Но истина, как вы знаете, это то, что делает мир проще, а отнюдь не то, что обращает его в хаос. Истина – это язык, помогающий постичь всеобщее. Ньютон вовсе не «открыл» закон, долго остававшийся тайной, - так только ребусы решают, а то, что совершил Ньютон, было творчеством. Он создал язык, который говорит нам и о падении яблока на лужайку, и о восходе солнца. Истина – не то, что доказуемо, истина – это простота.

«Военный лётчик» Антуан де Сент-Экзюпери

Просто сложные системы

Материалы, изложенные в предыдущих письмах, являлись прелюдией. Концепции инкапсуляции, полиморфизма, агрегации и наследования важны не только сами по себе, их сочетание образует объектную технологию, способную вывести разработку сложных систем на качественно иной уровень. Данное письмо посвящено рассмотрению вопросов, связанных с переходом на этот новый уровень.

Архитектура сложной системы

Любая сложная система иерархична. Иерархичность является важнейшим свойством системы. Она позволяет упорядочить связи и функции, тем самым, понизить сложность управления системой и сложность моделирования системы.

Иерархичность системы основана на подчинении более низких уровней более высоким уровням. Подчинение, как правило, подразумевает либо прямое включение, либо использование элементов более низкого уровня иерархии. Другими словами, существо иерархии системы отражает агрегация. Задачей именно агрегации является прямое или косвенное включение элементов, расположенных на нижнем уровне, и управление этими элементами в соответствии с логикой контейнера-владельца.

Применение агрегации позволяет сделать несколько важных заключений. Первое заключение состоит в том, что элементы нижнего уровня должны быть полностью независимы друг от друга и не должны самостоятельно вступать в какое-либо взаимодействие. Или, говоря другими словами, должна соблюдаться полная инкапсуляция. Любая логика взаимодействия между элементами является частью логики сущности, в которую вложены данные элементы или которой они подчинены. То есть, два объекта не должны иметь возможности обращаться друг к другу, если они принадлежат одному уровню. Логика взаимодействия этих объектов является частью логики более высокого уровня, логикой их контейнера.

Второе заключение состоит в том, что смежные уровни иерархии должны поддерживать интерфейс, который позволяет элементам этих уровней взаимодействовать между собой. Существо интерфейса отражает потребность верхнего уровня в некоторой функциональности, которая реализуется в элементах нижнего уровня. Но можно говорить и о том, что посредством интерфейса нижний уровень предоставляет функциональность, которой может воспользоваться верхний уровень.

Третье заключение говорит о том, что интерфейс с верхним уровнем определяет функциональность данного уровня, а значит и свойства сущностей данного уровня. Иными словами, межуровневый интерфейс закладывает полиморфизм свойств объектов каждого из уровней.

Четвертое заключение определяет возможность поэтапного развития интерфейса, соединяющего два уровня. На основании этого можно расширять свойства объектов каждого уровня. Полиморфизм и развитие свойств объектов лежат в основе концепции наследования. Из этого можно сделать вывод о том, что наследование является эффективным механизмом, позволяющим реализовать множество классов объектов на каждом уровне иерархии системы. Классы объектов, находящиеся на различных уровнях иерархии системы могут не иметь общего суперкласса. Наличие

общего суперкласса необходимо только самой системе для единообразного обращения к объектам любого уровня. Иной семантической нагрузки общий суперкласс нести не может, он, как правило, не существует в моделируемой предметной области.

Пятое заключение состоит в том, что все многообразие, как объектов системы, так и сущностей предметной области, образуется не на основе наследования, а на основе композиции объектов, на основе агрегации.

Шестое заключение определяет, что при разработке сложных систем акценты должны быть смещены с программирования на конструирование. Программирование, в привычном для нас виде, остается уделом только элементарных объектов, количество которых крайне невелико.

Седьмое заключение обосновывает переход от унарных универсальных языков программирования к многообразию языков и средств разработки, отражающих специфику каждого уровня иерархии сложной системы. В основе многообразия языков и средств разработки лежит логика межобъектной связи, которая может быть как очень простой, так и очень сложной, не имеющей формализации и включающей эмпирические правила. Обычно, с повышением уровня иерархии системы происходит значительное сокращение количества сущностей, но возрастает сложность описания логики их совместной работы.

Независимость объектов

Реальный мир состоит из множества элементарных сущностей, которые никак не связаны между собой и существуют обособленно. Буквы алфавита и ноты никак не взаимодействуют между собой, точно так же как и атомы в таблице Менделеева или структурные группы Ассура. Любая сложная система разложима на элементарные составляющие. Элементарных составляющих, как правило, очень и очень немного, но они образуют бесчисленное количество комбинаций, обладающих теми или иными свойствами. Так, например, два-три десятка букв образуют сотни тысяч слов, а семь нот - бесконечное количество музыкальных произведений, чуть больше сотни атомов создают огромное количество молекул. Но точно также ограниченное количество типов данных, поддерживаемых современными реляционными СУБД, лежит в основе хранения самой различной информации.

Можно рассмотреть ситуацию, когда, руководствуясь благими побуждениями, мы заложили связь между двумя элементарными объектами внутрь самих объектов, например, в виде обращения к одному объекту из метода другого объекта. Но наличие этой связи означает, что теперь нельзя использовать один объект без другого. Это равнозначно тому, что мы связали две буквы, например, «А» и «Б», но тогда мы не сможем использовать букву «А» без буквы «Б», это приведет к сокращению возможных применений, к сложностям при образовании слов. Понятно, что чем больше таких связей и зависимостей, тем труднее язык и тем сложнее им пользоваться. Если продолжать устанавливать прямые связи, то сложность будет возрастать многократно, а возможность повторного использования объектов в иных сочетаниях столь же сильно сократится.

Можно ослабить связи между объектами посредством виртуализации. Например, можно потребовать, чтобы один объект обращался не к конкретному объекту, а любому объекту некоторого класса или объекту, обладающему заданным набором интерфейсов. Такое решение расширит область повторного использования объектов, но крайне незначительно. Фактически, возможность повторного использования всегда будет ограничена теми рамками, которые заложили разработчики, что негативно сказывается при развитии и модернизации системы.

