ресурса.

Наличие общего ресурса для нескольких абстракций позволяет использовать такое обобщение в двух различных целях:

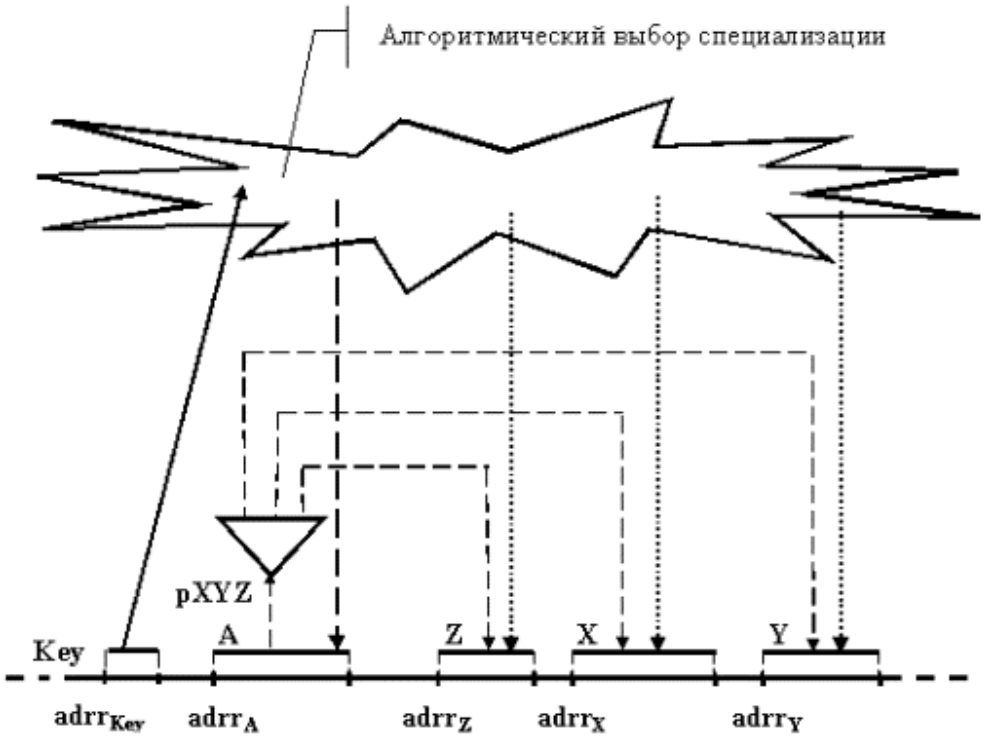
1. Хранение одного из альтернативных объектов. В этом случае выделенное пространство предоставляется экземпляру абстракции только одного типа, который хранит свое значение, никоим образом по семантике не коррелирующееся с семантикой альтернативных объектов. Поэтому, любая попытка альтернативной интерпретации общего ресурса ведет к семантической ошибке в программе.
2. Использование различных трактовок одного и того же пространства ресурсов. Может быть связано с различной семантической интерпретацией некоторой абстракции и обеспечивает разделение множества возможных операций над одним и тем же объектом на несколько непересекающихся групп. В частности, для математических операций целое число длиной два байта может интерпретироваться как единое слово, для операций сохранения-восстановления в двоичном формате, как младший и старший байты. Для операций сравнения и сдвига бывает необходимо отделять знаковый бит от значения. Для каждого из этих случаев возможна своя специализация. А вместе они могут формировать обобщение.

Обращение к конкретной специализации осуществляется алгоритмически после анализа значения ключа, указывающего на текущую альтернативу. При этом доступ обычно осуществляется не напрямую, а через оболочку обобщения (на схеме показано жирной пунктирной стрелкой).

Альтернативное связывание предполагает независимое размещение специализаций в рассматриваемом пространстве (рис. 7).



а) условное изображение на плоскости

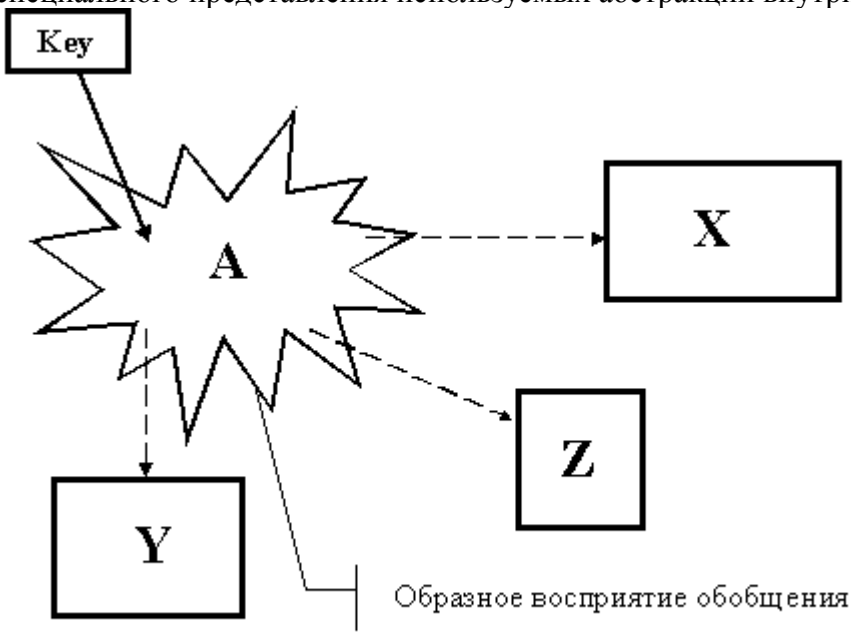


б) отображение на одномерное адресное пространство

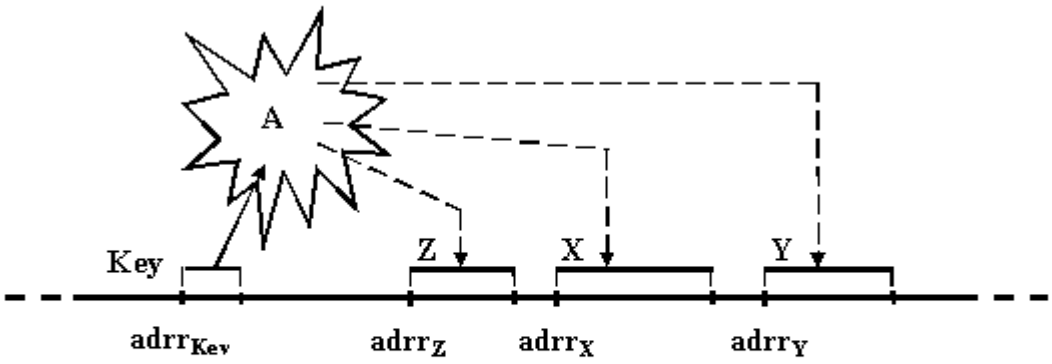
Рис. 7. Обобщение A из элементов X, Y, Z, построенное с использованием динамического вариантного связывания.

Однако использование ссылки или указателя обеспечивает доступ только к одному избранному объекту. При этом одновременно могут существовать все экземпляры специализаций или только часть из них. Таким образом, обобщение строится на основе динамического подключения одного независимого объекта. Как и косвенное связывание при агрегировании, альтернативное связывание обычно осуществляется алгоритмически, хотя допускается и статическая его реализация, если подключаемая специализация известна заранее. Выполнение алгоритма, осуществляющего связывание, происходит для отдельного экземпляра обобщения только один раз. Одновременно с выбором специализации осуществляется и установка ключа, указывающего признак специализации. Дальнейшая работа с обобщением осуществляется на основе предварительного анализа значения ключа. Это значение позволяет семантически правильно обработать специализацию через "обезличенный" указатель. Альтернативное связывание позволяет гибко формировать обобщения во время работы программы.

Образное обобщение (рис. 8), как и образное агрегирование, связано с мысленными ассоциациями программиста без специального представления используемых абстракций внутри программы.



а) условное изображение на плоскости



б) отображение на одномерное адресное пространство

Рис. 8. Обобщение A из элементов X, Y, Z, построенное на основе образного восприятия.

Используя ключевой признак, можно связать с отдельными специализациями подмножества разрозненных объектов программы, семантическая связь между которыми поддерживается на уровне алгоритма. При этом одни и те же программные объекты могут использоваться в альтернативных специализациях, имея при этом различную семантическую трактовку. Как и в случае с образным агрегированием, такой подход к написанию программ является "пережитком прошлого". Достаточно часто он использовался и для экономии памяти, когда одни и те же переменные интерпретировались различным образом.

На практике постоянно встречаются и смешанные обобщения, сочетающие различные варианты объединения специализаций. Особенно широко такой подход применяется в процедурном программировании для рационального использования памяти. Зачастую, специализации, имеющие схожий размер, обобщаются на основе единого ресурса. Те же специализации, размер которых значительно отличается в большую сторону, включаются на основе вариантного связывания.

Вариантное обобщение

Обозначим термином "***вариант***" основу обобщения данных в процедурном подходе. Обобщение, применяемое в процедурном подходе и построенное на основе варианта, назовем ***вариантным обобщением***. С каждым вариантом связан набор специализаций обобщения, построенный на основе уже существующих абстракций. Определим их как ***вариантные специализации***.

Обработка вариантных обобщений осуществляется независимыми процедурами, организующими доступ к внутренним переменным через экземпляр обобщения, получаемый, в качестве одного из аргументов списка параметров. Процедуры, обрабатывающие специализации обобщений, могут создаваться независимо от обобщающей процедуры. Они могут использоваться различными обобщающими процедурами. Каждая из таких процедур связана только со своими специализациями. В дальнейшем специализированные процедуры, используемые в процедурном подходе, будут называться ***обработчиками вариантов***.

Процедуры, осуществляющие обработку всего вариантного обобщения, используют алгоритмический механизм анализа вариантов по ключевому параметру, содержащему признак текущей вариантной специализации. Алгоритм анализа обычно строится с использованием условных оператором или переключателей. Анализ осуществляется всякий раз, когда запускается процедура, и заключается в проверке ключа, задающего признак специализации обобщения. После определения специализации запускается соответствующий обработчик варианта. Обобщающая процедура, осуществляющая обработку вариантного обобщения, называется ***вариантной процедурой***.

Процедурное программирование позволяет создавать варианты с использованием любого из описанных ранее методов формирования обобщений. Такая универсальность обуславливается гибкостью подхода, определяемая тем, что основной упор ложится на написание кода, обрабатывающего что угодно и как угодно. Абстракции данных изначально отсутствовали, добавляясь в ходе эволюционного развития. Поэтому, различные методы использования операций и операторов, основанные только на образном восприятии предметной области, появились с самого начала и лишь после стали постепенно вытесняться иерархическим абстрагированием.

Использование независимых вариантных процедур для создания кода ведет к централизации процесса обработки обобщений, разделяя его на отдельные задачи. Каждая из процедур обеспечивает решение одной из специализированных задач. Процедуры, решающие разные задачи, совершенно не связаны друг с другом. Декомпозиция работ внутри вариантной процедуры осуществляется в соответствии со специализацией обобщений. Каждая из работ выполняется отдельным обработчиком варианта.

Построение вариантного обобщения на основе общего ресурса

Использование общего ресурса позволяет строить вариантное обобщение на основе данных, размещаемых в едином адресном пространстве. Большинство процедурных языков программирования, имеющих абстрактные типы, поддерживают этот механизм. Рассмотрим создание обобщенной геометрической фигуры, используемой в задаче 1 для представления прямоугольника или треугольника. Первоначально создаются абстракции данных, определяющие конкретные геометрические фигуры:

```
//-------------------------------------------------------------
// прямоугольник
struct rectangle {
   int x, y; // ширина, высота
};
//-------------------------------------------------------------
// треугольник
struct triangle {
   int a, b, c; // стороны
};
//-------------------------------------------------------------
```

С каждой из специализаций связывается следующий набор обработчиков:

- Процедуры динамического создания прямоугольника и треугольника: ***rectangle *Create_rectangle(int x, int y)*** и ***triangle *Create_triangle(int a, int b, int c)***. Предназначены для создания указанных фигур по значениям их сторон в динамической памяти. Каждая возвращает указатель на созданный объект.

- Процедуры инициализации уже созданных прямоугольников и треугольников: ***void Init(rectangle***

&r, int x, int y) и ***void Init(triangle &t, int a, int b, int c)***. Исползуются в том случае, когда специализации созданы статически, автоматически или как составная часть обобщения.

