INTERVALE

125083, Москва, ул. 8-го Марта, 10/12
Тел./Факс: (095) 212-8238, 212-9975
http://www.intervale.ru

SObjectizer-4 Book

Е. Охотников

Оглавление

Часть I

Введение

Глава 1

Обзор

1.1 Что такое SObjectizer

SObjectizer — это:

- набор принципов, правил и ограничений на проектирование и реализацию программ в рамках агентного подхода, называемый агентной моделью;
- библиотека C++ классов, позволяющая реализовать приложение в терминах агентов, их состояний, сообщений и событий.

1.2 История SObjectizer

История SObjectizer состоит из двух этапов. Начало было положено в Гомельском Конструкторском Бюро Системного Программирования (КБ СП) в рамках проекта SCADA Objectizer.

Затем последовал годичный перерыв в развитии Objectizer. Он был вызван тем, что коллектив разработчиков SCADA Objectizer распался.

После того, как два разработчика SCADA Objectizer А. Лабыч и Е. Охотников оказались в компании Интервэйл, начался SObjectizer-4 — новый проект, основанный на переработанных идеях SCADA Objectizer.

1.2.1 Этап КБ СП

Проект, получивший название SCADA Objectizer, осуществлялся в лаборатории АСУ ТП как проект по созданию инструментального средства создания АСУ ТП. Подобные инструментальные средства относятся к классу SCADA-систем. От традиционных SCADA-систем Objectizer отличался тем, что в него закладывалась объектная ориентированность. Но не в чистом виде (инкапсуляция, наследование и полиморфизм), а с учетом особенностей предметной области — событийно-ориентированных АСУ ТП реального времени.

В результате попыток объединения принципов объектного подхода и событийной природы решаемых задач возникла агентная модель: агенты обмениваются сообщениями, сообщения порождают события, события обрабатываются в соответствии с приоритетом.

Работы над собственно SCADA Objectizer осуществлялись с 1996 по 2000 год. В 1998 году была завершена и в 1999 году опробована на реальном проекте первая версия SCADA Objectizer. В начале 2000 года была выпущена вторая версия, которая являлась лишь улучшенным вариантом первой версии. Первые версии состояли из двух частей (визуального конструктора и системы реального времени) и работали под OS/2. Летом 2000 года была выпущена новая экспериментальная третья версия, в которой вместо конструктора использовался событийно-ориентированный язык описаний и которая была перенесена под Windows, OS/2 и Linux.

Разработка SCADA Objectizer в КБ СП осуществлялась под руководством А. Косарева. В создании SCADA Objectizer в разное время принимали участие В. Гайдуков, А. Лабыч, Е. Охотников, С. Саевич.

1.2.2 Этап Интервэйл

При выполнении ряда проектов в компании Интервэйл в 2001-2002 гг. оказалось, что применение в них агентного подхода может дать отличные результаты. Понадобился инструмент, поддерживающий агентный подход. И в 2002 году произошла реинкарнация SCADA Objectizer в виде SObjectizer-4.

Несмотря на то, что SObjectizer базируется на тех же принципах, что и SCADA Objectizer, SObjectizer является совершенно самостоятельным, написанным с нуля, продуктом, который имеет много существенных отличий от SCADA Objectizer. Например, в SObjectizer, в отличие от SCADA Objectizer, существует наследование для агентов.

1.3 Причины продолжения развития SObjectizer

Вероятно, существует множество как объективных, так и субъективных причин того, что SCADA Objectizer получил свое логическое продолжение в виде SObjectizer. Однако, необходимо заострить внимание на одной, пожалуй, самой важной:

В объектно-ориентированном подходе существует разрыв между проектированием и программированием. Проектирование осуществляется в рамках объектов, состояний, сообщений. Программирование же на универсальных объектно-ориентированных языках программирования осуществляется только в терминах объектов и вызовов методов объектов. В результате оказывается, что проектная модель задачи отличается от программной модели.

В случае же с SObjectizer этот разрыв гораздо меньше – ведь и проектирование, и программирование осуществляется в терминах агентов, состояний, сообщений и событий.

1.4 Что предоставляет SObjectizer

1.4.1 Агентно-ориентированная модель

SObjectizer определяет агентно-ориентированную модель, в рамках которой осуществляется проектирование реализации конкретных задач. В рамках этой модели любое приложение рассматривается как совокупность именованных объектов — агентов. Каждый агент имеет заранее определенные состояния. Взаимодействие между агентами осуществляется посредством обмена сообщениями. Агент выбирает сообщения, которые он желает обрабатывать — подписывается на сообщения. Возникновение и последующая

обработка сообщения, на которое подписан агент, называется событием. Сообщение может породить несколько событий, которые будут обрабатываться в соответствии с назначенными им приоритетами — т.н. приоритетная диспетчеризация событий. Агенты, имеющие одинаковое множество состояний, событий и сообщений, образуют класс агентов.

1.4.2 Библиотека классов для C++

SObjectizer представляет из себя библиотеку классов для C++. Данная библиотека содержит средства для реализации обмена сообщениями и диспетчеризации событий, а также базовый класс для реализации агентов.

SObjectizer предоставляет возможность реализовывать собственные средства диспетчеризации событий и несколько уже реализованных диспетчеров.

1.4.3 Распределенные приложения. SObjectizer Protocol

SObjectizer предоставляет возможность реализации распределенных приложений. В этом случае приложение может состоять из нескольких автономных модулей, для взаимодействия которых используется специальный протокол — SObjectizer Protcol (SOP). Благодаря SOP прикладное приложение может вообще не иметь представления о том, что оно распределенное, т.к. обмен сообщениями между модулями осуществляется совершенно прозрачно и не требует усилий со стороны разработчика.

1.5 Чем SObjectizer не является

В процессе изучения SObjectizer может складываться впечатление, что SObjectizer очень похож на тот или иной программный продукт или технологию. Это действительно возможно, т.к. обмен сообщениями, приоритетные события, именованные объекты и т.д. широко используются в программном обеспечении. Тем не менее, SObjectizer не является аналогом тех продуктов, на которые он оказался похожим.

1.5.1 SObjectizer не повторяет работу MS Windows

Чаще всего при знакомстве с SObjectizer его сравнивают с MS Windows. Это происходит из-за того, что взаимодействие сущностей (агентов в SObjectizer и окон в MS Windows) происходит посредством обмена сообщениями.

В действительности же на этом сходство и заканчивается. Механизм обмена сообщениями в SObjectizer существенно отличается от механизма обмена сообщениями в MS Windows. Вот только несколько принципиальных отличий:

- в MS Windows выбором сообщения для обработки занимается функция окна (даже если этот процесс скрыт в глубине оконной библиотеки классов). Окна получают все широковещательные сообщения. И обработчик окна выбирает то сообщение, в котором он заинтересован. В SObjectizer, вследствие подписки агентов на сообщения, при возникновении сообщения уже известно, кто его обработает;

- сообщение в MS Windows – это идентификатор, которому сопоставлено два скалярных значения: wParam и lParam. Задача помещения в этот набор параметров

произвольных структур данных ложится на программиста. В SObjectizer сообщением может быть объект любого сложного типа (класса или структуры);

- в MS Windows нет возможности управлять приоритетами поступающих сообщений. В SObjectizer приоритетность обработки событий является одним из ключевых факторов.

1.5.2 SObjectizer не является реализацией UML

При более глубоком знакомстве с SObjectizer обнаруживаются аналогии между агентной моделью SObjectizer и UML, например, в понятиях состояний, переходах из состояния в состояние при возникновении событий, выполнении действий при входе/выходе в/из состояния. Может показаться даже, что SObjectizer предоставляет способ реализации проектов в терминах UML.

Это не так. Действительно, оказалось, что между SObjectizer и UML есть много общего, хотя разработка сначала SCADA Objectizer, а затем и SObjectizer, осуществлялась практически безотносительно к UML. Возможно, аналогия между SObjectizer и UML возникла из-за того, что разработчики SObjectizer и UML смотрели на одни и те же вещи, но с несколько разных точек зрения.

Поэтому, между SObjectizer и UML есть тонкие, но очень важные различия:

- в SObjectizer состояние не может иметь внутреннего действия. В SObjectizer состояние определяет только список допустимых для обработки событий;
- в SObjectizer нет различия между понятиями деятельности (activity) и действия (action). Вместо этого есть одно понятие — событие (event). Все операции в SObjectizer выполняются в обработчиках событий.
- в SObjectizer, при наличии обработчиков входа/выхода в/из состояния, операция смены состояния не рассматривается как атомарное. Смена имени текущего состояния для агента действительно осуществляется, как атомарная операция. Однако, во время работы обработчика входа (выхода) состояние агента можно изменить.

Кроме того, текущая реализация SObjectizer отличается от UML тем, что не поддерживаются вложенные состояния агентов.

Единственной частью SObjectizer, взятой из UML, является понятие активного объекта, реализованное в виде активного агента в диспетчере с активными агентами.

1.5.3 SObjectizer не является сервером приложений

SObjectizer не является сервером приложений. Он не является готовым продуктом промежуточного слоя (middle tier). SObjectizer — это инструмент для разработки приложений, в том числе и распределенных. Но SObjectizer не имеет готовых средств для обеспечения развертывания (deployment) приложений и не обеспечивает, например, таких сервисов промежуточного слоя, как безопасность (security), маршрутизация запросов и балансировка нагрузки (load balancing), доступ к корпоративным данным, долговременное хранение состояний объектов (stateful objects), восстанавливаемость (fail-over) и пр.

1.5.4 SObjectizer не является жестким каркасом приложения

SObjectizer можно назвать каркасом (framework) приложения, т.к. он диктует свои правила и ограничения на структуру и организацию приложения. Но, в отличие, например, от Microsoft Foundation Classes (MFC) или Qt[1], SObjectizer является гораздо более "мягким" каркасом, хотя бы потому, что SObjectizer можно использовать в приложениях, реализованных с помощью других framework. Например, SObjectizer успешно используется в приложениях, которые построены как на MFC, так и на Qt, а также в CGI-модулях.

SObjectizer не является жестким каркасом потому, что он не требует, чтобы все приложение было "написано на агентах". Напротив, SObjectizer позволяет реализовать лишь часть объектов приложения в виде агентов. Так, с помощью SObjectizer создавались CGI-модули, в которых SObjectizer запускался только для выполнения одного действия — запроса данных от другого приложения. После выполнения этого запроса SObjectizer останавливался, а CGI-модуль продолжал свою работу. При этом объем кода CGI, который работал с SObjectizer, составлял всего несколько процентов от общего объема кода модуля.

Также SObjectizer не накладывает органичений на способы интерактивного взаимодействия с пользователем. С помощью SObjectizer можно создавать "черные ящики", работающие вообще без какого-либо диалога с пользователем. Но можно создавать и полноценные GUI-приложения с применением готовых оконных библиотек, таких как MFC, ATL, WTL, Qt, wxWindows и др. При этом SObjectizer позволяет создавать агентов, которые сами являются элементами пользовательского интерфейса (окнами), хотя для этого может потребоваться наличие специального диспетчера[2]. Например, уже сейчас в состав SObjectizer входят диспетчеры главной нити приложения для Windows (предназначен для использования в MFC или WinAPI приложениях) и для Qt.

[1] `http://www.trolltech.com`

[2] Главной сложностью для таких агентов во многих оконных библиотеках является то, что запуск обработчиков событий этих агентов должен осуществляться только на контексте главной нити приложения, для чего требуется специализированный диспетчер.

Глава 2

Агентный подход

В данной главе предпринимается попытка дать более-менее формальные определения как для самого агентно-ориентированного подхода, так и для основных понятий и терминов, которые используются в SObjectizer. Но целью главы является не вывод строгих, формализованных определений, а объяснение того, что же понимали под агентно-ориентированным подходом сами разработчики SObjectizer.

Можно провести аналогию с объектно-ориентированным подходом. Для ООП нет единого (однозначного и общепризнанного) определения того, что же из себя представляет ООП. Как и нет единой общепризнанной объектной модели данных. Как и нет единого объектно-ориентированного языка программирования. Здравый смысл подсказывает, что объектно-ориентированный означает ориентированный на объекты, т.е. первичны именно объекты, все остальное вторично.

Аналогично, агентно-ориентированный — означает, что во главу угла ставятся агенты. Что же понимается под агентами в SObjectizer? Об этом и пытается рассказать данная глава.

По ходу изложения материала будут часто упоминаться термины “агентный подход” и “агентно-ориентированная модель”. Сложно сказать, что они реально означают для каждого пользователя или разработчика SObjectizer. Но можно считать, что:

- *агентно-ориентированная модель* вводит классификацию сущностей, которые используются при проектировании и реализации программного обеспечения;
- *агентный подход* — это процесс использования агентно-ориентированной модели при проектировании и реализации программного обеспечения.

2.1 Основные понятия

Изложение агентной модели SObjectizer является наиболее сложной частью описания SObjectizer. Дело в том, что речь идет о понятиях, интуитивно понятных для разработчиков и опытных пользователей SObjectizer. Для начинающего пользователя сложно дать точные определения основных понятий. Многие понятия перекликаются с аналогичными понятиями в объектно-ориентированном подходе. Поэтому для понимания агентной модели необходимо знание ООП.

Еще одной сложностью при определении основных терминов является то, что такие понятия как состояние, сообщение, событие и агент являются взаимосвязанными, не существующими друг без друга. Что не позволяет дать определение одному термину

без использования в нем остальных терминов. Поэтому, перед подробным объяснением каждого из понятий, нужно дать краткое описание агентно-ориентированной модели SObjectizer:

Агентно-ориентированная модель SObjectizer включает в себя понятия *агента*, *состояния*, *сообщения* и *события*. Агент – это именованный объект, имеющий заранее определенное множество состояний. Взаимодействие между агентами осуществляется путем обмена сообщениями. Агенты реагируют на сообщения посредством событий — возникновение сообщения есть событие. События обрабатываются с помощью обработчиков событий. Состояние определяет можество событий, на которые реагирует агент, находясь в данном состоянии.

Далее основные понятия рассматриваются более подробно, начиная от самого основополагающего понятия *состояние* и заканчивая наиболее емким понятием *агент*.

2.1.1 Состояние

Любой объект реального мира в любой момент времени находится в каком-то состоянии. Все множество состояний, в которых может находиться объект, образует множество состояний объекта. Когда объекты реального мира моделируются в программе, программные объекты также имеют множество состояний, поскольку программный объект является отражением реального объекта. Только в программе количество состояний ограничивается условиями задачи.

С точки зрения разработки программ важность состояния в том, что состояние определяет набор допустимых действий, а смена состояния приводит к изменению набора допустимых действий. Например, включенную электрическую лампочку можно выключить, но нельзя включить. И наоборот, выключенную лампочку можно включить, но нельзя выключить.

В общем случае под допустимыми действиями понимаются как действия, которые можно выполнить **над** объектом, так и действия, которые может выполнить **сам** объект.

В SObjectizer любые действия над агентом может выполнять только **сам агент**. Нет возможности что-либо сделать над агентом без его ведома. В частности, только агент может изменить свое состояние. Поэтому для того, чтобы что-то выполнить **над** агентом, необходимо определить в **самом** агенте событие, обработчик которого будет осуществлять необходимые действия. Т.е. в SObjectizer допустимые действия определяются описанными для агента событиями. Поэтому можно сказать, что *состояние* — это поименованное множество разрешенных к обработке событий.

В SObjectizer, в отличие от UML, сменой состояний управляет сам агент. Агент может неоднократно изменить свое состояние в обработчике любого своего события. При этом на уровне агентной модели не связывается состояние, событие и переход в новое состояние, как это делается на диаграммах состояний UML. Т.о. смена состояния агента выносится с уровня агентной модели на уровень реализации агентов.

Аналогично UML каждому состоянию можно сопоставить обработчики входа и выхода в состояние. Но работа данных обработчиков не рассматривается как мгновенное, атомарное действие — атомарным является только смена имени состояния агента. В частности, обработчик входа в состояние может инициировать переход в другое состояние. На уровне агентной модели это не запрещено, хотя на уровне реализации это может привести к неожиданным побочным эффектам.

