Методология разработки распределенных приложений на основе модели оболочки

В.П. Иванников, А.Н. Винокуров, К.В. Дышлевой,
В.Е. Каменский, А.В. Климов, С.Г. Манжелей,
В.А. Омельченко, Л.Б. Соловская

1. Введение

Предлагаемый подход ориентирован на разработку распределенных приложений с высокими требования к эффективности, и прежде всего, на приложения реального времени. В основе его лежат механизмы адаптации, обеспечивающие гибкие возможности оптимизации взаимодействий удаленных объектов.

На концептуальном уровне множество всех объектов распределенного глобального пространства представляется покрытым множеством специального вида подмножеств - оболочек. Каждая оболочка содержит некоторую совокупность семантически взаимосвязанных объектов. Оболочка также представляет собой особого вида объект. Могут существовать вложенные оболочки и федерации оболочек.

Любое взаимодействие двух удаленных объектов поддерживается прозрачным образом оболочками, содержащими эти объекты. Другими словами, объекты не видят оболочек, в которых они содержатся. Любой объект вызывает метод другого объекта независимо от взаимного расположения этих объектов (локального или удаленного), от свойств взаимодействующих оболочек (объектной модели, поддерживаемой конкретной оболочкой, организации мультидоступа к объектам оболочки, методов поддержки секретности внутри оболочки и т.п.) и от способов коммуникации между оболочками (может использоваться IIOP CORBA 2.0 [CORBA95] или любой другой специфический протокол).

Мы предполагаем, без потери общности, что используемый язык реализации является объектно-ориентированным языком со строгой типизацией. Следует четко различать две фазы в жизни объектов: фазу проектирования и фазу выполнения (вызовов методов объектов). Оболочки поддерживают обе фазы и организуют взаимное согласование объектов на обеих фазах. То есть, оболочки являются одновременно и средой проектирования объектов, и средой выполнения объектов.

Поскольку оболочки ответственны за поддержку согласованного прозрачного взаимодействия объектов, сами оболочки нуждаются в некоторой дисциплине согласования. В принципе, мы могли бы

представить себе однородный мир оболочек, отвечающих некоторому одному стандарту организации взаимодействия между объектами, например, CORBA [CORBA95]. Мы, однако, хотели бы предложить более разнообразный и богатый мир оболочечных моделей, обладающих не только разными свойствами, но и разными свойствами описания этих свойств, т.е. метасвойствами. Далее, поскольку мы имеем дело со строго типизированными языками, следует различать согласование оболочек на стадии проектирования объектов (статическое согласование оболочек) и на стадии выполнения (адаптивное динамическое согласование).

Для гибкого согласования оболочек естественно использовать технику метаобъектного контроля [KRB91]. Однако, как правило, метаобъектный контроль используется либо на фазе проектирования, либо на фазе выполнения.

В первом случае [Chi96] мы получаем высоко эффективный, но монолитный код, который мы не имеем возможности менять во время выполнения некоторым систематическим образом. Однако, в процессе распределенной обработки данных зачастую возникают ситуации, требующие проведения существенных изменений схем согласования оболочек. Например, это требуется в случаях миграции объекта (объект может переместиться в оболочку, имеющую другие свойства) или изменения свойств самой оболочки (например, изменения требований секретности в оболочке-предприятии). При второй альтернативе платой за исключительную гибкость (неограниченная возможность динамической трансформации) управления вызовами методов объектов во время выполнения является большая потеря производительности, по крайней мере в десятичный порядок [ZC95, ZC96]. Такая потеря является результатом использования интерпретации.

Наш подход в использовании метаобъектного контроля [IZKN96, IDZ97] есть некоторый компромисс между двумя этими альтернативами. Его идея заключается в следующем. На стадии проектирования мы подготавливаем различные варианты возможного метаконтроля для организации взаимодействия объектов (возможно и отсутствие метаобъектного контроля). И тем самым получаем высоко эффективные варианты кода. На фазе же выполнения используется тот или иной вариант. Более того, текущий вариант во время выполнения может быть заменен на другой. Это может понадобиться, например, когда требуется динамическое согласование оболочек.

Рассмотрим внутреннюю структуру оболочки. Как уже говорилось, она содержит некоторую совокупность семантически взаимосвязанных прикладных объектов. Кроме того, в ней находится метаконтекст,

содержащий метаобъекты. Метаобъекты могут управлять поведением прикладных объектов, обогащая их функциональность. Например, метаобъекты могут обеспечивать мультидоступ к прикладным объектам, или организовывать их долговременное хранение (persistency). В соответствии с принципом рефлективности, метаобъекты в свою очередь могут управляться другими метаобъектами.

Мы предполагаем также, что метаконтекст подвергнут ортогональной декомпозиции, т.е. метаконтекст разбит на функционально независимые (ортогональные) коллекции метаобъектов - метасервисы. Каждый из таких метасервисов обеспечивает тот или иной вид функциональности (например, организация мультидоступа, долговременного хранения данных, сбора статистики, процесса отладки программ).

Предположение об ортогональности чрезвычайно важно. Выполнение требования ортогональности метасервисов предоставляет следующие возможности:

- возможность естественным образом подключать прикладные объекты к различным функциональным элементам метаконтекста прозрачно для прикладных объектов;
- возможность развивать независимо различные функциональности, т.е. метасервисы;
- возможность добавлять новые метасервисы, т.е. проводить метанаращивание оболочек без изменения уже существующих метасервисов.

Необходимо отметить, что полная функциональная ортогонализация зачастую оказывается невозможной. Например, существуют алгоритмы, в которых поддержка мультидоступа и хранения объектов сильно взаимосвязаны. Будем действовать прагматически и для таких случаев вводить новый вид метасервисов, например, “мультидоступ-сохранность”.

Для подключения прикладных объектов к метаконтексту будем использовать специализированный метаобъект - интерфейсный объект (ИО). В терминах модели “клиент-сервер”, некоторый клиентский объект может вызывать метод серверного объекта либо непосредственно (не подключаясь к метаконтексту), либо через соответствующий ИО, выполняющий роль посредника. Вызов метода удаленного объекта всегда осуществляется через ИО (как минимум необходимо организовать пересылку параметров и результатов между объектами прозрачным для них обоих образом). Интерфейсный объект с одной стороны наследует интерфейс целевого (серверного) объекта, т.е. имеет те же методы, что и целевой объект. С другой стороны, как метаобъект, ИО имеет собственные методы для управления отработкой вызовов.

Отметим, что для того, чтобы подключение ИО было прозрачным для объектов приложений, в этих приложениях должен использоваться некоторый механизм позднего связывания объектов. То есть, объекты не связываются друг с другом статически (на этапе проектирования приложения). Объекты приложений устанавливают связь друг с другом и осуществляют собственно взаимодействие между собой посредством некоторой среды, не являющейся частью приложения (т.е. некоторой системной среды). Именно в функции такой среды и входит организация общения объектов через интерфейсные объекты, когда это необходимо.

Итак, ИО можно представлять себе как некоторый коммуникационный объект, который при любом вызове метода целевого объекта может осуществлять также вызовы метаобъектов в виде пред- (перед вызовом нужного метода целевого объекта) или пост- (после) функциональностей. Для организации таких пред- и пост-функциональностей существенным образом используется свойство ортогональности различных элементов метаконтекста.

Техника использования ИО позволяет поддерживать различного рода отношения между объектами: “один с одним” (клиентский и серверный объект работают через один, используемый только этой парой, ИО), “многие с одним” (все клиентские объекты вызывают методы одного серверного объекта через один общий ИО), “один с многими” (клиентский объект работает одновременно с несколькими целевыми объектами через один ИО) и “многие с многими” (несколько клиентских объектов работают с несколькими серверными через один ИО - multicasting).

Подчеркнем еще раз, что подключение к метаконтексту осуществляется прозрачно как для клиентских, так и для серверных объектов. Организация метаконтроля не затрагивает реализации приложений (клиента и сервера). В этом смысле можно говорить, что любой объект приложений рассматривается как “черный ящик”. Такая схема удобна для организации многих типов метаконтроля (например, синхронизация и сбор статистики). Однако, в некоторых случаях при организации метаконтроля может потребоваться доступ к инкапсулированным данным целевого объекта. Такой доступ, например, может быть необходим при обеспечении долговременного хранения объектов средствами метаконтроля (подобный пример обсуждается в Приложении С). В различных языках программирования внешний доступ к состоянию объекта может быть организован разными способами. Например, в языке C++ для того, чтобы объекты типа (класса) A имели доступ к состоянию объектов типа B, в объявлении класса B класс A должен быть указан как дружественный (friend class) [Str94].

В случае удаленных объектов, происходит естественное расщепление ИО на ИО клиента и ИО сервера (аналогия пары “стаб - скелетон” в архитектуре CORBA). Причем метаконтроль, производимый в этих двух ИО, может быть частично или полностью независимым. Когда нет полной независимости, необходимое согласование клиентского и серверного ИО обеспечивается соответствующими оболочками, в которых эти ИО находятся. Примером такого согласования может быть прозрачная компрессия-декомпрессия данных, передаваемых взаимодействующими объектами. Во время выполнения средствами оболочек может производиться одновременная согласованная трансформация связей клиентского и серверного ИО с метаобъектами, осуществляющими необходимую отработку данных. Такая дополнительная обработка данных может включаться или выключаться в ИО оболочками путем подсоединения к ним или отсоединения от них метаобъектов. Здесь ортогонализация также облегчает разделение зависимых и независимых компонент метаконтроля, и их изменение во время выполнения.

Коснемся теперь кратко вопроса реализации оболочечной модели и специфических свойств оболочек на стадии проектирования объектов. Для проектирования используется набор из трех языков: языка реализации, языка спецификаций и языка описания метаобъектных связей. Поскольку мы хотим иметь возможность получать надежный, высоко эффективный код, язык реализации должен быть объектно-ориентированным языком со строгой типизацией. В качестве языка спецификаций для описания типов объектов целесообразно использовать IDL. Такой выбор гарантирует выполнение одного из необходимых условий совместимости с CORBA. Мы не рассматриваем здесь способ получения IDL спецификаций целевых объектов: они могут быть написаны вручную, или быть получены автоматически из языка реализации обратной компиляцией в IDL.

Нам необходим также язык для описания различных вариантов подключения целевых объектов к метаконтексту, т.е. для описания наполнения интерфейсных объектов. Сформулируем кратко основные требования к этому языку:

- этот язык должен быть строго типизированным;
- он должен обеспечивать параметризацию описаний (использование generic types), позволяющую производить поэтапную детализацию (refinement) типа коммутации типа конкретного ИО;
- язык должен отражать ортогональность различных используемых метасервисов, т.е. он должен предоставлять возможность независимого описания подключения метаобъектов ортогональных метасервисов в процессе конкретизации типов;

- язык должен обеспечивать кластеризацию, т.е. описание различных вариантов метаконтроля целевых объектов.

Рабочее название предлагаемого языка - TL (Template Language).

Схема работы с этим языком состоит в том, что компилятор получает на входе тексты на TL и IDL, производя на выходе тексты интерфейсных объектов различного вида на языке реализации.

Для поддержки фазы проектирования оболочки необходимы описания ее объектов на всех трех языках, а также описания внешних оболочек и содержащихся в них объектах на языках IDL и TL. Процесс согласования некоторой оболочки A с некоторой внешней оболочкой B происходит путем конкретизации описаний на языке TL схем взаимодействия с объектами из оболочки B. При подстановке параметров в эти TL спецификации в качестве фактических параметров используются различные составляющие оболочки A.

Итак, предлагаемый подход конструирования распределенных систем базируется на нескольких основных идеях: ортогональной декомпозиции системы, метаобъектном контроле с использованием развитой техники объектных посредников, согласовании распределенных компонентов - оболочек.

Наиболее четко идея ортогональной декомпозиции формулируется в проекте аспектно-ориентированного программирования [KASP]. В работе подвергается ревизии и критике традиционный подход модульной декомпозиции программных систем. Действительно, в традиционном подходе смешиваются модули, соответствующие принципиально различным свойствам (аспектам) проектируемой системы: мультидоступа, коммуникациям, отладки и т.п. Авторы справедливо полагают, что именно это явление смешивания различных аспектов (tangling-of-aspects) является основной причиной сложности существующих программных систем. Аспектно-ориентированное программирование позволяет программисту описывать различные аспекты разрабатываемой системы в естественных, но разных формах (различных языках), а затем автоматически соединять эти описания, определив отношения между аспектами, в конечную исполняемую форму (на некотором языке программирования) с помощью специального синтезатора - Aspect Weaver.

В операционной системе Tigger [ZC96] также проводится ортогональная декомпозиция. Вызов метода любого объекта может быть перехвачен либо до, либо после обработки объектом вызова. Специальная системная компонента (Piglet Core) может направлять эти перехваты

метаобъектам, сгруппированным по ортогональным функциональностям (metaregions).

Метаобъектный контроль и рефлексия - широко используемая в объектно-ориентированном программировании техника проектирования гибких, динамически адаптивных программных систем (см. [KRB91], [MJD96]). Эта техника лежит в основе некоторых популярных программных продуктов [FDM94], [DF94], [NH96]. Иногда метаобъекты выступают посредниками между взаимодействующими объектами [GC96]. В последней работе производится специализация метаобъектов по категориям реификации: по вызову метода объекта, по доступу к состоянию объекта, по доступу к исходному коду во время выполнения и т.п.

Потребности трансформаций во время вызова метода объекта приводят к использованию межобъектных посредников. Иногда, следуя объектно-ориентированной технологии, эти посредники реализуются специальными объектами. Так, в ORBIX ([IONA96]) предусмотрено несколько видов такого рода посредников. Это прежде всего фильтры. Фильтры дают возможность добавлять пред- и постфункциональности как к избранному объекту, так и тотально ко всем работающим через брокер объектам. По существу, они реализуют промежуточный перехват (trapping) объектных вызовов и дают возможность добавлять данные к параметрам вызова на стороне клиента и использовать их на стороне сервера. Согласованность обработки клиентом и сервером не поддерживается и должна осуществляться приложениями. Другой вид посредника - это smart proxy. Этот объект программируется на исходном языке и осуществляет замену целевого объекта. В принципе, он сам может вызывать методы других объектов, включая и методы целевого. В CORBA Security ([COSS96]) предлагается очень близкий к фильтрам механизм (interceptor). Такого рода фильтры могут добавлять в передаваемое брокером сообщение информацию, идентифицирующую клиента, и соответственно, проверять ее на стороне сервера.

Предложения OMG по multiple interfacing привели к выделению еще одного вида объекта посредника - интерфейса ([MI97]). Основное его назначения - обеспечить объект несколькими интерфейсами и обеспечить к ним доступ со стороны клиента. Интерфейс не производит никаких трансформаций, служит только для транзита параметров и, по существу, обобщает понятие объектной ссылки.

Приступим теперь к более детальному описанию особенностей и возможностей предлагаемого оболочечного подхода.

2. Определение понятия оболочки

Для работы с семантически взаимосвязанными совокупностями объектов мы вводим понятие оболочки. Оболочка позволяет работать с объектами как на этапе проектирования системы, так и на этапе выполнения (run time). Каждая оболочка является отдельным объектом. Важной особенностью описываемого подхода является то, что он представляет собой технологию наращивания семантики приложений и их взаимного увязывания без внесения изменений в сами приложения. Другими словами, помещение готовых объектов в контекст оболочек происходит для этих объектов прозрачным образом. Таким образом, этот подход рассчитан на организацию окружения (среды) для объектов из некоторых приложений, которые могут быть написаны вне зависимости от этого оболочечного подхода.

