Меня поражает факт, которого никто не желает признать: жизнь Духа иногда прерывается. Только жизнь Разума непрерывна или почти непрерывна. Моя способность размышлять не претерпевает больших изменений. Для Духа же важны не сами вещи, а смысл, связывающий их между собой. Подлинное лицо вещей, которое он постигает сквозь внешнюю оболочку. И Дух переходит от ясновидения к абсолютной слепоте. Для того, кто любит свой дом, настаёт час, и он вдруг обнаруживает, что это не более чем скопище разрозненных предметов.

Антуан де Сент-Экзюпери

Параллелизм и связь между объектами

На основе объектной технологии становится проще разрабатывать системы с высоким уровнем параллелизма. Объект можно представить как некоторую обособленную сущность, взаимодействие с которой осуществляется с помощью объявленного интерфейса. Инкапсуляция защищает область данных объекта от случайного или умышленного изменения. Как следствие, несколько объектов могут работать параллельно, избегая негативного влияния друг на друга. Это действительно возможно, если правильно реализовать взаимодействие между объектами. В частности, надо избегать прямых или косвенных обращений к свойствам объектов (вызовов методов), отдавая предпочтение взаимодействию на основе сообщений. Говоря общими терминами, можно сказать, что синхронную форму связи желательно заменять асинхронной. При синхронном режиме взаимодействия управление передаётся тому объекту и тому свойству, к которым обращаются. Возврат управления происходит после того, как свойство завершило работу. Асинхронное взаимодействие не забирает управление у того объекта, который произвёл вызов. Вызывающий объект после асинхронного обращения продолжает работу.

Свойство объекта может возвращать или не возвращать результаты. При синхронной связи результат работы вызванного свойства передаётся в момент возврата управления. При асинхронной связи результат возвращается в некоторую точку синхронизации. Если расположить точку синхронизации непосредственно за асинхронным обращением, то результат будет аналогичен синхронному обращению. То есть, синхронное обращение к свойству объекта можно считать частным случаем асинхронного обращения.

Обычно синхронные обращения ассоциируются с прямыми или косвенными вызовами подпрограмм, в то время как асинхронные обращения, отождествляют с обменом сообщениями. На самом деле это не более чем удобные метафоры, и именно в силу своего удобства и простоты восприятия они и будут использоваться при дальнейшем изложении.

Синхронные вызова подпрограмм считаются более быстрыми и экономичными формами связи по сравнению с асинхронной передачей сообщений. Однако такое утверждение справедливо далеко не всегда. При последовательном режиме работы вызов подпрограммы действительно происходит и быстро, и экономично, например, он сведётся к следующему набору операторов:

```
        push    param1
        push    param2
        …
        push    paramN
        call    MyProc                          ; прямой вызов
        …
или
        push    param1
```



```
        push    param2
        …
        push    paramN
        call    [eax * 4 + VirtualTable]     ; косвенный вызов
        …
```

Но с переходом к параллельному исполнению, утверждение о более быстрой форме синхронных вызовов может утратить свою силу. Чтобы понять, почему так происходит, достаточно привести простой пример. Рассмотрим работу участкового врача. Пусть у него на попечении находится всего один больной. Приняв вызов от пациента, врач отправляется к нему на дом, и производит осмотр или другие медицинские процедуры. Эта схемы работы оптимальна. Теперь увеличим число больных. Если врач по-прежнему будет ходить на каждый вызов, то первых пациентов он, возможно, обслужит быстро, чего нельзя сказать про остальных. Здесь будет более удобна другая форма обслуживания. Врач, получив несколько вызовов, оптимизирует маршрут обхода и только после этого начинает посещения больных. Экономия времени и сил врача происходит не только за счёт того, что он не возвращается за каждым новым вызовом в поликлинику, но и за счёт того, что минимизировано расстояние до каждого следующего пациента.

Аналогичные рассуждения можно привести и в отношении программных систем. Если бы тот, кто нуждается в сервисе, предоставляемом системой, был в единственном числе, то форма прямых или косвенных вызовов была бы оптимальным решением. Но, предположив большее число «пациентов», нужно предположить и возможную неэффективность данной формы связи. В отличие от вызовов, сообщения не попадают непосредственно к обработчику. Сначала они помещаются в очередь, где порядок их обработки может быть изменён с целью оптимального исполнения всей совокупности поступивших сообщений. В качестве примера можно рассмотреть работу некоторых подсистем операционной системы.