2.1.2 Сообщение

Сообщение — это передача информации от агента или агенту. Поскольку речь идет об информации, то необходимо различать тип передаваемых данных и сами данные. Поэтому, под понятием *сообщение* в SObjectizer понимаются как тип передаваемых данных (т.е. класс сообщения), так и сами данные (т.е. экземпляр сообщения).

Владение сообщением

Каждое сообщение в SObjectizer принадлежит конкретному агенту. Этот агент называется *владельцем* сообщения. Каждое сообщение имеет имя, состоящее из имени агента-владельца и имени сообщения в агенте.

Понятие владения сообщением не связано напрямую с тем, получает ли агент-владелец данное сообщение или отправляет его. Владение означает только то, что сообщение существует в системе только тогда, когда в системе существует агент-владелец.

Например, агент-лампочка может иметь два сообщения `включить` и `выключить`. Эти сообщения относятся только к агенту-лампочка, и именно агент-лампочка является получателем данных сообщений. Если в системе таких агентов несколько (`лампочка_1`, `лампочка_2`, ..., `лампочка_N`), то как-то нужно различать сообщения, относящиеся к каждому из агентов. Достигается это тем, что каждый агент владеет своим сообщением и в системе оказываются сообщения `лампочка_1.включить`, `лампочка_1.выключить`, ..., `лампочка_N.включить`, `лампочка_N.выключить`. Т.е. агент-лампочка владеет сообщениями, которые не отсылает, а только получает.

Другим примером может являться агент-датчик-температуры. Этот агент может владеть сообщением `температура_изменилась`. Каждый агент-датчик-температуры только отправляет это сообщение, но не получает его.

Могут быть случаи, когда агент владеет сообщениями, которые он не отсылает и не получает. Т.е. некий агент вводит в систему свои сообщения, которые вместо него используют другие агенты. Например, может быть агент `пожарная_сигнализация`, который владеет сообщением `пожарная_тревога`. Это сообщение отсылают датчики температуры, а получают, например, агенты автоматической системы тушения.

Т.е. на уровне агентной модели понятие владения сообщением сводится к способу формирования имени конкретного сообщения — имя сообщения формируется из имени агента и имени сообщения.

Экземпляр сообщения

В описании владения сообщениями указывалось, что в системе существуют сообщения, имя которых строится из имени агента-владельца и имени сообщения в агенте. При этом подразумевалось, что понятие “сообщение” соответствует понятию типа возможного обмена информацией между агентами. Т.е. существование сообщения `лампочка_1.включить` означает, что в системе возможна отправка *экземпляров* данного сообщения. Но сам экземпляр сообщения возникает только в момент выполнения операции *отправки* сообщения и исчезает после того, как экземпляр сообщения будет обработан. По аналогии с языками программирования можно сказать, что сообщение — это тип данных, а экземпляр сообщения — конкретный объект данного типа.

Целенаправленные и широковещательные сообщения

Целенаправленное сообщение — это передача информации конкретному агенту-адресату. Например, если есть агент-файл, то для записи в файл нужно отправить этому агенту сообщение с передаваемыми данными. Причем сообщение должно быть отправлено конкретному агенту, т.к. может существовать несколько агентов-файлов, а запись нужно осуществлять только в один из них.

Широковещательное сообщение — это сообщение, для которого нет конкретного адресата. Например, при обнаружении ошибок ввода-вывода агент-файл может отослать сообщение о том, что он не работоспособен. С точки зрения агента-файла это сообщение в никуда, во внешний мир. Агент-файл не знает, кто будет получать это сообщение. И будет ли кто-то получать его вообще.

Асинхронность сообщений

Все сообщения в SObjectizer являются асинхронными.

2.1.3 Событие

Событие — это реакция агента на сообщение. Если агент заинтересован в получении какого-либо сообщения, он должен объявить у себя событие и *подписать* его на сообщение. После этого возникновение в системе экземпляра сообщения будет приводить к генерации всех событий, которые были подписаны на сообщение.

Инцидент события

Сообщение, которое порождает событие, называется *инцидентом* данного события. Событие может иметь несколько различных инцидентов. В этом случае событие возникает при появлении в системе экземпляра любого из инцидентов (т.е. инциденты объединяются по маске “логическое ИЛИ”).

Экземпляр события

При генерации события возникает т.н. *экземпляр события* — пара <событие, экземпляр сообщения>. По аналогии с языками программирования можно сказать, что событие — это функция, а экземпляр события — это конкретный вызов функции с конкретными параметрами.

Приоритет события

Каждому событию должен быть назначен приоритет. Когда в системе возникает экземпляр сообщения, может быть сгенерировано множество экземпляров событий, у которых данное сообщение было инцидентом. Возникшие события будут обрабатываться в порядке убывания приоритета.

Возникновение экземпляра сообщения, генерация и обработка экземпляра события называются *диспетчеризацией*. При этом *диспетчеризация сообщения* означает генерацию и последующую обработку событий, у которых данное сообщение было инцидентом. *Диспетчеризация события* означает выделение процессорного времени на обработку конкретного экземпляра события.

Агентная модель SObjectizer определяет только, что событие имеет приоритет и диспетчеризация события выполняется с учетом приоритета. Но агентная модель не определяет конкретных схем диспетчеризации событий. Вполне возможно, что для различных задач могут потребоваться различные схемы диспетчеризации. Поэтому детали диспетчеризации событий вынесены на уровень реализации SObjectizer.

2.1.4 Агент

Агент — это именованный объект, который:

- имеет набор состояний;
- может владеть набором сообщений;
- может иметь набор событий.

Агент является отражением действующего объекта реального мира. Под действующим подразумевается то, что этот объект реагирует на некие внешние воздействия и способен выполнять что-либо.

Фактически на понятии агента выясняются различия между агентно- и объектно-ориентированными подходами. В ООП существуют только понятия объектов. И объекты, которые реально что-то выполняют, не сильно отличаются от объектов, которые используются для выполнения действий. Например, элекрическая лампочка, электрическая проводка и выключатель в ООП являются равноправными объектами. Каждый из них имеет набор атрибутов и методов (свойств). С точки зрения предметной области каждый из этих объектов что-то выполняет: лампочка светит, проводка пропускает электроток, а выключатель замыкает/размыкает электрическую цепь. Но в программном проекте, зачастую, действие элекропроводки не учитывается. Т.е. с точки зрения программы важно, что лампочка может светить или не светить, а выключатель — включать или выключать. Хотя сам объект “электрическая проводка” из рассмотрения не теряется, т.к. имеет важные свойства, например, протяженность, геометрию, сопротивление и т.д. Просто само действие, которое выполняет объект “электрическая проводка” настолько тривиально, что не рассматривается.

Итак, на примере с лампочкой, проводкой и выключателем видно, что, если рассматривать систему с точки зрения состава и свойств объектов, то нужно учитывать все объекты. Если рассматривать систему с точки зрения выполняемых объектами действий, то нужно сосредоточиться только на двух из них.

В этом смысле агентно-ориентированный подход является надстройкой над ООП. Т.е. агентно-ориентированный подход появляется там, где в ООП появляются объекты, выполняющие какие-либо действия. Такие объекты объявляются агентами, и дальше с помощью агентно-ориентированного подхода идет рассмотрение только их поведения и взаимодействия.

Класс агента

Агенты, имеющие одинаковые наборы состояний, владеющие однотипными сообщениями и обрабатывающие одни и те же события, образуют *класс агентов*. Агенты, принадлежащие одному классу, различаются только именами, приоритетами событий и списками инцидентов событий. Имена состояний, имена сообщений, списки событий,

разрешенных к обработке в каждом из состояний, у всех агентов одного класса одинаковы.

Понятие класса агента введено в агентную модель по двум причинам. Во-первых, разработчикам SObjectizer, привычным к использованию строго типизированных языков программирования, было проще манипулировать сущностями, имеющими тип. С этой точки зрения класс агента описывает тип агента (т.е. набор свойств, множество допустимых значений, список разрешенных действий и т.п.). Далее понятие класса и типа будут означать одно и то же. Во-вторых, при проектировании систем, состоящих из множества однообразных агентов, проще описать свойства агента в классе однотипных агентов, а затем указывать, какому классу принадлежит агент. Поэтому, если в системе какой-либо агент существует в единственном экземпляре, для этого агента также должен быть определен класс агента.

Ранее указывалось, что имя каждого сообщения состоит из двух частей: имени агента-владельца и имени сообщения. Такое составное имя диктуется понятием класса. При определении класса агента описываются все сообщения, которыми будут владеть агенты данного класса. Каждому сообщению в рамках класса назначается уникальное имя — имя сообщения. Но на уровне экземпляров агентов необходимо различать сообщения агентов одного класса. Для этого имя экземпляра сообщения собирается из имени агента и имени сообщения в классе агента.

Для агентов одного класса совпадают наборы событий и списки допустимых для обработки событий в каждом из состояний. Но каждый агент может устанавливать собственный приоритет на каждое из своих событий. Для каждого события агент может назначать собственные списки инцидентов. Например, в системе может быть два агента `пожарная_сирена` (на разных этажах здания). Эти агенты принадлежат одному классу и имеют событие `срабатывание_сигнализации`. Но у одного агента это событие может быть подписано на сообщения датчиков температуры, а у второго — на сообщения датчиков дыма.

Наследование классов агентов

В агентной модели SObjectizer существует понятие наследования классов агентов. Класс агента может быть производным от одного (одиночное наследование) или нескольких (множественное наследование) суперклассов. Производность означает, что:

- множество сообщений производного класса является объединением множеств сообщений всех базовых классов и собственного множества сообщений. А пересечение всех этих множеств должно быть пустым. Т.е. является недопустимым, если в производном классе оказывается хотя бы два сообщения с одинаковыми именами. Например, если суперкласс `A` владеет сообщением `m`, и производный класс `B` пытается определить сообщение `m`, то наследование `B` от `A` является недопустимым;

- множество событий и состояний производного класса является объединением множеств событий и состояний из базовых классов и собственных множеств событий и состояний. Как события, так и состояния могут быть переопределены (перекрыты) в производном классе. Если при множественном наследовании оказывается, что наследуются несколько событий (состояний) с одинаковыми именами, то наследование является допустимым, только если эти события (состояния) будут перекрыты в производном классе. Например, если суперклассы `A` и `B` имеют

событие `e`, то произвести от них класс `C` можно, только если перекрыть событие `e` в классе `C`.

Граф наследования должен быть ациклическим графом.

Один класс может являться суперклассом для нескольких классов, которые, в свою очередь, используются как базовые при множественном наследовании. Например, класс `A` является суперклассом для `B` и `C`, которые являются суперклассами для `D`. В этом случае считается, что класс `D` наследует единственный экземпляр класса `A`. В C++ это называется виртуальным наследованием.

Проводя аналогию с C++ события и состояния при наследовании классов агентов выполняют роль виртуальных методов. Например, пусть событие `e` суперкласса `A` переводит агента в состояние `s`. Пусть производный от `A` класс `B` перекрывает состояние `s`, но не событие `e`. Тогда в событии `e` агента класса `B` будет осуществлен переход в состояние `s` класса `B`, несмотря на то, что обработчик `e` принадлежит классу `A`.

Аналогично и с состояниями. Производный класс `B` может перекрыть событие `e`, не перекрывая ни одного унаследованного состояния. Тогда при возникновении события `e` для агента класса `B` будет вызываться обработчик из класса `B`, несмотря на то, что моменты, в которых событие `e` разрешено к обработке, определяется в суперклассе.

2.2 Применение агентного подхода

Агентно-ориентированный подход является, скорее, дополнением к объектно-ориентированному подходу, нежели его заменой. Фактически агентно-ориентированный подход является дополнением ООП в области динамического взаимодействия объектов. Т.е. в объектной модели разрабатываемой системы выделяются самостоятельные объекты, взаимодействующие друг с другом. Эти объекты объявляются агентами. Далее проектирование осуществляется практически так же, как и в ООП (описываются состояния, сообщения и события агентов), только с учетом того, что каждая из используемых сущностей (состояние, сообщение, событие) будет конкретным, заранее известным способом перенесена из проекта в программу.

2.2.1 Простой пример с двумя агентами

Рассмотрим систему из электрической лампочки, электрической проводки и выключателя. Здесь есть три объекта: “лампочка”, “проводка”, “выключатель”. Пусть выключатель может выдавать две команды: “включить” и “выключить”. На эти команды реагирует объект-лампочка и изменяет свое состояние: “светит” или “не светит”. Т.е. получается два агента: агент-выключатель и агент-лампочка.

Далее необходимо рассмотреть, какими состояниями, сообщениями и событиями обладает каждый из агентов. Агент-лампочка имеет два состояния: `st_светит` и `st_не_светит`. Агент-выключатель также имеет два состояния: `st_включен` и `st_выключен`. Агент-выключатель должен иметь два события: одно должно приводить к его включению, а второе — к выключению. При этом оба эти события должны инициироваться одним действием — переключением выключателя. Т.е. агент-выключатель должен иметь одно сообщение: `msg_переключение`, которое является инцидентом двух событий: `evt_переключение_когда_выключен`, `evt_переключение_когда_включен`. Каждое из этих событий актуально только в одном состоянии, поэтому событие

evt_переключение_когда_выключен допускается к обработке в состоянии st_выключен, а evt_переключение_когда_включен — в состоянии st_включен.

С сообщениями и событиями агента-лампочки можно поступать по разному. Можно, например, решить, что агент-выключать будет иметь два сообщения: msg_включить и msg_выключить. Эти сообщения будет отсылать агент-выключатель, а агент-лампочка будет на них реагировать. В этом случае окажется, что агент-лампочка будет жестко привязан к агенту-выключателю, т.к. агент-лампочка должен подписаться на сообщения агента-выключателя.

Можно, чтобы сообщениями msg_включить и msg_выключить владел агент-лампочка, а отсылал их агент-выключатель. Тогда агент-лампочка подписывается на собственные сообщения, но агент-выключатель оказывается связан с агентом-лампочкой, т.к. он должен знать, чьи сообщения он рассылает.

Возможно, что агент-лампочка будет иметь только одно сообщение: msg_напряжение_в_сети. Это сообщение будет иметь один атрибут — величина напряжения. Агент-выключатель при своем включении отсылает это сообщение, указывая в нем не нулевую величину. При выключении — отсылает то же сообщение, но с нулевой величиной напряжения. В этом случае агент-лампочка будет иметь одно событие evt_изменение_напряжения, которое обрабатывается в обоих состояниях.

Получившуюся агентную модель можно выразить на некоем псевдоязыке следующим образом:

```
// Описание класса агента-лампочки.
класс a_лампочка
  {
    // Владеет одним сообщением.
    сообщение msg_напряжение_в_сети
      {
        // Всего один атрибут.
        double m_volts;
      }

    // Имеет одно событие.
    событие evt_изменение_напряжения
      {
        // У которого один инцидент.
        инцидент a_лампочка::msg_напряжение_в_сети;
      }

    // Имеет два состояния, в каждом из которых
    // обрабатывается одно и то же событие.
    состояние st_не_светит
      {
        событие evt_изменение_напряжения;
      }
    состояние st_светит
      {
        событие evt_изменение_напряжения;
      }
```

```
  }

// Описание класса агента-выключателя.
класс a_выключатель
  {
    // Владеет одним сообщением.
    сообщение msg_переключение
      {}

    // Имеет два события.
    событие evt_переключение_когда_выключен
      {
        // У которого один инцидент.
        инцидент a_выключатель::msg_переключение;
      }

    событие evt_переключение_когда_включен
      {
        // У которого один инцидент.
        инцидент a_выключатель::msg_переключение;
      }

    // Имеет два состояния, в каждом из которых
    // обрабатывается одно событие.
    состояние st_выключен
      {
        событие evt_переключение_когда_выключен;
      }
    состояние st_включен
      {
        событие evt_переключение_когда_включен;
      }
  }
```

2.2.2 Простой пример с тремя агентами

Предположим, что требования к изложенной в предыдущем примере задаче изменились. Теперь на систему может воздействовать электрический провод, который соединяет лампочку и выключатель. Например, нужно учитывать, поврежден он или не поврежден. Т.е. включение выключателя не приведет к загоранию лампочки, если провод поврежден.