- Процедуры ввода параметров прямоугольника и треугольника из входного потока: ***void In(rectangle &r)*** и ***void In(triangle &t)***. Обеспечивают установку значений сторон во время диалога с пользователем.
- Процедуры вывода параметров прямоугольника и треугольника: ***void Out(rectangle &r)***< и ***void Out(triangle &t)***. Предназначены для вывода в выходной поток имеющихся данных.
- Процедуры вычисления площадей треугольника и прямоугольника: ***double Area(rectangle &r)*** и ***double Area(triangle &t)***.

Следует отметить, возможность разработки указанных процедур независимо от контекста, связанного с последующим использованием, что обуславливается восходящим характером написания кода. При этом легко могут быть созданы процедуры, полезные с точки зрения разработчика, но не нужные в рамках решаемой задачи. Ниже представлены исходные тексты описанных процедур, сгруппированные по обрабатываемым типам данных.

```
//-----------------------------------------------------------------
// Процедуры, обеспечивающие работу с прямоугольником
//-----------------------------------------------------------------
// Динамическое создание прямоугольника по двум сторонам
rectangle *Create_rectangle(int x, int y)
{
   rectangle *r = new rectangle;
   r->x = x;
   r->y = y;
   return r;
}
//-----------------------------------------------------------------
// Инициализация уже созданного прямоугольника по двум сторонам
void Init(rectangle &r, int x, int y)
{
   r.x = x;
   r.y = y;
}
//-----------------------------------------------------------------
// Ввод параметров прямоугольника
void In(rectangle &r)
{
   cout << "Input Rectangle: x, y = ";
   cin >> r.x >> r.y;
}
//-----------------------------------------------------------------
// Вывод параметров прямоугольника
void Out(rectangle &r)
{
   cout << "It is Rectangle: x = "
        << r.x << ", y = "
        << r.y << endl;
}
//-----------------------------------------------------------------
// Вычисление площади прямоугольника
double Area(rectangle &r)
{
   return r.x * r.y;
}
//-----------------------------------------------------------------
// Процедуры, обеспечивающие работу с треугольником
//-----------------------------------------------------------------
```

```
// Динамическое создание треугольника по трем сторонам
triangle *Create_triangle(int a, int b, int c)
{
   triangle *t = new triangle;
   t->a = a;
   t->b = b;
   t->c = c;
   return t;
}
//---------------------------------------------------------------
// Инициализация уже созданного треугольника по трем сторонам
void Init(triangle &t, int a, int b, int c)
{
   t.a = a;
   t.b = b;
   t.c = c;
}
//---------------------------------------------------------------
// Ввод параметров треугольника
void In(triangle &t)
{
   cout << "Input Triangle: a, b, c = ";
   cin >> t.a >> t.b >> t.c;
}
//---------------------------------------------------------------
// Вывод параметров треугольника
void Out(triangle &t)
{
   cout << "It is Triangle: a = "
     << t.a << ", b = " << t.b
     << ", c = " << t.c << endl;
}
//---------------------------------------------------------------
// Вычисление площади треугольника
double Area(triangle &t)
{
   double p = (t.a + t.b + t.c) / 2.0; // полупериметр
   return sqrt(p * (p-t.a) * (p-t.b) * (p-t.c));
}
//---------------------------------------------------------------
```

После этого формируется вариантное обобщение на основе объединения, обеспечивающее использование экземпляром абстракции общего адресного пространства для хранения одной из фигур.

```
// Обобщение на основе разделяемого (общего) ресурса
union {
   rectangle r;
   triangle t;
};
```

Это объединение включается в агрегат, который также содержит признаки специализаций и ключевую переменную, предназначенную для хранения текущего признака каждого экземпляра.

```
//---------------------------------------------------------------
// структура, обобщающая все имеющиеся фигуры
  struct shape {
    // значения ключей для каждой из фигур
    enum key {RECTANGLE, TRIANGLE};
    key k; // ключ
    // Обобщение на основе разделяемого (общего) ресурса
```

```
    union { // используем простейшую реализацию
      rectangle r;
      triangle t;
    };
  };
  //---------------------------------------------------------------
```

Вариантные процедуры, работающие с построенным обобщением, осуществляют окончательное объединение обработчиков специализаций:

- Процедуры ***shape *Create_shape_rectangle(int x, int y)*** и ***shape *Create_shape_triangle(int a, int b, int c)*** динамически создают обобщенную фигуру, инициализируя полученные экземпляры ранее созданными обработчиками вариантов.
- Процедуры ***void Init_rectangle(shape &s, int x, int y)*** и ***void Init_triangle(shape &s, int a, int b, int c)*** инициализируют обобщенную фигуру, созданную статически или автоматически. При этом опять же используются инициализаторы специализаций.
- Процедура ***shape* In()*** осуществляет создание обобщенной фигуры и ввод ее типа, используемого при вводе данных конкретной специализации. Посредством простейшего диалога, осуществляется выбор конкретной фигуры, что определяет дальнейший ввод параметров.
- Процедура ***void Out(shape &s)*** обеспечивает вывод конкретного варианта, определяемого экземпляром обобщенной фигуры.
- Процедура ***double Area(shape &s)*** предназначена для вычисления площади заданного экземпляра вариантного обобщения.

Исходные тексты процедур представлены в следующем ниже листинге. Для большей наглядности, все необходимые сигнатуры специализаций сгруппированы в одном месте.

```
//---------------------------------------------------------------
// Процедуры, обеспечивающие работу с обобщенной фигурой
//---------------------------------------------------------------
// Сигнатуры, необходимые обработчикам вариантов.
void Init(rectangle &r, int x, int y);
void Init(triangle &t, int a, int b, int c);
void In(rectangle &r);
void In(triangle &t);
void Out(rectangle &r);
void Out(triangle &t);
double Area(rectangle &r);
double Area(triangle &t);
//---------------------------------------------------------------
// Динамическое создание обобщенного прямоугольника
shape *Create_shape_rectangle(int x, int y)
{
  shape *s = new shape;
  s->k = shape::key::RECTANGLE;
  Init(s->r, x, y);
  return s;
}
//---------------------------------------------------------------
// Инициализация обобщенного прямоугольника
void Init_rectangle(shape &s, int x, int y)
{
  s.k = shape::key::RECTANGLE;
  Init(s.r, x, y);
}
//---------------------------------------------------------------
// Динамическое создание обобщенного треугольника
shape *Create_shape_triangle(int a, int b, int c)
{
```

```
    shape *s = new shape;
    s->k = shape::key::TRIANGLE;
    Init(s->t, a, b, c);
    return s;
}
//-----------------------------------------------------------------
// Инициализация обобщенного треугольника
void Init_triangle(shape &s, int a, int b, int c)
{
    s.k = shape::key::TRIANGLE;
    Init(s.t, a, b, c);
}
//-----------------------------------------------------------------
// Ввод параметров обобщенной фигуры из стандартного потока ввода
shape* In()
{
    shape *sp;
    cout << "Input key: for Rectangle is 1, "
         "for Triangle is 2, else break: ";
    int k;
    cin >> k;
    switch(k) {
    case 1:
      sp = new shape;
      sp->k = shape::key::RECTANGLE;
      In(sp->r);
      return sp;
    case 2:
      sp = new shape;
      sp->k = shape::key::TRIANGLE;
      In(sp->t);
      return sp;
    default:
      return 0;
    }
  }
//-----------------------------------------------------------------
// Вывод параметров текущей фигуры в стандартный поток вывода
void Out(shape &s)
{
    switch(s.k) {
    case shape::key::RECTANGLE:
      Out(s.r);
      break;
    case shape::key::TRIANGLE:
      Out(s.t);
      break;
    default:
      cout << "Incorrect figure!" << endl;
    }
}
//-----------------------------------------------------------------
// Нахождение площади обобщенной фигуры
double Area(shape &s)
{
    switch(s.k) {
```

```
    case shape::key::RECTANGLE:
      return Area(s.r);
    case shape::key::TRIANGLE:
      return Area(s.t);
    default:
      return 0.0;
    }
  }
  //---------------------------------------------------------------
```

Следует отметить отсутствие в обработчиках вариантов прямой зависимости от специализаций, что обуславливается доступом через процедуры-посредники, предоставляющие только свои сигнатуры.
Существующие языки программирования по-разному поддерживают механизм формирования обобщения на основе общего ресурса. В языках C и C++ [Страуструп], для задания ключа приходится создавать дополнительную структуру. Языки программирования Pascal [Вирт85] и Modula-2 [Вирт87] позволяют сразу создавать вариантные записи с ключом, но контроль на соответствие между текущим значением ключа и хранимой специализацией отсутствует. В языке программирования Ada [Джехани] используются объединения, в которых значение ключа точно соответствует хранимому объекту и обеспечивает поддержку дополнительного контроля при доступе к объекту во время выполнения. Это позволяет сгенерировать исключение при попытке некорректно использовать экземпляр обобщения.
Исходный текст программы, использующей обобщение на основе общего ресурса, приведен в уже упомянутом архиве pp_examp1.zip.
Вариантные обобщения на основе общего ресурса формируются при написании программы и распознаются во время трансляции. Это позволяет заранее распределить память и обеспечить быстрый и непосредственный доступ к отдельным экземплярам. К недостаткам можно отнести неэффективное использование пространства памяти при различных размерах специализаций.
Особенностью такого обобщения также является то, что его основа, как программный объект формируется после создания всех специализаций. Это связано с необходимостью знания их размеров для распределения памяти во время трансляции. Таким образом, процесс построения языковой структуры вариантного обобщения совпадает по направленности с восходящим проектированием. В начале кодируются отдельные специализированные фрагменты, а уже затем формируется их совместное, более крупное, образование. Это никоим образом не говорит, что при процедурном подходе проблематично применять нисходящую разработку. Просто, нисходящее проектирование, при статических вариантных обобщениях, ведется таким образом, что в начале осуществляется полная проработка всей иерархии обобщения, а уж только потом можно приступать к его кодированию. Возможно, что эта особенность кодирования вариантных обобщений в какой-то мере послужила популярности водопадной модели, оказавшейся малоэффективной при разработке сложных программ.

Построение вариантного обобщения на основе альтернативного связывания

Использование альтернативного связывания позволяет, зачастую, обойтись без дополнительных обобщающих конструкций, что обеспечивает большую гибкость при эволюционном наращивании процедурной программы. В этом плане можно посмотреть, как изменится метод решения задачи, если использование общего ресурса заменить вариантным связыванием. Сразу отмечу, что эквивалентных вариантов реализации существует очень много, а проводить их сравнение и анализ вряд ли имеет смысл (что получилось, то и показываю:). Сохраним существующие специализации без изменения, но заменим обобщение:

```
//---------------------------------------------------------------

// структура, обобщающая все имеющиеся фигуры

  struct shape {

    // значения ключей для каждой из фигур

    enum key {RECTANGLE, TRIANGLE};

    key k; // ключ
```

```
      // используемые альтернативные указатели
      union { // используем простейшую реализацию
        rectangle *pr;
        triangle *pt;
      };
    };
  //-------------------------------------------------------------
```

Все дальнейшие изменения, проведенные в этой программе, по сравнению с первой версией, связаны со спецификой использования уже имеющихся абстракций, комментировать которые дополнительно не вижу смысла. В архиве pp_examp1b.zip лежат исходные тексты, демонстрирующие, что из этого получилось.

Следует также отметить, что гибкость механизма альтернативного связывания привела к его непосредственной, явной или опциональной, поддержке в ряде языков программирования. В языке программирования C++ реализован механизм идентификации типов во время выполнения (RTTI) [Страуструп]. Он включается опционально, что позволяет использовать абстрактные типы как с дополнительными полями, поддерживающими динамическую типизацию, так и без них. В языке программирования Оберон такая поддержка введена явно. При этом в языке отсутствуют вариантные записи, обеспечивающие поддержку наложения ресурсов. Вирт [Wirth] посчитал, что такой механизм является избыточным. Возможно, что он прав, когда дело касается языка, не ориентированного на серьезное (машиннозависимое) системное программирование (да, я знаю, что на Обероне написана его же операционная система и оболочка для работы с пользователем:).

Кстати, я отношу использование RTTI к процедурному программированию, что определяется его идеологией, связанной с поддержкой признака объекта на уровне языковой семантики и использованием данного признака при централизованном выборе альтернатив. Как не крути, но использование контроля типа во время выполнения поднимает те же проблемы эволюционного развития программы, что и процедурный подход. При этом не важно, каким образом реализовано получение информации о типе объекта. Она может быть получена через специальную функцию (например, typeid в C++ [Страуструп]) или дополнительную переменную, параметризирующую объект (как это сделано в языке Оберон-2 [MoessenboeckWirth]).