На этапе проектирования системы абстракция оболочек используется для выделения групп объектов различного типа, предназначенных для решения общих задач. Оболочку на этом этапе можно представлять себе как среду хранения информации, описывающей объекты, то есть метаинформации. В частности, от оболочки можно получить информацию о интерфейсах ее объектов, о типах данных, с которыми они работают. Здесь важно отметить, что мы предполагаем строгую типизацию этой информации об оболочке. Такой информации должно быть достаточно для организации обращений к оболочке и ее элементам на любом языке программирования со строгой типизацией (C, C++, Pascal, ...).

Кроме того, оболочка на этапе проектирования предоставляет описания типичных механизмов взаимодействия с элементами оболочки. Эти описания отражают семантику наполнения оболочки. Механизм взаимодействия описывается в виде параметризованного шаблона, в котором зафиксированы механизмы работы с некоторыми элементами оболочек по определенным стандартным правилам. Проектирование взаимодействия объектов из различных оболочек может зачастую сводиться к использованию в оболочке клиента таких шаблонов с подстановкой параметров, определяемых клиентской стороной (статическое согласование оболочек).

В качестве последнего информационного элемента оболочек на этапе проектирования отметим предоставление оболочкой информации о способах динамического регулирования (настройки) работы объектов, которые предусмотрены механизмом оболочки для этапа выполнения (информация о возможностях динамического адаптивного согласования оболочек).

Нужно заметить, что все указанные здесь виды сведений, содержащиеся оболочкой на этапе проектирования, необходимы как для разработки согласованного объектного пространства внутри этой оболочки, так и для проектирования других оболочек, объектные пространства которых будут взаимодействовать с данной оболочкой и ее объектами.

На этапе выполнения программы оболочка представляет собой среду существования и функционирования объектов, контролирующую и время их жизни (создание и удаление объектов, т.е. функции “фабрики” объектов), и организующую на протяжении всего этого времени их сопряжение с “окружающей средой”, то есть как с объектами этой же оболочки, так и с элементами других оболочек. Оболочка полностью контролирует все находящиеся в ней объекты. В частности, она организует доступ к этим объектам. Получить доступ к объектам оболочки можно лишь обратившись с соответствующим запросом к этой оболочке. Здесь важно отметить, что функционирование оболочек должно быть прозрачным как для клиентских, так и для серверных приложений. Так, любые клиентские и серверные объекты должны находиться в некоторых оболочках. Все обращения объектов друг к другу прозрачным (для самих объектов) образом отрабатываются в контексте этих оболочек. Вопросы взаимодействия объектов решаются путем согласования соответствующих оболочек.

Собственно для организации взаимодействия различных элементов оболочки между собой и с внешним миром в каждой оболочке предусмотрено коммуникационное ядро (часть объекта-оболочки, не обязательно выделенная в отдельный объект). Такое ядро обеспечивает некоторый базовый механизм организации взаимодействия между объектами. Коммуникационное ядро инкапсулирует в себе определение взаимного расположения объектов и организацию, в зависимости от результатов такого определения, использования того или иного механизма взаимодействия этих объектов (например, в целях повышения эффективности весьма по-разному может быть организовано взаимодействие между локальными и удаленными объектами). Таким образом, подобное ядро оболочки обеспечивает прозрачность взаимного расположения объектов (location transparency). При этом, работа объектов оболочки с ядром организуется с помощью интерфейсных объектов, прозрачное использование которых обеспечивается за счет механизма позднего связывания этих объектов (о котором уже говорилось ранее во Введении).

Что касается согласования с внешним миром на этапе выполнения, то здесь можно выделить функциональность, которую должна иметь любая оболочка, а именно: предоставление информации о содержащихся в ней объектах и организации поиска объектов по именам. То есть, оболочка выполняет функции сервиса именования. При этом, разрешение имен не обязательно должно происходить лишь в пределах данной оболочки. Может использоваться механизм именования в некотором более широком смысле, например, могут поддерживаться федерации оболочек (см. [ANSA93]).

Необходимо отметить, что оболочки содержат также некоторые служебные объекты, предназначенные для обеспечения функционирования объектов из приложений. Важной особенностью этих объектов является то, что они не видны для самого приложения, то есть объекты приложения не содержат их явных вызовов. Такие дополнительные объекты в каждой оболочке в совокупности представляют собой некоторый набор сервисов, таких как сервисы обеспечения одновременного доступа к объектам (concurrency), сервисы сбора статистики и обработки информации о использовании объектов, сервисы обеспечения долговременного хранения объектов (persistency) и другие. Таким образом, каждый управляемый оболочкой объект находится в некоторой среде, управляемой этой же оболочкой и состоящей из различных сервисов. При этом, и состав этой среды и механизмы увязывания объекта с ее элементами могут меняться оболочкой на этапе выполнения.

Очень важно выделить независимые (ортогональные) сервисы в оболочке. Это необходимо для удобства разработки схем взаимодействия с элементами оболочки на этапе проектирования и для обеспечения простых и естественных способов управления объектами оболочки с помощью этих сервисов на этапе выполнения. На обоих стадиях ортогональность сервисов позволяет работать в отдельности лишь с интересующей частью системного наполнения оболочки. Появляется возможность безболезненно добавлять (производить upgrade) новые сервисы на этапе разработки.

Приведем теперь простейший пример организации работы оболочек. В этом примере любое приложение, рассчитанное на использование в рамках оболочечного подхода, использует технику позднего связывания, обеспечиваемую наличием операций следующего вида (на языке IDL):

```
A get_objectA_ptr (in string object_name);
void release_objectA_ptr (in A obj);
```

Выполнение этих операций обеспечивается оболочкой, в которой находится клиент. При этом, операция get_objectA_ptr возвращает указатель на серверный объект некоторого типа A из оболочки A_cover. Идентификатор (имя) этого объекта определяется параметром операции. Для организации взаимодействия данного объекта (клиента) с внешним объектом (типа A) необходимо сначала получить ссылку на этот объект с помощью вызова подобной операции. Кроме того, клиент должен будет объявить о завершении использования выданной ему ссылки на объект с помощью вызова операции release_objectA_ptr.

Наличие такой техники работы с указателем на объект дает возможность системе программирования возвратить клиенту ссылку не на сам объект, а на его представитель (посредник) - интерфейсный объект.

Оболочка A_Cover в простейшем виде может представлять собой объект, управляющий однотипными прикладными объектами (типа A) и иметь кроме методов, обеспечивающих именование его элементов, также два метода:

```
status get_object (in string name, out A result);
status delete_object (in A arg);
```

Управление объектами типа A в оболочке A_cover состоит в поддержании некоторого массива M, содержащего пары “имя объекта - указатель на объект”. При этом, выполнение метода get_object заключается в выдаче указателя на объект, либо найденный по имени в массиве M, либо вновь созданный. В последнем случае появляется новая запись в массиве, описывающая этот объект. Метод delete_object удаляет указанный объект и соответствующий ему элемент массива M.

При этом выполнение операции get_objectA_ptr в клиентской оболочке сводится к обращению к методу get_object оболочки A_cover. Для общения с объектом, ссылка на который получена из этого метода, клиентской оболочкой создается ИО, ссылка на который и выдается клиенту как результат выполнения этой операции.

3. Реализация оболочек с помощью техники интерфейсных объектов

Итак, для реализации описываемой концепции оболочек необходим такой механизм организации оболочек, который бы подчинялся следующим принципам:

- системные сервисы оболочки должны быть *ортогональными*;
- должна поддерживаться *строгая типизация* данных, с которыми работает оболочка и составляющие ее объекты;

- должна обеспечиваться *прозрачность* добавления механизма оболочек как для серверных, так и для клиентских приложений;
- оболочка должна предоставлять *параметризованные* шаблоны для работы с ее элементами;
- оболочка должна обеспечивать возможность *динамически настраивать* механизмы работы с ее объектами.

Воспользуемся для реализации концепции оболочек расширенной техникой метаобъектного контроля. В основе этой техники лежит положение о том, что расширение базовой семантики приложений и организация взаимодействия между объектами различных оболочек происходит не за счет внесения изменений в клиентские или серверные объекты, а посредством особой отработки вызовов, с которыми клиентские объекты одной оболочки обращаются к серверным объектам некоторой, вообще говоря, другой оболочки. В процессе такой отработки к вызову некоторого объекта из приложения, который будем называть целевым объектом, прозрачно для пользователя добавляются вызовы других объектов оболочки, также не видимых для пользователя приложения. Эти объекты, которые мы будем называть метаобъектами, формируют метаобъектный контекст (метаконтекст) приложения. Некоторые составляющие метаконтекста (группы метаобъектов), имеющие родственную семантику, будем называть метасервисами. Примерами таковых могут служить обеспечение постоянного хранения объектов, поддержка многопользовательского режима доступа к объектам, авторизация доступа к объектам и т.п.

Можно также говорить о том, что с помощью подобного механизма мы производим подключение прикладного объекта к метаобъектной среде (метасреде), поддерживаемой оболочкой этого объекта. Под *метарасширением* приложения мы и будем понимать применение техники работы с оболочками для некоторого готового приложения за счет добавления дополнительных вызовов. Сам же процесс такой расширенной отработки вызовов к прикладному (целевому) объекту будем называть метаобъектным контролем (метаконтролем).

Кроме того, предлагаемая техника использования оболочек учитывает то, что любая программная система продолжает меняться и после того, как ее первый раз начинают использовать. Описываемая технология поддерживает эволюцию системы таким образом, чтобы процесс добавления к метаконтексту оболочек новых метасервисов (*метанаращивание*) был бы как можно более безболезненным. Добавление нового ортогонального сервиса не приводит к переписыванию уже используемых механизмов метаконтроля.

Нужно отметить, что и метаконтекст и схема работы с ним может меняться в процессе выполнения программы как в оболочке клиента, так и в оболочке сервера, но это не влияет на видимый клиентом интерфейс объектов приложения.

Метаобъектный контроль предлагается осуществлять единообразным, в высокой степени независимым от контролируемого приложения образом. Конечно же, и сами объекты, составляющие метаконтекст некоторого приложения, подобным образом могут стать предметом метаконтроля (выполняется свойство рефлективности системы). То есть, в метаконтекст могут входить не только объекты, разработанные специально для организации метаконтроля, но и ранее разработанные объекты, которые подобно самому приложению, ничего не знают о метаконтроле. Механизм работы с такими объектами в пределах оболочки также может быть зафиксирован за счет метарасширения.

Однако, когда мы переходим к рассмотрению вопроса включения объектов в оболочку, возникают серьезные проблемы связанные с эффективностью реализации подхода. Дело в том, что традиционный способ подключения объектов к метасреде приводит к возникновению вложенных слоев (мета)интерпретации, что и является основным источником неэффективности таких рефлективных систем [MMAY95].

В рамках предлагаемого подхода авторами статьи разработан метод оптимизации рефлексии, основанный на пересмотре механизма подключения объектов к метасреде в пределах некоторой оболочки. Основная идея метода состоит в использовании особой разновидности метаобъектов - интерфейсных объектов (ИО) в качестве посредников между клиентским и серверным объектом. ИО производят связывание объекта с метаконтекстом. Именно они содержат в себе механизм организации дополнительных вызовов объектов метаконтекста. ИО может находиться как в оболочке клиента (клиентский ИО), так и в оболочке сервера (серверный ИО).

На Рис. 1. приведена схема организации работы двух взаимодействующих объектов как через клиентский, так и через серверный ИО. При этом, реально клиенту доступен интерфейс (***i1***) клиентского ИО, клиентский ИО работает с методами интерфейса (***i2***) серверного ИО, последний, в свою очередь, обращается к методам, входящим в интерфейс (***i3***) собственно целевого объекта. Здесь и далее договоримся на рисунках объекты приложений обозначать кружками, ИО - в виде эллипсов, метаобъекты - двойными кружками.

Как не трудно видеть, при использовании обоих ИО прозрачность работы интерфейсных объектов для клиента и сервера обеспечивается и в

том случае, когда интерфейс серверного ИО отличается от интерфейса целевого объекта (об этом будут знать только сами ИО). Пример, когда такое отличие интерфейсов может быть необходимо, приводится в Приложении В.

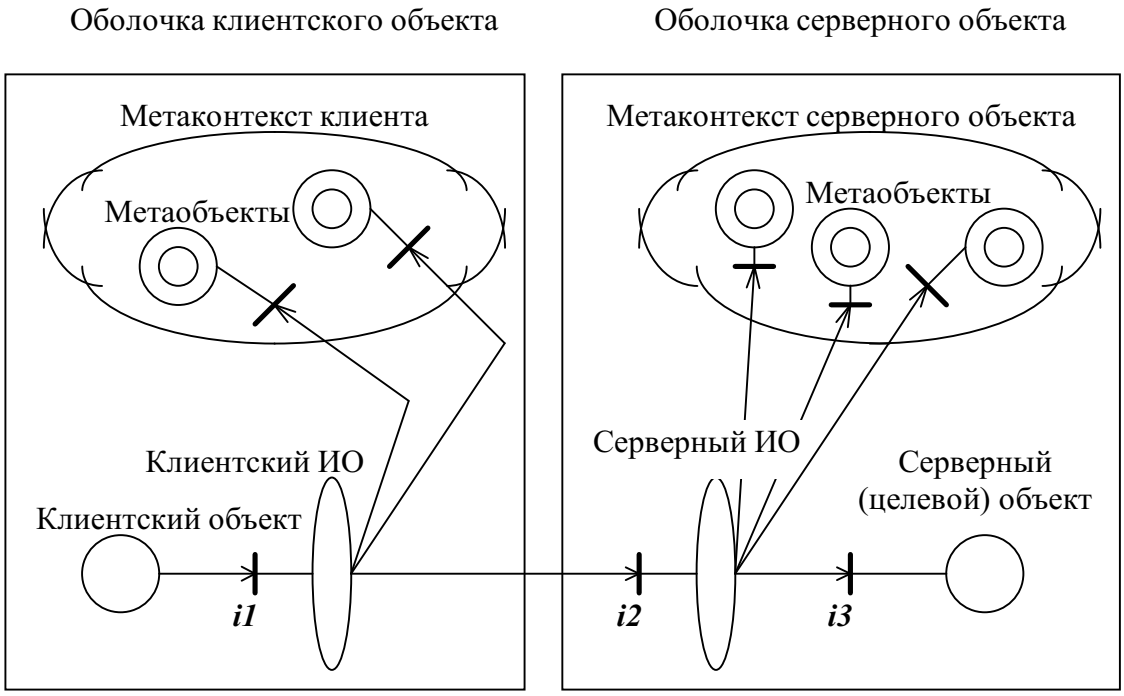

Рисунок 1. Схема организации метаконтроля.

В общем же случае возможны все четыре ситуации использования таких посредников:

- клиент работает с серверным объектом без посредников (ИО) - метаконтроль в этом случае не осуществляется;
- используется ИО в оболочке клиента - осуществляется подключение к метасреде клиента;
- используется ИО в оболочке сервера - осуществляется подключение к метасреде сервера;
- используется ИО в оболочке клиента и сервера - метаконтроль отрабатывается с помощью метаобъектов и клиентской, и серверной оболочек.