Пусть нам необходимо спроектировать менеджер памяти. В многозадачной операционной системе память, как правило, сильно фрагментирована. Для учёта свободных и занятых блоков или страниц существуют различного рода таблицы. При получении запроса на выделение блока памяти менеджер должен выполнить просмотр этих таблиц и найти либо наиболее подходящий блок, либо первый блок не меньшего размера. Когда количество запросов на выделение и освобождение памяти возрастает, то растёт и совокупное время, затрачиваемое на просмотр таблиц. Простая оптимизация очереди запросов позволит, во-первых, удовлетворить часть запросов на выделение памяти за счёт запросов на освобождение памяти. Во-вторых, можно ускорить обработку запросов посредством простой сортировки очереди, например, в порядке возрастания или убывания размеров запрашиваемых блоков. В этом случае можно удовлетворить все запросы за один проход по системным таблицам. Учитывая то, что количество фрагментов, как правило, существенно больше, чем число запросов поступающих за фиксированный промежуток времени, такая оптимизация может существенно ускорить работу менеджера памяти.

В качестве другого примера можно рассмотреть работу дисковой подсистемы. Пусть несколько задач будут одновременно что-то писать на диск и/или читать с диска. Очевидно, что основное время будет тратиться на пробег головки к требуемой области диска. Если предположить, что чтение и/или запись производятся в произвольных частях диска, и подсистема реагирует на каждый запрос, то среднее время исполнения запроса будет примерно равно N*T/2, где N – это число запросов за единицу времени, а T – время пробега по всему диску. Отказ от вызовов позволит ставить запросы в очередь. Сортировка очереди даёт возможность обслужить всё множество запросов за один пробег головки по диску, а, следовательно, среднее время исполнения запроса будет находиться в пределах - T/N.

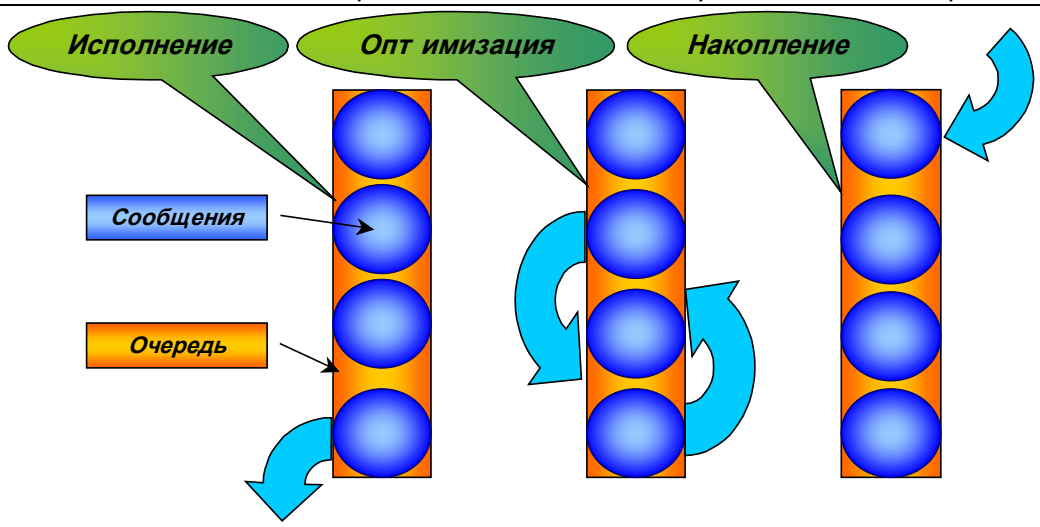

Рис. П04.01. Конвейер обработ ки сообщений

Наверное, имеет смысл отметить, что оптимизация очереди, обработка сообщений, которые в ней находятся, и регистрация новых сообщений - это различные по своей сути процессы, и они могут протекать параллельно. Для того, что сделать возможной параллельную работу можно использовать несколько очередей. Например, в то время, пока объект-обработчик обслуживает сообщения из первой очереди, сообщения во второй очереди оптимизируются, а третья очередь принимает новые сообщения. После того, как обслужены все сообщения из первой очереди (очередь пуста), на обработку поступает вторая очередь сообщений, третья очередь попадает под оптимизацию, а новые сообщения поступают в первую очередь, и т.д. Таким образом, можно получить высокоэффективный конвейер по обработке сообщений, все участки которого работают параллельно. В отличие от этого, при использовании механизма прямых и/или косвенных вызовов выполнить оптимизацию запросов практически невозможно.