Композиция

Композиция является основой многообразия окружающего нас мира. Мы соединяем различные элементы в единой сущности для придания ей требуемых свойств. Комбинируя элементы, можно изменять свойства сущности, которая их содержит. Поэтому нельзя считать контейнер простой механической смесью составляющих его элементов. Молекула состоит из атомов, и каждый атом значим. Изменяя количество или состав атомов, мы изменяем свойства молекулы, получая модификацию исходной молекулы или совершенно иную молекулу. Каждая буква в слове значима и, изменяя буквы или их местоположение, мы получаем другое слово.

Рассматривая композиции, нельзя не остановиться на вопросе качественных изменений. Мы можем заменить в контейнере один объект на другой, возможно даже, что новый объект будет относиться к совсем иному классу. Но при этом сам контейнер не претерпит каких-либо качественных изменений. Например, можно поменять двигатель в автомобиле с карбюраторного на дизельный (конечно, если позволяет интерфейс). При такой замене автомобиль остается автомобилем. Но если поменять шасси на воздушную «подушку», то автомобиль приобретет новое свойство – перемещение по бездорожью, в том числе и над водной поверхностью. При такой замене автомобиль превратится в вездеход, то есть будет получена сущность с новым набором свойств. Можно сформулировать наследование контейнеров, как замещение вложенных объектов или добавления новых объектов с целью получения новых свойств. Модификация контейнера, выполняемая ради получения новых свойств, аналогична модификации элементарного класса за счёт замены существующих или добавления иных методов, в результате создаётся новый подкласс, обладающий модифицированным набором свойств.

Одной из главных задач композиции является организация взаимодействия между вложенными элементами. Поскольку элементы не имеют возможности непосредственно взаимодействовать друг с другом, то это взаимодействие выполняется опосредованно через контейнер. Логика взаимодействия элементов при выполнении некоторой задачи является частью логики контейнера и реализуется в виде схемы. Важность опосредованного взаимодействия наиболее очевидна при рассмотрении контейнеров, расположенных на высоких уровнях иерархии системы. Здесь формы взаимодействия могут регулироваться контейнером в зависимости от состояния самого контейнера или состояния вложенных объектов.

Контейнеры, образуемые из элементов нижнего слоя иерархии, могут иметь разную продолжительность жизни. Устойчивые контейнеры используются различными задачами, и срок их жизни может быть бесконечно большим. Стабильные контейнеры создаются под задачу и уничтожаются при завершении задачи. Временные контейнеры создаются в отдельном программном блоке и завершаются при завершении исполнения этого блока.

Элементы, входящие в контейнер, могут существовать столько же, сколько и контейнер или меньше, но могут быть созданы и уничтожены вне данного контейнера, то есть иметь продолжительность жизни больше продолжительности жизни контейнера. Такие элементы, как правило, являются разделяемыми несколькими контейнерами.

Область видимости любого элемента контейнера ограничена. Система устанавливает прямую и обратную связь между контейнером и каждым из вложенных элементов. Если элементу требуются какие-то ресурсы, то он обращается к своему владельцу. Контейнер может предоставить эти ресурсы, может обобщить запросы от других элементов на тот же вид ресурсов, а может просто делегировать запрос на следующий высший уровень иерархии, если запрашиваемые ресурсы не являются его собственностью. При загрузке или создании контейнера система может оптимизировать транзитные запросы. Для этого достаточно проверить, в каких ресурсах нуждается каждый элемент контейнера. Если запрос на этот ресурс не обрабатывается контейнером, то система может сразу перенаправить запрос на тот уровень, где

производится управление требуемым ресурсом. Такое решение позволяет, не нарушая инкапсуляции, обеспечивать эффективность исполнения.

Конструирование

Конструирование – это процесс составления композиции. Как уже отмечалось выше, композиция собирается для придания контейнеру требуемых свойств. Конструирование – это абстрактный процесс. Можно говорить, что программист, разрабатывая метод элементарного класса, конструирует его из операторов языка с целью достижения необходимой функциональности. Точно также из методов конструируется объект, а из объектов контейнер.

Конструирование на разных уровнях иерархии системы существенно различается. Если конструирование низких уровней иерархии является уделом программистов, то конструирование высоких уровней должно стать прерогативой пользователей. Это заключение позволяет значительно сместить акценты при разработке сложных систем, о чем еще будет сказано в дальнейшем.

Очень важна тесная связь между конструированием и моделированием. Моделирование основано на использовании упрощенных или грубых конструкций, которые впоследствии могут быть заменены на более эффективные. Замена одних конструкций другими выполняется на основе конструирования. Аналогично текущая рабочая программная модель со временем может быть вытеснена иной, более совершенной моделью.

Моделирование открывает возможность итерационной разработки системы, когда от грубой первоначальной модели система развивается посредством уточнения и расширения.

Многообразие средств разработки

Конструирование позволяет использовать не только традиционные языки программирования, но и специализированные инструменты. Применение инструментальных средств значительно облегчает процесс разработки, делает более эффективным визуальный контроль. Повышение роли визуального контроля при организации различных контейнеров и их схем, автоматизация проверки соответствия интерфейсов дают основание для повышения качества системы.

Разработка инструментальных средств для конструирования контейнеров для самых разных приложений является самостоятельной задачей. Её качественное решение во многом определяет гибкость и удобство работы с системой. Крайне важно научиться смещать акцент с решения задач пользователей на разработку инструментов для решения пользовательских задач. То есть, вместо того, чтобы решать задачи за пользователей надо предоставить пользователям удобные инструменты для решения их задач. Такие инструменты, как построители расчётных и аналитических схем, средства создания различных диаграмм и интерфейсов, описатели систем и т.п. основываются на конструировании и применяются пользователями. Создание инструментов гораздо эффективнее решения отдельно взятой задачи при разработке системы.

Конечное программное обеспечение слишком погружено в детали конкретной задачи и не может иметь высокого абстрактного уровня. Как следствие, при таком подходе сложно находить общие элементы и специфицировать их для повторного использования. Если разработка конечного программного обеспечения разделена между несколькими разработчиками, то выделение общих составляющих сопряжено с ещё большими трудностями. В результате разработчики теряют много времени на то, чтобы многократно решать похожие проблемы в каждом конкретном конечном программном обеспечении. Это не только снижает производительность, но и существенно усложняет отладку, передачу разрабатываемого программного обеспечения между разработчиками, обучение пользователей, документирование и т.п.