Отметим, что в случае взаимодействия удаленных объектов, когда для передачи параметров и результатов данные необходимо трансформировать (произвести *маршалинг*) в соответствии с некоторым протоколом, клиентский и серверный ИО организуют передачу данных через коммуникационные ядра оболочек. Каждое такое ядро оболочки

организует все необходимые преобразования и собственно процесс передачи данных в ядро другой оболочки. При этом предусмотренный в ядре оболочки базовый набор механизмов передачи данных может расширяться интерфейсными объектами за счет использования специальных метаобъектов (подробнее об этом говорится далее в разделе 7 и в Приложении А). Сами же ИО локальны по отношению к соответствующим объектам, поэтому механизм взаимодействия объектов с ИО не требует преобразования данных и не зависит от взаимного расположения объектов. Таким образом, с помощью использования пары ИО на обоих сторонах удаленного взаимодействия может быть обеспечена прозрачность взаимного расположения объектов (location transparency). То есть клиентский и серверный объекты могут ничего не знать о местонахождении друг друга, использование нужного в каждой конкретной ситуации варианта организации “доставки” данных обеспечивается соответствующими ИО.

Перейдем теперь собственно к рассмотрению вопроса, какова должна быть механика ИО, как она может использоваться, и как ее можно описывать на этапе проектирования оболочек.

4. Средства спецификации подключений объекта к метасреде оболочки

Напомним, что мы договорились описывать интерфейсы всех объектов, о которых здесь идет речь, в том числе ИО и метаобъектов, с помощью языка IDL (Interface Definition Language). При описании метаконтроля мы будем использовать некоторые соглашения этого языка спецификаций и не будем ориентироваться ни на какой конкретный язык программирования. Мы предполагаем только, что это строго типизированный и, вообще говоря, объектно-ориентированный язык.

Рассмотрим теперь упомянутый также ранее язык TL. Это общий для всех оболочек язык, с помощью которого мы сможем единообразным, не зависящим от приложения и среды программирования образом, описывать различные варианты организуемого в оболочках метаконтроля для разных типов клиентской и серверной среды. С помощью этого языка мы будем иметь возможность для каждого типа объектов (т.е. объектов с одинаковым интерфейсом, специфицированным на языке IDL) описать несколько типов (версий) клиентских и/или серверных ИО. Каждый ИО сможет иметь возможность подстраиваться под метасреду, то есть иметь некоторый набор параметров, которые можно было бы задавать и изменять во время выполнения программы. С помощью таких параметров у оболочки будет возможность в динамике (на этапе выполнения

программы) задать и поменять при необходимости объекты метасреды, с которой работает ИО, а также выбрать некоторый из предусмотренных статически (на этапе проектирования) вариант метаконтроля, то есть выбрать одну из нескольких возможных последовательностей обращений к метаобъектам.

Итак, язык TL предназначен для описания подключения объектов к метасреде. Компилятор из TL в некоторый язык реализации кроме подобных описаний работы с метасредой на входе имеет также IDL спецификации интерфейсов всех используемых объектов. На выходе же у этого компилятора - тексты для работы с ИО на нужном языке реализации.

Для иллюстрации указанных возможностей предлагаемого языка рассмотрим следующий пример.

4.1. Пример спецификации метаконтроля

Напомним, что при спецификации методов в языке IDL мы можем использовать три вида параметров :

IN - входные параметры;

INOUT - входные - выходные параметры;

OUT - выходные параметры.

Договоримся также, что без потери общности все рассматриваемые здесь методы являются процедурами.

В качестве примера рассмотрим задачу организации обмена представляемой в виде строк информацией между некоторыми объектами O1 и O2. Вызываемый для осуществления этого обмена объект O2 реализует некоторый интерфейс T. Единственным интересующим нас методом этого интерфейса является метод, который мы и будем использовать для передачи информации:

```
exchange_info (IN string arg_info, OUT string res_info);
```

Будем рассматривать ситуацию организации удаленного взаимодействия объектов O1 и O2. Предметом возможного метаконтроля здесь является компрессия/декомпрессия передаваемой информации с целью уменьшения накладных расходов при ее передаче по сети. Для осуществления такого метаконтроля необходимо спроектировать клиентский и серверный ИО, каждый из которых будет прозрачно для обоих объектов O1 и O2 организовывать компрессию данных перед их посылкой и декомпрессию после получения данных из сети.

Собственно операции компрессии и декомпрессии данных производятся специальными однотипными метаобъектами, реализующими некоторый интерфейс ZIP с двумя следующими методами:

```
compress (IN string info, OUT string compressed_info);
decompress (IN string compressed_info, OUT string info);
```

В оболочках C1 и C2, содержащих объекты O1 и O2 соответственно, механизмами самих оболочек создается по одному метаобъекту типа ZIP и производится подключение каждого из них к соответствующему ИО.

Метаконтроль для операции exchange_info клиентского и серверного ИО может быть описан с помощью предлагаемого языка TL следующим образом :

```
T {
    // общие описания для клиентского и серверного ИО
    OBJECT Zip_metaobject : ZIP;

    SERVER IS {
        exchange_info (compressed_arg, compressed_res) USE {
            variable arg_info : string;
            variable res_info : string;

            Zip_metaobject . decompress (compressed_arg, arg_info);
            // вызов метода целевого объекта:
            send_info (arg_info , res_info);
            Zip_metaobject . compress (res_info, compressed_result);
        };
        // описание метаконтроля серверного ИО для других методов
        ...
    };

    CLIENT IS {
        exchange_info (arg_info, res_info) USE {
            variable compressed_arg : string;
            variable compressed_res : string;

            Zip_metaobject . compress (arg_info , compressed_arg);
            // вызов метода серверного ИО:
            send_info (compressed_arg, compressed_res);
            Zip_metaobject . decompress (compressed_res, res_info);
        };
        // описание метаконтроля клиентского ИО для других методов
        ...
    };
};
```

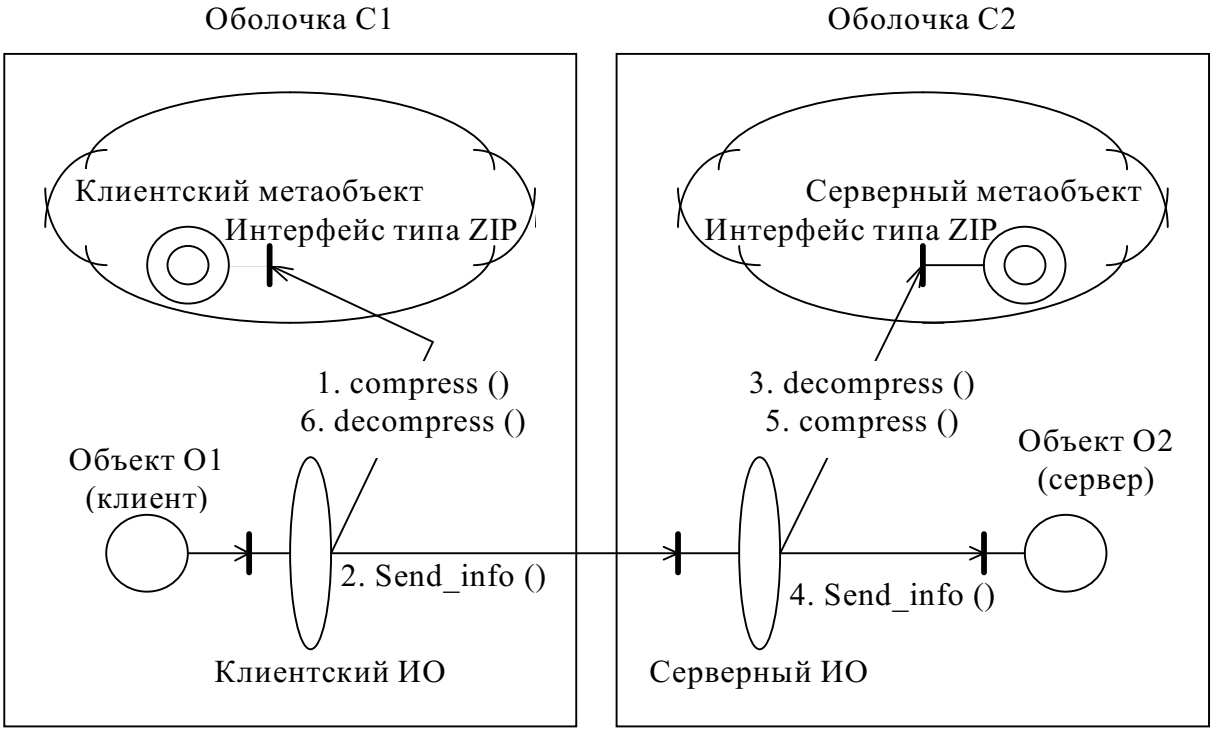

Рисунок 2. Пример организации метаконтроля.

В приведенном описании метаконтроля для обоих ИО (клиентского и серверного) зарезервирована объектная ссылка Zip_metaobject типа ZIP, которая представляет собой элемент состояния ИО, заполняемый оболочкой указателем на метаобъект, производящий преобразования данных. В обоих ИО в качестве реализации метода exchange_info указаны цепочки вызовов, содержащие два обращения к метаобъекту по этой объектной ссылке и вызов собственно метода целевого объекта (на Рис. 2 указаны все 6 вызовов, пронумерованных в порядке их выполнения). Для клиентского ИО таковым целевым объектом является серверный ИО, а для серверного ИО - это объект O2. Отметим, что для передачи данных между вызовами оказалось необходимо использовать временные переменные: compressed_arg и compressed_res в клиентском ИО, arg_info и res_info в серверном ИО.

4.2. Схемы выполнения операций ИО

В рассмотренном только что примере был указан единственный вариант метаконтроля, предусматривающий прозрачную для клиента и сервера компрессию и декомпрессию данных. Однако, выполнение этих дополнительных процедур также связано с потреблением машинных ресурсов. Поэтому имеет смысл предусмотреть еще один вариант метаконтроля для рассмотренного метода, когда данные пересылаются без

изменения. Таким образом, мы предоставим возможность на этапе выполнения оболочкам клиентского и серверного объектов согласованно решать, какой из двух вариантов, предусмотренных на этапе проектирования, необходимо использовать. При этом, оболочки будут иметь возможность поменять текущий вариант в динамике.

Итак, в рассмотренном примере в качестве альтернатив указанным последовательностям вызовов могут быть указаны последовательности из единственного вызова метода send_info. Для клиентского ИО:

```
{
    // вызов метода серверного ИО:
    send_info (compressed_arg, compressed_res);
};
```

и для серверного ИО:

```
{
    // вызов метода целевого объекта:
    send_info (arg_info , res_info);
};
```

Собственно механизм описания и использования подобных переключений вариантов реализован с помощью технологии кластеров и описан далее в п. 4.7.

Таким образом, в нашем подходе программист еще до компиляции задает возможный набор схем (вариантов) выполнения операций ИО. Каждая из этих схем имеет блочную структуру со своими переменными, инкапсулированными в блоках, и с четко “увязанной” по параметрам последовательностью вызовов. Во время работы интерфейсного объекта объемлющая его оболочка выбирает нужный вариант и именно этот вариант работает при вызове метода ИО. Важнейшей особенностью такого способа является возможность в любой момент поменять текущий вариант на любой другой из описанных в статике, согласованно в клиентской и серверной оболочках.

4.3. Базовые средства описания метаконтроля

На том же примере из п. 4.1. видно, что для описания вызовов в рамках метаконтроля необходимо использовать временные переменные для хранения промежуточных данных между вызовами методов. Это, конечно же, ставит вопрос об областях видимости таких переменных. Появляется необходимость выделения вызовов, работающих с одними и

теми же переменными в блоки, в которых такие переменные бы инкапсулировались.

Итак, любой вариант метаконтроля, то есть связанную, статически определенную последовательность вызовов объектов, в рамках предлагаемого нами языка описывается блоками, характеризуемыми следующими свойствами:

1. Каждый блок может быть вложен в некоторый другой блок.

2. В начале любого блока может присутствовать раздел описания локальных переменных.

3. Для вложенных блоков используются стандартные правила видимости переменных.

4. Для осуществления возможных переходов между блоками часть из них может иметь имя (метку).

Схематически описание блока выглядит следующим образом :

```
[Имя блока] {
    раздел объявления переменных;
        вызовы; описания блоков;
},
```

где каждый вызов представляет собой указание вызываемого объекта, некоторого из его методов, а также списка используемых в вызове параметров:

```
object . method (par_1, .. , par_n).
```

При этом, указание объекта может быть опущено лишь в том случае, если описывается обращение к целевому объекту конструируемого ИО:

```
target_method (par_1, .. , par_n).
```

Последовательность вызовов и вложенных блоков в TL может быть произвольной. Описания вложенных блоков имеют ту же структуру, то есть в них могут быть объявлены свои локальные переменные, кроме того, в них также доступны переменные всех объемлющих блоков по обычным правилам видимости.

4.4. Спецификация метаобъектной среды приложения

Конечно же, перед тем, как использовать в ИО объектные ссылки для организации вызовов методов метаобъектов, на которые эти ссылки указывают, все они должны быть описаны как элементы состояния ИО. Так, в приведенном в п. 4.1. примере было использовано следующее объявление:

```
OBJECT Zip_metaobject : ZIP;
```

Такое объявление вводит поле (элемент состояния) интерфейсного объекта - объектную ссылку Zip_metaobject типа ZIP. Также в интерфейсе этого ИО (доступном оболочке) появляется соответствующий этому полю атрибут (в терминах языка IDL) с теми же именем и типом. Этот атрибут предназначен для организации доступа к соответствующему полю. Однако, доступен он не для клиента, а для оболочки, содержащей ИО, точнее для некоторого метасервиса этой оболочки, управляющего согласованием этого ИО с метасредой этой оболочки и, возможно, согласованием с метасредой целевого объекта.

