Порядок ради порядка выхолащивает человека, лишая его основного дара – преображать мир и самого себя. Жизнь творит порядок, но порядок жизнь не творит.

Антуан де Сент-Экзюпери «Письмо к заложнику»

Основные положения

В этом письме будут рассмотрены концептуальные положения объектной технологии. Дело в том, что многие исследователи не только рассматривают различные положения, но и трактовка этих положений существенно различается. Как следствие получается расхождение в объектных моделях и, соответственно, в возможностях заложенных в этих моделях. Очень часто объектную модель воплощают в каком-то одном языке, и оценка модели основывается на оценке возможностей предоставляемых языком. Такая практика порочна по своему существу. Дело в том, что одна и та же модель может быть различно реализована в различных языках программирования. Поэтому более правильно обсуждать различные объектные модели и только в рамках одинаковых моделей сравнивать языки. Это с одной стороны. С другой стороны, сегодня надо отходить от универсальных языков высокого уровня, одинаково пригодных для любого применения. Современные системы слишком сложны и разносторонни, чтобы их можно было эффективно создавать с помощью какого-то одного языка. Никого не удивляет тот факт, что создание приложений ведётся с помощью 3GL. Работа с базами данных из того же приложения происходит с помощью SQL, реже с помощью QBE. Создание интерфейса с пользователем осуществляется на специальном языке (те, кто пишет под Windows, могут посмотреть на структуру RC-файлов). Наконец, межпрограммные и межкомпьютерные коммуникации реализуются посредством прикладного API, который тоже представляет собой псевдоязык. Список можно продолжить, рассматривая более специализированные системы и/или подсистемы.

При рассмотрении объектных моделей обычно выделяют три основных концептуальных положения: инкапсуляция, полиморфизм и наследование. Действительно, эти положения являются наиболее важными для описания основ объектной технологии, хотя их одних недостаточно. Поэтому в этом письме будут рассмотрены эти три положения, а следующее письмо введёт в круг основных положений агрегацию и следующие из неё положения: конструирование, локализация и абстракция.

Описание концептуальных положений важно не само по себе, а потому, что оно создаёт представление о том, какие возможности заложены в данную технологию, на какие горизонты эта технология может вывести процесс разработки программного обеспечения. Следует отметить, что к настоящему времени объектная технология не реализовала тот огромный потенциал, который в ней заложен, и связано это, прежде всего, с тем, что концептуальные положения, лежащие в основе этой технологии не раскрываются полностью или трактуются ошибочно.

Сложные системы

Объектная технология открывает возможность построения гораздо более сложных систем и программных комплексов, чем допускала технология структурного программирования. Разница в подходах к разработке принципиальна и она базируется именно на высокой сложности современного программного обеспечения. Здесь возможности структурного программирования и, основанных на нём, подходов к созданию программного обеспечения оказались недостаточны. Как результат применение структурной технологии приводит к слишком продолжительному циклу разработки, сложности в модернизации и управлении, сокращённому жизненному циклу и повышенной стоимости.

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

Наверное, имеет смысл сказать ещё несколько слов о сложных системах, создание которых становится возможным на основе объектной технологии. Г. Буч в своей книге «Объектно-ориентированный анализ и проектирование с примерами приложений на C++» со ссылкой на Брукса (Brooks, F. April 1987. No Silver Bullet: Essence and Accidents of Software Engineering. IEEE Computer vol. 20(4), p12.) отмечает, что «сложность вызывается четырьмя основными причинами: сложностью реальной предметной области, из которой исходит заказ на разработку; трудностью управления процессом разработки; необходимостью обеспечить достаточную гибкость программы; неудовлетворительными способами описания поведения больших дискретных систем» [стр. 22]. Далее Г. Буч, со ссылкой на Куртуа, (Courtois, P. June 1985. On Time and Space Decomposition of Complex Structures. Communications of the ACM vol. 28(6), p. 596) выделяет пять признаков сложной системы [стр. 29-30]:

1. Сложные системы часто являются иерархическими и состоят из взаимосвязанных подсистем, которые в свою очередь могут быть разделены на подсистемы, и т.д., вплоть до самого низкого уровня.
2. Выбор, какие компоненты в данной системе считаются элементарными, относительно произволен и в большей степени оставляется на усмотрение исследователя.
3. Внутрикомпонентная связь обычно сильнее, чем связь между компонентами. Это обстоятельство позволяет отделять «высокочастотные» взаимодействия внутри компонентов от «низкочастотной» динамики взаимодействия между компонентами.
4. Иерархические системы обычно состоят из немногих типов подсистем, по-разному скомбинированных и организованных.
5. Любая работающая сложная система является результатом развития работавшей более простой системы… Сложная система, спроектированная с «нуля», никогда не заработает. Следует начинать с работающей простой системы.