Разработка инструментального программного обеспечения наоборот ориентирована на множество однородных задач и изначально абстрагирована от деталей конкретных задач. Здесь от разработчиков требуется не столько знание специфики конкретной задачи, а то, какие способы решения могут быть использованы на всём множестве задач, знание того, что необходимо пользователю для решения данного множества задач. Таким образом, абстрагирование при данном подходе становится необходимым и важнейшим элементом разработки. При этом надо стремиться к тому, чтобы компоненты, полученные в результате анализа, можно было использовать и в других проектах.

Пример

Предположим, что необходимо реализовать расчёт заработной платы для некоторого предприятия. Можно пойти традиционным путём и написать программу начисления заработной платы. Для небольшого предприятия характерно относительно немного схем начисления, но на крупных предприятиях количество схем начислений заработной платы может достигать сотни и даже больше. Разобраться в деталях этих начислений совсем не просто, к тому же они достаточно часто видоизменяются. В результате, однажды написанная программа, будет требовать постоянного внимания со стороны разработчиков, отвлекая их от другой работы.

Попробуем разработать для пользователей не программу, но инструмент, чтобы они самостоятельно могли создавать расчётные схемы. Допустим, что пока у нас нет детального представления о том, как вообще происходит начисление заработной платы ни для одной из категорий служащих и рабочих. Поэтому возьмём чистый лист бумаги и пойдём к тому эксперту-пользователю из отдела труда и заработной платы (ОТиЗ), который должен ввести нас в тонкости расчёта. На листе бумаги слева и справа проведём прямые вертикальные линии. Левая часть листа предназначена для списка той информации, которая должна поступать на вход расчётной схемы (входная информация). Правая часть листа отведена для списка видов начисления и удержания, которые получатся в результате расчёта (выходная информация). На средней части листа будет отрисована схема начисления заработной платы для конкретной категории служащих или рабочих. Попросим эксперта-пользователя заполнить левую и правую часть листа. Скорее всего, первоначально информация может оказаться неполной, но полной информации нам не требуется. Гораздо важнее понять, как формируются эти части информации.

Очевидно, что входная информация должна получаться из базы данных, содержащей необходимую информацию о положении служащего, его контрактах, отработанном времени, льготах и т.п. Выходная информация, в свою очередь, тоже должна быть записана, возможно, в ту же базу данных (но не обязательно), поскольку она будет участвовать в последующих начислениях. Это определяет первые необходимые части нашей будущей разработки. В частности, нам необходимо создать такой инструмент, который бы позволил пользователю извлекать нужную информацию из базы данных и записывать результаты расчётов. Инструмент должен быть интегрирован с расчётной схемой и базой данных. Вполне вероятно, что в дальнейшем пользователям потребуется и другая информация из базы данных, отличная от той, что была перечислена при первой встрече. Количество начислений и удержаний тоже может меняться в различных схемах начислений заработной платы. Интерфейс с базой данных должен предусматривать и сохранение схем, созданных пользователем. Схемы должны быть связаны с категорией рабочих и служащих, а также иметь временную отметку, позволяющую отслеживать изменение расчётных схем во времени, а также версию и автора каждой схемы.

Визуальное отражение расчётной схемы должно быть максимально простым, понятным и эргономичным. Здесь можно воспользоваться той же метафорой чистого листа бумаги, содержащего три зоны: зона входной информации, зона выходной информации и наборное поле в центре листа. Существует относительно небольшое

количество элементов, с помощью которых пользователь может задавать суть и порядок расчётов. Прежде всего, к таким элементам относятся четыре арифметические операции (сложение, вычитание, умножение и деление) и вычисление процентов, а также определение различных коэффициентов (констант), используемых при расчёте. Помимо этого потребуется реализовать операцию выбора. Суть её состоит в том, что при известном входном значении должно формироваться одно выходное значение из допустимого множества. Эта операция является близким аналогом оператора «switch-case» в языке программирования Си. Возможно, в дальнейшем потребуются и иные операции, но в данный момент это не важно.

Результат любой операции может быть подан на вход следующей операции наравне с входными параметрами схемы или представлен как выходной параметр. Для того чтобы пользователю было удобно оперировать параметрами, их лучше поименовать. Желательно предоставить возможность указания, как сокращённого имени (аббревиатуры), так и полного (развёрнутого) названия.

После того как схема создана, она может быть сохранена и оттранслирована в удобный для исполнения вид. Транслировать схему можно в некоторый промежуточный или непосредственно в машинный код. Поскольку операции тривиальны, то трансляция даже в машинный код не вызывает особых проблем. Перед трансляцией необходимо представить всё множество входных параметров, промежуточных и конечных результатов, как набор переменных. Поскольку в данном случае все параметры представлены вещественными числами, то можно всё множество параметров представить, как один большой массив вещественных чисел с плавающей или фиксированной точкой.

Как видно из представленного выше описания, подобный инструментарий для построения расчётных схем применим не только для начисления заработной платы, но и во многих других расчётных задачах. Разработка этого инструментария кардинально отличается от разработки программного обеспечения для расчёта заработной платы. Для разработчиков не имеет значения логика той или иной схемы начисления, но необходимо обеспечить возможность и удобство разработки любой расчётной схемы. Полученный инструмент имеет много общего не только с аналогичными средствами моделирования других расчётных схем, но и со средствами моделирования схем, которые не имеют отношения к расчётам. Например, инструмент разработки схем технологических процессов или процессов управления. Здесь тоже входная информация должна получаться из базы данных, а полученная схема должна сохраняется в базе данных. Как и в расчётных схемах, при построении технологических схем удобно использовать наборное поле, где технологические операции связываются в виде графа (вершины – технологические операции, а рёбра, связывающие вершины, описывают порядок выполнения технологических операций). Соответственно появляется возможность унификации набора компонент, используемых при построении самых разных схем, и быстрая сборка новых инструментов из готовых (проверенных) компонентов. Отсюда можно сделать вывод, что абстрагирование от реальных расчётных схем позволяет увидеть общие элементы и подходы при разработке инструментальных средств самых разнообразных схем. В свою очередь, многократное использование общих элементов в различных инструментальных средствах, даёт возможность уменьшить объём кодирования и повысить надёжность программного обеспечения. Повышение надёжности связано с двумя основными факторами:

- повторное использование проверенного ранее кода;
- использование операции конструирования вместо кодирования.

Конструирование основано на разделении разработки элемента и схем связей, в которых участвует этот элемент. Это принципиально различные уровни, и их разделение способствует значительному снижению общей сложности разработки.