Другая важная особенность механизма сообщений заключается в том, что источник сообщения и его получатель могут находиться на разных компьютерах. Это позволяет строить распределённые системы почти столь же просто, как и создавать приложения, работающие на одном компьютере. Функция диспетчеризации сообщений от источника к получателю является одним из сервисов объектной системы. Иными словами, можно утверждать, что объектная система, построенная на сообщениях, может быть с минимальными усилиями преобразована в распределённую среду в рамках локальной или Intranet сети. Это тоже важное условие при построении сложных систем.

Системы, построенные на сообщениях, обладают большей гибкостью, чем системы, построенные на вызовах. Повышенная гибкость связана с возможностью перехвата и перенаправления сообщений, их предварительной обработкой, до того, как они поступят к получателю. Данное свойство может быть полезно, например, для того чтобы организовать проверку прав доступа источника сообщения к ресурсу, для проверки типов параметров запроса, для регистрации запросов, что очень помогает при отладке программных модулей и т.п. Отсюда можно сделать ещё один вывод, что системы, построенные на основе обмена сообщениями, обладают большей изолированностью участков кода. Применительно к объектным системам можно добавить, что при помощи сообщений поддерживается полная инкапсуляция. Разработчикам, использующим некий класс, достаточно иметь список сообщений, понимаемый данным классом. Но этот же механизм даёт возможность расширить понятие полиморфизма, то есть, не рассматривать его как часть механизма наследования.

Не менее интересно и то, что сообщения можно создавать заранее. Возможно, что данное свойство может показаться несколько странным, но именно благодаря этому, возможно, не только конструировать схемы обработки, которые будут рассмотрены позже, но и создавать схемы-шаблоны. Под схемой подразумевается порядок диспетчеризации исходного сообщения к нескольким объектам, то есть разложение исходного сообщения на серию сообщений к другим объектам. Схема-шаблон – это виртуальная схема, которая может быть использована при создании новых классов и/или назначена непосредственно во время работы объекта.

Возвращаясь к параллелизму, надо отметить, что параллелизм надо рассматривать на трёх уровнях. Первый уровень параллелизма основан на параллельном исполнении процессором нескольких машинных команд. Этот уровень с успехом используется при распараллеливании алгоритмов, позволяя значительно увеличивать скорость исполнения. Современные технологии, предлагаемые разработчиками процессоров, значительно расширяют возможности данного уровня параллелизма. Например, технология EPIC (Explicitly Parallel Instruction Coding) воплощённая фирмами Intel и Hewlett Packard в процессоре Itanium, является развитием идеи параллельного исполнения нескольких команд. Большое регистровое пространство с возможностью переименования регистров, создание и передача подпрограммам регистровых пулов позволяют параллельно исполнять не только отдельные машинные команды, но целые блоки операторов.

Второй уровень параллелизма состоит в параллельном исполнении задач на одном или нескольких процессорах. Существующие двух- и четырёхпроцессорные компьютеры будут постепенно вытесняться компьютерами, имеющими значительно большее количество процессоров. Однако при существующей SMP (симметричные мультипроцессоры) технологии, основанной на общей разделяемой памяти с единым физическим адресным пространством, рост количества процессоров будет сдерживаться ограничениями, присущими этой технологии. Использование общей разделяемой памяти требует постоянного согласования одних и тех же данных (информации расположенной в некоторой физической области памяти) изменяемых различными процессорами. Это приводит к интенсивному обмену информацией между процессорами. Как следствие, процессоры должны соединяться по схеме каждый-с каждым с помощью высокоскоростных каналов. С ростом количества процессоров возрастает и время, требуемое на синхронизацию данных, возрастает и количество каналов, и сложность арбитража. Как правило, количество процессоров в SMP-системах не превышает 32.

Следующий уровень состоит в переходе к системам с массовым параллелизмом (MPP-системы). Системы с массовым параллелизмом представляют собой вычислительные модули соединённые скоростными каналами связи. Каждый вычислительный модуль является либо отдельным процессором, либо SMP-системой. У каждого модуля есть своя собственная память, которая не разделяется с другими модулями. В силу этой особенности существует возможность выполнения разнородных задач на каждом из модулей при отсутствии взаимного влияния. Независимость задач исполняемых в различных MPP-модулях снижает необходимость в обмене информацией между модулями, поскольку исчезает потребность в регулярной синхронизации общих данных. Межзадачный обмен информацией происходит в виде обмена сообщениями (пакетами), и этот процесс напоминает взаимодействие компьютеров в сети.