Для учета этого нужно ввести в систему агента `a_провод`, который может находиться в двух состояниях: `st_не_поврежден` или `st_поврежден`.

В этом случае агент-выключатель должен отослать сообщение о том, что напряжение подано. Чье же это должно быть сообщение? В предыдущем примере это было сообщение агента-лампочки, т.к. считалось, что выключатель напрямую взаимодействует с лампочкой. Теперь же между выключателем и лампочкой находится провод. Т.е. сообщение о том, что выключатель подал (прекратил подачу) напряжение, должно

принадлежать либо агенту-выключателю, либо агенту-проводу.

Выбор того, какой из агентов будет владеть данным сообщением, зависит от того, что нам удобнее. Если нам удобнее, чтобы агент-выключатель знал, кем он управляет, тогда агент-провод будет владельцем сообщения (поскольку агенту-выключателю нужно знать, чье сообщение отсылать).

Если же нам удобнее, чтобы агент-провод знал, кто им управляет, то тогда агент-выключатель должен быть владельцем сообщения. Т.е. агент-выключатель широковещательно отсылает свое сообщение, не зная, кому оно достанется. Но на это сообщение реагирует агент-провод, для чего ему нужно знать про существование агента-выключателя.

Применительно к данному примеру у нас нет никаких предпочтений, поэтому пусть это сообщение принадлежит агенту-проводу. По аналогии пусть агент-провод отсылает сообщение агента-лампочки. Тогда получится, что каждый элемент цепи знает, кому он передает электроток, но не знает, от кого он его получает.

Т.о. пусть агент-провод владеет сообщением `msg_напряжение_в_сети`, которое имеет ту же структуру, что и сообщение `msg_напряжение_в_сети` агента-лампочки. Но эти сообщения являются независимыми, т.к. принадлежат к различным агентам различных типов.

Ранее было сказано, что агент-провод имеет два состояния: `st_не_поврежден` и `st_поврежден`. Реагировать на появление/исчезновение напряжения в сети он должен в любом состоянии, но выполнять при этом различные действия. Так, если агент-провод находится в состоянии `st_не_поврежден`, и в сети появляется напряжение, то он должен отослать сообщение `msg_напряжение_в_сети` агента-лампочки. Поэтому для обработки собственного сообщения `msg_напряжение_в_сети` агент-провод должен иметь два события: `evt_напряжение_когда_не_поврежден` и `evt_напряжение_когда_поврежден`.

Также агент-провод должен уметь переходить из одного состояния в другое. Для этого он владеет сообщениями: `msg_поврежден` и `msg_восстановлен`. Для каждого из сообщений достаточно одного события, обработчик которого будет изменять состояние агента и отсылать сообщение `msg_напряжение_в_сети` агента-лампочки с указанием текущего напряжения.

В результате получается следующий агент:

```
// Описание класса агента-провода.
класс a_провод
  {
    // Владеет сообщениями:

    // о появлении/исчезновении напряжения в сети;
    сообщение msg_напряжение_в_сети
      {
        double m_volts;
      }
    // о повреждении провода;
    сообщение msg_поврежден
      {}
    // о восстановлении провода;
    сообщение msg_восстановлен
      {}
```

```
    // Имеет события:

    // об изменении напряжения, когда провод поврежден;
    событие evt_напряжение_когда_поврежден
      {
        // У которого один инцидент.
        инцидент a_провод::msg_напряжение_в_сети;
      }
    // об изменении напряжения, когда провод не поврежден;
    событие evt_напряжение_когда_не_поврежден
      {
        // У которого один инцидент.
        инцидент a_провод::msg_напряжение_в_сети;
      }
    // о повреждении;
    событие evt_поврежден
      {
        // У которого один инцидент.
        инцидент a_провод::msg_поврежден;
      }
    // о восстановлении;
    событие evt_восстановлен
      {
        // У которого один инцидент.
        инцидент a_провод::msg_восстановлен;
      }

    // Имеет два состояния, в каждом из которых
    // обрабатываются события:
    состояние st_поврежден
      {
        событие evt_напряжение_когда_поврежден;
        событие evt_восстановлен;
      }
    состояние st_не_поврежден
      {
        событие evt_напряжение_когда_не_поврежден;
        событие evt_поврежден;
      }
  }
```

Описания остальных агентов изменять не нужно. Следует изменить только реализацию агента-выключателя, чтобы он отсылал сообщение агента-провода, а не агента-лампочки. Для агента-лампочки вообще ничего не изменится.

2.2.3 Простой пример наследования

Приведенные ранее примеры предполагали, что каждый из классов агентов должен знать хотя бы про один из остальных классов, т.к. агентам необходимо отсылать сообщения о напряжении в сети. Это создает трудности при сопровождении полученной агентной модели. Например, если мы захотим подключить выключатель к еще одному проводу. Или последовательно подключить несколько лампочек.

Без применения наследования агентов нам бы пришлось либо учитывать все возможные способы коммутации элементов электрической цепи (например, агент-провод должен был бы уметь рассылать сообщения `msg_напряжение_в_сети`, принадлежащие как агенту-лампочке, так и агенту-выключателю), либо применять не строго типизированные модели (например, рассчитывать на то, что каждый агент-элемент электрической цепи будет обладать сообщением `msg_напряжение_в_сети` с одной и той же структурой данных).

Если же применить наследование, то можно получить более простое, понятное и надежное решение. Например, пусть существует базовый класс `a_элемент_цепи`, который только владеет сообщением `msg_напряжение_в_цепи`:

```
класс a_элемент_цепи
  {
    // Владеет сообщениями:

    // о появлении/исчезновении напряжения в сети;
    сообщение msg_напряжение_в_сети
      {
        double m_volts;
      }
  }
```

Данный класс не имеет своих состояний и событий и является интерфейсом. От этого класса наследуются классы реальных агентов:

```
класс a_лампочка : a_элемент_цепи
  {
  ...
  }

класс a_выключатель : a_элемент_цепи
  {
  ...
  }

класс a_провод : a_элемент_цепи
  {
  ...
  }
```

В этом случае каждый из реальных агентов должен знать имена агентов, которые к нему подключены. И, при необходимости, осуществлять отсылку сообщения `msg_напряжение_в_сети`, которым гарантировано владеет агент-получатель сообщения.

Глава 3

Поддержка агентного подхода средствами SObjectizer

В данной главе дается общее представление о том, как в SObjectizer реализованы основные понятия агентно-ориентированного подхода. При этом SObjectizer рассматривается уже как библиотека C++ классов, позволяющая реализовать приложение в терминах агентов, их состояний, сообщений и событий. Описание ведется на довольно высоком уровне, не вдаваясь в технические детали, которые будут освещаться во второй части книги.

3.1 Поддержка основных понятий

3.1.1 Агент

Агентом в программе является объект пользовательского класса, производного от предоставляемого SObjectizer класса *so_4::rt::agent_t*.

Классы агентов

Для описания класса агента в C++ нужно подготовить два описания:

- описать C++ класс, которому будут принадлежать объекты-агенты. Этот класс должен быть производен (прямо или косвенно) от класса *so_4::rt::agent_t*:

```
class a_hello_world_t
  : public so_4::rt::agent_t
  {
  public :
    a_hello_world_t();
    virtual ~a_hello_world_t();

    virtual const char *
    so_query_type() const;

    virtual void
    so_on_subscription();
```

```
  void
  evt_start(
    const so_4::rt::event_data_t & data );
};
```

Такое описание называется *C++-описанием класса агента*, т.к. оно необходимо языку C++;

- описать класс агента для SObjectizer. Это описание делается с помощью специальных макросов прямо в исходном C++ тексте:

```
SOL4_CLASS_START( a_hello_world_t )
  SOL4_EVENT( evt_start )

  SOL4_STATE_START( st_normal )
    SOL4_STATE_EVENT( evt_start )
  SOL4_STATE_FINISH()
SOL4_CLASS_FINISH()
```

Такое описание называется *описанием класса агента для SObjectizer*. SObjectizer нуждается в таком описании, т.к., во-первых, SObjectizer не может брать описание классов агентов из C++-описаний и, во-вторых, для SObjectizer нужно описывать такие вещи, которые нельзя представить с помощью синтаксиса языка C++ (например, состояния).

Экземпляры агентов

Объекты классов, производных от *so_4::rt::agent_t*, могут быть агентами. Могут, а не являются, потому что создание объекта-агента в программе еще не означает, что SObjectizer сразу будет воспринимать его как агента. Для этого необходимо выполнить процедуру *регистрации агента*.

Важно отметить, что за создание и уничтожение экземпляров агентов (т.е. объектов соответствующих C++ классов) отвечает программист. Т.е. SObjectizer **не контролирует время жизни объектов-агентов**. Этим должен заниматься прикладной программист (хотя SObjectizer предоставляет средства для упрощения этой задачи в виде т.н. динамических кооперaций).

Регистрация агентов

Для того, чтобы SObjectizer узнал о существовании объекта-агента необходимо выполнить операцию *регистрации* агента. Эта операция сообщает SObjectizer о том, что появился объект, который должен восприниматься как агент. SObjectizer определяет тип нового агента посредством виртуального метода *so_4::rt::agent_t::so_query_type* и определяет, какими сообщениями, событиями и состояниями обладает данный агент.

После того, как агент успешно зарегистрирован, он начинает жить своей жизнью в рамках SObjectizer, т.е. его сообщения могут быть отправлены и получены, события могут возникать и обрабатываться.

Поскольку агентом является C++ объект, время жизни которого ограничено, необходимо позаботиться о *дерегистрации* агента перед тем, как C++ объект-агент будет уничтожен. Т.е. нужно выполнить операцию, обратную регистрации – указать SObjectizer, что данный объект перестает быть действующим агентом.

3.1.2 Сообщение

Типом сообщения может быть любая структура или класс языка C++. Экземпляром сообщения, соответственно, является объект этого типа. Ограничений на тип сообщения (например, в виде необходимого базового класса) нет. Главное условие – тип сообщения должен быть корректным типом языка C++ (структурой или классом).

Так же, как и класс агента, тип сообщения должен быть описан два раза:

- C++ описание типа, реализующего сообщение. Например:

```
// Сообщение, содержащее данные.
struct msg_with_data
  {
    int m_int_data;
    double m_array_of_doubles[ 15 ];
  };

// Сообщение, не содержащее данных.
struct msg_without_data
  {
  };
```

- описание в рамках описания класса агента для SObjectizer. При этом описание нужно делать в том классе агента, который владеет данным сообщением. Например:

```
SOL4_CLASS_START( a_some_agent_class_t )
  SOL4_MSG_START( msg_with_data, msg_with_data )
    SOL4_MSG_FIELD( m_int_data )
    SOL4_MSG_FIELD_ARRAY( m_array_of_doubles )
  SOL4_MSG_FINISH()

  SOL4_MSG_START( msg_without_data, msg_without_data )
  SOL4_MSG_FINISH()
SOL4_CLASS_FINISH()

SOL4_CLASS_START( a_another_agent_class_t )
  SOL4_MSG_START( msg_without_data, msg_without_data )
  SOL4_MSG_FINISH()
SOL4_CLASS_FINISH()
```

3.1.3 Событие

Событие и обработчик события в SObjectizer являются неразделяемыми понятиями. Нет события без обработчика и нет обработчика без события. Обработчиком события

является нестатический метод класса агента формата:

- ```
  void
  evt_handler();
  ```

  или

- ```
  void
  evt_handler( const so_4::rt::event_data_t & );
  ```

  или

- ```
  void
  evt_handler(
    const so_4::rt::event_data_t &,
    const some_msg_t * );
  ```

  или

- ```
  void
  evt_handler(
    const so_4::rt::event_data_t &,
    const some_msg_t & );
  ```

  или

- ```
  void
  evt_handler(
    const some_msg_t * );
  ```

  или

- ```
  void
  evt_handler(
    const some_msg_t & );
  ```

Для SObjectizer событие становится известным после явного указания имени метода-обработчика в описании класса агента для SObjectizer:

```
SOL4_CLASS_START( a_some_agent_class_t )
  SOL4_EVENT( evt_some_event )
  SOL4_EVENT_WITH_INCIDENT_TYPE(
    evt_another_event,
    msg_another_event_incident )
  SOL4_EVENT_STC(
    evt_yet_another_event,
    msg_yet_another_event_incident )
SOL4_CLASS_FINISH()
```

Макросы *SOL4_EVENT*, *SOL4_EVENT_WITH_INCIDENT_TYPE*[1] и *SOL4_EVENT_STC* указывают SObjectizer, что у агентов класса `a_some_agent_class_t` есть три события:

[1] Данный макрос является устаревшим и не должен использоваться. Если необходимо указать тип инцидента события, то следует применять макрос *SOL4_EVENT_STC*.

evt_some_event (для которого не определен тип инцидента), evt_another_event (инциденты которого реализуются C++ типом *msg_another_event_incident*) и evt_yet_another_event (инциденты которого реализуются C++ типом *msg_yet_another_event_incident*). Т.е. соответствующий C++ класс *a_some_agent_class_t* должен был быть описан, например, так:

```
class a_some_agent_class_t
  :  public so_4::rt::agent_t
  {
  public :
    ...
    void
    evt_some_event();

    void
    evt_another_event(
      const so_4::rt::event_data_t & data,
      const msg_another_data_incident * msg );

    void
    evt_yet_another_event(
      const so_4::rt::event_data_t & data,
      const msg_yet_another_data_incident * msg );

    ...
  };
```

3.1.4 Состояние

Состояние декларируется только в описании агента для SObjectizer. В состоянии перечисляются события, разрешенные к обработке в данном состоянии, и необязательные обработчики входа в состояние и выхода из состояния. Например:

```
SOL4_CLASS_START( a_some_connection_t )
  SOL4_MSG_START( msg_connected,
    a_some_connection_t::msg_connected )
  SOL4_MSG_FINISH()

  SOL4_MSG_START( msg_connection_lost,
    a_some_connection_t::msg_connection_lost )
  SOL4_MSG_FINISH()

  SOL4_EVENT( evt_start )
  SOL4_EVENT_STC(
    evt_connected,
    a_some_connection_t::msg_connected )
  SOL4_EVENT_STC(
    evt_connection_lost,
    a_some_connection_t::msg_connection_lost )
```

```
    SOL4_INITIAL_STATE( st_initial )

    SOL4_STATE_START( st_initial )
      SOL4_STATE_EVENT( evt_start )
    SOL4_STATE_FINISH()

    SOL4_STATE_START( st_not_connected )
      SOL4_STATE_EVENT( evt_connected )
      SOL4_STATE_ON_ENTER( on_enter_st_not_connected )
    SOL4_STATE_FINISH()

    SOL4_STATE_START( st_connected )
      SOL4_STATE_EVENT( evt_connection_lost )
      SOL4_STATE_ON_EXIT( on_exit_st_connected )
    SOL4_STATE_FINISH()
  SOL4_CLASS_FINISH()
```

Данное описание говорит SObjectizer, что существует класс агентов `a_some_connection_t`, агенты которого обладают:

- сообщениями:
  - `msg_connected`, которое реализуется C++ типом *`a_some_connection_t::msg_connected`*;
  - `msg_connection_lost`, которое реализуется C++ типом *`a_some_connection_t::msg_connection_lost`*;
- событиями:
  - `evt_start`,
  - `evt_connected`,
  - `evt_connection_lost`;
- состояниями:
  - `st_initial`, в котором разрешено для обработки событие `evt_start`;
  - `st_not_connected`, в котором разрешено для обработки событие `evt_connected`. При входе в данное состояние вызывается метод *`on_enter_st_not_connected`*;
  - `st_connected`, в котором разрешено для обработки событие `evt_not_connected`. При выходе из данного состояния вызывается метод *`on_exit_st_connected`*.

Стартовым состоянием агента является состояние `st_initial` (т.е. после регистрации агенты данного класса оказываются в состоянии `st_initial`).