Межуровневый интерфейс

Интерфейсы между смежными уровнями иерархии являются ключевым звеном в сложных системах. При разработке проекта сверху вниз верхний уровень определяет тот интерфейс, который должен быть реализован нижним уровнем. При разработке снизу вверх нижний уровень декларирует тот интерфейс, который он может предоставить верхнему уровню.

Серьёзность подхода к разработке интерфейса, соединяющего два уровня, обусловлена тем, что изменение интерфейса потребует внесения изменений в реализацию на каждом из смежных уровней. Хорошо продуманный интерфейс, наоборот исключает необходимость в синхронном внесении изменений и позволяет вести разработку каждого уровня независимо от других уровней. В свою очередь, отсутствие потребности в синхронизации работы различных групп разработчиков значительно ускоряет работу над проектом, снижает конфликтность в коллективе и делает работу каждого разработчика наиболее удобной и комфортной.

Интерфейс уровней иерархии сложной системы можно рассматривать в качестве механизма взаимной виртуализации. Одни и те же элементы нижнего уровня могут входить во многие контейнеры верхнего уровня, и контейнер верхнего уровня может включать различные элементы нижнего уровня. Таким образом, межуровневый интерфейс является тем механизмом, который обеспечивает необходимую для модернизации и развития системы гибкость.

С другой стороны, межуровневый интерфейс представляет собой спецификацию, которая описывает минимально необходимый функционал нижнего уровня. Фактически, совокупность межуровневых интерфейсов представляют собой важную часть документации по описанию архитектуры системы.

Функциональность уровня

Функциональность любого уровня специфицируется его интерфейсом с верхним уровнем, независимо от того, какого рода проектирование применяется: нисходящее или восходящее. Зная интерфейс уровня, достаточно просто выполнить его проекцию на свойства классов, населяющих данный уровень.

Интерфейс с нижним уровнем может требовать функционально различных сервисов. В таком случае интерфейс делится на ряд независимых интерфейсов, которые реализуются различными уровнями, подчинёнными данному уровню. Текущий уровень позволяет агрегировать объекты, полученные с различных уровней подчинённых текущему уровню.

Иерархия классов, реализующих интерфейс некоторого уровня, может развиваться в двух основных направлениях. Первое направление – это создание различных вариантов реализации интерфейса. Каждая из реализаций может быть ориентирована на некоторое специфическое применение. И эта специфика применения может расширяться по мере развития представлений о предметной области.

Второе направление связано с постепенной реализацией интерфейса с верхним уровнем. То есть, суперкласс уровня может включать не весь функционал, заявленный в интерфейсах для данного уровня. Дальнейшая реализация интерфейсов происходит в подклассах. Как отмечалось ранее, интерфейс с верхним уровнем может со временем наращиваться новым функционалом по мере расширения представлений о предметной области. Добавленные функции могут быть реализованы в подклассах уже существующих классов. Вполне допустимо одновременное развитие иерархии классов уровня в обоих направлениях одновременно.

Говоря о функциональности уровня, необходимо снова вернуться к средствам её реализации, к средствам разработки. Функции нижних уровней реализуются разработчиками-программистами с использованием типовых средств разработки, но

функции высоких уровней реализуются пользователями. Задачей программистов является предоставление пользователям удобных инструментов для реализации функций высоких уровней иерархии системы. Разработка и унификация инструментов и их компонент для различных уровней представляет собой самостоятельную серьёзную задачу, как это было отмечено в примере, приведённом ранее.

Разработка инструментального программного обеспечения нацеленного на предоставление средств для решения некоторого класса коренным образом отличается от разработки конечного программного, целью которого является решение конкретных задач. Следует особо отметить изменения во взаимодействии между пользователями и разработчиками. Фактически инструментальный подход стирает грань между пользователями и разработчиками, ибо именно пользователи реализуют в системе важнейшие функции верхних уровней, то есть становятся соразработчиками системы.

Можно выделить характерные особенности, каждого вида программного обеспечения:

Критерий	Конечное ПО	Инструментальное ПО
Процесс разработки	Дискретный. При изменении условий старое ПО заменяется новым	Непрерывный. ПО постоянно совершенствуется и развивается, охватывая все новые задачи
Качество в процессе доработки и развития	Понижение	Повышение
Взаимоотношения с пользователями	Противостояние	Кооперация
Область применения	Ограничена конкретной задачей	Ограничена классом задач
Уровень абстракций	Низкий	Высокий

Развитие интерфейса

Сложные системы редко находятся в завершённом состоянии. Меняются наши представления о предметной области, под воздействием внешних и внутренних факторов меняется сама предметная область. Одним из путей развития системы является развитие межуровневых интерфейсов. Последовательное развитие интерфейсов должно сопровождаться добавлением новых спецификаций, но не изменением существующих.

Развитие интерфейса не обязательно связано с развитием нижнего уровня. Новые спецификации могут быть удовлетворены за счёт существующего на данный момент функционала. Но развитие интерфейсов может быть основой развития иерархии классов на конкретном логическом уровне.

Разработка сложной системы

В пятом письме по теме «Проектирование» приведены проблемы, с которыми неизбежно сталкиваются разработчики при создании сложной системы методом “water-fall”. Сейчас, после рассмотрения архитектуры сложной системы, можно определить иную методику разработки.

Коренное отличие предлагаемой методики состоит в том, что работы по проектированию и кодированию происходят тогда, когда нет детального представления о том, что должна представлять собой система в конечном итоге. Мало того, такого

представления может вообще не существовать у разработчиков в силу того, что предметная область познаётся и формализуется непосредственно в процессе работы. В таком случае система должна развиваться вместе с расширением кругозора разработчиков. Как бы парадоксально не звучало, но мы должны начать разработку тогда, когда не существует полного представления о том, что же разрабатывается. Можно в качестве примера рассмотреть то, во что сегодня превратились компьютеры. Вряд ли разработчики первых компьютеров предполагали, что всего через несколько десятилетий компьютеры проникнут во многие сферы человеческой деятельности от автоматизации управления предприятиями, сложными техническими системами до центров развлечения, включая средства связи, издательскую и научную деятельность. Тем не менее, сегодня это свершившийся факт. Поэтому реально вопрос состоит не в том, можно ли сделать то, что пока неведомо, а в том, как обеспечить возможность развития системы вне рамок представлений разработчиков.