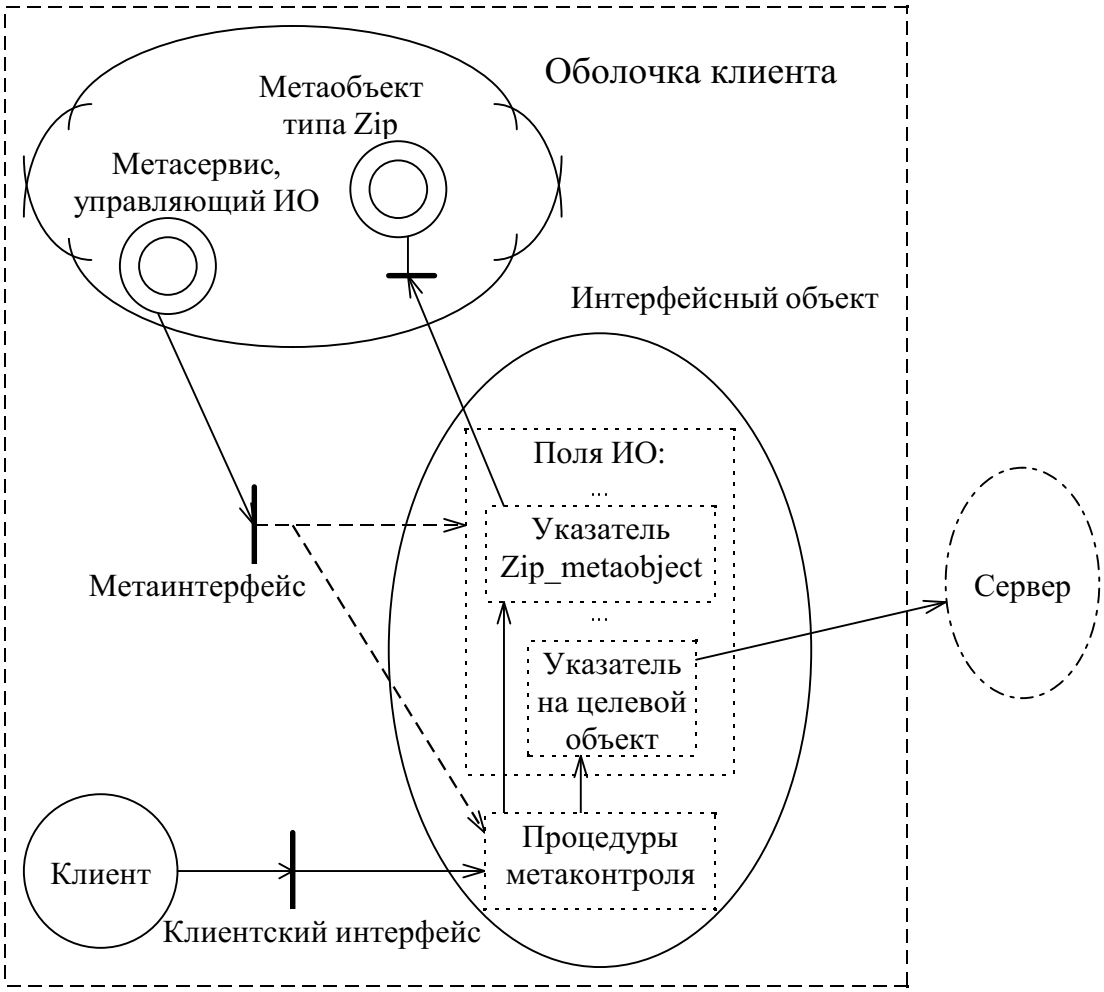

Рисунок 3. Схема работы интерфейсного объекта.

Таким образом, кроме видимого клиентом интерфейса ИО у этого же ИО есть недоступный и неизвестный клиенту интерфейс (метаинтерфейс), содержащий операции манипуляции состоянием этого ИО. Метаинтерфейс известен оболочке и используется ею и при

инициализации ИО в момент его создания, и для его перенастройки в дальнейшем. Объявления указанного вида с ключевым словом OBJECT наряду с обсуждаемыми ниже в п. 4.7. объявлениями кластерных множеств и являются источником формирования такого метаинтерфейса ИО.

На Рис. 3. схематически изображены оба интерфейса ИО. При этом под прерывистыми стрелками на рисунке подразумевается управление (возможность модификации), а под сплошными - лишь использование элементов, указываемых стрелкой.

Собственно метаконтроль на этапе выполнения будет осуществляться посредством вызовов в некоторой последовательности методов тех объектов, на которые указывают введенные таким образом объектные ссылки. Оболочка же, содержащая ИО, может с помощью динамической смены содержимого такого поля ИО, переключать метаконтроль на другой метаобъект, реализующий тот же интерфейс.

Нужно также отметить, что совокупность объявлений на языке TL таких четко типизированных объектных ссылок в качестве параметров ИО, на этапе проектирования можно воспринимать как отражение метасреды, в которую ИО может быть потенциально помещен оболочкой. То есть, все типы метаобъектов, с которыми потенциально может работать метаконтроль разрабатываемого ИО должны быть отражены в этих объявлениях.

4.5. Библиотеки шаблонов

При программировании схем метаконтроля, связанных с работой с одними и теми же типами объектов, часто приходится повторять практически совпадающие последовательности описаний. Это наводит на мысль о необходимости выделения некоторых последовательностей действий в *параметризованные* шаблоны (образцы), составляющие библиотеки шаблонов для того или иного типа объектов (сервисов). Такие библиотеки могут соответствовать как некоторым сервисам самим по себе, так и сервисам в контексте некоторого типа оболочек в том случае, когда оболочка доопределяет семантику использования сервиса и, таким образом влияет на возможные механизмы работы с этим сервисом. Совокупность таких библиотек для различных сервисов оболочки и может стать информационным элементом оболочки этапа проектирования, о котором мы говорили ранее в разделе 2.

Основная идея, на которой базируется построение таких библиотек, состоит в том, что семантика сервиса некоторой оболочки предопределяет определенный набор характерных для этих оболочки и сервиса

последовательностей обращения к методам этого сервиса. Так, например, для сервиса контроля доступа (concurrency) с операциями захвата объекта lock и освобождения объекта unlock возможна только одна последовательность вызовов сервиса:

```
lock ( rw_flag ) ;
Работа с контролируемым объектом ;
unlock ;
```

где параметр rw_flag задает необходимый режим захвата - на запись (rw_flag = WRITE) или на чтение (rw_flag = READ). В виде шаблона эта последовательность может быть задана следующим образом:

```
LockPattern ( rw_flag ) < concurrency_objref > { ControlledOperations } IS {
    A {
        concurrency_objref . lock ( rw_flag );
        DO ControlledOperations WITH EXCEPTION HANDLER B;
    }
    B {
        concurrency_objref . unlock ;
    }
}
```

Использованные в шаблоне идентификаторы имеют следующий смысл:

LockPattern - имя шаблона;

rw_flag - параметр шаблона - имя параметра операции lock;

concurrency_objref - параметр шаблона - имя сервиса контроля доступа;

ControlledOperations - параметр шаблона - контролируемые операции;

A, B - метки блоков;

WITH EXCEPTION HANDLER - ключевые слова, указывающие, что при возникновении исключительной ситуации (exception) во время выполнения контролируемых операций необходимо перейти на указанный блок (здесь блок B используется как отработчик исключительных ситуаций).

Таким образом, мы описали типовое подключение объекта из метасервиса concurrency к целевому объекту. Естественно, что описание конкретного подключения представляет собой некоторую *параметризацию* (конкретизацию) этого шаблона. Конкретизация же подобного шаблона состоит в указании конкретных имен параметров операций, имен объектных ссылок, а также в указании последовательности вызовов, место для которых зарезервировано с помощью параметров типа

ControlledOperations. Использование последнего параметра обеспечивает возможность, например, воспользоваться описанным шаблоном для различных операций одного и того же целевого объекта с однотипным метаконтролем. При спецификации интерфейсного объекта с помощью этого шаблона для всех таких операций в качестве фактического параметра ControlledOperations необходимо просто указать соответствующий вызов целевого объекта.

Для иллюстрации механизма использования описанного шаблона рассмотрим некоторый интерфейсный объект, интересующий нас атрибут которого (см п. 4.4.) описан следующим образом :

```
// ссылка на сервис контроля доступа (объект типа Concurrency)
OBJECT Concur_service : Concurrency;
```

Тогда для некоторого метода some_operation (in SomeType arg), который нужно всегда выполнять в режиме изменения объекта (записи), метаконтроль для этого ИО может быть описан с помощью указанного шаблона следующим образом :

```
some_operation (arg) USE {
    LockPattern (WRITE) < Concur_service > { OPERATION }
}
```

Отметим, что ключевое слово OPERATION, использованное в этой спецификации, обозначает обращения к тому методу целевого объекта, метаконтроль для которого описывается (в данном случае - some_operation) именно с теми параметрами, которые указаны для метода ИО (в этом примере - arg). Использование такого ключевого слова позволяет описывать в одной спецификации однотипный метаконтроль для целой группы методов:

```
method_1 (...), method_2 (...), ..., method_n (...) USE {
    ...
    OPERATION;
    ...
}
```

4.6. Использование композиций шаблонов

Итак, при проектировании метаконтроля для работы с сервисами из другой оболочки программист имеет возможность получить из этой оболочки параметризованные шаблоны для работы с нужным сервисом. В результате нет необходимости переписывать одни и те же тексты и

вдаваться в детали вызовов объектов этого сервиса. Кроме того, наличие таких библиотек позволяет *ортогонализовать*, то есть обособить, сделать независимым использование различных сервисов при описании метаконтроля. Программист теперь может, абстрагируясь от деталей организации обращений к методам сервисов, мыслить в терминах композиций конкретизаций шаблонов, каждый из которых воспринимается как некоторая крупная семантическая единица работы с конкретным сервисом из оболочки.

Рассмотрим теперь пример подключения ИО к двум независимым сервисам. Для этого продолжим рассмотрение примера с ИО, “оборачивающим” метод some_operation синхронизационным захватом, и допустим, что в этом ИО есть также атрибут - ссылка на сервис статистики типа Statistic:

```
OBJECT stat_obj : Statistic;
```

Этот новый сервис имеет методы begin_time и finish_time, назначение которых состоит в запоминании времен начала и окончания некоторой деятельности. Очевидно, что в библиотеке шаблонов для этого сервиса имеет смысл описать такой шаблон:

```
StatisticPattern <statistic_objref>      { ControlledOperations } IS {
    {
        statistic_objref . begin_time ;
        DO ControlledOperations ;
        statistic_objref . finish_time ;
    }
}
```

Композиция описанных выше шаблонов LockPattern и StatisticPattern может выглядеть так :

```
some_operation (arg) USE {
    LockPattern (WRITE) < Concur_service > {
        StatisticPattern < stat_obj > { OPERATION }
    }
}
```

Не вдаваясь в детали реализации используемых шаблонов, таким способом вызов some_operation можно “обернуть” и контролем доступа, и статистикой. Нужно заметить, что при указанной вложенности шаблонов вызовы сервиса статистики “обернуты” той же парой вызовов lock-unlock сервиса контроля доступа. То есть, фактически, в защищенном режиме

работает пара объектов - контролируемый серверный объект и сервис статистики. Если же нужно контролировать только вызов сервера, достаточно просто изменить композицию шаблонов :

```
some_operation (arg) USE {
    StatisticPattern < stat_obj > {
        LockPattern (WRITE) < Concur_service > { OPERATION }
    }
}
```

4.7. Использование кластеров

Как уже ранее отмечалось (см. п. 4.2), в ИО для каждой операции в статике (т.е. на этапе проектирования) может быть предусмотрено несколько различных вариантов (схем) метаконтроля. Конечно же, в ИО должен быть предусмотрен механизм выбора одного из этих вариантов. Очевидным решением этой проблемы является введение специальных полей ИО, представляющих собой, в терминах языка IDL, атрибуты перечислимого типа. Выбор текущего значения такого атрибута и определяет выбор некоторой определенной схемы метаконтроля. Более того, имеет смысл говорить о том, что смена такого атрибута меняет состояние ИО в целом, а не только схему выполнения отдельной его операции.

С помощью таких атрибутов, оказывается, очень удобно менять схему выполнения сразу нескольких, если не всех сразу, операций ИО. С этой точки зрения, можно говорить о том, что мы предусматриваем для данного ИО некоторый набор *кластеров*, каждый из которых влияет на схему метаконтроля для одной, или более операций. Каждый такой кластер получает свой собственный идентификатор. Множество идентификаторов кластеров и есть перечислимый тип - тип атрибута ИО.

Для иллюстрации механизма работы с одним множеством кластеров воспользуемся примером с сервисом статистики, описанном в п. 4.6. Кроме варианта использования схемы метаконтроля, описанного в шаблоне StatisticPattern, мы хотим также предусмотреть возможность выполнения контролируемой операции без сбора статистики. Для этого мы описываем множество кластеров StatisticClusterSet с двумя элементами (кластерами) MakeStatistic и NoStatistic. Сами же различные схемы метаконтроля для этих двух случаев описываются в шаблоне StatisticCasePattern. Все эти описания вместе с используемым шаблоном StatisticPattern помещаются в библиотеку сервиса статистики :

```
TEMPLATES LIBRARY StatisticLib {
    StatisticPattern <statistic_objref> { ControlledOperations } IS {
```

```
        statistic_objref . begin_time ;
        DO ControlledOperations ;
        statistic_objref . finish_time ;
    }

    CLUSTER SET StatisticClusterSet {
        DEFAULT MakeStatistic, NoStatistic };
    StatisticCasePattern <service_ref> { ControlledOperations } IS {
        MakeStatistic :
            StatisticPattern < service_ref> { ControlledOperations };
        NoStatistic : DO ControlledOperations;
    }
}
```

Здесь ключевое слово DEFAULT в описании множества кластеров указывает, что вариант MakeStatistic является инициализирующим. То есть, в момент создания ИО в качестве начального значения атрибута StatisticClusterSet указывается именно MakeStatistic. Поэтому, до явной смены этого кластера на другой, то есть до изменения значения этого атрибута, будет происходить сбор статистики, как это указано в первой схеме шаблона.

Теперь при описании метаконтроля, чтобы использовать описания из этой библиотеки, ее явным образом необходимо “подключить” :

```
USE TEMPLATES LIBRARY StatisticLib;
```

Кроме возможности использования шаблонов из этой библиотеки в результате такого подключения в ИО автоматически появляется атрибут StatisticClusterSet.

Если теперь, как и в примере из п. 4.6., у нас описан атрибут ИО - объектная ссылка stat_obj типа Statistic, метаконтроль для метода some_operation описывается следующим образом :

```
some_operation (arg) USE {
    StatisticCasePattern < stat_obj > { OPERATION }
}
```

При этом, с помощью атрибута StatisticClusterSet интерфейсного объекта его оболочка теперь в любой момент работы с этим ИО может поменять текущую схему метаконтроля для указанной операции.

Заметим, что механизм кластеров позволяет не только использовать разный метаконтроль при отработке одних и тех же методов, но он также позволяет делать доступными для клиента в разные моменты времени разные группы методов целевого объекта. Примером такого

использования кластеров может служить детально описанный в Приложении В пример организации доступа к объектам, инкапсулирующим работу с файлами. Клиентский ИО в этом примере содержит три кластера, активных когда 1) файл не открыт (доступен только метод открытия файла), 2) файл открыт на чтение (доступны методы чтения, позицирования и закрытия файла), 3) файл открыт на чтение и запись (клиент может вызывать методы чтения, записи, позицирования и закрытия файла).

В общем случае, любой ИО может одновременно включать произвольное количество различных библиотек с описаниями кластеров. То есть, в ИО может появляться любое количество атрибутов, значения которых меняются независимо друг от друга. Это дает возможность с помощью нескольких таких атрибутов независимо управлять работой с ортогональными (независимыми) метасервисами.