Объекты обладают важным, с точки зрения развития параллелизма, свойством, они представляют собой изолированные сущности. При правильном проектировании никакие два объекта не должны иметь пересечений ни по данным, ни по обслуживаемым ими ресурсам. Следовательно, благодаря объектной технологии можно существенно увеличить параллельность работы и осуществить переход к MPP системам.

Связи внутри контейнера

Взаимодействие объектов в рамках некоторого программного модуля может быть организовано различными способами. Например, один объект может прямо обращаться к свойствам другого объекта, посредством механизма синхронного или асинхронного вызова, через объявленный интерфейс или непосредственно. Примеры такого взаимодействия «объект-объект» можно встретить в литературе.

Однако недостатки такой формы связи вполне очевидны. Если в коде одного объекта встречается обращение к свойству другого объекта, то использовать эти объекты порознь невозможно. Как следствие, утрачивается гибкость и управляемость. Альтернативой для данной связи является связь через посредство контейнера. В этом случае, два объекта, принадлежащие одному контейнеру, не могут взаимодействовать друг с другом непосредственно. За их взаимодействие отвечает контейнер. Это позволяет отделить логику межобъектного взаимодействия от логики работы самих объектов. В результате внутренняя логика класса становится существенно проще, объекты этого класса можно использовать в любых сочетаниях с объектами других классов и, наконец, логика управления вложенными классами обретает независимость и гибкость. Эта логика взаимодействия и является логикой контейнера.

Важность данного положения трудно переоценить. И дело здесь не столько в том, что повторное использование классов значительно повышается, сколько в том, что существенно упрощается процесс разработки сложных систем и программных комплексов. Используемые в настоящее время классы логически просты, никто, пока не создаёт такие классы, как, скажем, предприятие. И, вследствие этого необходимость в разделении логик пока неочевидна. Но сложные составные классы могут иметь весьма нетривиальную логическую реализацию. Например, такой контейнер может находиться в различных состояниях и, в зависимости от своего состояния, так или иначе, обрабатывать входное сообщение. Мало того, изменение состояния может произойти непосредственно в процессе обработки сообщения, что может быть связано с изменением схемы дальнейшей трансляции. В такой ситуации очень сложно, или даже вообще невозможно, реализовать логику межобъектного взаимодействия внутри вложенных классов.

Инкапсуляция контейнеров подразумевает, в частности, что контейнер может эмулировать любой системный сервис для вложенных объектов. Например, если одному из вложенных объектов потребовалась память, то он обращается с запросом на выделение памяти к своему контейнеру. Безусловно, каждый контейнер не должен реализовывать весь системный сервис, но контейнеру, возможно, потребуется перехват обращений вложенных объектов за системным сервисом, например, в целях оптимизации работы всей совокупности вложенных объектов.

С другой стороны, подобное решение избавляет разработчиков классов вложенных объектов от необходимости знать о том, кто предоставляет тот или иной сервис. Всё что им необходимо для работы предоставляет контейнер – «хозяин» этих вложенных объектов. При загрузке или создании контейнера система определяет, будет ли контейнер перехватывать запросы вложенных объектов или эти запросы необходимо передать службам, минуя контейнер (обработка запросов по умолчанию). Такое решение повышает гибкость, не понижая при этом эффективности обработки запросов.

Вложенные в контейнер объекты могут быть как активными, так и пассивными. Активные объекты генерируют события, которые, в виде сообщений, поступают к контейнеру. Пассивные объекты не создают сообщений, а лишь выполняют распоряжения, поступающие от контейнера.

Генерация события активным объектом не обязывает контейнер обрабатывать соответствующее сообщение, то есть, данное сообщение может быть

проигнорировано контейнером. Однако не исключено, что с сообщением от вложенного объекта будет связана некоторая схема обработки. И поступление такого сообщения будет обработано аналогично обработке внешнего, по отношению контейнеру, сообщения.

При обработке внутренних сообщений необходим контроль для исключения бесконечных циклов, когда вложенный объект генерирует сообщение, схема обработки которого предусматривает обращение к этому же свойству вложенного объекта, которое может вновь создать то же самое внутреннее сообщение и т.д. Данный вид контроля должен выполняться системой автоматически при создании новой схемы обработки внутреннего сообщения.