Не привожу в качестве примера другие языки, так как данный механизм в большинстве из них - это не только дань моде. Это способ преодоления недостатков, присущих "чистому" объектно-ориентированному подходу! До этого момента мы еще доберемся, но уже сейчас можно сказать, что "чистых" ОО языков (без RTTI) практически не существует из-за низкой эффективности однорукого объектного программирования!

Построение вариантного обобщения на основе образного восприятия

Вряд ли, в наше время, имеет смысл писать пример, демонстрирующий образное восприятие обобщений, в стиле языков Фортран или Алгол-60. Такое сейчас редко увидишь даже в нормальных программах, написанных на Ассемблере (и туда уже дошли механизмы описания не только абстрактных типов данных, но и классов). Однако, достаточно часто встречаются приемы, обеспечивающие сочетания образного восприятия с другими методами формирования вариантов. Это делается как по незнанию, так и для получения кода, обладающего меньшей зависимостью от других частей. Следует отметить, что в последнем случае часто снижается семантический контроль программы во время компиляции. Ловите ошибки во время выполнения!

Для иллюстрации образного восприятия вариантов, введем в нашу программу следующие изменения:

- Создадим контейнер, используя указатели на любой тип данных (void). Будем подразумевать, что каждый элемент контейнера указывает на одну из разработанных фигур.
- Для идентификации специализаций, вместо перечислимого типа, используем целые числа, образно ассоциируя их с соответствующими фигурами (1 - прямоугольник, 2 - треугольник). Тогда, появление новой фигуры просто сведется к образной ассоциации с еще одним числом.

- Результаты таких умозрительных рассуждений отображены в следующих структурах контейнера и обобщений.

```
//---------------------------------------------------------------
struct container
{
   enum {max_len = 100}; // максимальная длина
   int len; // текущая длина
   // Контейнер обазных указателей на специализации, построенные
   // на основе распределенного ключа
   void* cont[max_len];
};
//---------------------------------------------------------------
// Использование образного восприяитя обобщения на основе ключа,
// разнесенного по отдельным дополнительным специализациям
//---------------------------------------------------------------
// шаблонная структура, обобщающая все имеющиеся фигуры
struct shape {
   // значения ключей для каждой из фигур
   int k;
   // образый ключ устанавливается для каждой вводимой фигуры.
   // Можно легко ошибиться, но гибкость - прекрасная!
   // А связь с фигурами отсутствует!
   // Предполагается, что они пропишутся отдельно и будут связаны
   // через образное наложение одинаковых структур памяти
 };
//---------------------------------------------------------------
// структура, обобщающая прямоугольник
struct r_shape {
   // значения ключей для каждой из фигур
   int k; // образый ключ  = 1
   // А здесь будет храниться прямоугольник
   rectangle r;
};
//---------------------------------------------------------------
// структура, обобщающая треугольнк
struct t_shape {
   // значения ключей для каждой из фигур
   int k; // образый ключ  = 2
   // А здесь будет храниться треугольник
   triangle t;
};
//---------------------------------------------------------------
```

Следует отметить ряд особенностей. Контейнер теперь совершенно не зависит от хранимых в нем фигур и может использоваться для чего угодно. Пользуйтесь и ошибайтесь на здоровье! Каждая из геометрических фигур использует дополнительный ключ, который тоже не контролируется. Структура ***shape*** введена для того, чтобы выделить общую часть всех обобщающих фигур (в данном случае - ключ). Она используется в качестве шаблона при анализе ключей экземпляров специализаций.

Использование "всеохватывающего" типа void ведет к потере информации о подключаемом типе, то есть, "растипизации". Это не позволяет в дальнейшем использовать объект без явного и неконтролируемого приведения типов, осуществляемого во всех обработчиках вариантов после анализа текущего значения признака. Надо хорошо постараться, чтобы учесть возможные преобразования и обеспечить их правильность. В остальном же процесс обработки альтернатив мало чем отличается от тех методов, которые использовались при построении обобщений на основе общего ресурса и альтернативного связывания. Текст данной версии программы расположен в архиве pp_examp1c.zip.

Примечание. *Возможно, я утомил Вас однообразными и тривиальными примерами. Но хотелось уменьшить количество ошибок в иллюстрациях. Да и не надо их смотреть, если и так все понятно.*

Как объектно-ориентированная альтернатива лишилась признака

Основным отличием ОО подхода от процедурного является возможность группирования процедур вокруг обрабатываемых ими данных. Оно, сочетаемое с динамическим связыванием объектов, обеспечивает возможность выбора альтернатив без дополнительного анализа признака специализации. При процедурном подходе анализ принадлежности объектов к определенному типу обычно осуществляется внутри процедур, что ведет к использованию алгоритмических методов обработки альтернатив.

Рассмотрим специфику различных способов группировки на простом примере. Предположим, что нам необходимо выполнить множество обобщающих процедур, доступных через множество их сигнатур **F(D)**:

$$\mathbf{F(D) = \{F_1(D), F_2(D), \dots , F_m(D)\}},$$

$\mathbf{D = \{D_1, D_2, \dots , D_n\}}$ - обобщающий аргумент одного из типов:

$$\mathbf{type(D_j) = t_j}, \text{ где } \mathbf{t_j \in T} \text{ и } \mathbf{T = \{t_1, t_2, \dots , t_n\}}.$$

При этом для выполнения процедуры $\mathbf{F_i}$, обрабатывающей аргумент типа $\mathbf{t_j}$, используется специализированная процедура $\mathbf{f_{ij}(D_j)}$. Процессор, обрабатывающий такую полиморфную процедуру, может использовать два различных варианта последовательного доступа к альтернативным параметрам. В первом случае обработка может начаться с определения значения процедуры, а лишь затем будет осуществляться анализ типа операнда. На некотором С-подобном языке такой вариант анализа можно представить в виде следующей схемы:

```
switch(F) {
  case F1:
    switch(type(D)) {
      case t1: f11(D1); break;
      case t2: f12(D2); break;
      ...
      case tn: f1n(Dn); break;
    }
  case F2:
    switch(type(D)) {
      case t1: f21(D1); break;
      case t2: f22(D2); break;
      ...
      case tn: f2n(Dn); break;
    }
    ...
  case Fm:
    switch(type(D)) {
      case t1: fm1(D1); break;
      case t2: fm2(D2); break;
      ...
      case tn: fmn(Dn); break;
    }
}
```

Тот же самый результат на выходе можно получить, если в начале проанализировать тип аргумента, а лишь затем – значение процедуры:

```
switch(type(D)) {
  case t1:
    switch(F) {
      case F1: f11(D1); break;
      case F2: f21(D1); break;
        ...
      case Fm: fm1(D1); break;
    }
  case t2:
```

```
    switch(F) {
      case F1: f12(D2); break;
      case F2: f22(D2); break;
      ...
      case Fm: fm2(D2); break;
    }
    ...
  case tn:
    switch(F) {
      case F1: f1n(Dn); break;
      case F2: f2n(Dn); break;
      ...
      case Fm: fmn(Dn); break;
    }
}
```

Первый способ анализа задает зависимость данных от исполняемых процедур. Именно таким образом происходит группировка внутри процедурных программ. Группировка процедур в зависимости от типов данных является спецификой объектного программирования. Эта специфика практически не проявляется, если мы имеем дело с некоторым процессором, занимающимся анализом произвольных комбинаций процедур и данных, поступающих на его вход.

Однако в случае с программами возникает несколько иная ситуация. Значения альтернативных типов данных обычно определяются в ходе выполнения программы. Они формируются случайным образом из разных источников, зависят от разных факторов, что и определяет их непредсказуемость. Расположение же обобщающих процедур задается программистом, и определяется алгоритмом решаемой целевой задачи. Это ведет к естественному отбрасыванию альтернативы, обеспечивающей выбор процедуры. Но обобщающие процедуры, располагаемые в процедурной программе на строго определенных местах, не могут иметь информации о типах фактических параметров-специализаций, формируемых во время выполнения программы. В этой ситуации вполне естественно, что обобщающие процедуры содержат в своих телах код, осуществляющий анализ признаков для альтернативных вариантов аргументов, чтобы выбрать и исполнить требуемую специализированную процедуру.

В ОО программах методы классов тоже размещаются в местах, определенных алгоритмом обработки данных. Но их привязка к типам обрабатываемых данных осуществляется во время кодирования и в декларативной манере посредством механизма виртуализации. В программе не требуется создание экземпляров конкретных объектов, но заготовки отношений между типами данных и процедурами уже хранятся в виде векторов отношений, в каждом из классов $\mathbf{C_i} \in \mathbf{C}$:

$$\mathbf{C_1 = (t_1, f_{11}, f_{11}, f_{m1}),}$$
$$\mathbf{C_2 = (t_2, f_{12}, f_{22}, f_{m2}),}$$

…

$$\mathbf{C_n = (t_n, f_{1n}, f_{2n}, f_{mn}).}$$

Поэтому, когда в ходе выполнения программы формируется экземпляр объекта $\mathbf{D_i}$ типа $\mathbf{t_i}$, создается и экземпляр отношения $\mathbf{C_i}$, которое содержит и процедуры, обеспечивающие требуемую обработку созданного объекта.

Однозначная зависимость между данными и процедурами их обработки, сгруппированными в одном объекте, позволяет каждому экземпляру класса в любой момент обратиться к его же методу через существующее отношение без дополнительного анализа признака при выборе специализированной процедуры.

Объектное обобщение

Таким образом, объектно-ориентированный подход предлагает такой метод организации альтернатив, который не требует анализа признаков специализаций. Он использует совокупность из двух статически связанных агрегатов (рис. 9), динамически подключаемых к указателю на альтернативу.

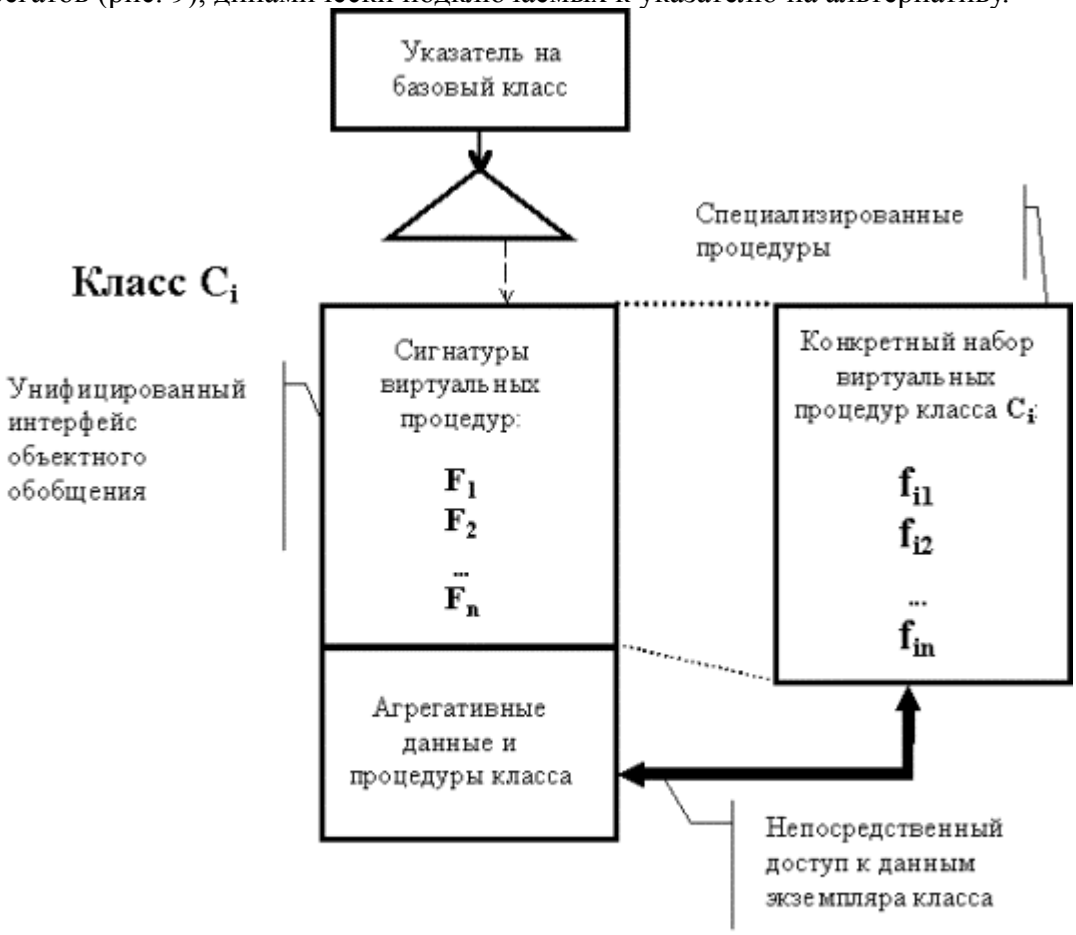


Рис. 9. Двойное связывание, обеспечивающее трактовку класса как альтернативы объектного обобщения.