Следует отметить некоторую случайность в этих пяти признаках и их плохую логическую связь. Сложные системы в программировании ничем не отличаются от других сложных систем, таких как биологические, социальные, технические и т.п. Сложность системы можно выразить в уровнях управления, чем сложнее система, тем больше уровней управления она имеет. Отсюда происходит тезис об иерархичности сложных систем. Однако большое количество уровней не означает, что те задачи, которые решает система, не могут быть решены более простой системой, с меньшим количеством уровней управления.

Любая сложная система состоит из некоторого количества элементарных компонент. Типов компонент не может быть много, но экземпляры этих компонент могут произвольно комбинироваться, отражая многообразие мира. Здесь вполне уместно вспомнить, что все многообразие цветов и оттенков обеспечивается комбинацией всего из трёх основных цветов; всё многообразие предприятий, не зависимо от их специализации, формы собственности, географического положения и т.п., обеспечивается комбинацией трёх составных частей: люди, оборудование и материалы. Все механизмы представляют собой комбинации групп Ассура, а все вещества состоят из элементов таблицы Менделеева. Таких примеров можно привести множество для любой предметной области, но при этом важно отметить, что выбор элементарных составляющих не может быть произвольным или случайным. От правильности выбора элементарных составляющих зависит принципиальная возможность создания системы и её сложность. Но данный тезис реально изменяет понятие сложности системы для любой предметной области. Система сложна

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

настолько, насколько она непонятна, и непонятна настолько, насколько нам сложно вычленить её элементарные компоненты и определить связи между ними.

Сложная система, как было отмечено ранее, отличается сложностью управления, то есть сложностью связей между компонентами. Каждый уровень управления обладает собственной логикой, которая не зависит от логики управления других уровней. Взаимодействие между уровнями управления в любой (не только программной) сложной системе основано на декларированном интерфейсе каждого уровня. Создание интерфейсов и связей на каждом уровне управления представляет собой отдельную задачу и требует отдельных проектных решений. Создание и развитие каждого уровня может и должно вестись параллельно с созданием и развитием других уровней. Развитие любого уровня начинается с декларации его интерфейса и только после определения интерфейса и увязки его с интерфейсами смежных уровней управления, можно переходить к реализации всех уровней. Интерфейс представляет собой формальную спецификацию возможностей данного логического уровня. Этот тезис переформулирует положение о трудностях, связанных с процессом разработки. Трудность разработки зависит от того, насколько точно определены основные логические уровни и того, насколько хорошо формализованы интерфейсы всех уровней.

Как следствие, сложная система может создаваться с «нуля» и развиваться произвольно долго, расширяя, углубляя и совершенствуя свои функциональные возможности. Но для выполнения этого требования необходимо тщательное соблюдение инкапсуляции, дабы отдельные компоненты системы можно было безболезненно заменять непосредственно во время её работы. Таким образом, инкапсуляция обеспечивает развитие и модифицируемость, то есть, гибкость системы. Но об этом речь пойдёт ниже.

Претензии к неудовлетворительным способам описания поведения больших дискретных систем нельзя воспринимать однозначно. До тех пор, пока делаются попытки описания системы в целом, без разделения её на отдельные уровни (деление на горизонтальные уровни) и подсистемы/компоненты/объекты (деление по вертикали), такие претензии безусловно оправданы. Однако тщательное деление и разработка логики каждого уровня не только серьёзно упрощает задачу, но и выявляет несостоятельность этих претензий.

Дополнительно хотелось бы отметить, что любая сложная система основана на процессах, которые определяют существо этой системы и которые остаются неизменными во времени. Однако реализация этих процессов может постоянно видоизменяться, принимая различные формы. Сложная система не может иметь детальной формализации в силу своей сложности. Попытка составить детальное формальное описание такой системы приводит к слишком высоким расходам человеческих, временных и финансовых ресурсов. Само описание неизбежно будет содержать многочисленные ошибки, возникновением которых мы опять обязаны сложности системы. За то время пока создаётся детальное формальное описание, предметная область или наши знания о ней могут существенно измениться, что сделает часть описания непригодным для использования. А так как любая предметная область находится в процессе постоянного развития, то нет никакой гарантии, что система, сделанная на основе детального формального описания, не устареет раньше, чем будет завершена. Сделаем выводы о том, что сложная система:

1. основана на ограниченном числе базовых процессов, определяющих существо системы;
2. не имеет детального формального описания;
3. обладает выраженной иерархичностью;
4. находится в постоянном развитии и, как следствие, не имеет завершённого состояния;
5. разложима на более простые составляющие, вплоть до элементарных;
6. определяется сложностью связей между её составляющими;

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

7. выполняет только функции управления по отношению к составляющим;

Отсюда можно заключить, что умение выполнять декомпозицию системы адекватно умению превращать сложную систему в простую. Так как система находится в постоянном развитии, то здесь неприменимы методы, используемые для разработки программы, то есть, законченного программного продукта. Система должна не только допускать возможность модификации, но содержать необходимые средства, позволяющие сделать процесс модификации максимально удобным и лёгким. Развитие системы должно происходить непрерывно от экспериментальной модели до моделей, находящихся в эксплуатации. Это одна сторона развития, вторая сторона связана с расширением функциональных возможностей системы до пределов, ограничивающих предметную область.

Теперь осталось показать, что объектная технология хорошо подходит для разработки таких систем. Для этого необходимо сделать описание основных положений, на которых основана объектная технология.

Инкапсуляция

Инкапсуляция подразумевает объединение и защиту кода и данных в некоторой сущности: объекте или модуле. Как правило, идея объединения кода и данных не вызывает серьёзных возражений. Это отчасти обусловлено тем, что исследователи понимают, что невозможно провести чёткой грани между кодом и данными: код обрабатывает данные, а данные управляют кодом. То или иное значение данных может вызвать разные ветки кода, но те или иные ветки кода могут, так или иначе, изменить данные. Идея объединения кода и данных восходит к модульному программированию (см. Письмо 01) и не является «изобретением» объектной технологии.

В отличие от идеи объединения кода и данных, идея защиты имеет разные толкования. Вот точка зрения на инкапсуляцию Г. Буча: «Скрытие информации – понятие относительное: то, что спрятано на одном уровне абстракции, обнаруживается на другом уровне. Забраться внутрь объектов можно; правда, обычно требуется, чтобы разработчик класса-сервера об этом специально позаботился, а разработчики классов-клиентов не поленились в этом разобраться. Инкапсуляция не спасает от глупости; она, как отметил Страуструп, «защищает от ошибок, но не от жульничества». Разумеется, язык программирования тут вообще ни при чём; разве что операционная система может ограничить доступ к файлам, в которых описаны реализации классов. На практике же иногда просто необходимо ознакомиться с реализацией класса, чтобы понять его назначение, особенно, если нет внешней документации» [ГБ. стр. 66]. Прежде чем соглашаться с подобной точкой зрения следует рассмотреть истоки проблемы.

Для чего вообще необходима инкапсуляция? Для того чтобы защитить реализацию класса. Если не обеспечивается должная защита, то прикладной программист, использующий данный класс, имеет возможность «привязаться» к конкретной реализации. В этом случае теряется возможность простой модификации не только данного класса, но и всей системы в целом. Практически любое изменение может катастрофическим образом сказаться на работоспособности, как других частей системы, так и прикладных программ. Однако такое положение противоречит изложенному выше требованию обеспечения модифицируемости. Чтобы осознать важность этого положения можно провести простой эксперимент. Предположим, что существует возможность обращаться к любому оператору подпрограммы из внешнего программного обеспечения. Это нарушает инкапсуляцию кода подпрограммы, но одновременно утрачивается и возможность модификации самой подпрограммы. Теперь нельзя изменить реализацию кода по той причине, что какое-то внешнее программное обеспечение могло обращаться к одному из удаляемых или изменяемых операторов. Аналогичная картина складывается при рассмотрении инкапсуляции на

уровне объектов (классов). Нарушая инкапсуляцию, мы обязательно утрачиваем модифицируемость, не зависимо от того, о каких сущностях идёт речь. Таким образом, «скрытие информации» является таким же относительным понятием, как и требования гибкости и модифицируемости.

С другой стороны инкапсуляция прямо связана и с требованием повышения надёжности систем (см. Письмо 01). Без соблюдения инкапсуляции различные модули и объекты могут произвольно менять данные других модулей или объектов, что неизбежно и негативно скажется на надёжности программного обеспечения. Можно рассмотреть простой пример, который показывает важность соблюдения инкапсуляции. Предположим, что нам необходимо разработать файловую подсистему.