Инкапсуляция контейнера и вложенных объектов

Теоретически контейнер может непосредственно включать в себя структуры вложенных объектов. Результат такого включения семантически близок к множественному наследованию и, как отмечалось ранее, такое решение очень неудобно в практическом применении. Следует отметить, что физическое вложение структур, если и возможно то, только для статических контейнеров. Динамические контейнеры потребуют очень больших накладных расходов для поддержания такой организационной схемы. Помимо этого, прямое включение не позволяет разделять один и тот же объект между несколькими контейнерами, что может осложнить реализацию параллельной обработки в соответствующих задачах.

Однако и со статическими контейнерами возникают серьёзные проблемы. Как отмечалось ранее, при рассмотрении темы «Полиморфизм», вложенные в статический контейнер объекты выступают в некоторых ролях и замена одного объекта на другой возможна, если новый объект способен выступать в тех же ролях, в которых выступал его предшественник. Однако одну и ту же роль могут играть совершенно разные объекты, в том числе и объекты, не находящиеся в связи наследования. Как следствие, структуры этих объектов могут значительно отличаться друг от друга, что вызовет определённые проблемы при замене одного объекта другим.

Включение по ссылке имеет очевидные преимущества по сравнению с непосредственным включением вложенных объектов в контейнер. Но остаётся вопрос о том, должна ли ссылка быть указателем на структуру объекта или она должна представлять собой некоторый идентификатор объекта, не имеющий прямой связи с местом расположения объекта в памяти. На первый взгляд, преимущества использования указателей состоят в большей эффективности, но здесь уместно вспомнить, что вложенные объекты инкапсулированы, и обращение к ним происходит на основе объявленных интерфейсов. В этом случае необходимость и эффективность использования указателей, вызывает сомнения. С другой стороны, наличие указателей исключает возможность использования в одном контейнере объектов, расположенных в различных адресных пространствах, например, на разных компьютерах. Это ограничение может создать проблемы при разработке распределённых систем. Кроме того, часто отладку кода новых объектов лучше производить изолировано от остального кода, в отдельном адресном пространстве. После того, как процесс отладки и опытной эксплуатации успешно завершены, новый объект может размещаться в том же адресном пространстве, что и другие, вложенные в этот контейнер, объекты. Процесс перемещения объекта из одного адресного пространства должен быть прозрачен не только программиста, но и для контейнера-владельца.

Идентификация объектов внутри контейнера без применения указателей способствует инкапсуляции вложенных объектов, поскольку от контейнера не требуется знать ни размеры вложенных объектов, ни их адреса. В таком случае, вложенные объекты «спрятаны» от контейнера, который ими владеет. Благодаря

этому становится возможным и использование несколькими контейнерами одного объекта, и произвольное размещение вложенных объектов в различных адресных пространствах, и высокая защищённость вложенных объектов, как от контейнера, так и от взаимного влияния. При этом указатель на объект является частным случаем идентификатора объекта, если учесть, что два объекта не могут располагаться по одному адресу. Следовательно, система может использовать указатели, если это необходимо и возможно, но от разработчиков не требуются выполнения операций манипуляции с адресами, которые могут привести к снижению надёжности системы.

В свою очередь, контейнер «невидим» для вложенных в него объектов. Вложенные объекты не должны зависеть от того, в какой контейнер их вложили. Это аналогично тому, что работоспособность подпрограммы не должна быть связана с тем, в какой программе её используют.

Связь с суперклассами

Вопрос о том должен ли подкласс «знать» структуру своего суперкласса довольно интересен и требует отдельного рассмотрения. Существует несколько решений, отражающих всё многообразие ответов, от однозначного «да», до столь же однозначного «нет». Например, сторонники того, чтобы подкласс «знал» структуру своего суперкласса, говорят, что незнание структуры своего суперкласса равносильно незнанию подклассом своей собственной структуры. И это абсолютно справедливо.

Оппоненты, в ответ на это, возражают, заявляя, что в этом случае нет, и не может быть инкапсуляции, то есть сокрытия реализации. Любой разработчик подкласса может обратиться к свойствам суперкласса, минуя интерфейс, но при этом не гарантируется правильность такого обращения. Например, суперкласс имел некоторое поле, которое должно принимать ограниченный набор значений, и это требование неукоснительно соблюдалось в суперклассе. Но разработчики подкласса могли не знать об этом требовании и, следовательно, могли выполнить присвоение значения вне заданного диапазона, минуя интерфейс. Теперь поведение подкласса может стать неопределённым, если хотя бы одно из свойств суперкласса, обращающееся к этому полю, осталось не перекрытым. Таким образом, инкапсуляция суперклассов должна соблюдаться и для подклассов и обращение к свойствам, наследованным от суперклассов, должно производиться только через объявленные интерфейсы.