Первый агрегат, являясь экземпляром производного или базового класса, содержит данные и процедуры, непосредственно размещенные в его теле. Для различных производных классов состав этих данных и процедур может отличаться, так как не они определяют облик формируемого обобщения.
Внутри этого агрегата существует вторая, статическая связь, формируемая во время компиляции. Она присоединяет агрегат, состоящий из виртуальных (переопределяемых) процедур, обеспечивающих обработку специализации обобщения. Этот агрегат процедур (а не таблица виртуальных функций с указателями на процедуры) может заменяться на другой при построении производного класса. Именно разделение на две конструкции способствует гибкой смене специализированных процедур в ходе разработки программы, которое внешне выглядит как замещение виртуальных процедур базового класса процедурами производного класса. При этом сигнатуры альтернативных процедур, определяемых базовым классом, не меняются. Непосредственный доступ из виртуальных процедур к переменным класса позволяет гибко подстраивать процесс обработки данных под специфику производных классов.

Реализация такого связывания осуществляется на основе механизмов наследования и виртуализации, являющихся общепринятыми в ОО подходе. Они позволяют создать иерархию классов, расширяемую по самым различным направлениям. Общая схема возможной иерархии классов приведена на рис. 10.

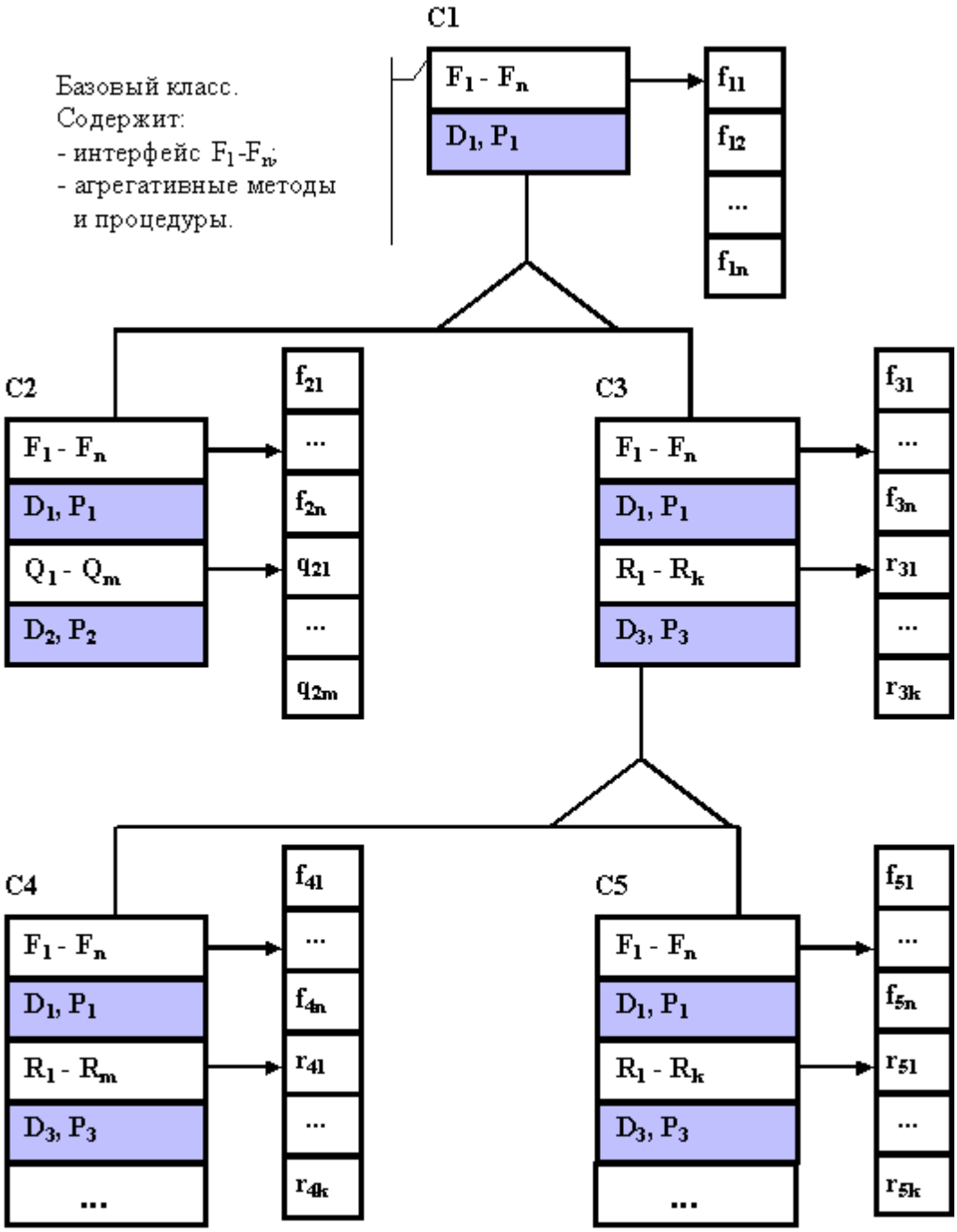


Рис. 10. Иерархия классов, построенная с применением наследования и виртуализации.

Каждый из классов этой иерархии может служить альтернативой в объектном обобщении, порождаемом на основе базового класса. Следует отметить, что, в общем случае, количество виртуальных процедур, а также агрегативных процедур и данных в порождаемых производных классах может увеличиваться. Однако, структура интерфейса, определяемая базовым классом, остается неизменной, что и позволяет, на этой основе, выстраивать механизм анализа альтернатив.

Основная задача выбора альтернативы заключается в вызове соответствующей процедуры. Она решается следующим образом. Через указатель на экземпляр базового класса, обеспечивающий динамическое связывание с любым экземпляром производного класса, осуществляется доступ к интерфейсу объекта. Назовем такой указатель ***опорой объектного обобщения***. Схема возможных динамических подключений для ранее рассмотренной иерархии классов представлена на рис. 11.

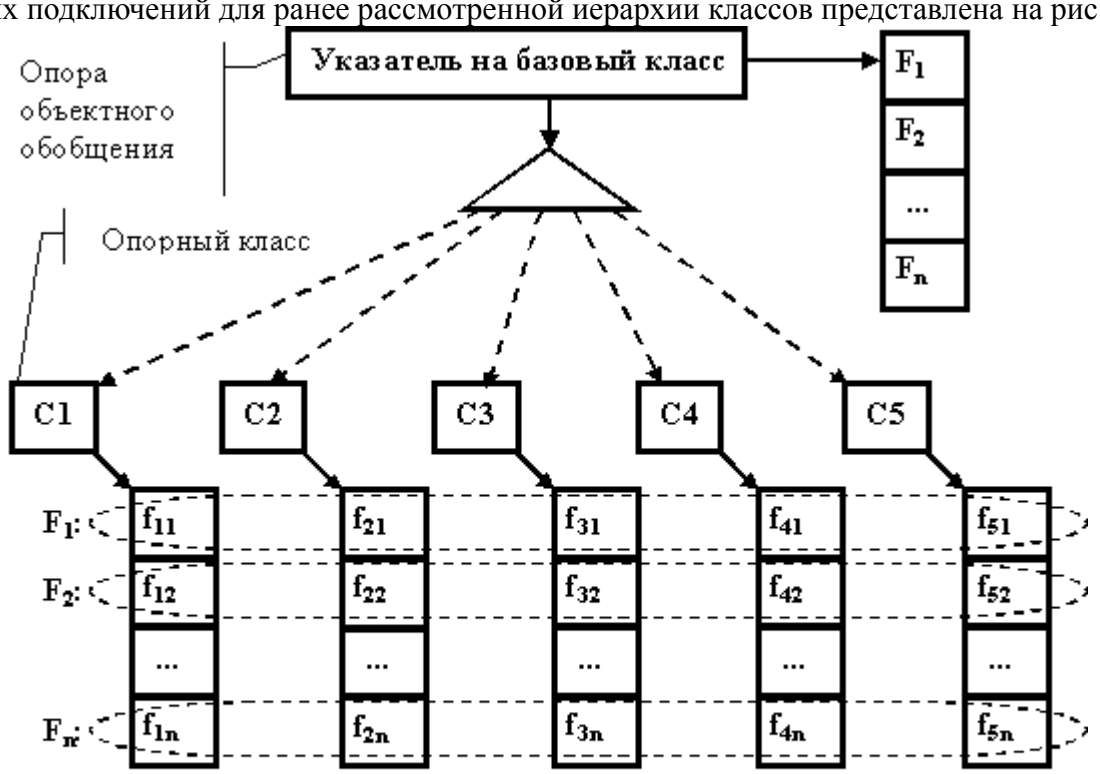


Рис. 11. Динамическое подключение альтернатив в объектном обобщении.

Альтернатива создается во время выполнения программы, предоставляя в наше распоряжение набор специализированных процедур, к которым происходит обращение через второй указатель, обеспечивающий статическое связывание. Как и многие предшествующие мне ораторы, хочу еще раз отметить, что подобная структура является не единственной, и двойное связывание может быть реализовано различными способами.

Таким образом, в объектном обобщении не нужен специализированный ключ, обеспечивающий анализ существующей альтернативы, так как решение проблемы безусловного выбора осуществляется через изменение порядка опроса объектов. Вместе с тем, можно считать, что такой ключ у каждого класса существует. Им является совокупность всех специализированных процедур, подключаемых через статически связанный агрегат.

Предположим, что обычный класс, определяющий агрегат, не содержит виртуальных процедур. Допустим также, что базовый класс позволяет порождать производные классы с применением иерархии наследования и содержит только виртуальные процедуры. Такое допущение позволяет независимо рассматривать особенности процесса модификации ОО программ для агрегатов и обобщений и не вносит искажений в предмет исследований. Базовый класс, используемый для построения объектного обобщения, назовем ***опорным классом***.

Процесс кодирования объектного обобщения начинается с формирования структуры опорного класса, определяющего единую основу. В нее входят общие переменные, задающие унифицированную часть состояния, общий интерфейс, состоящий из виртуальных процедур класса. Каждая процедура класса определяет одну из целевых функций обработки в соответствии с назначением обобщения. Виртуальная процедура опорного класса определяется как ***опорная процедура*** и играет роль обобщающей процедуры.

После построения опорного класса начинают формироваться все производные классы, каждый из которых доопределяет состояние в сторону своей специализации. Производные классы выступают в роли специализаций обобщения. При этом специализации могут добавляться к опорному обобщению в любое время, не изменяя его организации. Специализации обобщения, построенные на основе производных классов, назовем ***специализациями опоры***.

Направление процесса построения объектного обобщения совпадает с процессом нисходящего проектирования. Первоначально создается опорный класс, определяющий абстракцию верхнего уровня. После этого осуществляется детализация, связанная с построением производных классов, решающих частные задачи. Многоэтапное уточнение может сопровождаться построением иерархий классов на основе наследования. Такое совпадение позволяет проводить программирование почти одновременно с проектированием, создавая каркас приложения или законченную версию с неполными функциональными возможностями. Наращивание функциональных возможностей осуществляется за счет новых специализаций на следующих витках процесса проектирования. Это объясняет популярность возвратной модели проектирования при использовании ООМ.
Обработка обобщений обычно осуществляется виртуальными процедурами производных классов, которые переписываются в соответствии с их специализацией. Эти процедуры назовем ***расширяющими процедурами***. При этом процедура базового класса (опорная процедура) может использоваться там, где ее функциональных возможностей достаточно. В объектном обобщении осуществляется привязка процедур классов к объектам, что ведет к централизации функционального интерфейса. Декомпозиция функций производится путем распределения каждой процедуры по своим специализациям, заданным производными классами.