Последовательность

Создание сложной системы можно разделить на ряд шагов:

- определение контура предметной области
- выделение уровней иерархии
- описание интерфейсов уровней
- разработка инструментов
- разработка классов внутри уровней

Контур предметной области

Здесь специально не используется термин «границы предметной области», поскольку при создании сложной системы границы невозможно указать точно. На любой стадии необходимо исходить из того, что существующие представления о предметной области не полны в силу сложности и динамики развития самой предметной области, проблем её анализа и описания.

Рис. 1. Контур предметной области

Неоднозначность и неформальность контура предметной области не должна служить преградой для развития проекта. Если проект создаётся под заказ, то необходимо зафиксировать требования заказчика в виде цели и задач, которые он ставит перед системой. При этом необходимо заранее оговорить, что система может развиваться и совершенствоваться и после окончания работы над проектом силами самих пользователей и персонала, который занят сопровождением. Иными словами, возможность развития системы должна быть зафиксирована в документации, предваряющей работы над проектом. Требования пользователей, которые фиксируются

перед началом работ, не должны быть детализированы, они должны формулировать цели, но не детали частных решений.

В случае, когда проект является инициативной разработкой необходимо самостоятельно определиться с первоначальным представлением о системе, сформулировать в общем виде цель и задачи. Например, в последующих письмах будет рассмотрено несколько примеров проектов сложных систем. В качестве примеров предполагается использовать самые разные проекты, такие, например, как проект реляционной системы управления базами данных и система управления предприятием. Цель создания системы управления базами хранение, извлечение, обновление и удаление информации. Столь же обще можно определить и цель системы управления предприятием – автоматизация основных бизнес процессов управления предприятием. Такой формулировки цели на первом шаге вполне достаточно.

Уровни иерархии

Определение уровней иерархии является ответственной задачей. Как правило, иерархические уровни достаточно явно представлены в предметной области и задачей разработчиков является фиксация этих уровней в модели. Но бывают ситуации, когда декомпозиция по уровням неочевидна, однако дополнительное межуровневое деление не вызывает серьезных осложнений, так как все изменения локализованы в рамках одного уровня.

Уровни иерархии системы отличаются видами решаемых задач, а, следовательно, и теми интерфейсами, которые имеет каждый уровень. Любой верхний уровень реализует свою функциональность через интерфейсы нижних уровней. Верхний уровень может определять/использовать интерфейсы нескольких нижних уровней.

Уровни системы могут иметь подуровни вложенности. Объекты верхнего подуровня агрегируют объекты нижнего подуровня, так же как и при делении на основные уровни. Отличие подуровня от уровня состоит в том, что подуровень не имеет собственной спецификации интерфейса. То есть, все объекты одного уровня содержат реализацию того интерфейса, который специфицирован для данного уровня независимо от того, какому подуровню они принадлежат. Типичным примером деления на подуровни может служить уровень графических примитивов системы графического пользовательского интерфейса GUI. Этот уровень содержит элементарные графические примитивы: точки, прямые и кривые линии. Из элементарных графических примитивов складываются композитные графические примитивы, такие как, многоугольники, овалы, шрифты, образы (картинки). Но при этом и элементарные, и композитные примитивы остаются примитивами и реализуют один и тот же интерфейс. Другой пример деления на подуровни будет рассмотрен в примере проекта реляционной системы управления базами данных.

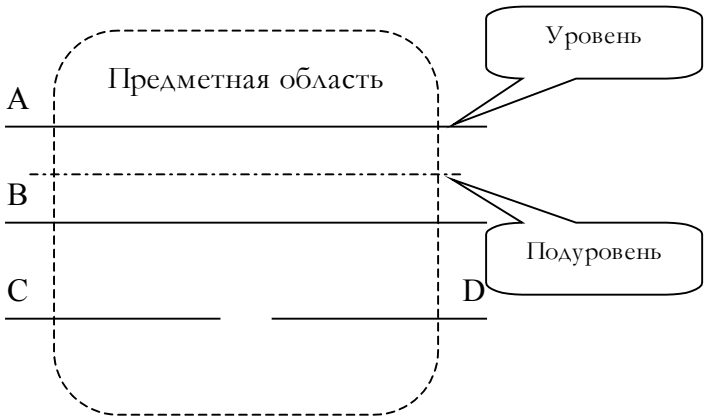

Рис. 2. Деление на уровни и подуровни

Определение подуровней происходит при разработке уровня, именно тогда определяется, что уровень содержит контейнеры, агрегирующие классы того же уровня.

Описание интерфейсов уровней

После того, как определены уровни, необходимо специфицировать интерфейсы каждого уровня. Спецификация интерфейса – это определение и описание семантики уровня в виде набора функций. При разработке спецификации интерфейса происходит определение задач, которые решаются на каждом уровне. Описание задач обязательно включает информацию о тех параметрах, которые необходимы для решения каждой из задач.

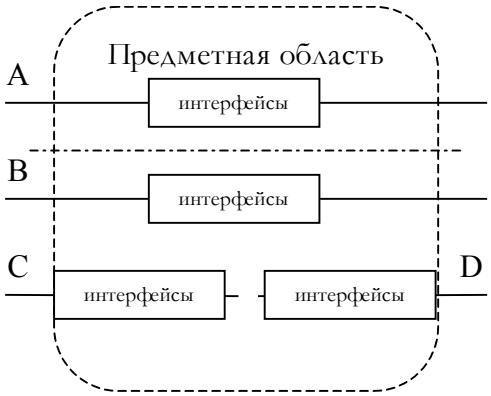

Рис. 3. Разработка интерфейсов между уровнями

Описание задач должно проводиться в направлении от общего к частному. То есть, первоначально определяются ключевые задачи, задачи, которые потребовали введения данного уровня, затем описываются частные и вспомогательные задачи.

Если последовательно рассматривать интерфейсы от верхнего к нижнему уровню, то они образуют спецификацию, отражающую декомпозицию задач предметной области. Развитие представлений о предметной области увеличивает спецификацию (количество интерфейсов различных уровней).

Разработка инструментов

После того, как определены уровни иерархии системы, и задачи, которые решаются на каждом из уровней, необходимо определить то, какие инструменты в наибольшей степени удовлетворяют условиям разработки каждого конкретного уровня. Для работы на низких уровнях достаточно традиционных языков программирования таких, как ассемблер, C, Pascal и т.п. На более высоких уровнях требуются средства разработки максимально приближенные к предметной области. Средства разработки должны иметь семантическое соответствие своему уровню. Это позволит подключить к решению конкретных задач пользователей.