Выделим структуру данных, которая необходима для работы с файлом, и создадим набор интерфейсных функций, которые будут осуществлять взаимодействие между программами и файлами. Очевидно, что набор интерфейсных функций мал. Как правило, он состоит из следующего перечня: создать файл, удалить файл, переименовать файл, открыть файл, закрыть файл, прочитать некоторое произвольное число байт из файла, записать некоторое произвольное число байт в файл, переместить указатель файла в новую позицию. Такой интерфейс позволил избежать необходимости в прямом доступе к внешним накопителям и, как следствие, снизить вероятность повреждения информации. В результате инкапсуляции структур данных и кода в одном модуле значительно упростился доступ к информации, и исчезла необходимость в дублировании кода доступа для каждой программы. Однако важно и то, что благодаря инкапсуляции работа с файлами стала намного надёжнее и безопаснее.

Интересно и другое, а именно то, что появилась возможность абстрагироваться не только от физических параметров внешних носителей, но от самих носителей! Собственно, какая разница, куда пишется информация: на внешний носитель вроде «жёсткого диска» или на иное устройство? Никакой! Следовательно, в виде файла вывода может выступать дисплей, принтер или другое устройство. Таким образом, в системе становится доступной возможность потокового ввода-вывода.

В книге Г. Буча инкапсуляция рассматривается в тесной связи с абстракцией, что абсолютно справедливо. Но, к сожалению, там ничего не сказано о том, что практическую пользу от инкапсуляции можно получить в полном объёме только тогда, когда мы создаём некоторый сервис, но не программу. Представьте, что инкапсуляция файла выполнена в одной единственной программе. Много ли пользы от этого?! Нет. Если же файловый сервис вынести в некоторое межпрограммное пространство, то возможности становятся несравненно больше, не говоря уже о том, что можно организовать работу с одним и тем же файлом нескольких программ одновременно.

Теперь можно сделать вывод о том, что частичная инкапсуляция – это миф. Инкапсуляция либо есть, либо её нет. Очень часто можно слышать разговоры о том, что для пользователей некоторого класса или модуля должен поддерживаться один уровень инкапсуляции, а для разработчиков, которые создают собственные классы или модули на основе данного, например, с помощью наследования, уровень инкапсуляции должен быть другим. Такой подход не кажется верным. С точки зрения создателей модуля или класса совершенно безразлично, почему новая реализация их программного обеспечения, полностью соответствующая заявленной спецификации, приводит к сбоям в работе другого программного обеспечения.

Полиморфизм

Полиморфизм подразумевает много форм реализации. Полиморфизм, как правило, рассматривают в тесной связи с виртуализацией. В Письме 01 говорилось о

виртуализации параметров и подпрограмм. Можно вернуться и посмотреть, что одно и тоже действие Draw принимало некоторую виртуальную фигуру и производило её рисование (первый пример). Далее было показаны преимущества перехода от виртуализации данных к виртуализации подпрограмм. В этом случае для каждой фигуры объявлялась своя собственная реализация действия Draw. Говоря другими словами, действие Draw стало полиморфным, имея отдельную форму (реализацию) для каждой из фигур.

Полиморфизм имеет большое значение для объектной технологии, так как он позволяет устанавливать аналогии между различными действиями (методами) различных классов, выявляя общие свойства этих классов и отождествляя их реакцию на один и тот же запрос. Вследствие этого, становится возможным создание схем или шаблонов поведения для класса. Схема или шаблон поведения связывают несколько методов элементарного класса. Например, для тех же геометрических фигур, рассмотренных в первом письме, можно задать абстрактную схему Change, в которой описать поведение класса в ответ на изменения каких-то параметров: координат якорной точки, цвета или толщины линии рисования, координат задающих конец линии (для линии); ширины или высоты (для прямоугольника), радиуса (для окружности). Поведение любой фигуры опишем следующей схемой:

- спрятать фигуру (Hide);
- сделать изменения (Change_Value);
- отобразить фигуру (Draw).

Не смотря на то, что все фигуры различны, данный шаблон поведения будет корректно работать для любой из этих фигур благодаря полиморфизму.