Пример использования объектного обобщения

Рассмотрим использование представленной выше схемы формирования объектного обобщения на примере решения задачи 1. Разработку обобщения начнем с опорного класса, который должен содержать необходимый интерфейс обобщенной геометрической фигуры. При этом создадим абстрактный класс, позволяющий обеспечить более надежный контроль интерфейсов производных классов.

```
//-----------------------------------------------------------------
// Класс, обобщающий все имеющиеся фигуры.
// Является абстрактным, обеспечивая, тем самым,
// проверку интерфейса
class shape {
public:
   virtual void In() = 0;  // ввод данных из стандартного потока
   virtual void Out() = 0; // вывод данных в стандартный поток
   virtual double Area() = 0; // вычисление площади фигуры
};
//-----------------------------------------------------------------
```

Опорные процедуры, располагаемые внутри опорного класса и специализаций опоры, предполагается в дальнейшем использовать следующим образом:

- Процедура ***virtual void In()*** предназначена для ввода исходных данных из стандартного потока ввода.
- Процедура ***virtual void Out()*** обеспечивает вывод информации о геометрической фигуре в стандартный поток вывода.
- Процедура ***virtual double Area()*** осуществляет вычисление площади геометрической фигуры.

Все эти виртуальные абстракции начинают обрастать конкретным содержанием только при реализации конкретных специализаций опоры. При этом, разработка специализаций может осуществляться независимо и поэтапно, так как между ними отсутствует какая-либо взаимосвязь. Поэтому, в начале разработаем прямоугольник и его методы.

```
//-----------------------------------------------------------------
// прямоугольник
class rectangle: public shape
{
   int x, y; // ширина, высота
public:
   // переопределяем интерфейс класса
   void In();     // ввод данных из стандартного потока
   void Out();    // вывод данных в стандартный поток
   double Area(); // вычисление площади фигуры
```

```
    rectangle(int _x, int _y); // создание с инициализацией.
    rectangle() {} // создание без инициализации.
};
//---------------------------------------------------------------
// Динамическое создание прямоугольника по двум сторонам
rectangle::rectangle(int _x, int _y): x(_x), y(_y) {}
//---------------------------------------------------------------
// Ввод параметров прямоугольника
void rectangle::In() {
    cout << "Input Rectangle: x, y = ";
    cin >> x >> y;
}
//---------------------------------------------------------------
// Вывод параметров прямоугольника
void rectangle::Out() {
    cout << "It is Rectangle: x = " << x << ", y = " << y << endl;
}
//---------------------------------------------------------------
// Вычисление площади прямоугольника
double rectangle::Area() {
    return x * y;
}
//---------------------------------------------------------------
```

Затем можно приступать к созданию треугольника.

```
//---------------------------------------------------------------
// треугольник
class triangle: public shape
{
    int a, b, c; // стороны
public:
    // переопределяем интерфейс класса
    void In();     // ввод данных из стандартного потока
    void Out();    // вывод данных в стандартный поток
    double Area(); // вычисление площади фигуры
    triangle(int _a, int _b, int _c); // создание с инициализацией
    triangle() {} // создание без инициализации.
};
//---------------------------------------------------------------
// Инициализация уже созданного треугольника по трем сторонам
triangle::triangle(int _a, int _b, int _c): a(_a), b(_b), c(_c) {}
//---------------------------------------------------------------
// Ввод параметров треугольника
void triangle::In() {
    cout << "Input Triangle: a, b, c = ";
    cin >> a >> b >> c;
}
//---------------------------------------------------------------
// Вывод параметров треугольника
void triangle::Out() {
    cout << "It is Triangle: a = "
      << a << ", b = " << b
      << ", c = " << c << endl;
}
//---------------------------------------------------------------
// Вычисление площади треугольника
double triangle::Area() {
```

```
    double p = (a + b + c) / 2.0; // полупериметр
    return sqrt(p * (p-a) * (p-b) * (p-c));
}
//-----------------------------------------------------------------
```

В контейнере имеется массив опор объектных обобщений, обеспечивающий подключение и обработку альтернатив. Следует отметить, что разработка контейнера, представленного при обсуждении агрегатов, осуществляется совершенно независимо от конкретных геометрических фигур. Необходимы знания только структуры базового класса и его интерфейса. Как и ранее, реализации методов вынесены за пределы классов, чтобы обеспечить меньшую зависимость между программными объектами, что существенно при эволюционном проектировании больших программных систем. Код всего примера представлен в уже упоминавшемся архиве oop_examp1.zip.

Отличие методов обобщения

Ниже рассмотрены отличительные особенности технических решений, определяющие специфику использования обобщений. Этот материал ни в коем случае не является полноценным анализом достоинств и недостатков процедурного и объектно-ориентированного программирования.

О безоговорочной победе объектного обобщения

На мой взгляд, именно обобщение с правой повергло процедурное программирование в глубокий нокаут, после которого ему, в основном, приходится выступать лишь на любительских и студенческих подмостках. Основным неоспоримым достоинством объектного обобщения является поддержка быстрого и безболезненного, для уже написанного кода, добавления новых альтернатив в существующие обобщающие процедуры. Там, где процедурный подход ведет к поиску и редактированию фрагментов программы, ООП довольствуется только созданием и легкой притиркой новых классов.

Рассмотрим добавление круга в уже написанную программу, обеспечивающую работу с геометрическими фигурами. При процедурном подходе подобное расширение ведет к изменению обобщения и всех процедур, непосредственно обрабатывающих его. Это связано с тем, что именно в телах процедур осуществляется выбор альтернатив и их последующее использование. Измененное вариантное обобщение, построенное с использованием общего ресурса, примет следующий вид:

```
//-----------------------------------------------------------------
// структура, обобщающая все имеющиеся фигуры
// Изменилась, в связи с добавлением круга
struct shape {
    // значения ключей для каждой из фигур
    enum key {RECTANGLE, TRIANGLE, CIRCLE}; // добавился признак круга
    key k; // ключ
    // используемые альтернативы
    union { // используем простейшую реализацию
      rectangle r;
      triangle t;
      circle c;  // добавился круг
    };
};
//-----------------------------------------------------------------
```

Для нашего небольшого примера добавление круга всего лишь к одному обобщению ведет к модификации все трех процедур, осуществляющих обработку обобщенной геометрической фигуры. Привожу их.

```
//-----------------------------------------------------------------
// Ввод параметров обобщенной фигуры из стандартного потока ввода
// Изменилась из-за добавления круга
shape* In()
{
    shape *sp;
    // Изменилась подсказка из-за добавления круга
    cout <<
```

```
    "Input key: for Rectangle is 1, for Triangle is 2,"
    " for Circle is 3 else break: ";
  int k;
  cin >> k;
  switch(k) {
  case 1:
    sp = new shape;
    sp->k = shape::key::RECTANGLE;
    In(sp->r);
    return sp;
  case 2:
    sp = new shape;
    sp->k = shape::key::TRIANGLE;
    In(sp->t);
    return sp;
  case 3: // добавился ввод круга
    sp = new shape;
    sp->k = shape::key::CIRCLE;
    In(sp->c);
    return sp;
  default:
    return 0;
  }
}
//-----------------------------------------------------------------
// Вывод параметров текущей фигуры в стандартный поток вывода
// Изменилась из-за добавления вывода круга
void Out(shape &s)
{
  switch(s.k) {
  case shape::key::RECTANGLE:
    Out(s.r);
    break;
  case shape::key::TRIANGLE:
    Out(s.t);
    break;
  case shape::key::CIRCLE: // добавился вывод круга
    Out(s.c);
    break;
  default:
    cout << "Incorrect figure!" << endl;
  }
}
//-----------------------------------------------------------------
// Нахождение площади обобщенной фигуры
// Изменилась в связи с добавлением круга
double Area(shape &s)
{
  switch(s.k) {
  case shape::key::RECTANGLE:
    return Area(s.r);
  case shape::key::TRIANGLE:
    return Area(s.t);
  case shape::key::CIRCLE: // добавился круг
    return Area(s.c);
  default:
```

```
    return 0.0;
  }
}
//---------------------------------------------------------------
```

При разработке же больших программных систем обработка обобщений может осуществляться не одной сотней процедур, каждую из которых потребуется изменить. А изменение написанного кода всегда связано с риском внести дополнительные ошибки. Кроме этого, изменения, связанные с добавлением сведений о новой специализаций могут затронуть различные единицы компиляции (без изменения располагаемых в них программных объектов), что тоже ведет к дополнительным затратам. На всякий случай я дописал и откомпилировал этот пример, чтобы убедиться в правильности его работы. Файлы проекта находятся в архиве pp_examp1_1.zip.

ОО подход, в аналогичной ситуации, позволяет провести добавление круга практически без изменений уже написанного кода. При этом, все добавления, связанные с новой фигурой могут осуществляться во вновь создаваемых единицах компиляции. Процесс добавления новой фигуры полностью аналогичен разработке уже созданных и не требует специальных комментариев. В рассматриваемом случае необходимо только изменить процедуру, обеспечивающую ввод новой фигуры. Хотя, и здесь можно было бы поступить более тонко.

```
//---------------------------------------------------------------
// Ввод параметров обобщенной фигуры из стандартного потока ввода
// Изменяется в связи с добавлением круга
shape* In()
{
  shape *sp;
  // Изменяется подсказка
  cout << "Input key: for Rectangle is 1, for Triangle is 2,"
    " for Circle is 3, else break: ";
  int k;
  cin >> k;
  switch(k) {
  case 1:
    sp = new rectangle;
    sp->In();
    return sp;
  case 2:
    sp = new triangle;
    sp->In();
    return sp;
  case 3: // добавлен ввод круга
    sp = new circle;
    sp->In();
    return sp;
  default:
    return 0;
  }
}
//---------------------------------------------------------------
```

Файлы этого проекта находятся в архиве oop_examp1_1.zip.

Естественно, такое впечатляющее техническое преимущество ОО подхода, при создании альтернатив, не могло остаться незамеченным, и привело к его плодотворной раскрутке, Помпезное воспевание оставило за кадром слабые стороны объектного и сильные стороны процедурного программирования.

Мощь обобщающего агрегирования

Кроме наращивания альтернатив, ОО обобщение прекрасно поддерживает эволюционное расширение агрегативных способностей объекта, при котором используется только одна производная альтернатива, заменяющая ту, которая эксплуатировалась до нее. Эта возможность вытекает из того, что обобщение - агрегат. А добавляемые или изменяемые в производных классах виртуальные методы можно трактовать как методы агрегата. Таким образом, применяя наследование, мы можем легко расширять внутреннюю структуру и функциональность объекта, предоставляемого клиентам. А сохранение интерфейса обеспечивает прозрачную для клиента подмену одного объекта другим. Методы любого производного класса могут использовать тот же интерфейс, что и методы ранее используемых базовых классов. Главное для клиента - это отсутствие изменений в интерфейсе связываемого объекта.
Использование процедурного подхода не позволяет осуществлять такую подмену, так как создание нового агрегата сопровождается и созданием для его обработки новых процедур. В связи с тем, что обращение к агрегату осуществляется через формальный параметр определенного типа, приходится менять интерфейс клиента. Можно, конечно, обратиться к подключаемому объекту через указатель, поддерживающий образное восприятие, но все равно, придется изменить тела процедур, вызываемых клиентом, чтобы они были "заточены" на обработку новых структур данных. Но и в этом случае встают дополнительные проблемы, если предполагаются дальнейшие, не предусмотренные регламентом, изменения подключаемого объекта (как нового, так и хорошо зарекомендовавшего себя старого).

Ахиллесовы пятки

Вместе с тем, и у объектных альтернатив есть свои узкие места, которые затрудняют их эффективное применение.