Так, например, мы можем подключить к разрабатываемому ИО кроме метасервиса сбора статистики также и метасервис, обеспечивающий одновременный доступ к целевому объекту (уже упомянутый ранее в п. 4.5. сервис Concurrency). При этом, в библиотеку шаблонов этого сервиса кроме описанного ранее в п. 4.5. шаблона LockPattern включаются также описания множества кластеров ConcurClusterSet и шаблона ConcurCasePattern, с помощью которых, как и для сервиса статистики описывается две альтернативы - нужно использовать метасервис (кластер MultiAccess) или нет (кластер NoConcurrency):

```
TEMPLATES LIBRARY ConcurrencyLib {
    … // описание шаблона LockPattern

    CLUSTER SET ConcurClusterSet {
        DEFAULT MultiAccess, NoConcurrency };

    ConcurCasePattern ( rw_flag ) <service_ref> { ControlledOperations } IS {
        MultiAccess :
            LockPattern ( rw_flag ) < service_ref> { ControlledOperations };
        NoConcurrency : DO ControlledOperations;
    }
}
```

При описании метаконтроля, чтобы работать с обоими метасервисами, нужно подключить их библиотеки:

```
USE TEMPLATES LIBRARY StatisticLib;
USE TEMPLATES LIBRARY ConcurrencyLib;
```

Определяются также два поля ИО:

```
OBJECT concur_service : Concurrency;
OBJECT stat_obj : Statistic;
```

Метаконтроль для метода some_operation, включающий работу с указанными двумя метасервисами, может быть описан теперь следующим образом :

```
some_operation (arg) USE {
    StatisticCasePattern < stat_obj > {
        ConcurCasePattern (WRITE) < concur_service > { OPERATION }
}
```

Как не трудно видеть, описание композиции метасервисов аналогично тому, которое использовало два шаблона без кластеров (см. п. 4.6.). В этом же случае появляются два атрибута ИО - ConcurClusterSet и StatisticClusterSet, значения которых могут меняться независимо. С каждым из атрибутов может работать отдельный управляющий метасервис, который даже не знает о существовании второго атрибута и соответствующего ему метасервиса.

4.8. Механизм последовательной конкретизации ИО

Обратим внимание на последовательность описания метаконтроля для ИО в предыдущем пункте. Как не трудно видеть, сначала отдельно была описана работа с метасервисом статистики, а затем был добавлен метасервис Concurrency. При этом, правда, пришлось переписать заново все, что было описано для первого метасервиса для организации работы с обоими. Однако, такой механизм пошаговой детализации ИО представляется естественным и при реальной разработке метаконтроля: в процессе разработки ИО его результирующее описание получается не сразу, а в результате постепенного добавления работы с очередными метасервисами. Возможность пошагового задания метаконтроля также включена в язык TL.

В процессе разработки ИО его описание может быть задано в виде последовательности уточнений, каждое из которых имеет свой идентификатор (имя) и является параметризованным описанием некоторой части метаконтроля. Так, в примере из п. 4.7. в качестве такого первого уточнения может быть указана работа лишь с метасервисом статистики (Interf - интерфейс, содержащий метод some_operation; работа с этим интерфейсом организуется в пределах одной оболочки, поэтому описывается только серверный ИО):

```
Interf {
    // Первое уточнение ИО
    USE TEMPLATES LIBRARY StatisticLib;
    OBJECT stat_obj : Statistic;

    SERVER FirstVersion < other_services {} >
    IS {
        some_operation (arg) USE {
            StatisticCasePattern < stat_obj > {
                other_services { OPERATION } }
        }
    }
}
```

Такое первичное описание ИО не содержит ничего, кроме подключения метасервиса Statistic. С помощью параметра *other_services* этого уточнения, названного FirstVersion, оставляется возможность в следующих уточнениях расширить отработку метода some_operation. В следующем же уточнении мы пользуемся теми описаниями, которые уже сделаны в первом (его подключение производится с помощью ключевого слова REFINES), а кроме того за счет конкретизации его параметра добавляем работу с новым метасервисом Concurrency:

```
Interf {
    // Второе уточнение ИО
    USE TEMPLATES LIBRARY ConcurrencyLib;
    OBJECT concur_service : Concurrency;

    SERVER SecondVersion
        REFINES FirstVersion <ConcurCasePattern(WRITE) <concur_service> >;
}
```

Очевидно, что такое разделенное описание ИО удобно не только для представления последовательности разработки ИО, но и позволяет упростить возможное репроектирование метаконтроля, поскольку оно локализует в разных уточнениях работу с разными метасервисами.

Кроме того, такой процесс уточнения (refinement) ИО предоставляет возможность естественным образом разделить разработку клиентского ИО в серверной и клиентской оболочках этапа проектирования. Для иллюстрации этого обратимся снова к примеру из п. 4.1., который связан с добавлением компрессии и декомпрессии данных при организации взаимодействия двух удаленных объектов.

В примере из п. 4.1. в пределах одного описания для интерфейса T на языке TL приводился как метаконтроль для серверного, так и для клиентского ИО. Напомним, однако, что подобные описания метаконтроля содержатся в серверной оболочке стадии проектирования (содержащей описания для серверного объекта типа T). Таким образом, реально лишь метаконтроль в серверном ИО, управляемом этой же оболочкой, может быть полностью задан в рамках такого описания. Что же касается клиентского ИО, то во многих случаях для него оказывается не достаточно того метаконтроля, который описан в серверной оболочке. Серверная оболочка задает в клиентском ИО работу лишь с теми метасервисами, которые необходимы для согласования взаимодействия с серверным ИО (в нашем примере - лишь использование метаобъекта для компрессии и декомпрессии данных на клиентской стороне). И в общем случае окончательное доопределение метаконтроля в клиентском ИО происходит в клиентской оболочке стадии проектирования. На стадии такого уточнения метаконтроля клиентская оболочка может как уточнять работу с метаобъектами, описанную в серверной оболочке, так и добавлять некоторые новые, известные только ей ортогональные метасервисы. При этом, такое уточнение и дополнение в клиентской оболочке может производиться поэтапно.

В рассматриваемом примере на стороне сервера описывается первое уточнение клиентского ИО с именем SeverVersion.

```
T {
    // Первая (серверная) версия описания метаконтроля для клиентского ИО
    CLIENT ServerVersion <compr_decompr_type, compress_op,decompress_op>
    IS {
        // поле ИО, тип которого определяется в клиентской оболочке
        OBJECT metaobject : compr_decompr_type;

        exchange_info (arg_info, res_info) USE {
            variable compressed_arg : string;
            variable compressed_res : string;

            Zip_metaobject . compress_op (arg_info , compressed_arg);
            // вызов метода серверного ИО:
            send_info (compressed_arg, compressed_res);
            Zip_metaobject . decompress_op (compressed_res, res_info);
        };
    };
};
```

Указанное уточнение содержит три параметра, имеющие следующий смысл:

- compr_decompr_type - тип метаобъекта для компрессии/декомпрессии;
- compress_op - метод метаобъекта для компрессии данных;
- decompress_op - метод метаобъекта для декомпрессии данных.

В качестве первого уточнения такого параметризованного описания ИО, полученного от серверной оболочки, клиентская оболочка этапа проектирования задает конкретный тип метаобъекта и названия двух его используемых методов.

```
T {
    // Вторая версия описания метаконтроля для клиентского ИО
    CLIENT FirstClientVersion
        REFINES ServerVersion <ZIP, compress, decompress >;
}
```

Обратим внимание на то, что это уточнение уже не содержит параметризации, но все еще может быть подвергнуто последующей конкретизации. Например, в следующем уточнении может быть зафиксировано желание использовать некоторый метасервис сбора статистики:

```
T {
    // Третья версия описания метаконтроля для клиентского ИО
    CLIENT SecondClientVersion { some_statistic_pattern {} }
        REFINES FirstClientVersion
    IS {
        exchange_info (arg_info, res_info) USE {
            some_statistic_pattern {
                FirstClientVersion::exchange_info (arg_info, res_info) }
        };
    };
};
```

Третье (второе на стороне клиента) уточнение имеет параметр some_statistic_pattern, который должен быть далее конкретизирован некоторым шаблоном, описывающим обращения к сервису статистики. Необходимо обратить внимание на то, что в этом уточнении при описании метаконтроля для метода exchange_info используется описание для того же метода из предыдущего уточнения (FirstClientVersion::exchange_info): шаблон сервиса статистики “оборачивает” предусмотренную ранее отработку этого метода. То есть, механизм конкретизации позволяет не только задавать параметры предыдущего уточнения, но и дает

возможность расширять метаконтроль для методов за счет добавления работы с новыми ортогональными метасервисами.

В следующем и последнем уточнении доопределяется работа с сервисом сбора статистики за счет введения ссылки на соответствующий метаобъект типа Statistic, а также использования шаблона StatisticPattern (см. п. 4.6.) в качестве конкретизации параметра предыдущего уточнения:

```
T {
    // объявляется использование библиотеки сервиса статистики
    // из клиентской оболочки
    USE TEMPLATES LIBRARY StatisticLib;

    // поле ИО - ссылка на сервис статистики
    OBJECT stat_obj : Statistic;

    // Окончательная конкретизация клиентского ИО
    CLIENT
        REFINES SecondClientVersion { StatisticPattern <stat_obj> };
};
```

Это уточнение уже более не может быть конкретизировано, поскольку оно не является именованным.

Компилятор языка TL в качестве последнего уточнения всегда воспринимает то, которое не содержит параметризации и не конкретизируется никаким другим уточнением. И именно вся совокупность уточнений вплоть до этого последнего определяют содержимое результирующего ИО.

5. Реализация оболочки

Мы работаем с оболочками и на этапе проектирования, и на этапе выполнения. На обоих этапах оболочка с точки зрения объектной модели является отдельным объектом, реализующим некоторый внешний IDL интерфейс. Для оболочек фазы проектирования под внешним интерфейсом имеется в виду набор методов, предназначенный для согласования будущего взаимодействия оболочек и входящих в них объектов. На этапе выполнения внешний интерфейс состоит из методов, назначение которых - согласовывать работу объектов различных оболочек. Кроме того, на этапе выполнения оболочки предоставляют также некоторый внутренний интерфейс, доступный только их собственным объектам.

Каждая оболочка этапа проектирования предназначена для разработки состава, механизмов работы с внутренними объектами и

объектами внешних оболочек для всех оболочек некоторого типа, которые могут возникнуть на этапе выполнения. Одной из важнейших задач, решаемых такими оболочками при проектировании системы является фиксация набора различных схем метаконтроля для потенциальных объектов соответствующих оболочек этапа выполнения. Кроме того, на этом этапе необходимо зафиксировать некоторый набор “статических” именованных объектов, то есть объектов известных типов, на которые при проектировании можно ссылаться по имени. Также должны быть разработаны механизмы динамического (на этапе выполнения) порождения объектов. При проектировании работы с такими объектами необходимо включать использование подобных механизмов порождения объектов перед их использованием.

Внешний интерфейс оболочки этапа проектирования включает методы, предназначенные для:

- предоставления информации о соответствующем интерфейсе оболочек этапа выполнения;
- описания типичных механизмов взаимодействия с объектами соответствующих оболочек этапа выполнения в виде совокупности библиотек шаблонов (см. п. 4.5.);
- описания возможностей динамического изменения метаконтроля для объектов оболочек этапа выполнения (имеется в виду смена кластеров и полей состояний ИО (см. п. 4.7.)).

Внешний интерфейс оболочки этапа выполнения предоставляет возможность работать из других оболочек с некоторой совокупностью сервисов оболочки. Часть из этих сервисов может быть предназначена для управления работой с объектами оболочки определенного типа. Такие сервисы управляют в частности и временем жизни объектов оболочки, то есть они включают в себя функции фабрики объектов (создание и удаление объектов). Кроме того, в оболочке может быть предусмотрена группа сервисов, необходимых для поддержания функционирования внутренних объектов оболочки, а также для предоставления информации о содержимом оболочки.

Пожалуй одним из наиболее важных из таких служебных сервисов является сервис именования. Это единственный сервис, который мы сочли нужным выделить как стандартный, который должен поддерживаться всеми видами оболочек. Для унифицированной работы с этим сервисом мы фиксируем набор методов, составляющих базовый интерфейс оболочек, от которого должны наследоваться интерфейсы всех оболочек

этапа выполнения. Далее приводится описание этого базового интерфейса на языке IDL:

```
module Covers {
	typedef string Name;
	typedef sequence<Name> CompoundName;
	struct Binding {
		Name	binding_name;
		Object	binded_obj;
	};

	typedef sequence<Binding> BindingList;

	interface SearchIterator {
		boolean	next_one (out Binding b);
		boolean	next_n (in unsigned long how_many, out BindingList bl);
		void	destroy();
	};

	interface BaseCover {
		void	register (in Name n, in Object obj);
		void	unregister (in Name n);
		Object	simple_resolve (in Name n);
		Object	compound_resolve (in CompoundName n);
		void	local_search (in unsigned long how_many,
				in Name search_pattern, out SearchIterator bi);
		void	full_search (in unsigned long how_many,
				in Name search_pattern, out SearchIterator bi);
	};
};
```

Указанный интерфейс предполагает идентификацию объектов как с помощью простых имен (Name), состоящих из одной строки (string), так и с помощью составных имен (CompoundName), представляющих собой последовательности строк. Простые имена используются в следующих методах:

- register - регистрация под указанным именем либо внутреннего объекта оболочки, либо внешнего (в т.ч. может регистрироваться внешняя оболочка);
- unregister - разрегистрация ранее зарегистрированного объекта;
- simple_resolve - разрешение имени в пределах данной оболочки по принятым в ней правилам (используется вложенность оболочек и федерации, как это описано далее в разделе 6).

Следующие два метода предназначены для организации поиска имени по шаблону, для навигации по результирующему списку ассоциаций “объект-имя” эти методы выдают поисковый итератор (SearchIterator):

- local_search - поиск лишь в пределах данной оболочки;
- full_search - поиск во всем пространстве ассоциаций “объект-имя”, доступном из данной оболочки (по тем же правилам, что и в методе simple_resolve).

Кроме того, предусмотрен метод compound_resolve для разрешения составного имени по стандартным правилам (каждая компонента составного имени, кроме последней, представляет собой имя оболочки, в которой нужно продолжить поиск по остатку имени).

Итак, мы предоставляем программисту IDL спецификацию интерфейса BaseCover и его реализацию на используемом им языке реализации. К этому базовому механизму разработчиком добавляются методы, специфичные для конкретного типа оболочек. Да и собственно реализация методов этого базиса может быть изменена. Например, метод simple_resolve по желанию разработчика оболочки может существенно расширить свою функциональность: при получении в качестве аргумента имени из некоторого множества специальных зарезервированных имен вместо поиска может производиться создание объекта определенного типа. То есть, механизм создания объектов может быть добавлен даже без добавления новых методов в интерфейс оболочки.

6. Расширение механизма именования

Однако, хотя механизм поиска по составным именам и позволяет производить поиск не только в той оболочке, где он был инициирован, но такой механизм все же требует от пользователя четкого знания пути в графе именования оболочек. Для того, чтобы максимально упростить работу пользователя с механизмом именования, необходимо ввести в него дополнительные средства обеспечения прозрачности поиска в пространстве оболочек.

Одним из возможных способов введения такой прозрачности является работа с иерархией оболочек. То есть, одни оболочки могут включать в себя другие также, как одни блоки могут включать в себя другие в модульных языках программирования. При этом, при поиске простого имени в некоторой оболочке, в случае, если непосредственно в этой оболочке имя не найдено, поиск продолжается в оболочках, которые содержат данную. То есть работает тот же стандартный принцип видимости имен, который используется для переменных в блоках.