Обычно полиморфизм рассматривают как составную часть механизма наследования. Считается, что полиморфными могут быть только свойства (методы) подклассов. Но столь узкая трактовка термина «полиморфизм» приводит к проблемам при классификации. Чтобы убедиться в этом, попробуем построить иерархию классов фауны. Предположим, что мы уже дошли до птиц. Большинство птиц имеют свойство «летать», но не все птицы летают. Многие птицы умеют плавать, но опять же не все. Наконец, есть птицы, которые умеют летать и плавать, а есть такие, которые не умеют ни того, ни другого. Можно создать подклассы: «птицы летающие» и «птицы плавающие», как подклассы класса «птицы». Но как создать птиц обладающих обоими свойствами одновременно? Сторонники множественного наследования, скорее всего, создадут некий класс, обладающий свойством летать, и класс, обладающий свойством плавать. Тогда можно получить класс плавающих и летающих птиц с помощью тройного наследования (птицы, нечто летающие и нечто плавающие). Можно считать, что решение найдено. Вот только непонятно, что это за классы: «нечто летающее» и «нечто плавающее»? Как эти классы вложить в иерархию? Кто будет их суперклассом? Наконец, надо решить, что делать, если таких неприкаянных «классов» будет много?

Вопросы, которые возникают при использовании множественного наследования, не имеют простых ответов. Тем не менее, в данном случае решение очевидно, для того, чтобы увидеть его достаточно сформулировать проблему. Два класса должны обладать одним и тем же свойством (методом), но при этом данное свойство (метод) может отсутствовать у их ближайшего суперкласса. Словосочетание «одним и тем же свойством», в данном случае, тождественно термину полиморфизм. То есть, говоря другими словами, два класса должны обладать полиморфным свойством, которое отсутствует у их ближайшего суперкласса. Отсюда и решение проблемы: полиморфизм – явление самостоятельное и глобальное. Оно не только представляет собой часть механизма наследования, но и может существовать вне иерархии классов.

Этот подход в большей степени соответствует реальности. Например, летать могут птицы, насекомые, рептилии и механические аппараты (самолёты, вертолёты,

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

дирижабли и т.п.). Означает ли это, что все они имеют общего летающего предка? Конечно, нет. Плавать могут не только рыбы, но и животные, и птицы, и суда. Свидетельствует ли это о том, что у их общего суперкласса есть это свойство? Снова, нет. Здесь вполне допустимо рассмотреть и тот пример, который приводится в книге Г. Буча [стр. 74]. Он рассматривает два класса растений: цветы и фрукты-овощи. И отмечает, что некоторые цветы имеют плоды, а некоторые фрукты-овощи имеют цветы, и на основании таких рассуждений приходит к выводу о том, что множественное наследование необходимо. Реально же проблема решается элементарно и без множественного наследования, вполне достаточно сказать, что «иметь цветы» и «иметь плоды» это полиморфные свойства растений.

Таким образом, допустив независимость полиморфизма, мы получили возможность строить иерархии объектов наиболее естественным образом, который соответствует реальности. Исчезла необходимость в создании каких-то невероятных классов и тем более, не стало проблемы привязывания этих классов к общей иерархии. Как следствие, проблема перемещения классов или их свойств внутри иерархии просто перестаёт существовать. В свою очередь, исключение случайности при проектировании иерархий классов, позволяет получать простые, понятные и стабильные иерархии. Работать с такими иерархиями существенно легче, а их сопровождение, модификация и развитие требуют значительно меньше усилий.

Интересный вопрос: почему такая возможность не была реализована ранее? В объектно-процедурных языках типа C++, OO Pascal и им подобных это связано с тем, что связь между объектами в этих языках осуществляется посредством косвенных вызовов, через виртуальные таблицы (см. Письмо 01). Полиморфизм обеспечивается постоянством индекса входа свойства в родственные виртуальные таблицы. Если всё то же свойство «летать» имело индекс входа в виртуальную таблицу равный, скажем, трём, то оно и оставалось таким, для всех подклассов данного суперкласса. Компилятор, встретив вызов свойства «летать» заменял его инструкцией косвенного вызова подпрограммы, адрес которой третьему индексу входа, реализующей данное свойство у данного класса. Правда, следует отметить, что при множественном наследовании, такая однозначность утрачивалась, что требовало от компиляторов несколько большего «интеллекта».

Данная схема связи не позволяла ввести самостоятельность для полиморфизма, так как терялась однозначность между полиморфным свойством (виртуальным методом) и его точкой входа в виртуальную таблицу. С другой стороны, в момент компиляции не всегда можно сказать, какой конкретно класс будет использоваться в некоторой ситуации. В рамках принятой схемы связи посредством вызовов проблема трудноразрешима. Но, отказ от вызовов подпрограмм в пользу передачи сообщений, серьёзно облегчает решение задачи.