Данному классу агентов должен соответствовать следующий C++ класс:

```
class a_some_connection_t
  : public so_4::rt::agent_t
  {
  public :
    a_some_connection_t();
    virtual ~a_some_connection_t();

    virtual const char *
    so_query_type() const;

    virtual void
    so_on_subscription();

    struct msg_connected {};
    struct msg_connection_lost {};

    void
    evt_start();
    void
    evt_connected(
      const so_4::rt::event_data_t & data,
      const msg_connected * cmd );
    void
    evt_connection_lost(
      const so_4::rt::event_data_t & data,
      const msg_connection_lost * cmd );

    void
    on_enter_st_not_connected(
      const std::string & state_name );
    void
    on_exit_st_connected(
      const std::string & state_name );
  };
```

Для поддержки состояний в C++ классе агента не нужно ничего делать — всем этим занимается SObjectizer. Даже имя текущего состояния хранится не в самом агенте, а внутри SObjectizer, и для определения имени состояния агента нужно воспользоваться API-методом *so_4::api::query_agent_state*.

3.1.5 Наследование агентов

Для наследования классов агентов необходимо использовать наследование на уровне C++ классов агентов:

```
// Класс агента-элемента электрической цепи.
class a_electric_item_t
  :   public so_4::rt::agent_t
  { ... };
```

```
// Класс агента-лампочки.
class a_lamp_t
  :  public a_electric_item_t
  { ... };

// Класс агента-выключателя.
class a_switcher_t
  :  public a_electric_item_t
  { ... };

// Класс агента-проводки.
class a_wire_t
  :  public a_electric_item_t
  { ... };
```

На уровне описаний классов агентов для SObjectizer необходимо перечислять все непосредственные базовые типы:

```
SOL4_CLASS_START( a_electric_item_t )
...
SOL4_CLASS_FINISH()

SOL4_CLASS_START( a_lamp_t )
  SOL4_SUPER_CLASS( a_electric_item_t )
  ...
SOL4_CLASS_FINISH()

SOL4_CLASS_START( a_switcher_t )
  SOL4_SUPER_CLASS( a_electric_item_t )
  ...
SOL4_CLASS_FINISH()

SOL4_CLASS_START( a_wire_t )
  SOL4_SUPER_CLASS( a_electric_item_t )
  ...
SOL4_CLASS_FINISH()
```

В описании класса агента для SObjectizer может указываться несколько суперклассов — при использовании множественного наследования. Но в этом случае каждый из базовых типов должен быть виртуально унаследован от *so_4::rt::agent_t*. При только одиночном наследовании не обязательно виртуально наследоваться от *so_4::rt::agent_t*. Это объясняется тем, что в классе *so_4::rt::agent_t* SObjectizer хранит часть нужной ему информации об агенте. И при множественном наследовании эта информация должна быть представлена только в единственном экземпляре (чего можно достичь только виртуальным наследованием).

3.2 Подписка событий

Указание инцидентов и приоритетов событий агентов называется *подпиской* агента. Агент может изменить подписку своих событий в любое время, после того, как агент зарегистрирован в SObjectizer. Но, как правило, подписка агента осуществляется только один раз — при регистрации агента и не меняется до дерегистрации агента. Для того, чтобы было удобно подписывать агента, предназначен чистый виртуальный метод *so_4::rt::agent_t::so_on_subscription()*. Он вызывается SObjectizer у каждого агента в момент регистрации.

3.3 Кооперации агентов

При выполнении подписки агентов существует важное ограничение: подписаться можно только на сообщения уже зарегистрированных агентов. В том числе и на собственные сообщения, т.к. метод *so_on_subscription* вызывается у уже зарегистрированного агента. В некоторых случаях необходимо зарегистрировать сразу нескольких агентов, использующих собщения друг друга в качестве инцидентов собственных событий (т.е. существует циклическая зависимость между агентами).

Например, попробуем представить себе часть системы управления банкоматом. В ней есть агент `a_bank`, управляющий взаимодействием с банком. Есть агент `a_bank_connection`, управляющий каналом связи с банком. Эти два агента должны подписываться на сообщения друг друга. Агент `a_bank` подписывается на сообщения `a_bank_connection` о состоянии линии связи. А агент `a_bank_connection` может подписываться на сообщения агента `a_bank` для их передачи по каналу связи.

Что происходило бы, если бы агенты `a_bank` и `a_bank_connection` регистрировались по очереди:

- регистрируется агент `a_bank`, который не может подписаться на сообщения `a_bank_connection`, т.к. `a_bank_connection` еще не зарегистрирован;
- регистрируется агент `a_bank_connection`, который подписывается на сообщения агента `a_bank`;
- агенту `a_bank` каким-то образом сообщается, что он может подписаться на сообщения агента `a_bank_connection`, и агент `a_bank` делает это.

Для упрощения операций регистрации агентов, имеющих циклические зависимости в инцидентах событий, введено понятие *кооперации* агентов. Кооперация — это совокупность агентов, которые регистрируются и дерегистрируются как единое целое.

С использованием кооперация приведенный выше пример реализуется гораздо проще. Агенты `a_bank` и `a_bank_connection` объединяются в одну кооперацию. Когда кооперация регистрируется, SObjectizer сначала регистрирует всех агентов кооперации, а затем последовательно вызывает у них методы *so_on_subscription*. Т.е., когда вызывается *a_bank.so_on_subscription*, в системе уже есть агент `a_bank_connection`, и на его сообщения можно подписаться. Аналогично и с агентом `a_bank_connection`.

Можно сказать, что если в системе есть несколько агентов, зависящих от сообщений друг друга, то вместе эти агенты образуют нечто большее, чем просто набор агентов, а именно — кооперацию. В этом смысле кооперация в SObjectizer аналогична понятию кооперации в UML.

3.3.1 Родительские и дочерние кооперации

В SObjectizer, начиная с v.4.2.7, появилась поддержка взаимоотношений "родитель-потомок" для коопераций. Для любой кооперации можно назначить одну кооперацию-родителя. Любая кооперация может иметь произвольное количество дочерних кооперации.

Необходимость в таких взаимоотношениях возникла из-за того, что SObjectizer не определяет порядка, в котором кооперации дерегистрируются и уничтожаются. Т.е., если сначала дать команду SObjectizer дерегистрировать кооперацию `A`, затем кооперацию `B`, а затем кооперацию `C`, то не гарантируется, что кооперации будут дерегистрированы именно в этом порядке. Например, может оказаться, что агенты из кооперации `C` выполняют длительные вычисления, и их дерегистрация произойдет только после окончания вычислений. К этому времени кооперации `B` и `A` будут дерегистрированы.

В некоторых случаях очень важен порядок дерегистрации и уничтожения кооперации. Например, если кооперация `A` для выполнения своих действий создала подчиненные кооперации `B` и `C`. Завершение работы кооперации `A` возможно только после того, как `B` и `C` будут дерегистрированы.

Еще одна проблема — это использование несколькими кооперациями общего ресурса, скажем, буфера памяти, в который одна кооперация записывает данные, а вторая — считывает. Если этот буфер создает кооперация `A`, то логично заставить ее уничтожить буфер при свой дерегистрации. Но тогда нужно гарантировать, что к этому моменту кооперации `B` уже нет.

Для решения этих проблем достаточно использования взаимоотношений "родитель-потомок". При дерегистрации родительской кооперации SObjectizer гарантирует, что сначала будут дерегистрированы и уничтожены все ее дочерние кооперации. А перед этим будут дерегистрированы и уничтожены дочерние кооперации дочерних кооперации и т.д.

С помощью взаимоотношений "родитель-потомок" приведенные выше примеры очень легко разрешаются. Так, если кооперация `A` объявляется родительской для кооперации `B` и `C`, то сам SObjectizer гарантирует кооперации `A`, что к окончанию ее работы кооперации `B` и `C` уже будут дерегистрированы. А значит кооперация `A` спокойно может уничтожить буфер, который она использовала совместно с кооперацией `B`.

Глава 4

Состав SObjectizer

Данная глава коротко описывает основные части SObjectizer для того, чтобы пояснить принципы работы SObjectizer.

4.1 SObjectizer Run-Time

Часть SObjectizer, отвечающая за функционирование агентов, условно называется SObjectizer Run-Time. SObjectizer Run-Time занимается регистрацией/дерегистрацией агентов, осуществлением отсылки сообщений, генерации событий, вызовов обработчиков событий и т.д. Для этих целей SObjectizer Run-Time использует *системный словарь*, *диспетчер* и *интерфейс прикладного программирования – SObjectizer API*.

SObjectizer Run-Time может быть либо остановлен, либо запущен. Работа агентов осуществляется только при запущенном SObjectizer Run-Time. Запуск SObjectizer Run-Time производится API-функцией *so_4::api::start()*. Приложение, написанное с использованием SObjectizer, само определяет, когда необходимо запустить SObjectizer Run-Time, а когда остановить.

Как правило, запуск SObjectizer Run-Time осуществляется при старте приложения, а останов SObjectizer Run-Time означает завершение работы приложения. Но могут быть случаи, когда SObjectizer Run-Time необходим для выполнения только некоторых действий приложения (например, для взаимодействия распределенных приложений, написанных с использованием SObjectizer Protocol). В этих случаях SObjectizer Run-Time запускается только для выполнения этих действий, после чего SObjectizer Run-Time останавливается.

4.2 Системный словарь

Системный словарь хранит описания всех классов агентов и всех зарегистрированных агентов (имя текущего состояния, списки сообщений, событий, информацию о подписке событий) и т.д.

Информация в системный словарь заносится при выполнении кода, скрытого макросами описания классов агентов для SObjectizer, и при регистрации кооперации.

Обращения к системному словарю могут осуществляться одновременно из разных нитей приложения. Поэтому системный словарь синхронизирует все обращения посредством использования специального синхронизирующего объекта (как правило, mutex-а), что потенциально является узким местом.

4.3 Диспетчер

Диспетчер отвечает за вызов обработчиков событий в соответствии с их приоритетами. Т.е. диспетчер предоставляет контекст, на котором будет осуществлен вызов обработчика события. Как правило, для этого диспетчер располагает собственной нитью (thread), запуск которой осуществляется при запуске SObjectizer Run-Time, а останов — при останове SObjectizer Run-Time.

SObjectizer не декларирует, какая дисциплина диспетчеризации событий должна использоваться. Вместо этого SObjectizer определяет интерфейс (в виде абстрактного C++ класса) диспетчера. При старте SObjectizer Run-Time должен быть указан объект, реализующий данный интерфейс, который будет использовать SObjectizer Run-Time для диспетчеризации событий.

В состав SObjectizer входят несколько уже реализованных диспетчеров. Например, простейший диспетчер с одной рабочей нитью. Данный диспетчер обладает только одной нитью, на которой, согласно приоритетам, запускаются обработчики событий всех зарегистрированных агентов. Обработка события не может быть прервана возникновением более высокоприоритетного события, т.к. все заявки на вызов обработчиков ставятся в очередь. Извлечение очередной заявки осуществляется согласно приоритету только после того, как свою работу завершил предыдущий обработчик.

Очевидно, что недостатком диспетчера с одной рабочей нитью является зависимость общей производительности SObjectizer Run-Time от производительности каждого из обработчиков событий. Если какой-либо из обработчиков выполняет длительные операции, то больше подойдет диспетчер с активными объектами. Некоторые агенты специальным образом объявляются активными объектами, и для каждого из них диспетчер создает отдельную нить. На нити активного объекта запускаются только обработчики событий данного агента. Поэтому, если какой-либо обработчик начнет выполнять длительную операцию, то это отразится только на событиях данного агента, а события остальных агентов будут выполняться параллельно на остальных нитях диспетчера.

Схем диспетчеризации может быть множество. SObjectizer специально не ориентирован на какую-то конкретную схему, чтобы SObjectizer можно было настраивать для различных задач. Это возможно, т.к. SObjectizer Run-Time работает с тем диспетчером, который был предоставлен ему приложением. Если среди поставляемых с SObjectizer диспетчеров нет подходящего, его можно создать, используя определяемый SObjectizer интерфейс.

При рассмотрении понятия диспетчера необходимо заострить внимание на нескольких принципиальных моментах.

4.3.1 Обеспечение приоритетности событий

Необходимо четко определить, что происходит при возникновении в системе сообщения, приводящего к генерации нескольких событий с разными приоритетами. Интуитивно хотелось бы, чтобы события с более высоким приоритетом отрабатывали до событий с низкими приоритетами. Но не все диспетчеры способны обеспечить это.

Например, рассмотрим диспетчер с активными объектами. Пусть в результате сообщения `m` сгенерированы события `A.e1` и `B.e0`. Агенты `A` и `B` являются активными объектами. Приоритет события `A.e1` выше приоритета события `B.e0`. Но порядок запуска обработчиков данных событий будет определяться рядом факторов:

- текущей ситуацией с очередью заявок на вызов обработчиков событий каждого из агентов. Может быть в очереди для агента `A` уже стоят несколько более приоритетных событий, а очередь заявок для агента `B` пуста;
- текущей ситуацией с планировщиком операционной системы. Может быть не известно, какой из нитей диспетчера будет выделено процессорное время. Это может зависеть от количества процессоров, от типа планировщика, от приоритетов каждой из нитей в операционной системе, от общей загруженности операционной системы и т.д.;
- операциями, выполняемыми каждым из обработчиков. Если обработчик `A.e1` начал выполнять длительную операцию ввода-вывода, то операционная система может выделить процессорное время нити, обслуживающей обработчик `B.e0`.

Можно разделить необходимость обеспечения приоритетности в рамках одного агента и в рамках всех агентов приложения. Часто достаточно, чтобы диспетчер гарантировал, что более высокоприоритетное событие одного агента будет обработано прежде менее приоритетных событий этого же агента. В каком порядке будут запускаться обработчики событий остальных агентов не важно. Как правило, гарантировать такую приоритетность в диспетчере возможно без существенных накладных расходов (как на создание диспетчера, так и на время работы диспетчера).

Если же нужно обеспечивать приоритетность в рамках всех агентов, то, как правило, это ведет к существенным накладным расходам. Простейшим из подобных диспетчеров является диспетчер с одной рабочей нитью. Но в этом случае все события обрабатываются строго последовательно, без какой-либо возможности распараллеливания обработчиков различных событий. А написание диспетчера, который бы позволял распараллеливать вызовы обработчиков событий, но при этом гарантировал бы эффективный механизм обеспечения приоритетности, является нетривиальной задачей.

Поэтому при выборе диспетчера для конкретного приложения необходимо внимательно отнестись к вопросу обеспечения приоритетности вызова обработчиков событий. А также к вопросу накладных расходов, которые можно считать допустимыми для выбранного механизма диспетчеризации.

4.3.2 Обеспечение целостности агентов

Задача обеспечения целостности агентов возникает, если диспетчер позволяет запустить два обработчика событий одного агента на разных нитях одновременно. В этом случае методы-обработчики будут работать на контекстах разных нитей, но при этом обращаться к одним и тем же данным.

Такая ситуация может возникнуть, например, в диспетчере, который каждому приоритету событий выделяет отдельную нить. На каждой из нитей запускаются обработчики событий любых агентов, но все события имеют одинаковый приоритет. Тогда на разных нитях могут одновременно работать обработчики различных событий одного и того же агента. А если приложение выполняется в многопроцессорной среде, то эти обработчики действительно могут работать параллельно, каждый на своем процессоре.

Поскольку возможна ситуация, когда несколько методов объекта обращаются к данным этого объекта, то возникает задача обеспечения синхронизации доступа к данным. Поскольку SObjectizer вообще не определяет дисциплины диспетчеризации, то SObjectizer не может и гарантировать подобную синхронизацию.

Обеспечением синхронизации может заниматься как сам объект-агент, так и диспетчер. При этом не всегда очевидно, какой способ окажется более эффективным или вообще реальным: иметь ли синхронизирующий объект в каждом из агентов или синхронизировать доступ к агенту на уровне диспетчера.

Более того, возможны ситуации, когда обеспечение подобной синхронизации вообще недопустимо. Например, в критически важных системах реального времени. Когда возникновение высокоприоритетного события должно прерывать выполнение обработчиков остальных низкоприоритетных событий. Достичь этого можно при помощи диспетчера, выделяющего каждому приоритету отдельную нить и назначающего каждой нити соответствующий приоритет операционной системы. При этом количество приоритетов, доступных событиям агентов, жестко ограничивается (например, допустимым количеством приоритетов нитей в операционной системе). Также необходима операционная система, поддерживающая действительно вытесняющую многозадачность.