Более сложным, но и более гибким способом расширения механизма именования является использование техники *федераций*. Федерация представляет собой группу оболочек, предоставляющих друг другу возможность поиска в их пространстве ассоциаций “объект-имя”. Простейшей формой такого объединения является федерация, в которой каждая оболочка предоставляет возможность поиска без ограничений во всем своем пространстве таких ассоциаций для всех членов федерации. В более общем случае оболочки, входящие в федерацию, могут ограничивать и доступ других членов федерации в свое пространство именованных объектов, и свой доступ к таким пространствам из других оболочек федерации (т.е. происходит сужение доступного пространства в целях ускорения поиска). Такие ограничения описываются с помощью списков экспорта (указание, что и каким оболочкам федерации доступно из данной оболочки) и импорта (указание, что и из каких оболочек доступно данной оболочке) ассоциаций “объект-имя”.

Напомним, что оба приведенных механизма расширения именования используются в методе full_search базового интерфейса оболочек. Поиск в федерациях и объемлющих оболочках имеет более низкий приоритет, чем поиск в оболочке, где он инициирован. То есть, при любом разрешении имени сначала должен отработать локальный сервис именования оболочки, и только если нужный объект не был найден в самой оболочке, осуществляется дополнительный поиск в соответствии с указанными механизмами.

Итак, с помощью такого расширенного именования оболочки могут предоставлять находящимся в них объектам возможность разрешать только простые имена относительно этой оболочки, не задумываясь о действительной распределенности пространства ассоциаций, в котором производится поиск. Для этого оболочке необходимо находиться в федерациях со всеми потенциально “интересными” для ее объектов оболочками.

Нужно заметить, однако, что в некоторых системах программирования клиент не всегда является объектом. В этом случае для того, чтобы унифицировать технику работы с оболочками и, в частности, обеспечить возможность единообразной работы с именованием, может понадобиться некоторое расширение объектной модели.

Например, в системе программирования ОС UNIX / язык C++ программы выполняются в рамках процессов, начинающих работу с вызова функции *main()*. Кроме такой инициализирующей процесс функции указанная система программирования допускает использование и других функций, не являющихся методами каких-либо объектов (наследие

языка С). Однако, необходимость работы с объектными ссылками и с именованием может возникать и в подобных функциях, и в методах объектов языка С++. Для того, чтобы предлагаемая технология работы с оболочками была унифицирована, будем считать особого вида объектами отдельные процессы ОС UNIX. Это чисто клиентские объекты, не имеющие интерфейса. Любая деятельность этапа выполнения происходит либо в подобных объектах, либо в объектах, которые вызываются в результате обращений по объектным ссылкам. *Все* такие объекты должны быть приписаны к некоторым оболочкам. Таким образом, любая работа с объектными ссылками в такой расширенной объектной модели происходит *в рамках* некоторого объекта, находящегося в некоторой оболочке.

Когда в используемой системе программирования обеспечен гарантированный контекст (объект, оболочка) для любой клиентской деятельности, можно говорить о единой схеме работы с внутренним интерфейсом оболочек. Методы такого внутреннего унифицированного для всех оболочек интерфейса вызываются единообразно в любом объекте. Выполнение любого метода этого интерфейса всецело зависит от контекста, в котором он вызван, то есть определяется механизмами текущей оболочки. Так, например метод *Object* ***resolve*** *(in Name n)* этого внутреннего интерфейса используется для разрешения указанного простого имени с помощью расширенного механизма именования *той* оболочки, в которой находится объект, в рамках которого производится обращение по этому методу.

Важной особенностью внутреннего интерфейса оболочек является то, что в допускающих это языках реализации, методы этого интерфейса могут быть реализованы как функции, не относящиеся к конкретному объекту. Например, в С++ указанный метод реализуется функцией *Object* ****resolve*** *(const Name &n).*

Использовать такие вызовы методов не по объектной ссылке весьма удобно и при описании метаконтроля. Например:

```
obj_ptr = RESOLVE (name);
```

Такой вызов не требует наличия заполненной заранее объектной ссылки, а выполняется он в соответствии с алгоритмом, предусмотренным оболочкой, в которой находиться ИО на этапе выполнения. Этот механизм, в частности, может решать проблему инициализации ИО. То есть подобные обращения к текущей оболочке могут использоваться для заполнения полей состояния ИО (см. 4.4.) при его инициализации. Для этого на этапе проектирования необходимо зафиксировать для ИО лишь некоторые простые имена метаобъектов в подобных вызовах. А на этапе

выполнения каждая оболочка, в которой создается ИО такого типа, должна обеспечить разрешение этих имен, используя, возможно механизмы федераций и иерархии оболочек.

7. Согласование оболочек

Как уже отмечалось ранее, интерфейс каждой оболочки этапа выполнения кроме базового интерфейса включает также интерфейсы различных входящих в нее сервисов. Эти сервисы могут быть в том числе предназначены для манипулирования некоторыми отдельными группами взаимосвязанных объектов, например объектами некоторого одного типа. Подобное включение интерфейсов производится путем наследования спецификаций в языке IDL. Нужно заметить, что и сами интерфейсы сервисов могут образовывать графы наследования.

Каждый элемент объектного пространства находится в некоторой оболочке, которая кроме порождения и удаления собственно объектов также обеспечивает функционирование необходимой ее объектам метасреды и настраивает находящиеся в ней интерфейсные объекты и в момент их создания и в процессе их работы. Методы различных сервисов в частности могут использоваться как для управления извне созданием и удалением объектов оболочки, так и для разных видов управления метасредой внутри оболочки.

Для обеспечения возможности управлять функционированием объекта в пределах некоторой оболочки для каждого объекта (в том числе и интерфейсного) предусмотрен специальный метод интерфейсного объекта BaseCover get_cover (), выдающий ссылку на оболочку, в которой находится целевой объект этого ИО. Необходимо отметить, что при дальнейшей работе с объектной ссылкой на оболочку, полученной с помощью такой операции, нет необходимости точно знать тип оболочки, т.е. весь ее интерфейс, можно работать лишь через интерфейс нужного сервиса. Это обеспечивается механизмом наследования интерфейсов.

Одним из важнейших применений описываемой технологии работы с оболочками является согласование метасреды различных объектов для обеспечения их взаимодействия. Это может сводиться к согласованию оболочек, содержащих эти объекты, проводимом с помощью соответствующих сервисов этих оболочек.

Напомним, что необходимость в согласовании деятельности оболочек уже возникала в примере из п. 4.2. В этом примере на этапе проектирования как для клиентского, так и для серверного ИО описывались две различные схемы метаконтроля: одна из них включала в себя компрессию и декомпрессию данных, другая работала с данными, не

изменяя их. При этом, технология метаконтроля предполагает, что при использовании первого варианта для ИО соответствующая оболочка подключает к этому ИО метаобъект, производящий необходимые преобразования данных. Очевидно, что клиентский и серверный ИО могут гарантированно “понимать” друг друга лишь в том случае, когда содержащие их оболочки “договариваются” о том, какой вариант метаконтроля необходимо использовать. То есть, эти оболочки должны включать сервис согласования метаконтроля для рассмотренного типа объектов. Процедура согласования метаконтроля оболочек может происходить не только в момент появления нового клиента (в этом случае клиентская оболочка обращается к серверной для инициализации однотипного метаконтроля). Может быть предусмотрен и такой механизм согласования, который может быть использован обеими оболочками в динамике уже после установления связи между клиентом и сервером. Подобная необходимость может возникать в любой из оболочек, например, в силу изменения условий существования находящегося в этой оболочке ИО. Так, наиболее очевидной ситуацией, когда необходимо динамическое согласование оболочек, является миграция (перемещение) одного из двух объектов.

Более сложный пример согласования оболочек детально описан в Приложении А. В этом примере идет речь о том, как при работе в рамках технологии CORBA (см. далее раздел 8) можно воспользоваться техникой оболочек для добавления новых, более эффективных механизмов организации взаимодействия между объектами. В частности, описывается способ организации работы объектов из разных процессов одной машины (рассматривается ОС UNIX) через разделяемую память с помощью клиентского и серверного метаобъектов. Конечно же, и здесь для нормальной совместной работы клиентского и серверного ИО необходима согласованная настройка метаконтроля этих ИО, обеспечиваемая специальными сервисами содержащих их оболочек.

8. Согласование с технологией CORBA

Обсудим теперь, как организовывать вышеописанный механизм работы с оболочками и использовать метаконтроль, опираясь на широко известную архитектуру организации работы объектов в разнородных распределенных системах CORBA. Заметим, что используемый нами язык спецификаций интерфейсов IDL изначально был разработан именно в рамках этой архитектуры. В современном стандарте CORBA определен как сам язык IDL, так и правила его отображения (меппинг) в некоторые языки реализации, в частности C и C++. То есть, при применении

механизма оболочек в рамках CORBA стандарт определяет схему работы не только со спецификациями интерфейсов на IDL, но и многие элементы организации работы интерфейсов на основе описаний на языке TL. В частности, в меппинге зафиксированы и отображение понятия объектных ссылок из IDL в нужный язык реализации, и различные механизмы работы с ними, такие как:

- механизмы работы с наследованием;
- организация работы с памятью, в том числе, при передаче параметров в методы;
- использование простейших объектов-посредников типа A_var для работы с объектными ссылками, соответствующим произвольному интерфейсу типа A.

Очевидно, что даже только перечисленные требования стандарта существенно влияют на реализацию интерфейсных объектов, функциональность которых в основном и состоит в работе через объектные ссылки с различными метаобъектами.

Кроме того, архитектура CORBA предусматривает использование посреднических объектов при организации работы через объектные ссылки. На стороне клиентского объекта таким посредником является стаб, на стороне сервера - скелетон. Единственное назначение этих посредников - обеспечение прозрачности по расположению объектов (location transparency). То есть, в зависимости от взаимного расположения клиента и вызываемого объекта стаб и скелетон обеспечивают использование одного из предусмотренных в ORB (Object Request Broker - брокер объектных заявок) вариантов организации вызова метода. Как не трудно видеть, при организации метаконтроля на базе CORBA технологии коммуникационные ядра оболочек (см. раздел 2) могут строиться на основе таких брокеров.

Отметим, что предусмотренный в CORBA механизм работы с объектными ссылками представляет собой технику позднего связывания объектов. Таким образом, приложениям, написанным в рамках архитектуры CORBA, ориентирующимся на использование метаконтроля, достаточно организовывать взаимодействие между объектами только с помощью объектных ссылок (то есть, например, не стоит “оптимизировать” взаимодействие локальных объектов с помощью организации их непосредственных вызовов в обход ORB’а). Более того, поскольку ORB и подготавливаемые компилятором IDL посредники (стабы и скелетоны) не является частью самого приложения (они фактически представляют собой часть системы программирования), мы можем реализовывать метаконтроль без внесения изменений в

приложение за счет расширения системы программирования даже для приложений, изначально не ориентировавшихся на работу с метаконтролем (своего рода legacy). Достаточно того, чтобы различные объекты приложения взаимодействовали между собой и с объектами других приложений через объектный ссылки. На этом примере с CORBA видно, что в общем случае, для произвольной системы программирования, написанные в ней приложения могут быть прозрачным образом (для самих приложений) “обернуты” техникой оболочек в том случае, если эта система программирования обеспечивает позднее связывание объектов приложения за счет использования предоставляемых *ею* посредников.

Что же касается собственно CORBA, здесь остается вопрос о том, как же согласуются ИО, которые в нашем подходе также являются посредниками между клиентскими и серверными объектами, с понятиями стабов и скелетонов.

Рассмотрим для начала серверную сторону. Как сами объекты приложений, так и серверные ИО создаются и управляются оболочкой, в которой они находятся. Таким образом, если следовать CORBA, оболочка также должна создавать скелетон для серверного объекта и произвести их связывание. В результате любое обращение к этому объекту будет проходить через посреднический объект - скелетон. Не трудно видеть, что у серверного ИО и скелетона совпадают и схема порождения и их место при организации работы с подконтрольным (целевым для ИО) объектом. Таким образом, на серверной стороне эти два посредника могут быть совмещены, то есть, можно сказать, что мы добавляем механику ИО к базовому механизму скелетона.

Что же касается клиентской стороны, то здесь отношение между стабом и клиентским ИО определить несколько сложнее. Дело в том, что стаб, в отличие от скелетона не может быть порожден оболочкой. Стаб создается внутренними механизмами ORB’а при получении клиентом объектной ссылки от других объектов. Кроме того, объект - стаб зачастую не используется получившим его объектом (т.е. он не вызывает методы стаба), а играет лишь роль объектной ссылки, которую нужно передать в другой объект. В таком случае, необходимость работы с клиентским ИО вообще не возникает.

Итак, если бы мы попытались на стороне клиента также, как и на стороне сервера, совместить стаб и ИО в одном посреднике, то мы потеряли бы возможность управлять временем жизни ИО с помощью оболочки, не смогли бы выбирать нужный тип ИО для данного клиента. Также мы бы имели громоздкий стаб и в том случае, когда через него не

производятся обращения и все механизмы и поля ИО, в него вошедшие являются явно лишним “грузом”.

Итак, клиентские объекты-посредники (стаб и клиентский ИО) не могут быть совмещены в одном объекте. Поскольку необходимость использования ИО проявляется лишь при первом обращении клиента через стаб, необходимо также расширить механизм работы стаба следующим образом: при выполнении клиентом первого обращения к методу стаба, то есть, когда клиент демонстрирует в первый раз, что собирается использовать стаб для обращений к серверу, в стабе происходит обращение к оболочке клиента с запросом на получение ИО для этого стаба. По типу стаба для конкретного клиента оболочка выбирает нужный тип клиентского ИО, возможно, согласовывая его с оболочкой соответствующего стабу целевого объекта (т.е. в оболочках может быть и сервис согласования типов ИО). Конечно, в оболочке могут быть предусмотрены и различные алгоритмы ассоциации ИО со стабами. Так, все однотипные стабы могут работать через один и тот же экземпляр ИО, а может и использоваться схема “1 стаб - 1 клиентский ИО”.

Итак, стаб получает ссылку на “приписанный” ему оболочкой клиентский ИО, после чего все вызовы методов стаба, включая самый первый, передаются в этот ИО. Конечно же, оболочка может выяснить, что указанный клиент должен работать с указанным сервером без метаконтроля на клиентской стороне, в этом случае стаб получает незаполненную (nil) ссылку на ИО и будет просто вызывать указанный клиентом метод целевого объекта (который еще может быть перехвачен серверным ИО!).

Заметим, что обеспечение прозрачности расположения объектов, производимое в соответствии со стандартом CORBA в стабах и скелетона, нами декларировалось как одна из базовых функциональностей ИО. Конечно же, при указанном объединении посредников технологии CORBA и интерфейсных объектов, последние могут использовать для обеспечения такой прозрачности как уже зафиксированные в стабах и скелетонах механизмы, опирающиеся на ORB, так и добавлять новые способы организации передачи данных за счет использования техники метаконтроля. Например, это может производиться за счет согласованного использования метаобъектов, обеспечивающих работу через разделяемую память, как это описано в Приложении А.