Поскольку не существует высокоуровневых программных инструментов для любой предметной области, то их разработка является самостоятельной и важной задачей, решаемой при разработке проекта сложной системы. Ниже будут изложены те преимущества и особенности, которые имеет подход к разработке проекта, ориентированный не столько на решение конкретных задач, сколько на разработку инструментов для решения классов задач.

Разработка классов внутри уровня

После того, как полностью или частично определены интерфейсы между уровнями иерархии, возможна разработка классов объектов, расположенных на каждом из уровней. Интерфейсы, с одной стороны, образуют спецификацию свойств объектов нижнего уровня, а, с другой стороны, являются механизмом виртуализации объектов.

Механизм виртуализации позволяет объектам верхнего уровня иерархии гибко манипулировать включёнными объектами нижнего уровня, добавляя, заменяя или удаляя их по мере необходимости.

Интерфейсы с верхним уровнем обычно образуют свойства абстрактного класса, суперкласса всех классов данного уровня (*Рис. 4A*.). Но вполне допустимо постепенная реализация интерфейсов в процессе наследования (*Рис. 4B*). В этом случае подклассы реализуют те интерфейсы, которые не были реализованы в суперклассах. Тогда компилятор или программная система должны проверять возможность использования конкретных объектов в тех или иных ролях, на заданном подмножестве интерфейсов.

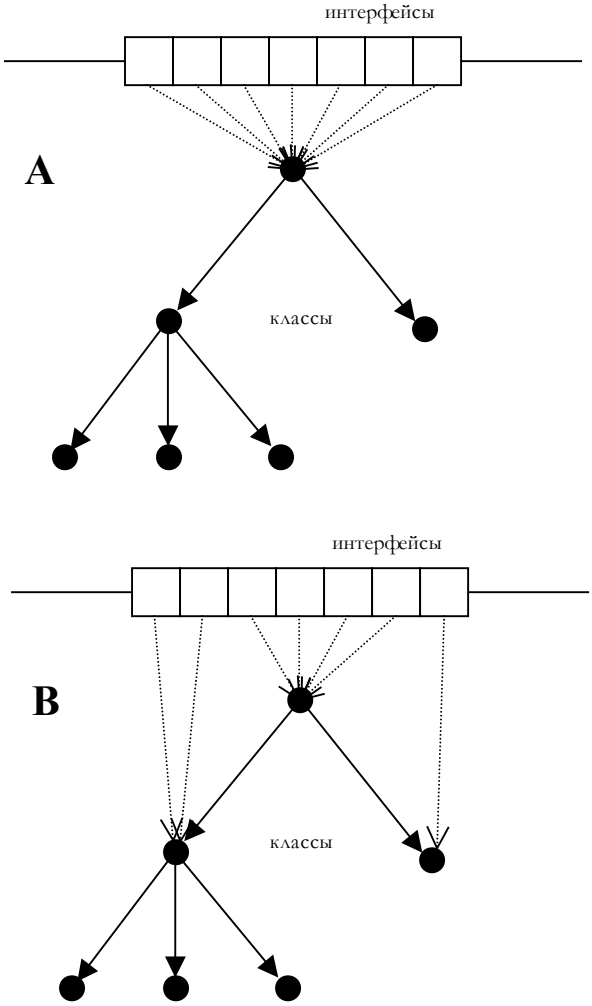

Рис. 4. Проекция интерфейса уровня на свойства классов

Развитие интерфейса с верхним уровнем, как правило, связано и с развитием классов нижнего уровня. Новая функциональность определяет направление развития классов нижнего уровня. Но развитие классов нижнего уровня может происходить и при неизменном интерфейсе с верхним уровнем. Такое развитие можно назвать вариантным, то есть, таким, когда идёт совершенствование классов, приспособлением их для различных требований и условий при сохранении неизменного интерфейса. Так, например, развитие автомобилей (транспортный уровень) в рамках задачи перевозок, происходило в двух направлениях повышение комфорта при перевозке людей и повышение грузоподъемности при перевозке грузов.

Цикличность

В известной статье Никлауса Вирта («Program development by stepwise refinement», CACM, 1971. Vol. 14, N 4. Apr. p. 221-227) разработка программного обеспечения рассматривается как последовательность уточняющих шагов. Первоначально даётся грубый метод решения, который затем уточняется до тех пор, пока не будет получен удовлетворительный результат. К такому же выводу, но с иной стороны, приходит и Фредерик Брукс в известной книге («Мифический человеко-месяц или как создаются программные системы», СПб, Символ-Плюс, 1999). В частности он пишет: «Поэтому проблема не в том, создавать или нет опытную систему, которую придётся выбросить. Вы всё равно это сделаете». А далее, в разделе «Постоянны только изменения», он добавляет: «После уяснения того, что опытную систему нужно создать, а потом выбросить, и что перепроектирование с новыми идеями неизбежно, полезно обратиться к изменению как явлению природы. Первый шаг – признание того, что изменение – это образ жизни, а не постороннее или досадное исключение» (стр. 108-109). К этим словам имеет смысл прислушаться.

Рассматривая проект сложной системы так, как предложено выше, имеет смысл ещё раз акцентировать внимание на том, что начало работ по проекту не предусматривает детального анализа предметной области. Развитие системы может продолжаться бесконечно долго, в том числе и без участия разработчиков системы, силами самих пользователей. Процесс развития системы имеет ярко выраженный итерационный характер. При этом могут добавляться новые уровни и подуровни, наращиваться интерфейсы, образовываться новые классы и т.д. После того, как определён уровень и интерфейсы, происходит его непрерывное развитие от простейших реализаций до самых сложных.

Циклы разработки различных уровней независимы друг от друга и могут иметь различную протяжённость во времени. Поскольку взаимодействие между уровнями осуществляется только посредством интерфейса, то нет никаких причин осуществлять синхронизацию работы различных команд, создающих смежные уровни. Точно также любой из уровней может быть завершён задолго до окончания работ по всему проекту. Разработчики могут считать, что работы по созданию того или иного уровня иерархии системы завершены, если уровень реализует весь заявленный в интерфейсе функционал с должным качеством, а также содержит необходимые инструменты для развития этого уровня силами пользователей или других разработчиков.