В таком приложении возникновение высокоприоритетного события свидетельствует о критической ситуации в реальном мире. И ничто не должно задерживать начало обработки этой критической ситуации. В том числе и синхронизация с обработкой уже произошедших низкоприоритетных событий. Исходя из этого, состав, назначение и принципы работы агентов должны проектироваться так, чтобы изменение общих данных агента в обработчике высокоприоритетного события не приводило к катастрофическим последствиям.

4.4 Интерфейс прикладного программирования

Поскольку SObjectizer является библиотекой C++ классов, было бы правильно говорить, что вся открытая часть этой библиотеки является интерфейсом прикладного программирования SObjectizer. Тем не менее, содержимое библиотеки SObjectizer разбито на несколько пространств имен, одно из которых называется *so_4::api*. Содержимое именно этого пространства имен и подразумевается под SObjectizer API.

SObjectizer API включает в себя следующие функции:

deregister_coop – дерегистрация кооперации;

make_global_agent – регистрация т.н. глобального агента (относится к возможностям SObjectizer Protocol);

query_agent_state – определение имени текущего состояния агента;

register_coop – регистрация кооперации;

send_msg – отсылка сообщения;

shutdown – принудительный останов SObjectizer Run-Time;

start – запуск SObjectizer Run-Time;

subscribe_event – подписка события агента.

Данные функции выделены в отдельную категорию, поскольку они относятся не только к агентам, но и ко всему SObjectizer Run-Time. Для реализации агентов необходимо знание, например, таких классов, как *so_4::rt::agent_t* и

so_4::rt::event_data_t. Но для выполнения многих действий, включая управление реализованным на основе SObjectizer приложением, достаточно использования только функций SObjectizer API.

Например, можно представить себе GUI-приложение, написанное с использованием MFC. Часть этого приложения будет использовать SObjectizer и реализовывать прикладную логику работы. Другая часть приложения отвечает только за интерфейс пользователя и управляет прикладной логикой — т.е. отсылает сообщения в SObjectizer. Именно для этой части и предназначен SObjectizer API.

4.5 Принцип работы SObjectizer

Выполнение практически всех функций из SObjectizer API требует обращения к системному словарю. Но доступ к системному словарю синхронизируется посредством mutex-подобного синхронизирующего объекта. Поэтому при каждом выполнении функции SObjectizer API системный словарь на некоторое время блокируется. Это означает, что обращения к функциям SObjectizer API из других нитей так же окажутся заблокированными. Ниже описывается, как именно это происходит в различных случаях.

4.5.1 Обработка описания класса агента

За макросами описания класса агента для SObjectizer (например, *SOL4_CLASS_START*, *SOL4_CLASS_FINISH*, *SOL4_MSG_START*, *SOL4_MSG_FINISH*, ...) скрывается код по регистрации/дерегистрации описания класса агента в системном словаре.

Данный код представляет из себя набор глобальных переменных. Конструкторы этих переменных регистрируют описания класса в системном словаре. Деструкторы — изымают описания классов из словаря. При каждой из этих операций системный словарь блокируется.

То, что описания классов агентов в системном словаре формируются при помощи глобальных переменных, нужно учитывать при работе с динамически-загружаемыми библиотеками. Если в DLL находятся описания классов агентов, и данная DLL линкуется к исполнимому модулю посредством библиотеки импорта, то классы агентов будут автоматически регистрироваться при загрузке приложения (т.е. когда SObjectizer Run-Time еще не запущен) и дерегистрироваться при выгрузке приложения (т.е. когда SObjectizer Run-Time уже остановлен).

Но, если DLL с описаниями классов агентов загружается вручную при запущенном SObjectizer Run-Time, то описания классов агентов попадают в уже использующийся системный словарь. Это не является проблемой — после успешной загрузки DLL можно регистрировать агентов этих классов. Проблемы могут возникнуть при выгрузке DLL с описаниями классов агентов при запущенном SObjectizer Run-Time — если останутся зарегистрированными агенты данных классов. Получится, что системный словарь уничтожит описания классов у себя, но это не будет отражено во внутренних структурах работающих агентов. И при первой попытке обращения к внутренним описаниям агента со стороны SObjectizer, скорее всего, произойдет крах приложения (из-за доступа по неверному указателю).

Поэтому нужно иметь в виду, что во время работы SObjectizer Run-Time можно свободно загружать и выгружать DLL с описаниями классов агентов, но только, если в этот момент нет зарегистрированных агентов данных классов.

4.5.2 Регистрация кооперации

При регистрации кооперации выполняются следующие действия:

1. Блокируется системный словарь.
2. Проверяется уникальность имени кооперации.
3. Проверяется уникальность имен всех агентов кооперации. Проверяется наличие описаний классов всех агентов кооперации.
4. Проверяется имя родительской кооперации. Если имя родительской кооперации задано, то родительская кооперация должна быть уже зарегистрирована.
5. Все агенты кооперации и сама кооперация регистрируются в системном словаре.
6. Системный словарь разблокируется.
7. У всех агентов кооперации последовательно вызывается метод *so_on_subscription* для осуществления первоначальной подписки.
8. Всем агентам кооперации рассылается специальное сообщение, означающее, что агент успешно зарегистрирован в системе.

То, что подписка агентов осуществляется при незаблокированном системном ядре, означает, что внутри *so_on_subscription* агенты могут обращаться к любым функциям SObjectizer API.

Осуществление подписки агентов при незаблокированном ядре может привести к интересным эффектам. Например, если в кооперации есть агенты `A` и `B`, и подписка для агента `A` уже завершилась, то агент `A` может начать обрабатывать свои события, не дожидаясь, пока будет подписан агент `B`. Если в такой ситуации агент `A` отправит сообщение агенту `B`, то это сообщение до адресата не дойдет, т.к. агент `B` еще не подписан.

4.5.3 Дерегистрация кооперации

Дерегистрация кооперации является более сложным процессом, чем регистрация. Дело в том, что в момент обращения к функции *so_4::api::deregister_coop()*, агенты, входящие в кооперацию, могут обрабатывать свои события, сообщения этих агентов, могут являться инцидентами событий других агентов и т.д. Поэтому, в отличие от регистрации кооперации, возврат из *so_4::api::deregister_coop()* еще не означает, что кооперация дерегистрирована.

При дерегистрации кооперации выполняются следующие действия:

1. Блокируется системный словарь.
2. Кооперация и все входящие в ее состав агенты помечаются как дерегистрируемые. Пока агент считается дерегистрируемым нельзя отправлять или подписываться на его сообщения. Но также нельзя зарегистрировать другого агента с таким же именем.
3. Все дочерние кооперации и входящие в них агенты, а также дочерние кооперации дочерних кооперация и их агенты и т.д. объявляются дерегистрируемыми.

4. Системный словарь разблокируется.

5. SObjectizer Run-Time с некоторым темпом просматривает все дерегистрируемые кооперации. Если оказывается, что ни один из агентов кооперации никаким образом больше не используется, и нет ни одной дочерней кооперации, то кооперация окончательно дерегистрируется и изымается из системного словаря. Просмотр, проверка и окончательная дерегистрация происходят при заблокированном системном словаре.

6. У каждого агента полностью дерегистрированной кооперации вызывается виртуальный метод *so_on_deregistration*, который означает, что агент полностью изъят из SObjectizer Run-Time.

7. О каждой из полностью дерегистрированных кооперациий широковещательно рассылается специальное сообщение.

Если какой-то агент подписан на сообщения дерегистрируемого агента, то SObjectizer автоматически "отписывает" этого агента во время проведения окончательной дерегистрации кооперации.

То, что возврат из *so_4::api::deregister_coop()* не гарантирует полной дерегистрации кооперации, нужно учитывать при ручной выгрузке DLL во время работы SObjectizer Run-Time. Как показано выше, если не убедиться, что больше нет агентов, принадлежащих реализованным в DLL классам агентов, то можно спровоцировать крах приложения. Поэтому решение о возможности выгрузки DLL нужно принимать после получения сообщения о том, что кооперация полностью дерегистрирована.

4.5.4 Отсылка сообщения

При отсылке сообщения выполняются следующие действия:

1. Блокируется системный словарь.

2. Проверяется наличие отсылаемого сообщения.

3. Разблокируется системный словарь.

4. Если нужно, проверяется корректность экземпляра сообщения.

5. Блокируется системный словарь.

6. Генерируются все события, инцидентами которых является данное сообщение. Для каждого из сгенерированных событий диспетчеру ставится заявка на вызов обработчика данного события.

7. Системный словарь разблокируется.

Важно отметить, что в SObjectizer нет очереди ожидающих диспетчеризации сообщений. Отсылка сообщения есть диспетчеризация всех событий, инцидентами которых является данное сообщение. Но, если из нескольких нитей одновременно будет вызвана функция *so_4::api::send_msg()*, то сама операционная система будет выстраивать эти вызовы в некое подобие очереди на блокировку системного словаря.

Поскольку диспетчер и системный словарь никак не синхронизированы между собой, может оказаться так, что все сгенерированные сообщением события будут обработаны еще до возврата из функции *so_4::api::send_msg()*.

4.5.5 Смена состояния агента

Смена состояния агента осуществляется следующим образом:

1. Блокируется системный словарь.
2. Проверяется возможность перехода в новое состояние.
3. Агент переводится в новое состояние.
4. Разблокируется системный словарь.
5. Если заданы, вызываются обработчики выхода из старого состояния.
6. Если заданы, вызываются обработчики входа в новое состояние.

Данный механизм смены состояния агента означает:

- что сама смена состояния выполняется атомарно;
- что обработчики выхода/входа запускаются уже тогда, когда агент находится в новом состоянии;
- обработчики выхода/входа могут обращаться к любым функциям SObjectizer API.

Важно, что изменить состояние может только сам агент и только в обработчике любого из своих событий и только во время обработки события (т.е. тогда, когда обработчик запущен диспетчером на одной из своих нитей).

4.6 Normal- и insend-события

Начиная с версии 4.2.7, в SObjectizer события делятся на два типа: *normal*-события и *insend*-события. От типа события зависит диспетчер, который SObjectizer Run-Time будет использовать для запуска обработчика события.

При диспетчеризации normal-событий все экземпляры событий передаются диспетчеру, который был указан при старте SObjectizer. А этот диспетчер определяет, на какой нити обработчик события будет запущен. Как правило, в этом случае экземпляр события ставится в очередь заявок на одну из рабочих нитей диспетчера. Время, когда normal-событие будет обработано, зависит от множества факторов: типа диспетчера, количества заявок в очереди, приоритета рабочей нити, общей загруженности операционной системы. В результате, событие может быть обработано и до возврата из *send_msg*, и через час после вызова *send_msg*. Для многих задач это вполне приемлемо.

Но бывают случаи, когда необходимо выполнить какие-то действия до возврата из *send_msg*. Например, какой-то агент сначала информирует о своем появлении в системе, а затем о своем текущем состоянии. Тот, кто заинтересован в данном агенте, должен гарантировано успеть подписаться на сообщение о состоянии агента. Но сделать это можно только внутри вызова *send_msg* с сообщением о появлении нового агента. С помощью normal-событий это сделать не получится. В таком случае полезными оказываются insend-события.

Для insend-события SObjectizer гарантирует, что:

- запуск обработчика осуществляется на контексте нити, вызвавшей *send_msg*;
- обработчик отрабатывает до возврата из функции *send_msg*.

Достигается это за счет того, что при отсылке сообщения все normal-события передаются на диспетчеризацию назначенному диспетчеру. Но insend-события в диспетчер не попадают. Вместо этого их обработчики запускаются прямо внутри *send_msg* согласно своих приоритетов.

Обработчики для normal- и insend-событий не различаются. Т.е. один и тот же обработчик может использоваться как для insend-, так и для normal-события. Различие состоит только в способе подписки insend-события.

При использовании insend-событий следует обращать внимание на несколько важных особенностей, речь о которых пойдет ниже.

4.6.1 Insend-события и многопоточность

В случае normal-событий за выбор нити, на которой осуществляется запуск обработчика события, отвечает диспетчер. Ряд диспетчеров берут на себя решение задачи по обеспечению целостности агентов в многопоточной среде. Например, штатные диспетчеры SObjectizer с одной рабочей нитью, с активными объектами и активными группами, диспетчеры главной нити приложения для Windows и Qt гарантируют, что все события агента запускаются только на одной нити и только последовательно с учетом приоритета.

В случае insend-события агент может быть уверен в том, что о его целостности **никто не заботится**. Какой бы диспетчер не использовался, на скольких бы нитях агент сейчас не работал, все равно insend-событие будет запущено на контексте нити, вызвавшей *send_msg*. Поэтому агент сам должен обеспечивать собственную целостность. В частности, защищаться с помощью mutex-а, если это необходимо.

Особое внимание защите с помощью mutex-а следует уделить, если возможны вложенные вызовы insend-событий, т.е., если из insend-события агента отсылается сообщение, которое приводит к вложенному вызову еще одного insend-события этого агента. В случае использования нерекурсивных mutex-ов это приведет к блокировке приложения.

4.6.2 Insend-события и отложенные сообщения

Если инцидент insend-события отсылается как отложенное или периодическое сообщение, то insend-событие запускается на обработку как normal-событие. Связано это с тем, что обеспечение точного отсчета времени для отложенных и периодических сообщений выполняется на контексте специальной таймерной нити диспетчера, на которой недопустим запуск посторонних задач, таких, как обработчики insend-событий.

4.6.3 Конфликты insend- и normal-событий

Если при диспетчеризации сообщения SObjectizer определяет, что у одного агента на это сообщение одновременно подписаны normal- и insend-события, то SObjectizer преобразует insend-событие в normal-событие и передает оба события на обработку диспетчеру. А уже диспетчер определяет, какое из событий может быть обработано в соответствии с текущим состоянием агента. Такое поведение выбрано для того, чтобы

диспетчер мог сделать предположение о предсказуемости приложения[1].

4.6.4 Insend-события и синхронность

Insend-события не предназначены для организации синхронного взаимодействия между агентами. Главная задача insend-событий — это предоставление возможности подписчикам сообщения сделать что-то до того, как отправитель сообщения продолжит свою работу.

Insend-события не позволяют возвращать какие-либо значения отправителю сообщения, т.к. отправитель сообщения при широковещательной рассылке может даже не знать обо всех своих подписчиках. Это делает передачу значений между insend-обработчиком и отправителем сообщения неочевидной. Можно, например, передавать в сообщении указатель на приемник ответа. Но как гарантировать, что обработчик будет только один? И как гарантировать, что обработчик будет insend-событием?

Insend-события предназначены для подписчика, а не для отправителя сообщения. Отправитель даже не знает, кто и как подписан на отсылаемое им сообщение. Поэтому попытка реализовать с помощью insend-событий синхронное взаимодействие между несколькими агентами с передачей возвращаемого значения в обратном направлении, скорее всего, приведет к некрасивому и несопровождаемому решению.

[1] Подробнее вопрос предсказуемости рассматривается при обсуждении примера chstate (см. 12 на стр. 102).

Глава 5

SObjectizer Protocol

5.1 SObjectizer и распределенные приложения

Отличительной чертой построенных на SObjectizer приложений является то, что взаимодействие между агентами происходит посредством асинхронного обмена сообщениями. Доставкой сообщений занимается SObjectizer. Поэтому агент при отправке сообщения не знает, где физически расположен агент-получатель сообщения. Также, получая сообщение, агент не знает, где расположен агент-отправитель, и существует ли агент-отправитель к этому времени. Эта особенность SObjectizer позволяет создавать распределенные приложения, размещая агентов в различных исполнимых модулях и на различных компьютерах.

Для того, чтобы агенты, находящиеся в разных исполнимых модулях, могли обмениваться сообщениями, необходим транспорт, осуществляющий передачу сообщений между модулями. Таким транспортом является SObjectizer Protocol, сокращенно *SOP*.

SOP является прикладным протоколом, реализуемым SObjectizer над конкретными физическими протоколами передачи данных (например, TCP или SSL). Он отвечает за преобразование отсылаемого сообщения в некий промежуточный формат и формирование полученного сообщения из данного формата. Такое преобразование осуществляется прозрачно для приложения — агенты по-прежнему оперируют понятиями сообщений, не особенно заботясь о том, каким образом осуществляется доставка сообщений.