Таким образом, описанная технология расширения функциональности стабов и скелетонов позволяет организовать с помощью оболочек симметричное управление работой клиентских и серверных ИО. Оболочки сами заведуют порождением обоих типов ИО, и

имеют возможность выбирать нужный тип (версию) как клиентского, так и серверного ИО. То есть, описанный подход расширяет предусмотренные в CORBA базовые механизмы выполнения вызовов объектов, которые как на клиентской, так и на серверной стороне не содержат ничего кроме организации вообще говоря удаленных вызовов объектов. Механизм оболочек и интерфейсных объектов обеспечивает возможность не только модифицировать на стороне сервера поведение приложения, но он дает возможность симметрично (равноправно!) разнести *дополнительные состояния и функциональности приложения* как на серверную, так и на клиентскую части. То есть появляется возможность семантически разделить ответственность клиентской и серверной сторон за различные вопросы организации работы с приложением, что упрощает проектирование системы и делает ее работу более эффективной.

Нужно отметить, что привнесение техники оболочек в модель CORBA производится не за счет изменения реализации ORB'а, а с помощью согласования работы компилятора из IDL в нужный язык реализации с компилятором метаспецификаций (см. раздел 1).

При реализации техники оболочек на основе CORBA возникает также вопрос о соотношении между метасервисами оболочек и стандартными сервисами CORBA [COSS96].

Как отмечалось ранее, единственным метасервисом, который зафиксирован нами в качестве базового, т.е. реализуемого во всех оболочках, является сервис именования. Как известно, в стандарте CORBA зафиксирован интерфейс CosNaming::NamingContext объектов сервиса именования и описана в основных чертах его семантика. Для сохранения совместимости реализуемого в оболочках именования с CORBA базовый интерфейс оболочек BaseCover наследуется от этого стандартного интерфейса. Таким образом, базовый интерфейс оболочек предоставляет возможность воспринимать оболочки как контексты именования в терминах CORBA. Кроме того, оболочки поддерживают и предложенный ранее, отличный от CORBA механизм именования.

Что же касается остальных стандартных сервисов CORBA, технология метаконтроля позволяет использовать их, как и любые другие сервисы, как в качестве метасервисов оболочек, так и в качестве составляющих пользовательских приложений. Такое использование нами никак не фиксируется и остается на усмотрение программиста, проектирующего объектную систему в рамках оболочечной модели.

Литература

[ANSA93] Rob van der Linden. The ANSA Naming Model. Architecture report APM.1003.01, Poseidon House, Cambridge, 15 July 1993

[Cahill96] Vinny Cahill. An Overview of the Tigger - Object-Support Operating System Framework. SOFSEM'96: Theory and Practice of Informatics, Lecture Notes in Computer Science, volume 1175, Springer-Verlag, Berlin/Heidelberg, November, 1996, p. 34-35. (ftp://ftp.dsg.cs.tcd.ie/pub/doc/dsg-102.ps.gz)

[Chi96] Shigeru Chiba. OpenC++ Programmer's Guide for Version 2. Technical Report SPL-96-024, Xerox PARC, 1996. (http://www-masuda.is.s.u-tokyo.ac.jp/openc++.html)

[CORBA95]The Common Object Request Broker: Architecture and Specification. Revision 2.0, July 1995.

[COSS96] CORBA services: Common Object Services Specification, OMG Document 95-3-31, 1995. Updated: November 22, 1996.

[DF94] Scott Danforth, Ira R. Forman. Reflections on Metaclass Programming in SOM. OOPSLA 94- 10/94 Portland, Oregon USA

[FDM94] Ira R. Forman, Scott Danforth, Hari Madduri. Composition of Before/After Metaclasses in SOM. OOPSLA 94- 10/94 Portland, Oregon USA

[GC96] B.Gowing, V.Cahill. Metaobject protocols for C++: the Iguana approach. Proceedings of Reflection 96 Conference. 1996.

[IDZ97] В.П. Иванников, К.В. Дышлевой, В.И. Задорожный. Спецификация метанаращиваний для эффективного метаобъектного контроля. Программирование, N 4, 1997 г.

[IONA96] The Orbix Architecture. IONA Technologies. November 1996. Http://www-usa.iona.com/Orbix/arch

[IZKN96] Ivannikov V., Zadorozhny V., Kossmann R., Novikov B. Efficient Metaobject Control Using Mediators. 2nd Int.A.Ershov’s Conf. , Novosibirsk, 1996

[KASP] Gregor Kiczales et.al. Aspect-Oriented Programming. A Position Paper from Xerox Parc.

[Klein96] Jan Kleindienst, Frantisek Plasil, Petr Tuma. Lessons Learned from Implementing the CORBA Persistent Object Service. OOPSLA’96 conference proceedings, San Jose, California, October 6-10, 1996.

[KLMKM93] Gregor Kiczales, John Lamping, Chris Maeda, David Keppel, and Dylan McNamee. The Need for Customizable Operating Systems. In Proceedings of the Fourth Workshop on Workstation Operating Systems, pages 165--169. IEEE Computer Society Technical Committee on Operating Systems and Applications Environment, IEEE Compuer Society Press, October 1993.

[KRB91] Kiczales G., des Rivieres J., Bobrow D. The Art of the Metaobject Protocol. MIT Press, 1991.

[MI97] Multiple Interfaces and Composition. OMG Document orbos/97-02-06. February 1997

[MJD96] Malenfant J., Jacques M., Demers F.-N. A Tutorial on Behavioral Reflection and its Implementation. Proceedings of the Reflection'96 Conference, San Francisco, USA, April 21-23,1996

[MMAY95] Masuhara H., Matsuoka S., Asai K., Yonezava A. Compiling Away the Meta-Level in Object-Oriented Concurrent Reflective Languages Using Partial Evaluation. OOPSLA'95. 1995. p. 300-315.

[NH96] Farshad Nayeri, Ben Hurwitz. Generalizing Dispatching in a Distributed Object Systems. ECOOP'96.

[SoSt95] Soley R., Stone C. Object Management Architecture Guide. Third edition. John Wiley and Sons, Inc. 1995.

[Str94] D.Stroustrup. The C++ Programming Language 2nd Edition. Addison-Wesley, 1995.

[ZC95] Chris Zimmermann and Vinny Cahill. How to Structure Your Regional Meta: A New Approach to Organizing the Metalevel. In Proceedings of META '95, a workshop held at the European Conference of Object-Oriented Programming, 1995.

[ZC96] Chris Zimmermann and Vinny Cahill. It's Your Choice - On the Design and Implementation of a Flexible Metalevel Architecture. Proceedings of the International Conference on Configurable Distributed Systems, IEEE, Annapolis, Maryland, May, 1996. (ftp://ftp.dsg.cs.tcd.ie/pub/doc/dsg-100.ps.gz)

Приложение А

Это приложение детально описывает реализацию примера согласования оболочек, приведенного в основной части в разделе 7. Рассматриваются две оболочки: ClCover типа ClientCover и SvCover типа ServerCover (см. Рис. 4). Оболочки рассчитаны на возможность установления с помощью метасервисов более эффективной связи с серверными объектами некоторого типа ServObj, чем с помощью стандартного механизма ORB. Все серверные объекты указанного типа могут работать в оболочке SvCover, которую, таким образом, имеет смысл считать серверной. Рассматриваемый клиент действует в рамках оболочки ClCover (клиентская оболочка). В частности, если клиент и сервер находятся в разных процессах, но на одной машине, для них создаются специальные метаобъекты, выполняющие роль посредников, организующих передачу данных не с помощью механизма сообщений, как это делает ORB, а через сегмент разделяемой памяти, общий для клиентского и серверного процессов.

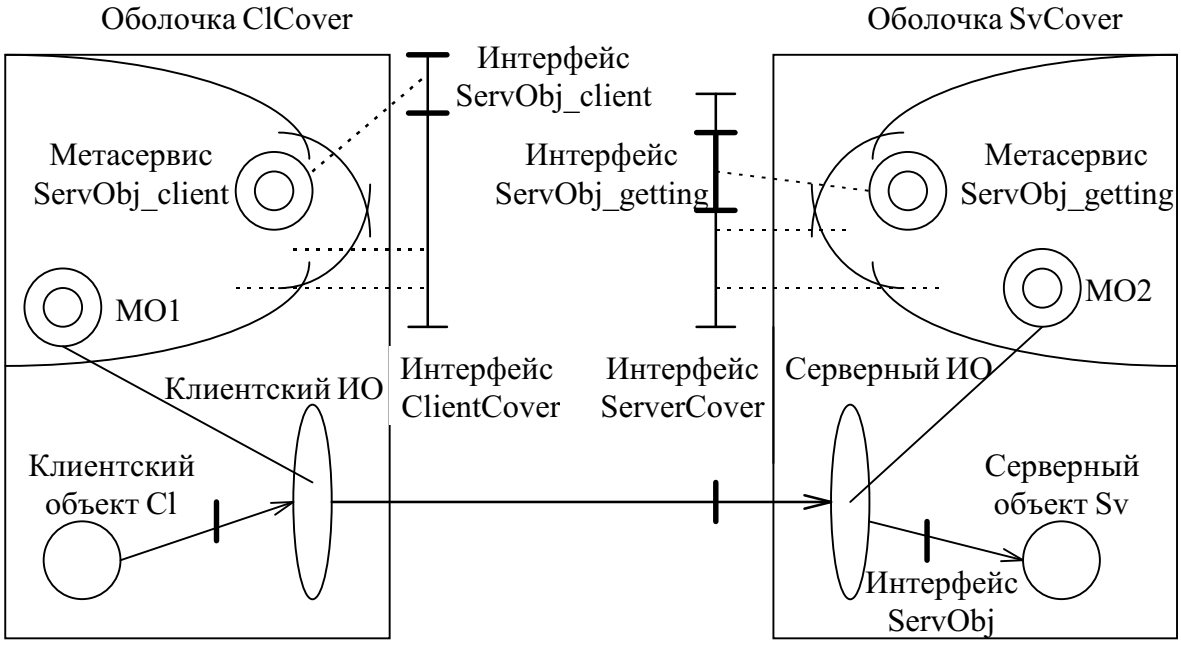

Рисунок 4. Согласование оболочек для организации работы с разделяемой памятью.

Интересующая нас часть IDL спецификаций для интерфейсов оболочек выглядит следующим образом:

```
// для серверной оболочки
interface ServObj_getting {
    ...
    ServObj use_optim_connection (in ServObj_IO client);
```

```
};
interface ServerCover : ..., ServObj_getting, ... {
        ...
};
// для клиентской оболочки
interface ServObj_client {
        ...
        void use_shared_mem (in ServObj_IO client, in long sh_mem_segm);
};
interface ClientCover : ..., ServObj_client, ... {
        ...
};
```

То есть, в серверной оболочке имеется сервис ServObj_getting, отвечающий за различные типы порождения объектов типа ServObj. Один из методов этого сервиса - use_optim_connection - позволяет серверной оболочке выбрать наиболее эффективный на ее взгляд способ взаимодействия с указанным в ее параметре клиентским объектом (интерфейсным объектом). В клиентской же оболочке есть сервис ServObj_client, позволяющий контролировать интерфейсные объекты типа ServObj_IO. Метод use_shared_mem этого сервиса настраивает метаконтекст указанного в качестве первого параметра интерфейсного объекта на использование для работы со своим целевым объектом сегмента разделяемой памяти, идентификатор которого указан вторым параметром.

Итак, получаем следующий алгоритм установления связи между клиентом и сервером, использующий согласование оболочек:

- клиентская оболочка ClCover создает интерфейсный объект типа ServObj_IO;
- в рамках инициализации клиентского ИО с помощью сервиса именования клиентской оболочки находится объектная ссылка на серверную оболочку SvCover;
- через сервис ServObj_getting инициализируемый интерфейсный объект вызывает метод оболочки use_optim_connection, передав ссылку на себя в качестве параметра;
- при отработке метода use_optim_connection серверная оболочка создает объект типа ServObj и серверный ИО для работы с этим объектом; далее она определяет по объектной ссылке на клиентский ИО, что клиент находится на этой же машине, и поэтому далее создает серверный метаобъект MO2 для работы через разделяемую память;
- серверная оболочка создает сегмент разделяемой памяти и передает его идентификатор вместе с ссылкой на клиентский ИО в оболочку этого

ИО (для получения ссылки на оболочку используется описанная выше операция get_cover), а точнее, вызывает метод use_shared_mem сервиса ServObj_client;

- клиентская оболочка по запросу серверной создает клиентский метаобъект-посредник МО1 и переключает ИО на работу через этот метаобъект;
- происходит возврат управления в серверную оболочку, а затем - в клиентский ИО, при этом возвращается ссылка на созданный серверный ИО.

Итак, клиентский ИО при инициализации обращается к серверной оболочке и получает от нее ссылку на созданный ею объект. При этом, с помощью согласования между оболочками, без ведома ИО, настраивается его окружение в соответствии с выбранным серверной стороной способом дальнейшей работы ИО с создаваемым серверным объектом.

Приложение В

В этом приложении приводится пример использования внутреннего интерфейса при общении клиентского и серверного ИО (интерфейс серверного ИО отличается от интерфейса целевого объекта). Кроме того, пример демонстрирует использование механизма кластеров для выделения нескольких групп методов интерфейса, с которым работает клиент. Возможность использования того или иного метода этого интерфейса определяется текущим состоянием клиентского ИО.

В этом примере мы допускаем, что имеется некоторое приложение для работы с файлами. Это приложение функционирует в рамках оболочки, которую мы можем ассоциировать, например, с понятием отдельного каталога файловой системы. Нас интересуют объекты приложения, входящие в такую оболочку, представляющие отдельные файлы соответствующего каталога. Именование всех таких однотипных объектов осуществляется с помощью базового сервиса именования оболочки (см. раздел 5.). Именно с помощью этого метасервиса клиент получает доступ к интересующему его файлу. Другие вопросы, связанные с работой с таким каталогом-оболочкой (создание, удаление файлов, логическое связывание файлов и т.п.), мы в этом примере не рассматриваем.

Все объекты-файлы являются реализациями следующего интерфейса:

```
enum Mode {READ, READ_WRITE};
typedef sequence<octet> Data;
typedef long Status;
```

```
interface File {
        Status      open (in Mode use_mode);
        Status      close ();
        Status      read (in long byte_cnt, out Data result);
        Status      write (in Data info);
        Status      seek (in long byte_cnt);
};
```

Объекты приложения, реализующие интерфейс File, рассчитаны на работу с единственным клиентом, которому предоставляется возможность:

- открыть (*open*) сессию работы с файлом, указав при этом режим его использования - только чтение данных (READ), либо чтение и запись (READ_WRITE) (тип использования может определять, например, текущий способ буферизации данных);
- закрыть сессию (*close*);
- прочитать (*read*) нужное количество байтов, начиная с текущей позиции в файле; после выполнения этой операции текущая позиция в файле находится непосредственно после прочитанного куска данных;
- записать (*write*) данные в файл (операция выполняется корректно лишь в том случае, если текущая сессия была открыта в режиме READ_WRITE), начиная с текущей позиции в файле; после выполнения этой операции текущая позиция в файле находится непосредственно после записанного куска данных;
- поместить (*seek*) текущий указатель в файле с указанным смещением относительно начала файла.

При этом, все методы в качестве результата возвращают статус окончания (значение типа Status).

При применении к такому приложению оболочечной модели мы хотим не только управлять временем жизни таких объектов, но и организовывать одновременный доступ (мультидоступ) различных клиентов (возможно, удаленных) к таким объектам. Как уже было отмечено, само приложение на мультидоступ не рассчитано, поэтому для решения задачи необходимо использовать технику интерфейсных объектов.