Параллелизм

Одним из важных принципиальных отличий предлагаемого метода разработки проектов сложных систем является высокий параллелизм работы всех групп разработчиков. Следствием параллельной работы является высокая скорость разработки, которая определяется времени разработки самого сложного и трудоёмкого уровня. Не менее важно и то, что параллелизм разработки существенно снижает затраты на администрирование и управление проектом, делает более простым контроль, что в конечном итоге повышает качество проекта.

Но параллельность допускается не только между уровнями, но и внутри любого уровня. Поскольку ветви иерархии наследования, определяемые на каждом из уровней, также разрабатываются независимо, то и при разработке уровня тоже происходит распараллеливание работ.

При создании проекта под заказ, внедрение проекта происходит не тогда, когда работы над ним завершены, а непосредственно в процессе разработки. Пользователи осваивают продукт параллельно с его развитием и совершенствованием. Это является дополнительным фактором снижения времени и стоимости проекта. Мы опробовали такую методику внедрения на проекте системы управления предприятием.

Организация работы над проектом

Упрощение администрирования проекта позволяет сделать качественные изменения в организации работ. Действительно, как уже отмечалось в пятом письме, при методе "waterfall" требовалось постоянное административное вмешательство для синхронизации работы различных групп разработчиков. Теперь все группы работают независимо и параллельно, ориентируясь на конечный результат, что позволяет говорить о процессной организации работы. Не существует никаких причин, по которым не следовало бы передавать право принимать проектные решения для любого уровня иерархии системы соответствующей группе разработчиков. Конечно, такая передача потребует от разработчиков большей творческой отдачи, большей квалификации и взаимозаменяемости, но при этом и работа становится более интересной. Таким образом, данный подход стимулирует повышение уровня подготовки и творческой самореализации разработчиков.

Изменения, связанные с развитием проекта, на каждом из уровней обсуждаются и принимаются внутри группы (пожалуй, теперь уже надо говорить – бригады) разработчиков. И задачей администрирования, в таком случае, является только контроль достижения цели проекта и проверка соответствия решений поставленным задачам. Роль руководителя проекта сводится к координации направлений развития проекта. Руководство проектом лучше переложить на координационный совет, в который могут входить войти архитектор системы, руководитель системы качества и руководители групп. Задачей совета является выработка и принятие стратегических проектных решений, согласование глобальных изменений, а также контроль сроков выполнения проекта.

Количество разработчиков в каждой из групп зависит от сложности уровня, на котором она работает. При завершении работ на одном из уровней разработчики могут и должны быть распределены по другим группам. Это одна из важнейших форм ротации, а ротация в любой форме способствует повышению мастерства разработчиков и созданию у них полной картины о создаваемой системе.

Заслуживает внимания и снижение конфликтности в коллективе разработчиков, по сравнению с традиционным методом последовательной разработки, типа "waterfall". Устранение зависимостей одних групп разработчиков от других, с одной стороны, снижает аритмию работ, а самостоятельность принятия проектных решений группой, с другой стороны, возлагает всю полноту ответственности на группу и исключает перекладывание ответственности на другие группы.

Документация

Проектная документация должна отражать существо работ по проекту. Как уже отмечалось ранее, сложная система является иерархичной и каждый уровень является независимым от других, осуществляя связь со смежными уровнями только через декларированный интерфейс. Архитектура системы должна быть отражена и в проектной документации, где отмечаются уровни иерархии системы, задачи каждого уровня и интерфейсы между уровнями. По каждому уровню формируются показатели, которые должны быть достигнуты, и методики проверки реальных значений данных показателей. Показатели, как правило, тесно связаны с тем интерфейсом, который представляется текущим уровнем своему верхнему уровню.

Разработка каждого уровня иерархии оформляется собственным набором документов. В этой документации фиксируются задачи, решаемые уровнем, модель уровня, средства разработки, сроки и стоимость работ, а также разработчики и их функции. В процессе разработки в проектную документацию вносят описания классов, которые населяют данный уровень, а также логику схем контейнеров.

Переход к итерационной методике разработки связан с привнесением понятия версии для каждого из уровней системы. В силу независимости уровней, количество

итераций разработки на каждом уровне может отличаться от количества итераций на любых других уровнях. Для каждой итерации в документации указывается её начало, решаемые в рамках итерации задачи, а также те значения контрольных параметров, которые должны быть достигнуты по завершению этапа разработки.

Взаимодействие с конечными пользователями

Взаимодействие с конечными пользователями характерно в случаях, когда проект сложной системы реализуется под заказ. Приняв за основу предлагаемый подход к разработке проектов, можно пересмотреть роль конечных пользователей. Традиционно конечные пользователи рассматриваются только в роли экспертов-покупателей. Они оценивают возможности и качество программного продукта или предлагаемого решения и принимают решение о покупке/заказе. В процессе работы над проектом пользователи привлекаются в качестве экспертов для оценки того или иного проектного решения. Собственно этими функциями обычно ограничивается роль конечных пользователей, не считая конечно, их повседневной работы с выбранным программным обеспечением.

Существо изменений по отношению к конечным пользователям состоит в том, что они становятся полноправными разработчиками проекта и продолжают развивать систему даже после того, как основная команда разработчиков завершила работу над проектом.

Разработчики обычно лучше представляют архитектуру будущей системы, а пользователи – структуру задач, решение которых должна обеспечить система. Совместная работа пользователей и разработчиков обеспечивает полноту представлений о будущей системе. Взаимодействие с конечными пользователями начинается с первого этапа – формирования контура предметной области. На данном этапе конечные пользователи формируют цель проекта системы, а разработчики на этой основе предлагают архитектуру системы.

В случае если архитектура соответствует поставленной цели, то есть в ней выделены все необходимые уровни, то переходят к следующему этапу – формированию интерфейсов каждого из уровней. При формировании интерфейсов уровней пользователи должны сформулировать классы задач, которые решаются на каждом из уровней. Разработчики, сделав анализ этих задач, должны представить интерфейсы каждого из уровней, а также предложить пользователям модель инструментальных средств, которые адекватно отражают существо решаемых задач. Инструментальные средства должны ориентироваться на весь класс задач, но не на отдельное подмножество. И, конечно, инструментальные средства должны быть эргономичны и удобны в работе.