Рассматривая полиморфные свойства, надо вернуться к рассмотрению инкапсуляции. Полиморфные свойства – это не только код, но и необходимые данные. Как отмечалось в Письме 01, подпрограммы могут иметь локальные данные, существующие только во время исполнения подпрограммы, локальные данные существующие постоянно, глобальные данные и параметры, передаваемые при вызове. При переходе к объектной технологии глобальные данные, необходимые для организации взаимодействия обслуживающих и интерфейсных подпрограмм, образуют структуру данных класса. Обслуживающие и интерфейсные подпрограммы преобразуются в свойства класса. Но, вот интересно, что происходит с локальными данными, существующими постоянно? Как отмечалось, эти данные нужны подпрограмме. Если подпрограмма превратилась в полиморфное свойство, которое может принадлежать различным классам, то её локальные данные, существующие постоянно, также принадлежат структурам этих классов. При передаче полиморфного свойства между классами, не связанными узами наследования, структура локальных данных, существующих постоянно, может переопределяться или вообще не использоваться какими-то классами. Но, если передача полиморфных

свойств происходит в результате наследования, то структура локальных данных, существующих постоянно, полиморфного свойства может быть только расширена! Таким образом, повышая функциональные возможности класса за счёт присоединения новых полиморфных свойств, возможно и изменение структуры данных класса. При отсутствии инкапсуляции введение такого механизма было бы трудноразрешимой задачей.

Есть ещё один интересный аспект, который надо рассмотреть, допустив самостоятельность полиморфизма. На этапе проектирования очень сложно сказать, какое свойство должно быть полиморфным, поскольку невозможно предвидеть все возможные пути развития среды. Но тогда следует предположить, что полиморфным может быть любое свойство класса. И это правильно. Но тогда зачем нужно наследование? Действительно, имея самостоятельный механизм полиморфизма, можно совершенно произвольно конструировать классы. Однако, это не лучший стиль. Наследование отвечает за передачу классообразующих (общих для данных классов) свойств, структуры и поведения класса. Благодаря наследованию достигаются высокие уровни абстракций. Одного полиморфизма для этого мало.

В сочетании с сообщениями (асинхронными вызовами) полиморфизм свойств даёт ещё одну уникальную возможность, а именно допускает возможность развивать класс, насыщая его новыми свойствами не только на стадии проектирования, но и при эксплуатации. Это кардинально меняет подход к проектированию. Теперь совершенно не обязательно изначально пытаться представить себе полный набор свойств класса. Любое свойство может быть добавлено в любое время и в любое место иерархии. Если класс, к которому добавляют свойство, имеет подклассы, то данное свойство будет распространяться и на подклассы, так будто оно было унаследовано ими. Поэтому на первой стадии вполне достаточно реализовать только минимально необходимую функциональность класса. А далее можно будет добавлять новые свойства, когда в них появится потребность. Такая возможность, во-первых, позволяет быстро собрать прототип системы и отработать основные функции, а, во-вторых, любое увеличение функциональности не повлечёт за собой ни переписывания кода, ни перекомпиляции, что очень важно для поддержания работоспособности существующих свойств. Если бы развитие классов шло традиционным путём, то достичь подобной мощности и гибкости было бы сложно, хотя бы потому, что пришлось бы перестраивать виртуальные таблицы. К сожалению, даже эту простую операцию выполнить без перекомпиляции кода класса нельзя.

Итак, основное достоинство полиморфизма заключается в том, что он позволяет различным классам иметь семантически однородные свойства. Благодаря этому становится возможным однотипно взаимодействовать с различными классами.

Наследование

Наследование, как отмечалось ранее, – это передача подклассу свойств, структуры и поведения суперкласса. Пожалуй, наследование представляет самую красивую концепцию в объектной технологии. Однако передача свойств и структуры – это механизм или средство, но не цель. Реальных целей, достигаемых с помощью наследования, несколько. К ним относится, например, упорядочивание типов сущностей, то есть классификация. Действительно, не так просто разобраться в нюансах, отличающих один тип сущности от другого типа, если не представлять иерархию их развития, её идеологическую основу. Как и любая другая классификация, иерархия классов значительно упрощает использование сущностей, их модификацию и сопровождение.

В основе разработки иерархии классов, как правило, лежит их функциональное родство. Так, например, при разработке среды хранения нас в первую очередь интересуют классы способные сохранять, искать, выбирать и

удалять информацию, представленную самыми различными типами. А при разработке среды графического интерфейса основу иерархии образуют классы способные отображать информацию в том или ином виде. В свою очередь, коммуникационная среда базируется на иерархии классов, которые умеют передавать информацию, распространяя её между компьютерами, связанными произвольным образом. Функциональная однородность, лежащая в основе построения иерархий классов, позволяет получать простые и эффективные реализации классов, но она может служить причиной образования множества независимых, не имеющих общего суперкласса, иерархий.