Воспользуемся уже предусмотренным в самом приложении разделением типов работы с объектом на работу в режиме чтения (READ) и режиме, допускающем запись (READ_WRITE). Для синхронизации клиентов на стороне сервера предусмотрим использование метаобъектов следующего интерфейса:

```
interface Concurrency {
        Status      connect (in Mode use_mode);
        Status      disconnect ();
        Status      lock () ;
        Status      unlock ();
};
```

Объекты, реализующие интерфейс Concurrency, обеспечивают возможность множеству клиентов одновременно работать с целевым объектом в режиме чтения, либо только одному клиенту - в режиме записи. Для открытия сессии работы в нужном режиме используется метод connect, для закрытия сессии - disconnect. Методы lock и unlock используются для единовременного захвата объекта на время выполнения метода в рамках сессии. При этом, мы предполагаем, что вызовы различных клиентов отрабатываются в различных нитях управления (т.е. в режиме multithread), т.е. в пределах одного UNIX процесса такие нити управления могут быть отложены в случае, когда некоторый клиент уже работает в режиме READ_WRITE

Для работы с методами lock и unlock целесообразно наличие в библиотеке шаблонов для сервиса Concurrency следующего шаблона (по аналогии с примером из основной части п. 4.5.):

```
TEMPLATES LIBRARY ConcurrLib{
        LockPattern < concurrency_objref > { ControlledOperations } IS {
                A {
                        concurrency_objref . lock;
                        DO ControlledOperations WITH EXCEPTION HANDLER B;
                }
                B {
                        concurrency_objref . unlock ;
                }
        }
}
```

Однако, вспомним, что исходная семантика объектов приложения такова, что все они рассчитаны на работу с единственным клиентом и хранят текущую позицию в файле для этого клиента. Очевидно, что при организации мультидоступа такой способ работы возможен только в режиме READ_WRITE. Когда же одновременно несколько клиентов читают данные из файла, все они должны работать так, если бы имели монопольный доступ к файлу. Обеспечение такой абстракции (т.е. прозрачности метаконтроля) как раз и является одной из основных задач

оболочечного подхода. То есть, с помощью механизма ИО необходимо организовать наличие характерного состояния (позиции в файле) для каждого отдельного клиента, которое бы учитывалось при выполнении каждой операции чтения.

Итак, мы приходим к тому, что в клиентском ИО нужно завести глобальную (доступную из всех методов, сохраняющую значение между обращениями к ним) переменную current_position типа long для хранения текущей позиции в файле.

Клиентский ИО может находиться в трех различных состояниях, определяющих и его поведение, и возможности клиента по его использованию (этим состояниям соответствуют кластера с указанными именами):

- NOT_CONNECTED - в этом состоянии ИО находится, когда клиент, работающий через него, еще не находится в рамках сессии работы с целевым объектом (сессия еще не открыта, либо уже закрыта);
- READING - состояние, в котором файл доступен только на чтение;
- WRITING - в этом состоянии клиент работает с объектом-файлом в монопольном режиме и может его модифицировать.

Очевидно, что в каждом из этих состояний клиенту доступна лишь часть методов:

NOT_CONNECTED - open();

READING - read(), seek(), close();

WRITING - read(), write(), seek(), close();

Как уже отмечалось при описании технологии работы с кластерами, попытка вызова метода, не предусмотренного в клиентском ИО в текущем кластере приводит к выдаче ошибки без работы с метаобъектами и целевым объектом.

Конечно же, в кластерах READING и WRITING для их общих методов read() и seek() должен осуществляться различный метаконтроль. В кластере READING, в отличие от WRITING, необходимо работать с переменной current_position. При этом выполнение метода seek() в рамках кластера READING приводит просто к сохранению его параметра в этой переменной, обращаться к целевому объекту нет необходимости. А выполнение read() в этом же кластере должно сводиться к установке текущей позиции в файле на основе значения переменной current_position и собственно чтения данных, начиная с этой позиции. Однако, реализовывать это с помощью последовательного обращения к методам seek() и read() нельзя, поскольку между этими вызовами любой другой клиент может по-своему изменить состояние целевого объекта. То есть, эти два действия должны производиться как единая операция на стороне

сервера. Таким образом, необходим дополнительный метод серверного ИО, предназначенный для чтения данных в режиме мультидоступа. Этот метод должен в качестве дополнительного параметра получать текущую позицию в файле от конкретного клиентского ИО:

```
Status      read_from_position (in long position, in long byte_cnt, out Data result);
```

Указанный метаконтроль для клиентского и серверного ИО можно на языке TL описать следующим образом:

```
File {
    SERVER IS {
        // спецификация изменений интерфейса серверного ИО
        ADD METHOD Status read_from_position (
                        in long position, in long byte_cnt, out Data result);
        // спецификация использования библиотеки шаблонов:
        USE TEMPLATES LIBRARY ConcurrLib;
        // объявление глобальных переменных (элементов состояния ИО)
        OBJECT concur_serv : Concurrency;

        // описание метаконтроля для методов в серверном ИО
        open (use_mode) USE {
            concur_serv . connect (use_mode);
        };
        close () USE {
            concur_serv . disconnect ();
        };
        read (byte_cnt, result), write (info), seek (byte_cnt) USE {
            LockPattern <concur_serv> { OPERATION; }
        };
        read_from_position (position, byte_cnt, result) USE {
            LockPattern <concur_serv> {
                seek (position);
                RESULT = read (byte_cnt, result);
            };
        };
    };

    CLIENT IS {
        CLUSTER SET CurrentState {
            DEFAULT NOT_CONNECTED, READING, WRITING };

        // объявление глобальных переменных
        variable current_position : long;

        // описание метаконтроля для методов в серверном ИО
        open (use_mode) USE {
            NOT_CONNECTED : OPERATION;
        };
```

```
        write (info) USE {
            WRITING : OPERATION;
        };
        close () USE {
            READING : OPERATION;
            WRITING : OPERATION;
        };
        read (byte_cnt, result) USE {
            READING : RESULT = read_from_position (
                                    current_position, byte_cnt, result);
            WRITING : OPERATION;
        };
        seek (byte_cnt) USE {
            READING : {
                current_position = byte_cnt;
                RESULT = 0; // признак корректной отработки метода
            };
            WRITING : OPERATION;
        };
    };
};
```

Нужно заметить, что в этом примере метаобъект типа Concurrency, используемый некоторым ИО, также имеет доступ к методам целевого объекта этого ИО. Дело в том, что методы *open* и *close* целевого объекта должен вызывать именно этот метаобъект (как видно по описанию метаконтроля, сам серверный ИО эти два метода никогда не вызывает), поскольку только он знает, когда необходимо переводить целевой объект из одного режима в другой (такое переключение режима происходит, когда заканчиваются сессии всех только читающих клиентов и открывается сессия пишущего клиента, и наоборот).

Заметим также, что в более сложном случае организации мультидоступа данные, необходимые для восстановления состояния сервера для работы с клиентом (в данном примере - смещение в файле), могут иметь сложную структуру. В этом случае требуется более сложная обработка данных, и поэтому имеет смысл для работы с такими данными пользоваться не состоянием ИО, а отдельным метаобъектом.

Кроме того, может быть не достаточно методов целевого объекта для указания характеристик каждого отдельного клиента. Например, если бы объект-файл не предоставлял метод seek() для смены текущей позиции, (просто производил бы последовательную работу с данными файла, сохраняя в своем состоянии текущую позицию в файле), то для установки характерной для клиента позиции в файле потребовалась бы

непосредственная манипуляция с состоянием целевого объекта. Для этого понадобился бы специальный метаобъект, работающий с содержимым объекта наподобие метасервиса, обеспечивающего долговременное хранение объектов (persistency) из Приложения С.

Приложение С

Описываемый здесь пример организации работы с метасервисом долговременного хранения данных (persistency [COSS96, Klein96]) демонстрирует вариант метаконтроля, когда невозможно обойтись без модификации текста исходных программ.

В этом примере приложение представляет собой совокупность объектов, реализующих интерфейс Store.

```
typedef long Status;
typedef long Label;
typedef sequence<octet> Info;

interface Store {
    Label save (in Info info_piece);
    Status read (in Label what, out Info result);
    Status remove (in Label what);
}
```

Каждый такой объект предназначен для использования одним клиентом в качестве хранилища данных произвольного размера. Такой серверный объект может использоваться для сохранения данных из других процессов, которые не могут сохранять большие порции информации в собственной оперативной памяти. Например, это может быть процесс, контролирующий получение видео информации, работающий на устройстве, вырабатывающем поток таких данных, но не имеющем собственной памяти для сохранения этих данных. Такой серверный объект может использоваться и локально как посредник при работе с динамической памятью.

Методы объекта предназначены для сохранения (save) данной порции информации, переданной в виде последовательности байтов. При таком сохранении для данных выделяется кусок динамическая память нужного размера. Идентификатор (метка) этого куска памяти, где были сохранены полученные данные, возвращается в качестве результата работы метода save. Метод read предназначен для чтения ранее сохраненных данных. Метод remove удаляет данные, когда они становятся не нужны.

Итак, такой объект реально представляет собой простейший посредник в работе с динамической памятью процесса и для удаленного, и для внутреннего доступа.

Метарасширение здесь заключается в организации работы с сервисом Persistency. Оно осуществляется *практически* без внесения изменений в готовую реализацию Store_realization.

Для организации Persistency в оболочке используется объект типа:

```
interface PM {
    typedef string PID;

    Status persist_save_anywhere (in Label elem_id, out PID pid_res);
    Status persist_save (in Label elem_id, in PID pid_arg);
    Status persist_restore (in PID pid, out Label elem_id);
    Status persist_delete (in PID pid);
}
```

Такой объект позволяет сохранить в файле (PID задает имя файла) либо все текущее состояние объекта типа Store (elem_id=-1), либо только некоторый его элемент. При этом метод persist_save позволяет задать место сохранения (идентифицируемое значением pid), а persist_save_anywhere находит место сохранения самостоятельно. Метод persist_restore позволяет восстановить состояние объекта (или один элемент этого состояния), а метод persist_delete удаляет указанное сохранение (файл).

Подчеркнем, что этот пример демонстрирует возможность организации с помощью метаконтроля долговременного хранения не только всего состояния объекта, но и некоторой части этого состояния. Методы объекта типа Persistency позволяют сохранять в файле отдельно указанную порцию информации, а также дает возможность затем ее восстановить. При этом, вообще говоря, изменяется метка, идентифицирующая такие данные (новое значение метки выдается через OUT параметр метода persist_restore).

Также пример иллюстрирует весьма важную возможность, предоставляемую предлагаемым механизмом метаконтроля. Она состоит в том, что видимый пользователем интерфейс (клиентского ИО) можно изменять также, как интерфейс серверного ИО. В этом случае, правда, для сохранения возможности прозрачной работы метаконтроля (пользователь должен иметь возможность работать с клиентским ИО так, будто ему доступен непосредственно интерфейс приложения) необходимо производить только добавление новых методов с наследованием всех методов интерфейса приложения. Очевидно, что новые методы

клиентского ИО могут выполняться полностью в обход целевого объекта, т.е. они могут строиться лишь на работе с метаконтекстом.

Итак, описываемое метарасширение объектов типа Store приводит к расширению интерфейса, доступного клиенту, но клиент может работать с этими объектами и как раньше, даже не зная о происшедшем изменении интерфейса.

Проиллюстрируем некоторые особенности описываемого в этом примере особого типа метаконтроля на некоторых частях программ на языке C++.

Будем предполагать, что объекты, реализующие интерфейс Store, являются реализациями класса Store_realization:

```
class Store_realization {
protected:
	Info *info_arr;
	long arr_length;
public:
	Label save (const Info &info_piece);
	Status read (Label what, Info *&result);
	Status remove (Label what);
};
```

Итак, каждый объект указанного типа хранит интересующую нас информацию в виде массива структур, каждая из которых предназначена для указания на одну хранимую в динамической памяти порцию данных (значение типа Label указывает номер элемента этого массива). Поле info_arr ссылается на такой массив (размер этого массива, конечно же, может динамически изменяться в соответствии с некоторым алгоритмом). Все три метода этого класса работают с этим массивом и не имеют никаких собственных состояний, интересующих сервис Persistency. Конечно же, в отличие от самих методов, инкапсулируемое объектом типа Store_realization состояние не доступно извне этого объекта ни для чтения, ни для модификации.

Для работы с такого типа объектами мы разрабатываем класс

```
class StorePersistency {
protected:
	Store_realization *_target_obj;
public:
	// конструктор
	StorePersistency (Store_realization *target) : _target_obj (target) {};

	Status persist_save_anywhere (Label elem_id, PM::PID &pid_res);
	Status persist_save (Label elem_id, const PM::PID pid_arg);
	Status persist_restore (const PM::PID pid, Label &elem_id);
```

```
    Status persist_delete (const PM::PID pid);
};
```

Объект указанного типа хранит ссылку на объект типа Store_realization. Эта ссылка заполняется (через параметр конструктора) оболочкой при создании объекта. Однако, доступные извне методы класса Store_realization не дают возможность непосредственно работать с состоянием объектов этого типа. Для того, чтобы объекты типа StorePersistency имели доступ к этому состоянию, в объявлении класса Store_realization необходимо добавить следующую строку:

```
friend class StorePersistency;
```

Такое объявление делает доступным (и на чтение и на модификацию) состояние объектов типа Store_realization всем объектам типа StorePersistency. Добавление такой строки представляет собой единственное необходимое в этом примере изменение в текстах исходных программ приложения.

Приведем теперь часть возможной реализации методов класса StorePersistency :

```
Status
StorePersistency::persist_save_anywhere (Label elem_id, PM::PID &pid_res) {
    // поиск, где можно сохранить данные
    ...
    pid_res = ...;
    return persist_save (elem_id, pid_res);
}
Status
StorePersistency::persist_save (Label elem_id, const PM::PID pid_arg) {
    FILE *file = fopen (pid_arg, “wb”);
    if (file == 0)
        return -1;
    if (elem_id < 0) (
        // сохранение всего состояния подопечного объекта
        Info *cur_info = _target_obj-> info_arr;

        fprintf (file, “%d”, _target_obj-> arr_length);
        for (int i = 0; i < _target_obj-> arr_length; i++, cur_info++) {
            // запись длины и содержимого текущего элемента
            fprintf (file, “%d”, cur_info->length());
            fwrite (&((*cur_info)[0]), cur_info->length(), 1, file);
        }
    }
    else {
        fprintf (file, “%d”, -1);
```

```
            ...// сохранение одного указанного элемента
        }
        return 0;
}
Status
StorePersistency::persist_restore (const PM::PID pid, Label &elem_id) {
        elem_id = -1;
        FILE *file = fopen (pid_arg, “rb”);
        if (file == 0)
            return -1;

        long elem_cnt;
        fscanf (file, “%d”, &elem_cnt);
        if (elem_cnt < 0) {
            ...// восстановление одного элемента
}
else {
            ...// восстановление всего массива _target_obj-> info_arr
        }
        return 0;
}
Status
StorePersistency::persist_delete (const PM::PID pid) {
        ...// удаление файла с именем pid
}
```