Естественно, что в действительности для написания распределенного приложения необходимо учитывать особенности SOP. Это касается как проектирования, так и реализации агентов приложения.

Сообщения не всех агентов могут быть переданы из одного модуля в другой. Передаются только сообщения т.н. *глобальных агентов*. Поэтому при проектировании распределенного приложения сначала выделяются глобальные агенты, сообщения которых будут передавать информацию из одного модуля в другой. Остальные агенты приложения реализуются так, чтобы обмениваться информацией друг с другом посредством сообщений глобальных агентов.

Поскольку SOP является только протоколом преобразования данных, в приложении необходимо позаботиться о транспорте. SObjectizer предоставляет готовые средства, обеспечивающие транспорт SOP-пакетов по некоторым протоколам передачи данных (например, TCP/IP). Некоторые другие протоколы (например, SSL/TLS поверх TCP/IP) доступны в виде дополнительных библиотек.

Исходя из требующегося приложению транспорта данных выбираются необходимые

средства обеспечения связи между модулями приложения. После чего, с учетом особенностей выбранного способа коммуникации, проектируется и реализуется поддержка взаимодействия между модулями.

Например, при использовании TCP/IP необходимо учитывать, что должны быть модули, создающие серверные сокеты. И должны быть модули, создающие клиентские сокеты, для подключения к серверным сокетам. Это может оказать влияние на всю архитектуру приложения. Так, могут быть выделены серверные модули, владеющие серверными сокетами и обслуживающие подключающихся к ним клиентов, при этом два серверных модуля не соединяются друг с другом. Либо каждый модуль владеет двумя типами сокетов — серверным и клиентским, и каждый модуль приложения устанавливает соединение с остальными модулями.

5.2 Глобальные агенты

Взаимодействие между агентами в SObjectizer осуществляется асинхронными сообщениями. Из-за этого даже в рамках одного исполнимого модуля сложно рассчитывать на существование того или иного агента в конкретный момент времени (если только эти агенты не входят в одну кооперацию). Действительно, можно подписаться на сообщения некоторого агента, но он тут же может быть дерегистрирован, и SObjectizer автоматически уничтожит только что созданную подписку. Или агенту отправляется два сообщения, но во время отправки первого сообщения агент-получатель еще может быть не подписан на него.

Еще более сложная ситуация складывается, если агенты работают в разных модулях, на разных территориально удаленных друг от друга компьютерах. Время на доставку пакета данных с одного компьютера на другой может превышать время жизни агента, отправившего сообщение. В таких условиях возникает вопрос о том, как же агенты, находящиеся в одном модуле, будут подписываться на сообщения агентов из другого модуля.

С учетом возможных накладных расходов в различных видах транспорта использование общего системного словаря для всех модулей невозможно. Сама по себе задача поддержки в каждом модуле системного словаря, в котором отражается ситуация в остальных модулях приложения, очень сложна. Требуется синхронизировать изменения словаря сразу со всеми модулями, учитывая возможные разрывы связи. А также пределы пропускной способности каналов данных. Например, если какой-то модуль регистрирует и дерегистрирует агентов сотнями в секунду, то рассылка всей информации для синхронизации системных словарей с таким темпом может быть физически невозможна.

Более того, не обо всех агентах конкретного модуля нужно знать остальным модулям. Как правило, подавляющее большинство агентов модуля занимается специфической задачей данного модуля. И им не только не нужно сообщать о себе внешнему миру, но и получать информацию от других модулей. На самом деле, информационные потоки между модулями четко выделяются из общей массы всех сообщений приложения. Поэтому возможно выделить несколько агентов, сообщения которых и будут переносить информацию между модулями приложения. Именно такие агенты и являются глобальными агентами в SObjectizer.

Глобальный агент в SObjectizer отличается от обычного агента. Глобальный агент служит только для владения сообщениями. Он регистрируется специальным образом.

Он не может заниматься обработкой событий. Он не может быть дерегистрирован. Он просто предоставляет свои сообщения в распоряжение всех желающих. Обычно глобальный агент регистрируется при старте модуля и существует все время работы модуля.

Для взаимодействия модулей приложения в каждом модуле нужно зарегистрировать одного и того же глобального агента. Т.е. этот агент как бы существует в приложении безотносительно к границам модулей — является глобальным агентом. Локальные агенты в каждом из модулей подписываются на его сообщения.

Когда в модуле возникает сообщение глобального агента, оно не только обрабатывается локальными агентами данного модуля, но и автоматически ретранслируется всем модулям, с которыми в данный момент есть соединение. И в каждом из получивших сообщение модулях оно обрабатывается так же, как и обычные сообщения.

5.3 Транспортные агенты и агент-коммуникатор

SOP спроектирован как прикладной протокол, который можно было бы реализовать поверх различных транспортных протоколов, с помощью различных средств межпроцессовых коммуникаций (IPC). Например, потоковых сокетов TCP, программных каналов (pipes), разделяемой памяти и т.д. Но для каждого средства IPC существуют свои особенности, которые необходимо учитывать. Поэтому поддержка SOP-коммуникаций в SObjectizer реализована агентами двух категорий:

- **транспортные** агенты, отвечающие за работу с конкретным механизмом IPC. Эти агенты занимаются только получением/передачей SOP-пакетов. Созданием транспортных агентов должно заниматься приложение, т.к. средства коммуникации выбираются именно на уровне приложения;
- **агент-коммуникатор**, который отвечает за формирование/разбор SOP-пакетов. Этот агент является частью SObjectizer, и его не нужно создавать. Он создается автоматически при старте SObjectizer Run-Time.

При появлении глобального агента агент-коммуникатор подписывается на сообщения этого агента. При возникновении сообщения генерируется событие агента-коммуникатора. Обработчик этого события преобразует сообщение в SOP-пакет. SOP-пакет рассылается как широковещательное сообщение агента-коммуникатора. Заинтересованные в данном SOP-пакете транспортные агенты обрабатывают сообщение агента-коммуникатора и передают SOP-пакет через контролируемый ими канал связи.

Когда транспортный агент получает из канала связи SOP-пакет, он отсылает специальное сообщение агенту-коммуникатору. Получив это сообщение агент-коммуникатор распаковывает полученный SOP-пакет и, если возможно, рассылает содержащиеся в нем сообщения глобальных агентов.

Особенностью данной схемы является то, что при возникновении сообщения глобального агента осуществляются обычные действия по генерации всех подписанных на него событий. Т.е. агент-коммуникатор является только одним из многих агентов, которые могут быть подписаны на сообщения глобального агента. Например, если внутри модуля `m` на сообщения глобального агента подписаны агенты `Am` и `Bm`, то при возникновении этого сообщения обработчики событий агентов `Am` и `Bm` будут запущены таким же

образом, как и обработчик агента-коммуникатора. Аналогичная картина может быть и в модуле `n`, где на это же сообщение подписаны агенты `Cn` и `Dn`. Если же со временем объединить модули `m` и `n`, то на доставку сообщения глобального агента всем подписавшимся на него агентам не потребуется дополнительных накладных расходов. Т.е. для доставки сообщения глобального агента от `Am` к `Cn` в этом случае не будет задействован коммуникационный уровень SObjectizer.

5.4 Идентификация коммуникационных каналов

Транспортные агенты дают каждому существующему соединению уникальный идентификатор, называемый *идентификатором коммуникационного канала*. Каждый экземпляр сообщения в SObjectizer помечается идентификатором коммуникационного канала, из которого этот экземпляр был получен. Если сообщение было сформировано в самом модуле, а не получено извне, то сообщение помечается специальным значением идентификатора — *localhost*.

Указание для каждого экземпляра сообщения идентификатора канала-источника сообщения необходимо, прежде всего, для корректной работы агента-коммуникатора. Агент-коммуникатор анализирует идентификатор канала-источника сообщения и, если идентификатор равен localhost, распространяет это сообщение через SOP. Если бы такой проверки не было, возникновение одного экземпляра сообщения глобального агента привело бы к тому, что агенты-коммуникаторы в разных модулях занимались бы пересылкой данного экземпляра друг другу.

Но идентификатор канала-источника сообщения можно использовать и по аналогии с широковещательной и целенаправленной рассылкой сообщений. Для этого идентификатор канала-источника сделан доступным для агента-получателя сообщения посредством метода *`event_data_t::channel()`*.

5.4.1 Broadcast-взаимодействие модулей

По умолчанию отсылка сообщения глобального агента подразумевает рассылку данного сообщения во все существующие в данный момент каналы связи. Это напоминает широковещательную рассылку сообщения агента. Только вместо агентов получателями являются модули приложения.

Broadcast-взаимодействие модулей удобно использовать, когда модули обмениваются сообщениями-уведомлениями. Например, датчик температуры может рассылать всему миру сообщение о повышении температуры. И ему не важно, кто и где на это сообщение среагирует. Агент, получивший данное сообщение, может принять решение о том, объявлять ли пожарную тревогу или нет. Сообщение об объявлении пожарной тревоги так же рассылается всему миру. И среагировать на него могут различные агенты в различных модулях. Один может включить сирену, второй — систему пожаротушения, третий — связаться со службами экстренного вызова.

5.4.2 Peer-to-Peer-взаимодействие модулей

В некоторых случаях нужно точно указать, в какой модуль требуется доставить экземпляр сообщения. Например, есть клиент-серверная система управления доставкой грузов. В ней существует один выделенный сервер, обладающий всей информацией о текущей ситуации. А также некоторое количество клиентских терминалов, с которых

осуществляется управление доставкой грузов получателю. Обычный сценарий работы предполагает, что с клиентского терминала делается запрос на сервер для получения списка невыполненных заказов. После выбора заказа клиентский терминал используется только для работы с этим заказом.

Для данной системы модули, реализующие клиентские терминалы, используют broadcast-взаимодействие с центральным серверным модулем. Т.е. в клиентском модуле просто отсылается сообщение глобального агента без заботы о том, какой именно идентификатор у текущего соединения с сервером. Это сообщение доходит до серверного модуля и обрабатывается неким агентом, который должен отослать соответствующее сообщение со списком заказов в ответ. Но эта отправка должна быть осуществлена только тому модулю, который прислал запрос. Для этого ответное сообщение отсылается в тот канал, из которого пришло сообщение-запрос.

При peer-to-peer взаимодействии сообщение сразу направляется в канал, идентификатор которого был указан при отсылке. Для самого модуля, в котором этот экземпляр сообщения был порожден, сообщения как бы не существует. Ни один из агентов данного модуля (за исключением агента-коммуникатора) не сможет получить данный экземпляр для обработки.

5.5 SOP и сообщения агентов

SOP является протоколом преобразования сообщения агентов в некое транспортабельное представление и обратно. Поэтому SObjectizer должен знать тип, реализующий сообщение, и набор полей данного типа, которые должны обрабатываться посредством SOP. Информация о типе и полях сообщения передается SObjectizer описанием сообщения в описании класса агента для SObjectizer:

```
SOL4_CLASS_START( a_some_class_t )
  SOL4_MSG_START( msg_empty, a_some_class_t::msg_empty )
  SOL4_MSG_FINISH()

  SOL4_MSG_START( msg_full, a_some_class_t::msg_full )
    SOL4_MSG_FIELD( m_name )
    SOL4_MSG_FIELD_ARRAY( m_bin_data )
  SOL4_MSG_FINISH()
  ...
SOL4_CLASS_FINISH()
```

Подобное описание сообщает SObjectizer, что агенты типа `a_some_class_t` обладают двумя сообщениями: `msg_empty` и `msg_full`. Первое реализуется C++ типом *`a_some_class_t::msg_empty`* и не имеет полей, доступных для SOP. Второе реализуется C++ типом *`a_some_class_t::msg_full`* и имеет два доступных для SOP поля: *`m_name`* и *`m_bin_data`*.

Имея такое описание SObjectizer посредством SOP способен передать информацию о возникновении этих сообщений из одного модуля в другой. Причем для сообщения `msg_empty` будет передано только имя сообщения. На принимающей стороне будет создан экземпляр сообщения `msg_empty`, в который SObjectizer ничего помещать не будет. При отправке сообщения `msg_full` SObjectizer, кроме имени сообщения, передаст

и значения полей *m_name* и *m_bin_data*. На принимающей стороне эти значения будут помещены в созданный экземпляр сообщения `msg_full`.

SOP накладывает ряд ограничений на поля сообщений. Например:

- доступные для SOP поля должны быть public-полями C++ типов;
- доступные для SOP поля должны относиться либо к скалярным C++ типам (вроде char, short, int и т.д.), либо к некоторым более сложным типам (вроде std::string).
- SOP не может работать с полями-указателями.

Практика использования SObjectizer показала, что даже с такими ограничениями SOP позволяет создавать работоспособные распределенные приложения. Кроме того, SOP поддерживает типы, сериализуемые средствами *ObjESSty*[1], что позволяет не обращать внимания на приведенные выше ограничения.

5.6 Воздействие на приложение через SOP

Агент-коммуникатор ориентирован на сообщения глобальных агентов только для отсылки сообщений. Получив же входящий SOP-пакет агенту-коммуникатору все равно, находятся ли в SOP-пакете сообщения глобальных или локальных агентов. Агент-коммуникатор просто проверяет наличие такого сообщения, после чего пытается создать и заполнить экземпляр сообщения. Если это удалось, созданный подобным образом экземпляр отправляется на диспетчеризацию.

Т.е., если подключиться к входящему каналу какого-нибудь модуля и соответствующим образом формировать SOP-пакеты, то можно отправлять в этот модуль сообщения не только глобальных, но и локальных агентов. Технически это возможно благодаря специальным утилитам, которые подключаются к работающему модулю и позволяют отправлять в него произвольные SOP-пакеты. В состав SObjectizer входит утилита `so_send_stdin` для TCP/IP транспорта.

Практика использования SObjectizer выявила два случая, когда подобное воздействие на приложение через SOP является удобным и необходимым:

Отладка. Зачастую при отладке сложно воспроизвести или имитировать какую-то сложную ситуацию. Например, когда часть приложения еще не написана. Или когда нужно определить поведение приложения при условиях, не воспроизводимых в действительности (например, авариях, критических сбоях и т.д.). В этом случае можно отсылать в приложение сообщения, которые бы возникали в реальной ситуации.

Управление черным ящиком. Иногда приложение представляет из себя “черный ящик”, реализующий конкретную задачу. Он не имеет пользовательского интерфейса, но должен каким-то образом управляться. Например, должна быть возможность изменить режим работы или завершить работу совсем. Для этого в приложении можно не создавать глобальных агентов, но открыть входной канал. В этот канал при помощи специальных утилит (`so_send_stdin`, например) можно отсылать сообщения, которые будут восприниматься “черным ящиком”.

[1] OBJect Entity Serialization & StabiliTY – отдельный проект для автоматической сериализации/десериализации сложных структур данных.

Часть II

SObjectizer в примерах: шаг за шагом

Глава 6

Введение

Данная часть является учебником по программированию с использованием SObjectizer. Материал подается в виде подробного разбора примеров, входящих в состав SObjectizer. Каждому примеру посвящена отдельная глава. При освещении очередного примера внимание заостряется только на тех характерных особенностях SObjectizer, для демонстрации которых был создан пример. Поэтому при первом прочтении рекомендуется читать главы последовательно, т.к. они представлены в порядке усложнения материала:

Глава 7. Пример hello_world. Дает общее представление о программировании с использованием SObjectizer.

Глава 8. Пример hello_all. Показывает различные аспекты работы с сообщениями. В первую очередь — целенаправленную и широковещательную рассылку сообщений.

Глава 9. Пример hello_delay. Демонстрирует применение отложенных сообщений.

Глава 10. Пример hello_periodic. Демонстрирует использование периодических сообщений.

Глава 11. Пример dyn_reg. Демонстрирует использование динамических кооперации.

Глава 12. Пример chstate. Демонстрирует различные аспекты смены состояний агента.

Глава 13. Пример inheritance. Демонстрирует наследование классов агентов.

Глава 14. Пример subscr_hook. Демонстрирует использование т.н. *hook-ов подписки* и диспетчера с активными агентами.

Глава 15. Пример filter. Демонстрирует возможность построения распределенного приложения с использованием коммуникационных возможностей SObjectizer.