Далее разработчики создают прототипы уровней, где реализуются первые варианты классов и инструментальных средств. Первые варианты классов включают абстрактные и наиболее существенные и простые для реализации классы. Соответственно инструменты должны позволять собирать классы и описывать их логику в виде схем. Созданные инструментальные средства передаются конечным пользователям для апробации и тестирования. Замечания и предложения пользователей относительно инструментальных средств оформляются в виде протокола, на основании которого происходит дальнейшее совершенствование и развитие классов, а также инструментальных средств каждого уровня.

Раннее вовлечение конечных пользователей в работу над проектом позволяет решить целый ряд серьёзных проблем. Во-первых, пользователи знакомятся и начинают работать с системой на очень ранних стадиях, что практически исключает необходимость в последующем обучении, или, по крайней мере, значительно сокращает стадию обучения. Во-вторых, конечные пользователи, являясь соразработчиками системы, хорошо представляют не только возможности системы, но и пути её развития и совершенствования. А так как развитие системы определяется

именно конечными пользователями, то их знание системы способствует быстрому и успешному развитию. В-третьих, при совместной работе разработчиков и пользователей исчезает психологический барьер неприятия системы конечными пользователями. Пользователи видят, как их предложения и рекомендации находят отражение в системе, и это порождает у них чувство уверенности и даже некоторого патриотизма.

Какие проблемы могут подстерегать на этом пути? Наиболее распространённая ошибка состоит в том, что разработчики идут «на поводу» у пользователей, произвольно меняя структуру системы в процессе работы. Если требования пользователей противоречат логике системы, то это означает, что, либо система концептуально непродуманна и противоречива, либо пользователи хотят применять систему для тех задач, которые не соответствуют или противоречат декларированной цели. Если возникает данная проблема, то необходимо вернуться к формулированию цели проекта (определению контура предметной области), заново проверить правильность декомпозиции по уровням и согласовать требования пользователей. В ситуации, когда и цель была сформулирована верно, и декомпозиция по уровням выполнена правильно, то надо показать пользователям противоречие, которое порождается новыми требованиями и исходной целью проекта.

Другая проблема связана с многократной модификацией некоторых элементов проекта из-за появления новых требований, которые не противоречат исходной цели проекта. Эта проблема, как правило, вызывается тем, что разработчики вместо инструментов предлагают пользователям готовое решение. На примере с расчётом заработной платы, который рассматривался ранее, можно видеть, что если вместо инструментального средства предложить пользователям готовое решение, то это решение придётся модифицировать каждый раз, как только появятся новые условия начисления заработной платы или произойдёт модификация схемы начисления. Единственный способ решения данной проблемы состоит в создании удобного инструмента для пользователей, инструмента, который был бы адекватен решаемым задачам.

Одной из важнейших задач, которую необходимо решить при создании сложной системы под заказ, является задача составления бизнес-плана и календарного плана разработки. Заказчик должен иметь возможность контроля сроков и качества выполнения проекта. Разработка проекта итерационным методом имеет свои особенности, которые необходимо учитывать при подготовке проектной документации. Основная особенность состоит в том, что на каждый новый этап составляется своя спецификация, где должно отражаться не только существо работ, но и те показатели, которые должен иметь проект по завершении текущего этапа. Эти показатели имеют тенденцию к уточнению и расширению от одной стадии к другой, что позволяет говорить о методике разработки с повышением качества, в отличие от методики «water-fall», которая является методикой разработки с понижением качества. Формирование спецификаций на каждый новый этап с указанием, контрольных значений и методик проверки должно происходить совместно со специалистами заказчика. Соответственно и оплата работ по каждому из этапов должна выполняться с учётом составленных спецификаций.

Заключение

Преодоление сложности при построении систем является ключевой задачей и её невозможно решить за счёт исключительно организационных мероприятий. Можно посадить за работу сотни и тысячи программистов, и они напишут миллионы строк кода, но проблемы, связанные со стыковкой кода, с управлением изменениями, с согласованием графиков работы различных групп программистов, станут столь серьёзными, что получить работающую систему станет невозможно. Данная проблема подробно описана в книге Ф. Брукса.

Если же присмотреться к окружающей действительности, к тем самым сложным системам, которые окружают нас в повседневной жизни, то иерархичность любой сложной системы становится совершенно очевидной. Иерархичность систем является одним из важнейших свойств, на основе которого преодолевается их сложность. И наша задача, как разработчиков, состоит в том, чтобы моделировать системы, приближаясь к оригиналу и описывая те же иерархические уровни.

Разделив проект на независимые уровни иерархии, можно получить совокупную сложность всех уровней намного меньшую, чем сложность проекта моносистемы. Количество уровней не может быть большим даже для столь сложных систем, как живые существа, а сложность каждого уровня относительно невелика, по сравнению со сложностью всей системы.

Декомпозиция системы по иерархическим уровням основывается на агрегации. Агрегация позволяет очень просто создавать композиции, обладающие различными свойствами. Именно за счёт композиции достигается то фантастическое многообразие окружающего нас мира, которое поражает воображение, которое лежит в основе образования новых сущностей с новыми свойствами, и в конечном итоге, в основе развития мира.

Композиция требует иных описательных средств, отличных от традиционных языков программирования. Мало того, каждый уровень иерархии сложной системы может потребовать собственной инструментальной основы, и противление этому ведёт только к усложнению проекта. Изучением атома, молекулы, клетки и живого существа занимаются совершенно разные науки, с уникальной методологической и инструментальной базой. И было бы совершенно неправильно заставлять их всех использовать только одну основу.

Композиция требует развитых средств конструирования, позволяющих осуществлять сборку новых сущностей не только быстро и комфортно, но и с учётом тех законов и правил, которые существуют на каждом из уровней иерархии сложной системы. Однако появление развитых инструментальных средств неразрывно связано с передачей возможности развития системы конечным пользователям. Задачей разработчиков является создание основы системы, принципов, правил (законов) и инструментальных средств. Такое распределение ролей между разработчиками и пользователями в большей степени способствует повышению эффективности совместной работы. Пользователи при этом в минимальной степени зависят от программистов-разработчиков, а программисты перестают решать задачи за пользователей, сосредоточив основное внимание на архитектуре систем, совершенствованию внутренних механизмов, элементарных классов, интерфейсов и инструментальных средств.

Именно на этом пути следует искать ответ на вопрос о том, как сделать разработку сложной системы простой и эффективной. «Истина – не то, что доказуемо, истина – это простота».