Пята первая: расширение функциональности альтернатив

Снова вернемся к исходной постановке задачи (чтобы не расширять до бесконечности пример). Предположим, что нам необходимо добавить в программу вычисление периметров геометрических фигур. И здесь процедурное программирование слегка отыгрывается на ниве эволюционного расширения программы. Добавление новой обобщающей процедуры некоим образом не связано с изменением уже написанного кода. Создаются специализированные процедуры, обеспечивающие получение периметров для прямоугольников и треугольников. После этого формируется обобщающая процедура, использующая их результаты после анализа признака текущей альтернативы. Да и агрегат (уже рассмотренный нами контейнер), при необходимости вычислить суммарный периметр, изменять не надо. Исходные тексты примера лежат в архиве pp_examp1_2.zip. Процедуры, обеспечивающие вывод периметров отдельных фигур, обобщения и суммарный периметр для фигур, расположенных в контейнере, выглядят следующим образом:

```
//-------------------------------------------------------------
// Вычисление периметра прямоугольника
double Perimetr(rectangle &r)
{
   return (r.x + r.y) * 2.0;
}
//-------------------------------------------------------------
// Вычисление периметра треугольника
double Perimetr(triangle &t)
{
   return t.a + t.b + t.c;
}
//-------------------------------------------------------------
// Нахождение периметра обобщенной фигуры
double Perimetr(shape &s)
{
   switch(s.k) {
   case shape::key::RECTANGLE:
     return Perimetr(s.r);
   case shape::key::TRIANGLE:
     return Perimetr(s.t);
```

```
    default:
      return 0.0;
    }
  }
  //---------------------------------------------------------------
  // Вычисление суммарного периметра для фигур,
  // размещенных в контейнере
  double Perimetr(container &c)
  {
    double a = 0;
    for(int i = 0; i < c.len; i++) {
      a += Perimetr(*(c.cont[i]));
    }
    return a;
  }
  //---------------------------------------------------------------
```

Что происходит при ОО подходе? Необходимо включить в базовый класс новую виртуальную процедуру, расширяющую исходный интерфейс. Далее требуется вставить во все производные классы методы, осуществляющие непосредственное вычисление периметров.

```
  //---------------------------------------------------------------
  // Класс, обобщающает все имеющиеся фигуры.
  // Изменился в связи с добавлением метода вычисления периметра
  class shape
  {
  public:
    virtual void In() = 0;     // ввод данных из стандартного потока
    virtual void Out() = 0;    // вывод данных в стандартный поток
    virtual double Area() = 0; // вычисление площади фигуры
    // добавлено вычисление периметра фигуры
    virtual double Perimetr() = 0;
  protected:
    shape() {};
  };
  //---------------------------------------------------------------
  // Измененный прямоугольник (вычисляет периметр)
  class rectangle: public shape
  {
    int x, y; // ширина, высота
  public:
    // переопределяем интерфейс класса
    void In();     // ввод данных из стандартного потока
    void Out();    // вывод данных в стандартный поток
    double Area(); // вычисление площади фигуры
    double Perimetr(); // добавлено вычисление периметра
    rectangle(int _x, int _y); // создание с инициализацией.
    rectangle() {} // создание без инициализации.
  };
  //---------------------------------------------------------------
  // Измененный треугольник
  class triangle: public shape
  {
    int a, b, c; // стороны
  public:
    // переопределяем интерфейс класса
    void In();     // ввод данных из стандартного потока
    void Out();    // вывод данных в стандартный поток
```

```
    double Area(); // вычисление площади фигуры
    double Perimetr(); // добавлено вычисление периметра фигуры
    triangle(int _a, int _b, int _c); // создание с инициализацией
    triangle() {} // создание без инициализации.
};
//-----------------------------------------------------------------
// Вычисление периметра прямоугольника
double rectangle::Perimetr() {
    return (x + y) * 2.0;
}
//-----------------------------------------------------------------
// Вычисление периметра треугольника
double triangle::Perimetr() {
    return a + b + c;
}
//-----------------------------------------------------------------
```

После этого, из-за изменения множества объектов необходимо перекомпилировать практически всю программу. Исходные тексты проделанной мною работы положены в архив oop_examp1_2.zip. Если эволюция программной системы зашла достаточно далеко и насчитывает несколько сотен альтернатив (чего мелочиться: будем манипулировать числами такого же порядка, как и при критике процедурного подхода), то объем проделанных изменений является достаточно впечатляющим. Если же вводимая в базовый класс процедура не является чистой, то можно легко упустить вставку реального кода в один из производных классов.

Однако, этот недостаток объектного подхода обычно считается не столь существенным. Во-первых, его сторонники не видят особой разницы между добавлением отдельных методов в классы и централизованным объединением множества новых независимых процедур в один огромный переключатель. И в том и в другом случаях можно легко что-то напутать. Ведь процедурный переключатель тоже займет не одну страницу. К тому же, последовательный анализ вариантов замедляет время выбора альтернативы. Во-вторых, модификация классов является более контролируемой во время трансляции, если добавлять чистый метод, ориентируясь на наследование интерфейса. Тогда сам компилятор будет сигнализировать о том, что тот или иной производный класс не содержит переопределения требуемой процедуры. Ну а если перекомпиляция и сборка проекта, после проделанной работы и займет пару суток, то освободившееся время всегда можно с пользой потратить на пару ящиков пива.

Другим методом решения этой проблемы, предлагаемым Бучем [Буч98], является более тщательное проектирование интерфейсов классов, осуществляемое до начала кодирования. По его мнению, именно проектированию интерфейсов надо уделять большее внимание, и тогда поставленная проблема почти отпадет. Метод, конечно разумный, хотя бы тем, что предлагает подумать и минимизировать дальнейшие затраты, прежде давить на клавиатуру. Однако, вряд ли он будет сильно полезен при экстремальном программировании [Beck]. Да и при любом методе проектирования больших систем всегда существует вероятность того, что ряд возможных функциональных расширений останутся неучтенными.

Пята вторая: добавление специализированных действий

Предположим, что нам надо выводить все прямоугольники, расположенные в контейнере. Соответствующая процедура должна "выявлять" прямоугольник из множества фигур всех видов и запускать специализированную процедуру вывода. Использование процедурного подхода позволяет добавить новый метод без каких-либо проблем, так как его отличие от других обработчиков альтернатив заключается только в анализе одного признака. Ее легко сформировать путем вырезания лишнего кода из уже существующей процедуры вывода произвольной альтернативы.

```
//-----------------------------------------------------------------
// Вывод только обобщенного прямоугольника
// Процедура добавлена без изменений других объектов
bool Out_rectangle(shape &s)
{
    switch(s.k) {
```

```
    case shape::key::RECTANGLE:
      Out(s.r);
      return true;
    default:
      return false;
    }
  }
  //---------------------------------------------------------------
```

Что из этого получилось, можно посмотреть в архиве pp_examp1_3.zip.

Объектно-ориентированный подход не позволяет, в данном случае, получить элегантное решение. Это связано с тем, что основной спецификой его использования является ориентация на массовое применение полиморфизма: когда множество всех объектов или один из этого множества объектов обрабатываются внешне одинаковым методом. Когда же требуется выделить специфический экземпляр производного класса, приходится прибегать к дополнительным ухищрениям.

Одним из наиболее эффективных и простых вариантов является использование динамического анализа типа объекта. Однако, как уже отмечалось выше, это процедурные штучки! Следовательно, однорукого программирования явно не хватает. Приходится, наряду с объектным обобщением, использовать и элементы вариантного обобщения, что ведет к смешению стилей и появлению признаков, сопровождающих классы по всей программе (даже тогда, когда эти признаки не используются). А раз так, то возможны проблемы, связанные с внутренними изменениями ряда таких процедур при добавлении новых производных классов. Эти проблемы, возможно, не столь критически, когда процедура обрабатывает только одну специализацию, что, скорее всего, является наиболее типичным случаем. Хотя, исключать возможность появления внутри таких процедур нескольких различных специализаций не стоит.

Менее изящным, но чисто объектным решением является перенос интерфейса специализированного обработчика в базовый класс и закрепления за ним функций ничего неделания, используя для этого, например, пустое тело метода (в нашем случае возвращается булевский признак, указывающий на отсутствие вывода прямоугольника). Такой подход используется даже в образцах проектирования [Гамма]. Переопределение метода только в нужном производном классе позволяет решить проблемы специализации. Ниже показано, как переопределяется вывод только прямоугольника.

```
//---------------------------------------------------------------
// Класс, обобщающает все имеющиеся фигуры.
class shape {
public:
  virtual void In() = 0;     // ввод данных из стандартного потока
  virtual void Out() = 0;    // вывод данных в стандартный поток
  // Добавлен вывод только прямоугольника как заглушка
  // Метод не является чистым и вызывается там где не переопределен
  virtual bool Out_rectangle() {
    return false;
  };
  virtual double Area() = 0; // вычисление площади фигуры
protected:
  shape() {};
};
//---------------------------------------------------------------
// Прямоугольник переопределяет вывод себя
class rectangle: public shape
{
  int x, y; // ширина, высота
public:
  // переопределяем интерфейс класса
  void In();     // ввод данных из стандартного потока
  void Out();    // вывод данных в стандартный поток
  // Переопределен вывод только прямоугольника
  bool Out_rectangle(); // вывод только прямоугольника
```

```
    double Area(); // вычисление площади фигуры
    rectangle(int _x, int _y); // создание с инициализацией.
    rectangle() {} // создание без инициализации.
};
//-----------------------------------------------------------
// Вывод только прямоугольника
bool rectangle::Out_rectangle() {
    Out();
    return true;
};
//-----------------------------------------------------------
```

В архиве oop_examp1_3.zip показано, как такой подход обеспечивает вывод всех прямоугольников. Основным недостатком примененного технического решения является "разбухание" интерфейсов базового и производных классов. Ему приходится поддерживать множество методов, полезных только в специфичных ситуациях. Наряду с этим возникают проблемы, связанные модификацией базового класса и полной перекомпиляции всех зависимостей при добавлении каждого нового специализированного метода.

В ряде случаев проблему можно решать и чисто программистскими методами, связанными с добавлениями в программу новых объектов. Например, для поддержки вывода только прямоугольников, можно ввести специальный контейнер, предназначенный для их хранения. Но в каждом из таких случаев следует внимательно учитывать специфику решаемой задачи.

Как "прикинуться" объектным обобщением

Использование процедурного подхода не позволяет явно "запихнуть" методы в агрегаты или обобщения. Однако практически любой процедурный язык, имеющий развитые абстрактные типы данных, позволяет легко сымитировать объектное обобщение. Именно этот подход используется при трансляции объектно-ориентированных программ в машинный код. Здесь стоит вспомнить компилятор cfront, разработанный Страуструпом [Страуструп2000]. Пример кода, порождаемого подобным компилятором можно подсмотреть в книге Голуба [Голуб]. Подобный прием наверняка использовался многими программистами еще до наступления эры ООП с целью создания более гибкой программы. Правда, я не согласен с высказыванием Страуструпа, что компилятор в данном случае порождает более эффективный код, чем тот, который может написать программист. Как и в случае программирования на ассемблере, имитация ОО стиля в процедурном подходе позволяет написать более эффективную программу, если учитывать при этом специфику решаемой задачи. Но также нет сомнений, что эта эффективность будет достигаться с большими затратами. А отсутствие контроля во время компиляции ведет к тому, что многие ошибки будут выявлены только при выполнении программы.

Почему-то я решил, что мои заметки не будут полными без простенькой процедурной программки в ОО стиле. Поэтому, не уделяя внимания оптимизации, я привожу свой вариант для рассматриваемого примера. Надеюсь, что программа наглядно демонстрирует те затраты, от которых удалось избавиться, перейдя на объектно-ориентированные языки. Небольшой нюанс проявляется в том, что я попытался по максимуму использовать структуры данных, написанные ранее для вариантного обобщения. Таблица виртуальных функций базового класса ***shape*** в данном случае выделена в отдельную структуру, что позволяет на ее основе генерировать конкретные таблицы виртуальных функций для вновь создаваемых специализаций. Код моделируемого объектного обобщения выглядит следующим образом:

```
//-----------------------------------------------------------
// Структура таблицы виртуальных функций базоваого класса
// Содержит указатели на функции, переопределяемые
// в моделях производных классов.
struct shape_vtbl {
    void (*In)(shape *_this);    // ввод данных из стандартного потока
    void (*Out)(shape *_this);     // вывод данных в стандартный поток
    double (*Area)(shape *_this);  // вычисление площади фигуры
    void (*Destroy)(shape *_this); // удаление обобщенной фигуры
};
//-----------------------------------------------------------
// Структура, обобщающая все имеющиеся фигуры.
```

```
// Моделирует абстрактный базовый класс.
  struct shape {
    shape_vtbl *vtbl; // Указатель на таблицу виртуальных функций
};
//---------------------------------------------------------------
```