Благодаря классификации становится возможным введение и широкое практическое использование абстракции. Значение абстракции при проектировании сложных систем переоценить невозможно. Абстракция даёт возможность исключить из рассмотрения малозначительные детали и сосредоточиться на том, что действительно важно на данном уровне рассмотрения. Как правило, абстракция используется для определения интерфейса, который позже реализуется и детализируется в подклассах. Например, можно реализовать абстрактное предприятие и определить необходимый интерфейс, который позволит взаимодействовать предприятию с внешним миром: другими предприятиями, государственными службами и фондами, и т.п. При этом совершенно неважно, чем будет заниматься это предприятие: выпуском продукции, предоставлением услуг, торговлей или чем-то ещё.

Абстракцию также полезно использовать для задания поведения класса. Здесь под поведением понимается последовательность действий по обработке какого-то сообщения (или реакция класса на некоторое воздействие). Подклассы наследуют реализацию данного алгоритма, что приводит не только к значительной экономии кода за счёт его многократного повторного использования, но и позволяет при необходимости менять поведение множества подклассов простой заменой одного алгоритма другим у общего суперкласса. Такая замена может успешно применяться при переходе от модели или макета будущей системы к рабочей реализации.

Несмотря на неоспоримые достоинства и логическую простоту, споры вокруг механизма наследования не утихают. Безусловно, главной причиной дебатов является множественное наследование. Введение этого механизма было избыточным, а его применение порождает больше проблем, чем отказ от него. При рассмотрении концепции полиморфизма отмечалось, что часть ситуаций, когда применяют множественное наследование, исключаются, если рассматривать полиморфизм, как самостоятельное явление, а ни как составную часть концепции наследования. Другая часть ситуаций решается за счёт агрегации, о которой речь пойдёт в следующем письме. При агрегации вводятся связи вложенности (принадлежности), которые могут естественным образом заменять связи наследования, используемые при множественном наследовании. Этого вполне достаточно, чтобы отказаться от механизма множественного наследования. Однако следует оговориться, что подмена будет полной только в случае, если объектное проектирование будет следовать правилу локализации, которое тоже будет рассмотрено в дальнейшем.

В чём суть множественного наследования?! Любая сущность может рассматриваться с самых различных позиций, и, как следствие, классифицироваться по различным признакам. То, что одна и та же сущность может находиться в нескольких классификаторах, означает, что она наследует свойства изначально разнородных сущностей. Это и является идеологической основой множественного наследования. Но при проектировании системы всегда есть возможность локализовать различные точки зрения на одну и ту же сущность, исходя из семантики её применения. Это с одной стороны, а с другой стороны, множественное наследование является частным и худшим вариантом агрегации. Наконец, исходя из правила локализации, можно отметить, что в семантически однородной среде не может существовать нескольких способов классификаций каких-либо сущностей, а,

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

следовательно, и необходимость в механизме множественного наследования отсутствует.

Не исключено, что приверженность механизму множественного наследования некоторых видных исследователей в области объектной технологии, связана с тем, что формирование иерархий классов производится ими достаточно произвольно, что отражено в книге Г. Буча. Он, в частности, пишет: «Трудно сразу расположить классы и объекты на правильных уровнях абстракции. Иногда, найдя важный класс, мы можем передвинуть его вверх в иерархии классов, тем самым, увеличивая степень повторности использования кода». Однако последовательное применение методов объектной композиции позволяет упорядочить процесс построения иерархий классов.

Не менее интересно высказывание Г. Буча по поводу агрегации. Он рассматривает два вида агрегации: агрегация на основе физического включения и ссылочная агрегация. Суть первого вида агрегации состоит в том, что класс-контейнер включает в себя вложенные в него классы, аналогично тому, как одна структура данных вкладывается в другую. Ссылочная агрегация подразумевает, что класс-контейнер включает в себя только ссылки на вложенные объекты. Так вот, рассуждая по поводу первого вида агрегации, Г. Буч пишет: «Часто агрегацию путают с множественным наследованием. Действительно, в C++ скрытое (защищённое или закрытое) наследование почти всегда можно заменить скрытой агрегацией экземпляра суперкласса. Решая, с чем вы имеете дело – с наследование или агрегацией – будьте осторожны. Если вы не уверены, что налицо отношение общего и частного (is a), вместо наследования лучше применить агрегацию или что-нибудь ещё» [ГБ стр. 133] . Из этой цитаты можно сделать вывод о том, что, по мнению Г. Буча, разница между агрегацией и множественным наследованием может быть столь тонкой, что её трудно различить. А механизмы агрегации и множественного наследования столь похожи, что не во всякой ситуации можно сделать правильный выбор между ними. Какой ещё аргумент может быть более весомым, чтобы отказаться от множественного наследования, нежели ситуация полной неопределённости и двусмысленности?