Другим важным элементом связи с суперклассами является наличие конструкторов, которые работают при создании объектов. В рамках конструктора подкласса происходит каскадный вызов конструкторов всех суперклассов. Это достаточно бессмысленная и протяжённая операция.

Более простой и эффективный способ инициализации объектов состоит в инициализации по шаблону. Данный способ не требует ни каскадных вызовов, ни знания структуры создаваемого объекта. Всё, что требуется от разработчика класса, это указать значения, которые будут присвоены полям при создании нового объекта. На основании этих значений формируется шаблон инициализации для конкретного класса. Если этот класс имеет суперкласс, то полный шаблон данного класса будет состоять из шаблонов суперкласса, плюс собственный шаблон. Шаблон хранится в системе вместе с описанием класса. При создании объекта, на выделенную под объект область памяти накладывается шаблон данного класса, после чего идентификатор нового объекта возвращается его владельцу. Шаблон не нарушает инкапсуляцию, поскольку не несёт в себе информации ни о количестве, ни о типе полей какого-то класса. Шаблон представляет собой просто битовую маску.

Заключение

Связи между сущностями характеризуют сложность любой системы. Поэтому локализация связей и ограничение их видов способствуют понижению сложности системы. Отсюда проистекает рекомендация локализации связей элементарного класса. Элементарные классы должны поддерживать связи только между своими свойствами, но сами не должны иметь непосредственных связей с другими классами. Это обеспечивает простоту и более высокий уровень повторного использования класса. С другой стороны, независимость элементарных объектов позволяет проще организовывать их параллельную работу.

Взаимодействие между элементарными классами осуществляется посредством контейнера. Контейнер на основе определения роли каждого вложенного объекта определяет связи между ними при обработке внешних запросов. Эти связи выражаются в виде схем (графов), где можно легко определить этапы последовательного и параллельного исполнения ветвей. Вынесение связей между вложенными объектами на уровень контейнера имеет своей целью отделения логики работы элементарного класса, от логики управления этим классом. Связи, представленные в виде схем, образуют логику контейнера, существо которой состоит в управлении вложенными объектами для выполнения той или иной совместной работы (цели). Разделение логики исполнения от логики управления принципиально важно. И вынесение связей между вложенными объектами на отдельный более высокий уровень, является основой образования логики управления.

Аналогично тому, как локализация связей между свойствами является основой построения элементарных классов, точно также локализация связей между объектами внутри контейнера является основой для построения контейнеров. Контейнер является не только агрегатом вложенных объектов, но и проводником необходимого сервиса для этих объектов. Таким образом, любой контейнер образует локализованную среду исполнения для вложенных в него классов. Тот факт, что реальное предоставление того или иного сервиса может осуществляться не контейнером, а внешним программным обеспечением, остаётся скрытым от разработчиков элементарных классов.

Физическое представление связей между контейнером и вложенными в него объектами тоже должно быть прозрачным как для разработчиков конкретных контейнеров, так и для разработчиков элементарных классов. Конкретное представление связи может быть сформировано только в момент создания (загрузки в память) контейнера для исполнения. Как следствие, увеличивается гибкость использования вложенных объектов. Например, для одного и того же контейнера в одном сеансе вложенные объекты могут располагаться в одном адресном пространстве, а в другом сеансе они могут быть разнесены по нескольким адресным пространствам. Это положение станет актуально, когда будет понято преимущество (необходимость) использования перемещаемых (кочующих) ресурсов.

Основной формой связи должны быть асинхронные вызовы, поскольку с одной стороны, они при необходимости могут быть сведены к синхронным вызовам, а с другой стороны, данная форма связи обеспечивает максимальную гибкость, производительность и защищённость системы в целом. В тех случаях, когда требуется последовательная работа объектов в одном адресном пространстве, система может самостоятельно привести асинхронные вызова к синхронным в момент загрузки той или иной схемы контейнера.

Локализация, как элементарных объектов, так и контейнеров, а также связь на основе сообщений позволят строить параллельные (в том числе и распределённые) системы столь же просто и естественно, как и обычные системы, не обладающие параллелизмом. Мало того, уровень параллелизма может изменяться непосредственно при работе системы, в случае, например, если становятся доступными новые вычислительные ресурсы. В свою очередь, переход к системам с

массовым параллелизмом позволит не только наращивать производительность пропорционально увеличению количества процессоров, но использовать иные подходы при решении сложных задач.