Глава 16. Пример high_traffic. Демонстрирует возможность передачи данных в распределенных приложениях при помощи сообщений глобальных агентов.

Глава 17. Пример raw_channel. Демонстрирует использование коммуникационных средств SObjectizer для работы с т.н. *raw*-соединениями.

Глава 18. Пример parent_insend. Демонстрирует использование родственных отношений между кооперациями и применение *insend*-событий.

Глава 19. Пример dyn_coop_controlled. Демонстрирует, как можно контролировать время жизни любых динамически созданных объектов при помощи динамической кооперации.

Глава 20. Пример destroyable_traits. Демонстрирует создание собственных свойств (traits) агентов и назначение агенту этих свойств в виде динамически созданных объектов.

Для получения базовых знаний о SObjectizer достаточно знакомства с примерами hello_world (стр. 54), hello_all (стр. 69), hello_delay (стр. 85), hello_periodic (стр. 89), dyn_reg (стр. 93), chstate (стр. 102) — это базис, без которого невозможно пользоваться SObjectizer.

Примеры filter (стр. 130), high_traffic (стр. 158) и raw_channel (стр. 166) знакомят с коммуникационными возможностями SObjectizer. В частности, в этих примерах показывается, как создавать простейшие распределенные приложения на основе SObjectizer. А также раскрываются особенности транспортных агентов SObjectizer, что позволяет использовать их для реализации собственных протоколов.

Примеры inheritance (стр. 116), subscr_hook (стр. 123) и dyn_coop_controlled (стр. 179) относятся к редко используемым, но не менее важным и удобным, возможностям SObjectizer, которые временами серьезно облегчают применение SObjectizer.

Пример parent_insend (стр. 173) затрагивает такую относительно новую возможность SObjectizer, как insend-события. В некоторых случаях применение insend-событий способно не только упростить проектное решение за счет предсказуемости моментов запуска обработчиков сообщений, но также позволяет увеличить производительность, поскольку для запуска обработчика insend-события не требуется прохождения заявки через очереди диспетчера.

Пример destroyable_traits (стр. 184) описывает понятие свойств агентов. Свойства агентов редко используются при решении прикладных задач. Но в некоторых случаях использование свойств агентов очень удобно. Например, с помощью свойств можно создавать мониторинговые инструменты, которые контролируют состояние агента и/или значение какого-либо из атрибутов агента.

Примеры описываются в порядке возрастания их сложности и важности для практического использования SObjectizer. Исключения составляют только примеры dyn_coop_controlled и destroyable_traits, которые описаны последними из-за своей специфичности.

Исходные тексты примеров приведены в приложении А.

Глава 7

Пример hello_world

Пример отображает на стандартный поток ввода сообщение 'Hello, World' и завершает свою работу.

Пример hello_world состоит из одного файла `main.cpp` (стр. 191).

7.1 Разбор файла main.cpp

Исходный код примера начинается с загрузки необходимых заголовочных файлов.

```
#include <so_4/rt/h/rt.hpp>
#include <so_4/api/h/api.hpp>

#include <so_4/timer_thread/simple/h/pub.hpp>
#include <so_4/disp/one_thread/h/pub.hpp>
```

- `so_4/rt/h/rt.hpp`. Содержит описания классов и функций, касающихся агентов и их содержимого. Т.е. описывает классы, являющиеся частью SObjectizer.
- `so_4/api/h/api.hpp`. Описывает функции SObjectizer API.
- `so_4/timer_thread/simple/h/pub.hpp`. Описывает простейшую нить таймера.
- `so_4/disp/one_thread/h/pub.hpp`. Описывает простейший диспетчер с одной рабочей нитью.

Загрузка сразу четырех заголовочных файлов – частный случай для файла `main.cpp`. Обычно загружаются только те заголовочные файлы, которые необходимы прикладному коду. Например, если в C++ файле реализуется код агента, то потребуется загрузить `so_4/rt/h/rt.hpp` и `so_4/api/h/api.hpp`. Если реализуется некий управляющий фрагмент, например, пользовательский интерфейс, то достаточно загрузить только `so_4/api/h/api.hpp`. Описание диспетчера загружать необходимо только в тех местах, где осуществляется запуск SObjectizer.

SObjectizer описывает свое содержимое в нескольких пространствах имен. Самым объемлющим (верхним) из них является пространство имен *so_4*. Остальные пространства имен вложены в пространсто имен *so_4*.

В SObjectizer принято соглашение о том, что каждый каталог, в котором содержатся доступные заголовочные файлы, вводит собственное пространство имен. Например,

so_4 — пространство имен *so_4*; so_4/rt — пространство имен *so_4::rt*; so_4/api – пространство имен *so_4::api* и т.д.

Далее следует C++ описание класса агента:

```
class a_hello_t
  : public so_4::rt::agent_t
  {
    // Псевдоним для базового типа.
    typedef so_4::rt::agent_t base_type_t;
  public :
    a_hello_t()
      :
      // Сразу задаем имя агента.
      base_type_t( "a_hello" )
      {}
      virtual ~a_hello_t()
      {}

    virtual const char *
    so_query_type() const;

    virtual void
    so_on_subscription()
      {
        ...
      }

    // Обработка начала работы агента в системе.
    void
    evt_start()
      {
        ...
      }
  };
```

Класс *a_hello_t* — это класс единственного агента данного примера. Каждый агент должен относиться к классу, производному от *so_4::rt::agent_t*. Класс *so_4::rt::agent_t* должен быть доступным (*public*) базовым классом. Класс *so_4::rt::agent_t* может быть не единственным базовым классом, т.е. класс агента с помощью множественного наследования может быть производен еще от нескольких классов. Важно, чтобы класс *so_4::rt::agent_t* был доступным (*public*) базовым классом!

Базовый класс *so_4::rt::agent_t* содержит набор методов, два из которых являются чистыми виртуальными методами — *so_query_type* и *so_on_subscription*. В данном примере необходимо создать объект (агент) типа *a_hello_t*, т.е. *a_hello_t* не может быть абстрактным, а значит не может иметь чистых виртуальных методов. Поэтому в описании *a_hello_t* методы *so_query_type* и *so_on_subscription* описываются как обычные виртуальные методы.

Метод *evt_start* — это действие, которое будет соответствовать событию evt_start.

Далее следует описание класса агента для SObjectizer:

```
// Описание класса агента для SObjectizer-a.
SOL4_CLASS_START( a_hello_t )

  // Одно событие.
  SOL4_EVENT( evt_start )

  // И одно состояние.
  SOL4_STATE_START( st_normal )
    // С одним событием.
    SOL4_STATE_EVENT( evt_start )
  SOL4_STATE_FINISH()

SOL4_CLASS_FINISH()
```

Макрос *SOL4_CLASS_START* начинает описание класса агентов. Параметром макроса является имя C++ класса. Это имя должно быть именем реального C++ класса. Кроме того, имя должно быть достаточным видимым C++ именем в точке применения *SOL4_CLASS_START*. Т.е., если C++ класс описан в каком-либо *namespace*, то *SOL4_CLASS_START*:

- должен быть указан в том же namespace, что и C++ класс:

```
namespace some_namespace
  {
    class some_class_t
      : public so_4::rt::agent_t
      {
        ...
      };

    SOL4_CLASS_START( some_class_t )
    ...
    SOL4_CLASS_FINISH()
  }
```

- должен получать в качестве параметра полное имя C++ класса:

```
namespace some_namespace
  {
    class some_class_t
      :  public so_4::rt::agent_t
      {
        ...
      };
  }
```

```
SOL4_CLASS_START( some_namespace::some_class_t )
    ...
SOL4_CLASS_FINISH()
```

- должен следовать после инструкции using namespace:

```
namespace some_namespace
  {
    class some_class_t
      :  public so_4::rt::agent_t
      {
        ...
      };
  }

using namespace some_namespace;
...
SOL4_CLASS_START( some_class_t )
...
SOL4_CLASS_FINISH()
```

Макросы *SOL4_CLASS_START* и *SOL4_CLASS_FINISH* помещают в системный словарь описание класса агентов. Описание класса агентов сохраняется в системном словаре под тем именем, которое было параметром *SOL4_CLASS_START*. Это означает, что для нормальной работы необходимо, чтобы в SObjectizer использовались только уникальные имена типов. Для небольших проектов достичь этого не трудно. Однако, в больших проектах в нескольких пространствах имен могут оказаться классы агентов с одинаковыми именами:

```
namespace io_subsys
  {
    class a_supervisor_t
      :  public so_4::rt::agent_t
      {
      ...
      };
    SOL4_CLASS_START( a_supervisor_t )
    ...
    SOL4_CLASS_FINISH()
  }

namespace user_subsys
  {
    class a_supervisor_t
      :  public so_4::rt::agent_t
      {
      ...
```

```
    };
  SOL4_CLASS_START( a_supervisor_t )
  ...
  SOL4_CLASS_FINISH()
}
```

Макросы *SOL4_CLASS_START* не различают, в каком пространстве имен они использованы. Поэтому SObjectizer проигнорирует одно из приведенных выше описаний класса `a_supervisor_t`. Для того, чтобы этого не происходило, рекомендуется всегда указывать полное имя C++ класса в *SOL4_CLASS_START*:

```
namespace io_subsys
  {
    class a_supervisor_t
      :  public so_4::rt::agent_t
      {
      ...
      };
    SOL4_CLASS_START( io_subsys::a_supervisor_t )
    ...
    SOL4_CLASS_FINISH()
  }

namespace user_subsys
  {
    class a_supervisor_t
      :  public so_4::rt::agent_t
      {
      ...
      };
    SOL4_CLASS_START( user_subsys::a_supervisor_t )
    ...
    SOL4_CLASS_FINISH()
  }
```

Макросы *SOL4_CLASS_START* и *SOL4_CLASS_FINISH* содержат в себе реализацию виртуального метода *so_query_type* приблизительно такого вида:

```
#define SOL4_CLASS_START(type_name)\
...\
const char *\
type_name::so_query_type() const {\
  return #type_name;\
}\
...
```

Поэтому метод *so_query_type* нужно только задекларировать в описании C++ класса, а его реализация будет автоматически сгенерирована при описании класса агента.

Макрос *SOL4_EVENT(evt_start)* указывает, что агенты класса a_hello_t будут обрабатывать всего одно событие, именуемое evt_start.

В SObjectizer обработчики всех событий должны быть нестатическими (возможно, виртуальными) методами C++ класса. Методы-обработчики, описываемые при помощи макроса *SOL4_EVENT* должны иметь формат:

```
void
evt_handler();
```

или

```
void
evt_handler(
    const so_4::rt::event_data_t & data );
```

В описании класса агента для SObjectizer макросы:

```
SOL4_STATE_START( st_initial )
    SOL4_STATE_EVENT( evt_start )
SOL4_STATE_FINISH()
```

указывают, что у агентов класса a_hello_t есть только одно состояние — st_initial, в котором обрабатывается только одно событие — evt_start.

Конструктор и деструктор класса *a_hello_t* определяются следующим образом:

```
a_hello_t()
  :
  // Сразу задаем имя агента.
  base_type_t( "a_hello" )
  {}
virtual ~a_hello_t()
  {}
```

Конструктор базового класса *so_4::rt::agent_t* имеет один обязательный аргумент: *const std::string & agent_name* — имя агента. Имя агента задается один раз при создании объекта-агента и не может быть изменено в дальнейшем. Имя агента должно быть уникальным и может состоять из любых не пробельных ASCII символов.

В данном примере будет использоваться только один агент, поэтому выбор имени для него не составляет труда. Но в более сложных проектах возможны ситуации, когда в различных подсистемах имена агентов могут совпадать (например, для агентов a_supervisor в пространствах имен *io_subsys* и *user_subsys*). Если таким агентам дать простое имя *"a_suprevisor"*, то SObjectizer Run-Time позволит зарегистрировать только одного из них. Для избежания этого рекомендуется давать агентам полные имена, включающие пространства имен, имена подсистемы и т.д. Например, *"io_subsys::a_supervisor"*, *"user_subsys::a_supervisor"*.

Метод *a_hello_t::so_on_subscription* реализует подписку агента:

```
virtual void
so_on_subscription()
```

```
{
  // Нужно подписать наше единственное событие.
  so_subscribe( "evt_start",
    so_4::rt::sobjectizer_agent_name(),
    "msg_start" );
}
```

Виртуальный метод *so_on_subscription* отвечает за подписку событий агента. В описании класса агента указываются только события агента, но не говорится, какие инциденты будут у каждого события. Объясняется это тем, что агенты одного и того же класса могут быть подписаны на совершенно разные сообщения. Более того, каждый агент может изменять список инцидентов своего события в любой момент. Поэтому, в общем случае, невозможно при описании события в описании класса указать инциденты события.

Однако, работа агента в системе осуществляется именно посредством событий. Поэтому, если в системе появится агент, ни одно из событий которого не подписано на сообщения, то этот агент не сможет работать в рамках SObjectizer Run-Time. Метод *so_on_subscription* предназначен для того, чтобы только что зарегистрированный в системе агент получил возможность подписать свои события и стать полноценным участником прикладной системы. Метод *so_on_subscription* вызывается внутри функции *so_4::api::register_coop* после того, как в словарь системы успешно будут занесены все члены регистрируемой кооперации. Это означает, что SObjectizer Run-Time известен сам агент, у которого вызван метод *so_on_subcription*, и остальные агенты той же кооперации. Т.е. агент может подписаться как на свои собственные сообщения, на сообщения агентов из своей кооперации, так и на сообщения любых других агентов прикладной системы.

В данном примере событие `evt_start` подписывается на одно сообщение - сообщение `a_sobjectizer.msg_start`[1].

Сообщение `a_sobjectizer.msg_start` является специальным сообщением, указывающим агенту, что он зарегистрирован в системе и может начинать свою работу. Это сообщение рассылается всем агентам зарегистрированной кооперации после того, как для каждого из агентов этой кооперации будет вызван метод *so_on_subscription*. В данном примере продемонстрирован универсальный способ определения момента, когда агент может начать свою работу в системе — событие `evt_start`, инцидентом которого является сообщение `a_sobjectizer.msg_start`. Т.е., когда агент будет зарегистрирован, система отошлет ему сообщение `a_sobjectizer.msg_start`. Это приведет к генерации события `evt_start`. Агент находится в состоянии `st_initial`, и событие `evt_start` разрешено для обработки в этом состоянии — следовательно, событие будет обработано вызовом обработчика для события — метода *evt_start*. Т.е. событие `evt_start` происходит и обрабатывается в самом начале работы агента в системе.

Строго говоря, сразу после появления агента в словаре системы у агента вызывается метод *so_on_subcription*. Поэтому вызов этого метода может использоваться как индикатор того, что агент появился в системе и может начать свою работу. Но использовать такой способ определения момента начала работы не рекомендуется, т.к. на этот момент времени остальные агенты регистрируемой кооперации могут быть еще не

[1] Функция *so_4::rt::sobjectizer_agent_name* возвращает имя специального агента `a_sobjectizer`, использование этой функции вместо имени `a_sobjectizer` предпочтительней, т.к. в будущих версиях SObjectizer имя агента `a_sobjectizer` может измениться.

подписаны. Например, если регистрируется кооперация, содержащая агенты `A`, `B` и `C`, и агент `B` в своем методе *so_on_subscription* попробует отправить сообщение агенту `C`, то не гарантируется, что это сообщение будет обработано. Агент `C` в этот момент может быть еще не подписан. Если же агент `B` будет отправлять сообщение агенту `C` в событии `evt_start`, то сообщение будет обработано, т.к. агент `C` уже будет подписан.

Метод *so_subscribe*, осуществляющий подписку, унаследован из базового класса *so_4::rt::agent_t*. Он имет два варианта:

```
void
so_subscribe(
  //! Имя подписываемого события.
  const std::string & event,
  //! Имя агента-владельца инцидента.
  const std::string & owner,
  //! Имя сообщения-инцидента.
  const std::string & msg,
  //! Приоритет события.
  int priority = 0,
  //! Поток для отображения сообщений об ошибках
  //! подписки. Если равен 0, то сообщения об
  //! ошибках не игнорируются.
  std::ostream * err = &std::cerr );
```

который используется для подписки на “чужие” сообщения (т.е. сообщения, которыми владеют другие агенты), и второй вариант:

```
void
so_subscribe(
  //! Имя подписываемого события.
  const std::string & event,
  //! Имя сообщения-инцидента.
  const std::string & msg,
  //! Приоритет события.
  int priority = 0,
  //! Поток для отображения сообщений об ошибках
  //! подписки. Если равен 0, то сообщения об
  //! ошибках не игнорируются.
  std::ostream * err = &std::cerr );
```

который используется для подписки на собственные сообщения.