Возможно, имеет смысл рассмотреть эту ситуацию более пристально. Предположим, что мы имеем два суперкласса: “First” и “Second”. От этих двух суперклассов создадим на основе множественного наследования подкласс “Third”. Очевидно, что подкласс “Third” наследует структуры и свойства обоих суперклассов. Реально это означает, что в структуру подкласса вкладываются структуры суперклассов или, говоря другими словами, подкласс становится агрегатом суперклассов. Но в отличие от описанных в следующем письме контейнеров, этот агрегат имеет существенные и неоправданные ограничения. Например, нельзя заменить один суперкласс на другой без перекомпиляции подкласса, даже если замещающий суперкласс обладает интерфейсом аналогичным интерфейсу замещаемого суперкласса. Как следствие, теряется гибкость и модифицируемость. Другой негативной стороной подобной агрегации является то, что вместо схем надо использовать методы, которые, в отличие от схем, пишутся, а не конструируются. Таким образом, теряется простота создания агрегатов. И, что не менее важно, утрачивается возможность разделения логик: логики исполнения (логика вложенных классов) и логики управления (логика контейнеров). Но данное разделение имеет ключевое значение, поскольку на её основе становится возможным объектная декомпозиция, которая понижает трудоёмкость проектирования и разработки систем. Более подробно эта тема раскрывается в следующем письме.

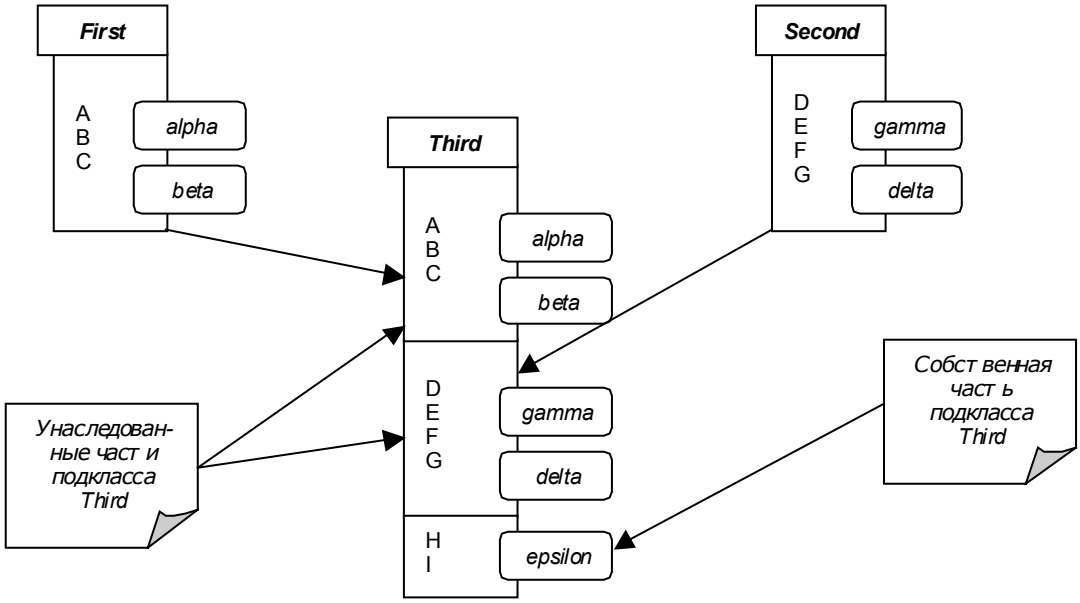

Рис. 02.01. Схема множест венного наследования

Итак, наследование позволяет классифицировать используемые программные сущности, вводить высокоуровневые абстракции, передавать свойства и структуру суперкласса своим подклассам, что позволяет существенно экономить код. Наконец, наследование определяет направление развития классов, что существенно облегчает разработку иерархии классов, отражающей частный, но важный, случай взаимосвязи между сущностями реальной предметной области.