Имитируя наследование, можно построить прямоугольник и треугольник.

```
//---------------------------------------------------------------
// Функции, инициализирующие таблицу прямоугольника
void In_shape_rectangle(shape* _this); // ввод данных
void Out_shape_rectangle(shape* _this); // вывод данных
double Area_shape_rectangle(shape* _this); // вычисление площади
void Destroy_shape_rectangle(shape* _this); // удаление фигуры
//---------------------------------------------------------------
// Таблица виртуальных функций прямоугольника.
// Задается одна для всех прямоугольников. Спрятана в файле
// Инициализируется специально сформированными расширяющими функциями
static shape_vtbl rect_vtbl = {
   &In_shape_rectangle,     // ввод данных из стандартного потока
   &Out_shape_rectangle,    // вывод данных в стандартный поток
   &Area_shape_rectangle,   // вычисление площади фигуры
   &Destroy_shape_rectangle // удаление динамически созданной фигуры
};
//---------------------------------------------------------------
// Структура, моделирующая прямоугольник - наследник фигуры.
struct shape_rectangle {
   shape base;  // База специализированной фигуры
   rectangle r; // Значимая часть. Может быть написана независимо.
};
//---------------------------------------------------------------
// Функции, инициализирующие таблицу треугольника
void In_shape_triangle(shape* _this);     // ввод данных
void Out_shape_triangle(shape* _this);    // вывод данных
double Area_shape_triangle(shape* _this); // вычисление площади фигуры
void Destroy_shape_triangle(shape* _this); // удаление фигуры
//---------------------------------------------------------------
// Таблица виртуальных функций треугольника.
// Задается одна для всех треугольников. Спрятана в файле
// Инициализируется специально сформированными расширяющими функциями
static shape_vtbl trian_vtbl = {
   &In_shape_triangle,     // ввод данных из стандартного потока
   &Out_shape_triangle,    // вывод данных в стандартный поток
   &Area_shape_triangle,   // вычисление площади фигуры
   &Destroy_shape_triangle // удаление динамически созданной фигуры
};
//---------------------------------------------------------------
// Структура, моделирующая треугольник - наследник фигуры.
struct shape_triangle {
   shape base;  // База специализированной фигуры
   triangle t; // Значимая часть. Может быть написана независимо.
};
//---------------------------------------------------------------
```

Таблицы виртуальных функций имитируются посредством соответствующих статических переменных. Это приводит к тому, что существует несколько одинаковых таблиц размещенных в разных единицах компиляции. Но я специально не стал вводить одну глобальную переменную, чтобы не думать о том, в каком файле ее определить, а в каких - объявить. Кстати, такой прием может использоваться и компилятором [Страуструп2000].

В методах фигур-наследников используется явное приведение обобщающего типа к типу специализаций. После этого осуществляется доступ к структуре, определяющей конкретную фигуру и обработка ее данных ранее написанными методами. Техника достаточно проста, а исходные тексты всего примера размещены в архиве pp_examp1o.zip.
Следует отметить, что окончательная структура объекта, имитирующего класс, формируется только во время выполнения программы. Это связано с тем, что таблица виртуальных функций связывается через указатель, значение которому может быть присвоено только соответствующим кодом. Этот код размещается в процедурах создания и инициализации структур, определяющих производные типы данных. Вот как они выглядят:

```
//-----------------------------------------------------------------
void Init(rectangle &r, int x, int y);
// Динамическое создание прямоугольника - наследника по двум сторонам
shape_rectangle *Create_shape_rectangle(int x, int y)
{
   shape_rectangle *sr = new shape_rectangle;
   // Привязка к виртуальной таблице
   sr->base.vtbl = &rect_vtbl;
   Init(sr->r, x, y);
   return sr;
}
// Инициализация прямоугольника - наследника по двум сторонам
void Init_shape_rectangle(shape_rectangle &sr, int x, int y)
{
   // Привязка к виртуальной таблице
   sr.base.vtbl = &rect_vtbl;
   Init(sr.r, x, y);
}
//-----------------------------------------------------------------
void Init(triangle &t, int a, int b, int c);
// Динамическое создание треугольника - наследника по трем сторонам
shape_triangle *Create_shape_triangle(int a, int b, int c)
{
   shape_triangle *st = new shape_triangle;
   // Привязка к виртуальной таблице
   st->base.vtbl = &trian_vtbl;
   Init(st->t, a, b, c);
   return st;
}
// Инициализация треугольника - наследника по трем сторонам
void Init_shape_triangle(shape_triangle &st, int a, int b, int c)
{
   // Привязка к виртуальной таблице
   st.base.vtbl = &trian_vtbl;
   Init(st.t, a, b, c);
}
//-----------------------------------------------------------------
```

Наличие такой динамической привязки к таблице указателей на функции объясняет, почему классы должны иметь процедуры по умолчанию.

Другой вариант

Кстати, можно обойтись и без внешней таблицы виртуальных функций, разместив указатели на сменяемые функции внутри структуры, имитирующей базовый класс, который будет выглядеть следующим образом:

```
//-----------------------------------------------------------------
// Структура, обобщающая все имеющиеся фигуры.
// Моделирует абстрактный базовый класс.
```

```
struct shape {
  // Непосредственное размещение указателей на заменямые функции
  // вместо таблицы виртуальных функций
  void (*In)(shape *_this);      // ввод данных
  void (*Out)(shape *_this);     // вывод данных
  double (*Area)(shape *_this);  // вычисление площади фигуры
  void (*Destroy)(shape *_this); // удаление обобщенной фигуры
};
//---------------------------------------------------------------
```

В этом случае можно обойтись без статических или глобальных переменных, определяющих таблицы виртуальных функций для специализаций обобщения:

```
//---------------------------------------------------------------
// Структура, моделирующая прямоугольник - наследник фигуры.
struct shape_rectangle {
   shape base;  // База специализированной фигуры
   rectangle r; // Значимая часть. Может быть написана независимо.
};
//---------------------------------------------------------------
// Структура, моделирующая треугольник - наследник фигуры.
struct shape_triangle {
   shape base;  // База специализированной фигуры
   triangle t; // Значимая часть. Может быть написана независимо.
};
//---------------------------------------------------------------
```

Но в этом случае формирование связей с обработчиками специализаций придется осуществлять во время инициализации. То есть, модель конструктора производного объекта станет более громоздкой. Вот как они могут выглядеть для рассмотренных структур:

```
//---------------------------------------------------------------
void Init(rectangle &r, int x, int y);
// Функции, инициализирующие таблицу прямоугольника
void In_shape_rectangle(shape* _this);  // ввод данных
void Out_shape_rectangle(shape* _this); // вывод данных
double Area_shape_rectangle(shape* _this); // вычисление площади фигуры
void Destroy_shape_rectangle(shape* _this); // удаление фигуры
// Динамическое создание прямоугольника - наследника по двум сторонам
shape_rectangle *Create_shape_rectangle(int x, int y)
{
   shape_rectangle *sr = new shape_rectangle;
   sr->base.In = &In_shape_rectangle;
   sr->base.Out = &Out_shape_rectangle;
   sr->base.Area = &Area_shape_rectangle;
   sr->base.Destroy = &Destroy_shape_rectangle;
   Init(sr->r, x, y);
   return sr;
}
// Инициализация прямоугольника - наследника по двум сторонам
void Init_shape_rectangle(shape_rectangle &sr, int x, int y)
{
   sr.base.In = &In_shape_rectangle;
   sr.base.Out = &Out_shape_rectangle;
   sr.base.Area = &Area_shape_rectangle;
   sr.base.Destroy = &Destroy_shape_rectangle;
   Init(sr.r, x, y);
}
//---------------------------------------------------------------
void Init(triangle &t, int a, int b, int c);
```

```
// Функции, инициализирующие таблицу треугольника
void In_shape_triangle(shape* _this); // ввод данных
void Out_shape_triangle(shape* _this); // вывод данных
double Area_shape_triangle(shape* _this); // вычисление площади фигуры
void Destroy_shape_triangle(shape* _this); // удаление фигуры
// Динамическое создание треугольника - наследника по трем сторонам
shape_triangle *Create_shape_triangle(int a, int b, int c)
{
   shape_triangle *st = new shape_triangle;
   st->base.In = &In_shape_triangle;
   st->base.Out = &Out_shape_triangle;
   st->base.Area = &Area_shape_triangle;
   st->base.Destroy = &Destroy_shape_triangle;
   Init(st->t, a, b, c);
   return st;
}
// Инициализация треугольника - наследника по трем сторонам
void Init_shape_triangle(shape_triangle &st, int a, int b, int c)
{
   st.base.In = &In_shape_triangle;
   st.base.Out = &Out_shape_triangle;
   st.base.Area = &Area_shape_triangle;
   st.base.Destroy = &Destroy_shape_triangle;
   Init(st.t, a, b, c);
}
//---------------------------------------------------------------
```

Этот подход позволяет ускорить выполнение программы, так как отсутствует дополнительное обращение через ссылку к таблице виртуальных функций. Однако эффект достигается за счет расточительного использования памяти, так как каждая структура должна содержать все указатели. Кроме того, усложняются и становятся более медленными функции, выполняющие конструирование, так как теперь необходимо осуществлять привязку обработчиков специализаций для каждого создаваемого объекта, имитирующего производный класс. К сожалению, здесь нельзя воспользоваться статическими полями структуры. Исходные тексты примера размещены в архиве pp_examp1od.zip.

Заключительные ассоциации

Сопоставление процедурного и объектно-ориентированного подходов показывает их самодостаточность при написании "чистого парадигматического" кода. Однако необходимость эффективного эволюционного развития программы и повторное использование некоторых ее фрагментов ведут, на практике, к совместному использованию этих парадигм. Программирование уже давно является многоруким независимо от того, каким считается язык: процедурным, объектным или мультипарадигматическим. А то, что объектно-ориентированная методология трактует со своих позиций сочетание различных стилей, не столь важно с точки зрения парадигм. В конце концов, любой компилятор (по крайней мере, для C++) транслирует написанную программу в обычный процедурный код.

Повторяются библейские истории. Было время, когда процедурный и объектный подход мирно сосуществовали под одной крышей, названной строителями этого очага "Структурное программирование" [Дал75]. Однако в дальнейшем им стало тесно, и молодой, объектно-ориентированный, Каин, дав в морду своему брату Авелю, покинул семейный очаг, спровоцировав Великое противостояние и Вавилонское столпотворение. Однако, даже если нам кажется, что разговор идет на одном языке, мы все равно используем смешение стилей, когда пишем реальный код.

Краткое содержание этой серии

В заключение этого раздутого текста хотелось бы кратко выразить то, о чем я пытался так долго говорить.

1. В предлагаемых заметках анализируется техника построения и использования программных объектов. При этом я попытался абстрагироваться от методологических, ресурсных и функциональных аспектов.
2. Рассматривается применение техники построения программных объектов в эволюционном программировании (кодировании). Упор сделан на процедурное и объектно-ориентированное программирование. Эволюционное программирование используется при разработке больших программных систем. Оно позволяет расширять функциональные и структурные возможности программы с минимальными изменениями уже написанного кода.
3. Проведено разделение техники кодирования на процедурную и объектно-ориентированную. Это чисто субъективное решение основано на использовании ряда методов построения программ в "дообъектную" эпоху.
4. Независимо от техники и парадигм программирования мы создаем программные объекты и отношения между ними, используя такие понятия как агрегат и обобщение. К одному и тому же конечному результату можно прийти различными путями.
5. Рассмотрена техника создания агрегатов и обобщений при процедурном и объектно-ориентированном подходах. Сделана попытка проиллюстрировать ее серий мелких и тривиальных примеров.
6. Процедурное агрегирование обладает большей гибкостью по сравнению с объектно-ориентированным, а метафора автономного полнофункционального объекта не всегда удобна, так как вступает в противоречие с ассоциациями, необходимыми при построении эволюционно наращиваемых программ. При этом можно, используя процедурный подход, вообразить, что мы работам с объектами, или изменить язык программирования таким образом, чтобы обычные внешние процедуры "прикидывались" внутренними методами объектов.
7. Объектное обобщение обладает неоспоримыми преимуществами при создании эволюционно расширяемых альтернатив и агрегатов с неизменяемым интерфейсом. Эти достоинства, на мой взгляд, и предопределили ход дальнейшего практического использования парадигм программирования. Хотя, и у объектных обобщений имеется ряд недостатков, вместо исправления которых проще использовать методы процедурного программирования.
8. Высокоэффективное эволюционное программирование должно быть многоруким. Иначе, оно не будет эффективным. Поэтому, даже в языках, считающихся чисто объектными, всегда существуют механизмы поддержки процедурного стиля, к коим можно отнести анализ типа объекта во время выполнения (даже, если внешне он прикидывается внутренним методом).
9. Будущее за мультипарадигматическим программированием (по Страуструпу). Следовательно, и методологии скоро вновь начнут "плыть" в сторону поддержки смешанных способов представления программных объектов, обеспечивая большую гибкость при проектировании.

Список использованных источников

1. [Appleton] Appleton Brad. Patterns and Software: Essential Concepts and Terminology. Электронная версия документа расположена по адресу: http://www.enteract.com/~bradapp/docs/patterns-intro.html.
2. [Bobrow] Bobrow D., Stefik M. Perspectives on Artificial Intelligence Programming. Science vol. 231, p. 951, February 1986.
3. [Buschmann] Buschmann F, Meunier R, Rohnert H, Sommerlad P, Stal M. Pattern-Oriented Software Architecture: A System of Patterns, Wiley, Chichester UK, 1996.
4. [Gamma] Gamma E., Helm R., Johnson R., Vlissides J. Design Patterns: Elements of Reusable Object-Oriented Software. - ISBN 0-201-63442-2 Hardback, 416 pages ©1995.
5. [Goldberg] Goldberg A., Robson D. Smalltalk-80: The Language. Reading, MA: Addison-Wesley, 1989.
6. [Hejlsberg] Hejlsberg A., Scott W. C# Language Reference.
7. [Jackson] Jackson M. Principles of Program Design. London, Academic Press, 1975.
8. [Beck]Beck Kent. Extreme Programming Explained: Embrace Change. - Addison Wesley, 190 pages, 2000.
9. [King] King G. Object-Oriented really is better than Structured. Электронная версия документа расположена по адресу: http://www-eksl.cs.umass.edu/~gwking/whyoop.htm.
10. [Kuzmin…] Kuzmin D.A., Kazakov F.A., Legalov A.I. Description of parallel-functional programming language. - Advances in Modeling & Analysis, A, AMSE Press, Vol.28, N3, 1995, pp.1-17.
11. [Meyer] Meyer M. Object-Oriented Software Construction. Second Edition. ISE Inc. Santa Barbara (California).
12. [Meyers] Meyers Scott. How Non-Member Functions Improve Encapsulation.
13. [MSF] Microsoft Solutions Framework
14. [Moessenboeck] Moessenboeck H. Object-Oriented Programming in Oberon-2. - / Springer-Verlag. (c) 1993
15. [MoessenboeckWirth] Moessenboeck H., Wirth N. The Programming Language Oberon-2. Institut fur Computersysteme, ETH Zurich July 1996.
16. [Wirth] Wirth N. The Programming Language Oberon. Электронная версия документа расположена по адресу: ftp://ftp.inf.ethz.ch/pub/software/Oberon/OberonV4/Docu/OberonReport.Text
17. [Алгоритмы] Алгоритмы, математическое обеспечение и проектирование архитектур, многопроцессорных вычислительных систем. Под ред. А.П. Ершова. - М: Наука, 1982.
18. [Андрианов] Андрианов А.Н., Бычков С.П., Хорошилов С.И. Программирование на языке симула-67. – М.: Наука. Глав. ред. физ.-мат. лит., 1985. – 288 с.
19. [АРНФС] Англо-русско-немецко-французский толковый словарь по вычислительной технике и обработке данных, 4132 термина. Под. ред. А.А. Дородницына. М.: 1978. 416 с.
20. [Архангельский] Архангельский А. Программирование в Borland C++ Builder 4. - М.: "Бином". - 928 с.
21. [Барендрегт] Барендрегт Х. Ламбда-исчисление. Его синтаксис и семантика. /Пер. с англ. - М.: Мир, 1985. - 606 с.
22. [Бек] Бек Кент. Экстремальное программирование. Открытые системы, №1-2, 2000. Адрес электронной версии документа: http://www.osp.ru/os/2000/1-2/059.htm.
23. [Бердж] Бердж В. Методы рекурсивного программирования. /Пер. с англ. - М.: Машиностроение, 1983. - 248 с.

24. [Боэм] Боэм Б., Браун Дж., Каспар Х. и др. Характеристики качества программного обеспечения. /Пер. с англ. М.: Мир, 1981. - 208 с., ил.

25. [Буч92] Буч Г. Объектно-ориентированное проектирование с примерами применения. /Пер. с англ. - М.: Конкорд, 1992. - 519 с., ил.

26. [Буч98] Буч Г. Объектно-ориентированный анализ и проектирование с примерами приложений на C++, 2-е изд./Пер. с англ. - М.: "Издательства Бином", СПб: "Невский диалект", 1998 г. - 560 с., ил.

27. [Васкевич] Васкевич Д. Стратегии клиент/сервер. Руководство по выживанию для специалистов по реорганизации бизнеса. - К.: "Диалектика", 1996. - 384 с.

28. [Вендров] Вендров А.М. CASE - технологии. Современные методы и средства проектирования информационных систем. - Сервер информационных технологий (www.citforum.ru).

29. [Вирт85] Вирт Н. Алгоритмы + структуры данных = программы. - М.: Мир, 1985.

30. [Вирт87] Вирт Н. Программирование на языке Модула-2. /Пер. с англ. - М.: Мир, 1987. – 244 с.

31. [Вирт89] Вирт Н. Алгоритмы и структуры данных. - М.: Мир, 1989.

32. [Вирт96] Вирт Н. Долой "жирные" программы. "Открытые системы", №6 (20), 1996, стр. 26-31.

33. [Гамма] Гамма Э., Хелм Р., Джонсон Р., Влиссидес Дж. Приемы объектно-ориентированного пректирования. Паттерны проектирования. /Пер. с англ. - СПб: Питер, 2001. - 368 с.

34. [Голуб] Голуб А.И. C, C++. Правила программирования. М: Бином. 1996. - 272 с.

35. [Гофман] Гофман В., Хомоненко А. Delphi 5 в подлиннике. - СПб.: "BHV – Санкт-Петербург", 1999. - 800 с.

36. [Дал69] Дал У.И., Мюрхауг Б., Нюгорд К. Симула-67. Универсальный язык программирования. - М.: Мир, 1969. - 99 с.

37. [Дал75] Дал У., Дейкстра Э., Хоор К. Структурное программирование. /Пер с англ. - М.: Мир, 1975. - 247 с.

38. [Дарнел] Дарнел JavaScript. Справочник. - СПб.: "Питер", 2000. - 192 с.

39. [Дейкстра78] Дейкстра Э. Дисциплина программирования./Пер. с англ. - М.: Мир, 1978.

40. [Джехани] Джехани Н. Язык Ада. /Пер. с англ. – М.: Мир, 1988. – 522 с.

41. [Джоунз] Джоунз Г. Программирование на языке Оккам: Пер. с англ. - М.: Мир, 1989. - 208 с.

42. [Зиглер] Зиглер К. Методы проектирования программных систем: Пер. с англ. - М.: Мир, 1985. -328 с.

43. [Йордон] Йордон Э. Путь камикадзе. Как разработчику программного обеспечения выжить в безнадежном проекте. /Пер. с англ. - М.: Издательство "ЛОРИ", 2000. - 256 с.

44. [Казаков…] Казаков Ф.А., Кузьмин Д.А., Легалов А.И. Параллельный язык управления потоков данных. - Математическое обеспечение и архитектура ЭВМ: Сб. научных работ. Вып. 2. КГТУ, Красноярск, 1997. с. 105-113.

45. [Катленд] Катленд Н. Вычислимость. Введение в теорию рекурсивных функций. /Пер. с англ. - М.: Мир, 1983. - 256 с.

46. [Кауфман] Кауфман В.Ш. Языки программирования. Концепции и принципы. - М.: Радио и связь, 1993. - 432 с.

47. [Легалов] Легалов А.И., Казаков Ф.А., Кузьмин Д.А. Водяхо А.И. Модель параллельных вычислений функционального языка. Известия ГЭТУ, Сборник научных трудов. Выпуск 500. Структуры и математическое обеспечение специализированных средств. С.-Петербург, 1996. с. 56-63.

48. [Легалов97-1] Легалов А.И. Разработка программ на основе объектно-реляционной методологии. - Математическое обеспечение и архитектура ЭВМ: Сб. научных работ. Вып. 2. КГТУ, Красноярск, 1997. с. 223-235.

49. [Легалов97-2] Легалов А.И. Сочетание процедурного и объектного подходов при разработке программ. - Вестник Красноярского государственного технического университета. Сб. научных трудов. Вып. 10. Красноярск, 1997. с. 102-109.

50. [Легалов99] Легалов А.И. Модель конструирования понятий. - Достижения науки и техники - развитию сибирских регионов: Тезисы докладов Всероссийской научно-практической конференции с международным участием; В 3 ч. Ч 3. Красноярск, КГТУ, 1999. с. 197-198.

51. [Легалов99-2] Легалов А.И. Процедурно-параметрическое программирование. - Проблемы информатизации региона. ПИР-99: Сб. научных трудов пятой Всероссийской научно-практической конференции. Красноярск: КГТУ, 1999. с. 13-27.

52. [Лег2000-1] Легалов А.И. Процедурно-параметрическая парадигма программирования. Возможна ли альтернатива объектно-ориентированному стилю? - Красноярск: 2000. Деп. рук. № 622-В00 Деп. в ВИНИТИ 13.03.2000. - 43 с.

53. [Лингер] Лингер Р., Миллс Х., Уитт Б. Теория и практика структурного программирования. - М.: Мир, 1982. - 406 с.

54. [Маклаков] Маклаков С.В. BPwin и ERwin. CASE-средства разработки информационных систем. - М.: ДИАЛОГ-МИФИ, 1999. - 256 с.

55. [Маслов] Маслов В.В. Основы программирования на языке Перл. - М.: Радио и связь, Горячая линия - Телеком. 1999. - 144 с.

56. [Маурер] Маурер У. Введение в программирование на языке ЛИСП. - М.: Мир, 1976. - 104 с.

57. [Мейерс2000-1] Мейерс С. Эффективное использование C++. 50 рекомендаций по улучшению ваших программ и проектов: Пер. с англ. - М.: ДМК, 2000. - 240 с.

58. [Мейерс2000-2] Мейерс С. Наиболее эффективное использование C++. 35 новых рекомендаций по улучшению ваших программ и проектов: Пер. с англ. - М.: ДМК Пресс, 2000. - 304 с.

59. [Нортон] Нортон П. Программирование на Java. Руководство П.Нортона (в 2-х томах). "СК-Пресс", 1998 - 900 с.

60. [Остераут] Остераут Дж. Сценарии: высокоуровневое программирование для XXI века. Открытые системы, №3, 1998.

61. [Пешио] Пешио К. Никлаус Вирт о культуре разработки ПО. "Открытые системы", №1, 1998, с. 41-44.

62. [Райтингер] Райтингер М. Visual Basic 6: полное руководство. - Киев: "BHV-Киев", 1999. - 720 с.

63. [Роджерсон] Роджерсон Д. Основы COM / Пер. с англ. - М.: Издательский отдел "Русская редакция" ТОО "Channel Trading Ltd.", 1997. - 376 с.

64. [Стерлинг] Стерлинг Л., Шапиро Э. Искусство программирования на языке Пролог /Пер с англ. - М.: Мир, 1990. - 235 с.

65. [Страуструп] Страуструп Б. Язык программирования C++. Третье издание. /Пер. с англ. - СПб.; М.: "Невский диалект" - "Издательство БИНОМ", 1999. - 991 с.

66. [Страуструп2000] Страуструп Б. Дизайн и эволюция C++: Пер. с англ. - М.: ДМК Пресс, 2000. - 448 с.

67. [Фролов] Фролов Г.Д., Олюнин В.Ю. Практический курс программирования на языке PL/1. – М.: Наука, 1983.

68. [Хендерсон] Хендерсон. Функциональное программирование. /Пер. с англ. М.: Мир, 1983.

69. [Хиллер] Хиллер С. Microsoft Visual Basic, Scripting Edition в действии. /Пер. с англ. - СПб.: Питер, 1997. - 448 с.

70. [Хоар] Хоар Ч. Взаимодействующие последовательные процессы. Пер. с англ. - М.: Мир, 1989. -264 с.

71. [Цикритзис] Цикритзис Д., Лоховски Ф. Модели данных. Пер. с англ. - М.: Финансы и статистика, 1985. - 344 с.

72. [Шагурин] Шагурин И.И., Бродин В.Б., Мозговой Г.П. 89386: описание и система команд. – М.: МП “Малип”, 1992. – 160 с.

73. [Элджер] Элджер Дж. C++: библиотека программиста. Пер. с англ. - СПб.: ЗАО "Издательство Питер", 1999. - 320 с.

74. [Языки] Языки программирования. Под ред. Женюи Ф. /Пер. с англ. М.: Мир, 1972. 406 с.