Последний аргумент методов *so_subscribe* определяет поток, на который могут отображаться сообщения о возникающих во время подписки ошибках. По умолчанию в качестве такого потока используется стандартный поток ошибок.

Единственное событие агента `a_hello` реализуется следующим образом:

```
void
evt_start()
  {
    std::cout << "Hello, world! This is SObjectizer v.4."
```

```
      << std::hex << __SO_4_VERSION__ << std::dec << std::endl;

    // Завершаем работу примера.
    so_4::api::send_msg(
      so_4::rt::sobjectizer_agent_name(),
      "msg_normal_shutdown", 0 );
  }
```

На стандартный поток вывода отображается строка “Hello, World!” и номер версии SObjectizer. Затем посредством функции *so_4::api::send_msg* агенту a_sobjectizer отправляется сообщение msg_normal_shutdown. Получив это сообщение агент a_sobjectizer завершает работу прикладной системы и SObjectizer Run-Time — происходит возврат из функции *so_4::api::start*, вызванной в *main()*.

После описания и реализации класса агента *a_hello_t* следует функция main:

```
int
main()
  {
    // Наш агент.
    a_hello_t a_hello;
    // И кооперация для него.
    so_4::rt::agent_coop_t  a_hello_coop( a_hello );

    // Запускаем SObjectizer Run-Time.
    so_4::ret_code_t rc = so_4::api::start(
      so_4::disp::one_thread::create_disp(
        so_4::timer_thread::simple::create_timer_thread(),
        so_4::auto_destroy_timer ),
      so_4::auto_destroy_disp,
      &a_hello_coop );
    if( rc )
      {
        // Запустить SObjectizer Run-Time не удалось.
        std::cerr << "start: " << rc << std::endl;
      }

    return int( rc );
  }
```

В начале функции main объявляются переменные *a_hello* и *a_hello_coop*. Первая переменная создает экземпляр объекта-агента, а вторая — экземпляр объекта-кооперации агентов. Для *a_hello_coop* используется конструктор с одним параметром. Это означает, что в кооперацию будет входить только один агент, и имя кооперации будет совпадать с именем агента.

Создавать объекты-агенты и объекты-кооперации должен прикладной программист, т.к. SObjectizer Run-Time не занимается созданием агента, а только использованием уже существующих объектов. При этом сам прикладной программист должен заботиться о том, чтобы время жизни объекта-агента превышало время, которое агент

зарегистрирован в системе. В данном случае агент `a_hello` будет существовать только внутри функции main. Поэтому агент и кооперация представлены локальными переменными.

В данном примере используется тривиальный диспетчер с одной рабочей нитью, который создается при помощи функции *so_4::disp::one_thread::create_disp*. Диспетчеру для своей работы требуется объект нити таймера, который используется для обработки отложенных и периодических сообщений. Объект нити таймера создается при помощи функции *so_4::timer_thread::simple::create_timer_thread*.

Для того, чтобы можно было обеспечить адаптацию SObjectizer к различным задачам, SObjectizer Run-Time написан так, чтобы не знать ничего о внутренностях диспетчера, с которым предстоит работать. Даже диспетчеры, входящие в состав SObjectizer, не пользуются скрытыми деталями реализации SObjectizer Run-Time – применяются только доступные классы и интерфейсы, описанные в `so_4/rt/h/rt.hpp` и пространстве имен *so_4::rt*. Оборотной стороной подобной гибкости SObjectizer стало то, что прикладному программисту самому необходимо заботиться о создании объекта диспетчера, даже, если используется какой-либо из диспетчеров SObjectizer.

Аналогичная ситуация и с нитью таймера. Нить таймера используется для отсылки отложенных сообщений. SObjectizer спроектирован так, что нитью таймера управляет диспетчер. Но в большинстве случаев не выгодно, чтобы диспетчер сам реализовывал нить таймера. По сути, диспетчер и нить таймера являются ортогональными понятиями. Т.е. иногда может протребоваться диспетчер с активными объектами и таймер с низкой точностью отсчета времени (каким является простейший таймер в составе SObjectizer). Иногда требуется тривиальный диспетчер и очень точный таймер. Для упрощения настройки SObjectizer некоторые диспетчеры требуют, чтобы им указывали уже существующий объект таймер. При использовании таких диспетчеров на прикладного программиста ложится задача создания объекта таймера и обеспечения того, чтобы таймер существовал все время, пока в программе живет объект диспетчер.

Как и в случае с агентами, SObjectizer Run-Time не занимается созданием и уничтожением объектов диспетчеров и нитей таймера. Но SObjectizer Run-Time облегчает работу программиста тем, что по желанию программиста может автоматически уничтожить диспетчер и нить таймера, когда они станут не нужными. Достигается это указанием флага *so_4::auto_destroy_timer* при обращении к функции *so_4::disp::one_thread::create_disp* (т.е. диспетчеру предписывается уничтожить нить таймера в своем деструкторе) и указанием флага *so_4::auto_destroy_disp* при обращении к функции *so_4::api::start* (т.е. SObjectizer Run-Time предписывается уничтожить диспетчер после завершения работы).

Важно отметить, что заголовочные файлы `so_4/timer_thread/simple/h/pub.hpp` и `so_4/disp/one_thread/h/pub.hpp` не содержат описаний реальных классов таймерной нити и диспетчера. Они описывают только функции, которые создают соответствующие объекты (в данном случае *so_4::timer_thread::simple::create_timer_thread()* и *so_4::disp::one_thread::create_disp()*). Возвращаемые данными функциями объекты принадлежат классам, производным от абстрактных базовых классов (интерфейсов) *so_4::timer_thread::timer_thread_t* и *so_4::rt::dispatcher_t*. Реальные классы таймерной нити и диспетчера скрыты от прикладного программиста, чтобы обеспечить максимальную независимость прикладных программ от деталей реализации текущей версии SObjectizer.

Функция *so_4::api::start* имеет формат:

```
so_4::ret_code_t
start(
  /*! Диспетчер, который должен применяться для работы
    Run-time SObjectizer.
    Должен быть указателем на динамический объект, если
    so_4::auto_destroy_disp == destruction_flag. */
  so_4::rt::dispatcher_t * disp,
  /*! Признак необходимости уничтожения диспетчера
    перед возвратом из функции. */
  so_4::destroy_disp_flags_t destruction_flag,
  /*!
    Стартовая кооперация, которая должна быть зарегистрирована
    сразу после запуска Run-time.

    Если содержит 0, значит нет стартовой кооперации.

    Если не 0, и регистрация завершается неудачно, то
    Run-time останавливается, и функция start возвращает
    код ошибки.
  */
  so_4::rt::agent_coop_t * initial_coop );
```

Обязательный аргумент *disp* — это диспетчер, который должен применяться для работы SObjectizer. В данном случае используется простейший диспетчер с одной рабочей нитью и приоритетной обработкой событий на этой нити.

Последний аргумент *initial_coop* — это указатель на т.н. стартовую кооперацию. Если *start_coop* равен 0, то считается, что при старте SObjectizer Run-Time не нужно создавать никаких прикладных агентов.

Функция *so_4::api::start* служит для запуска SObjectizer Run-Time и выполняет следующие действия:

- запускает диспетчер. Диспетчер создает все необходимые ему нити и становится готовым для диспетчеризации событий и отложенных сообщений;
- создает несколько агентов, которые являются неотъемлимой, встроенной частью SObjectizer Run-Time. Часть из этих агентов может использоваться пользовательским кодом. Наиболее важным из встроенных агентов является агент с именем `a_sobjectizer`;
- если указана стартовая кооперация, то осуществляет попытку ее регистрации;
- ожидает завершения работы диспетчера.

Если на каком-либо из шагов возникает ошибка, то функция *start* завершается. Если все идет нормально, то возврат из *start* происходит только после завершения работы диспетчера — т.е. только после завершения работы прикладной системы, построенной на основе SObjectizer. Это означает, что нить, вызвавшая функцию *start*, блокируется до завершения прикладной системы. В данном примере это именно то, что нужно — главная нить приложения ожидает завершения работы примера.

После обращения к *so_4::api::start* проверяется код возврата функции *start*:

```
if( rc )
  {
    // Запустить SObjectizer Run-Time не удалось.
    std::cerr << "start: " << rc << std::endl;
  }

return int( rc );
```

Если код возврата отличен от 0, значит не удалось нормально стартовать SObjectizer. Сообщение об этом выводится на стандартный поток ошибок. Тип *so_4::ret_code_t* является структурой, для которой определены операторы преобразования в *int* и вывода в *std::ostream*.

7.2 Результат работы примера

В результате работы примера на стандартный поток вывода печатается сообщение (при использовании SObjectizer v.4.2.7):

```
Hello, world! This is SObjectizer v.4.20700
```

7.3 Резюме

- Агенты реализуются в программе C++ классами, производными от *so_4::rt::agent_t*. Класс *so_4::rt::agent_t* должен быть доступным (*public*) базовым классом. Класс *so_4::rt::agent_t* может быть не единственным базовым классом.
- В C++ классе должно быть описано два виртуальных метода:

```
virtual const char *
so_query_type() const;
virtual void
so_on_subscription();
```

- Класс агента для SObjectizer должен быть описан внутри макросов *SOL4_CLASS_START* и *SOL4_CLASS_FINISH*. Описание класса агента для SObjectizer включает описание сообщений, событий и состояний.
- Имя типа, указанное в макросе *SOL4_CLASS_START*, должно быть именем C++ типа. Это же имя будет использоваться для идентификации типа агентов в системном словаре SObjectizer. Имя типа в системном словаре должно быть уникальным. Поэтому в больших проектах рекомендуется использовать пространства имен, а в *SOL4_CLASS_START* указывать полное имя с указанием имен всех пространств имен.
- Макросы *SOL4_CLASS_START* и *SOL4_CLASS_FINISH* содержат в себе реализацию метода *so_query_type*, поэтому этот метод реализовывать не нужно.

- Понятие события и обработчика события являются неразделимыми, т.е. не может быть события без обработчика. Поэтому для имен событий в описании класса агента используются имена методов-обработчиков.

- Методом-обработчиком должен быть нестатический, возможно, виртуальный, *public* - метод формата:

```
void
evt_handler();
```

или

```
void
evt_handler(
  const so_4::rt::event_data_t & data );
```

Примечание. Другие типы обработчиков событий будут рассмотрены в следующих главах.

- В описании класса агента должны быть указаны все события класса. Для описания события предназначен макрос *SOL4_EVENT(event_name)*.

Примечание. Другие макросы для декларации событий агентов будут рассматриваться в следующих главах.

- В описании класса агента должны быть указаны все состояния агентов данного класса. Для описания состояния используются макросы *SOL4_STATE_START(state_name)* и *SOL4_STATE_FINISH()*.

- В каждом состоянии должны быть перечислены события, которые допускаются для обработки в данном состоянии. Осуществляется это использованием макросов *SOL4_STATE_EVENT(event_name)* между макросами *SOL4_STATE_START* и *SOL4_STATE_FINISH*.

- События агента изначально являются неподписанными — т.е. не имеющими инцидентов. Для подписки событий предназначен виртуальный метод:

```
virtual void
so_on_subscription();
```

Этот метод вызывается при регистрации агента в SObjectizer Run-Time.

- Для подписки события предназначены методы *so_subscribe*, унаследованные из базового класса *so_4::rt::agent_t*[2].

[2]Для подписки события могут также использоваться макросы *SOL4_SUBSCR_EVENT_START* и *SOL4_SUBSCR_EVENT_FINISN_CERR* (*SOL4_SUBSCR_EVENT_FINISN*), между которыми должны использоваться по одному макросу *SOL4_SUBSCR_EVENT_MSG* или *SOL4_SUBSCR_EVENT_MSG_SELF* на каждого инцидента события. В первых реализациях SObjectizer эти макросы широко использовались для подписки событий. Подробную информацию по ним можно получить в SObjectizer Reference Manual.

- Если событию назначено несколько инцидентов, то событие возникает при появлении в системе хотя бы одного сообщения-инцидента (инциденты объеденены по маске ИЛИ) [3]. Но методы *so_subscribe* предназначены для событий, которые имеют только одного инцидента.

- После того, как агент зарегистрирован, SObjectizer Run-Time отправляет агенту сообщение `a_sobjectizer.msg_start`. Агент может использовать это сообщение для определения момента начала своей работы в прикладной системе.

- Для определения имени системного агента `a_sobjectizer` рекомендуется использовать функцию:

```
namespace so_4::rt {

const std::string &
sobjectizer_agent_name();

} /* namespace so_4::rt */
```

- Для завершения работы прикладной системы (т.е. SObjectizer Run-Time) необходимо отослать сообщение `a_sobjectizer.msg_normal_shutdown`.

- SObjectizer не занимается созданием объектов-агентов и не может контролировать время жизни объектов-агентов. Это означает, что прикладной программист сам должен создать объект-агент перед регистрацией и обеспечить агенту время жизни большее, чем время, которое агент будет зарегистрирован в SObjectizer Run-Time.

- Агенты регистрируются в системе внутри кооперaций. Если нужно зарегистрировать одного агента, создается кооперация, содержащая только одного агента.

- Часть прикладной системы, занимающаяся регистрацией агентов, может не иметь понятия о реальном составе регистрируемых кооперaций — для регистрации достаточно иметь только ссылку (указатель) на объект-кооперацию.

- SObjectizer Run-Time нужно указывать объект-диспетчер, который будет заниматься диспетчеризацией событий и отложенных сообщений. За создание объекта-диспетчера отвечает прикладной программист. В некоторых случаях диспетчеру необходимо передать ссылку на объект таймер (таймерную нить). За создание таймерной нити отвечает прикладной программист.

- Время жизни объекта-диспетчера должно превышать время работы SObjectizer Run-Time!

- Для запуска SObjectizer Run-Time необходимо вызвать функцию *so_4::api::start*.

[3] События с несколькими инцидентами можно подписывать, например, используя макросы *SOL4_SUBSCR_EVENT_START* и *SOL4_SUBSCR_EVENT_FINISN_CERR* (*SOL4_SUBSCR_EVENT_FINISN*) вместо метода so_subscribe.

- Если запустить SObjectizer Run-Time не удается, то функция *start* возвращает отличное от нуля значение.
- Если запустить SObjectizer Run-Time удалось, то возврат из функции *start* произойдет только после завершения работы диспетчера – т.е. после завершения работы прикладной системы. Это означает, что вызвавшая *start* нить блокируется на все время работы SObjectizer Run-Time.

Глава 8

Пример hello_all

Пример hello_all демонстрирует различные аспекты, связанные с сообщениями агентов:

- описание;
- проверку;
- отправку;
- приоритетное получение и обработку;
- широковещательную и целевую отправку.

Также пример hello_all показывает, как регистрировать статическую кооперацию, состоящую из нескольких агентов.

Пример hello_all состоит из одного файла `main.cpp` (стр. 193).

8.1 Что делает пример

В примере работают три основных агента (`a_receiver_one`, `a_receiver_two`, `a_receiver_three`). При своем появлении в системе они рассылают всем сообщение о том, что они появились (сообщение `hello_to_all`). В этом сообщении каждый из агентов указывает свое имя. Три основных агента подписаны на сообщение `hello_to_all`. Получив его агент отправляет сообщение `hello_to_you` тому агенту, который отправил `hello_to_all`.

В результате оказывается, что каждый из основных агентов получает по три сообщения `hello_to_all` (включая и отправленное им самим) и по два сообщения `hello_to_you` от других основных агентов.

Основные агенты работают на приоритете, отличном от самого низкого, нулевого, приоритета. На нулевом приоритете функционирует специальный агент `a_shutdowner`, который подписан на сообщение `msg_hello_to_all`. По получении этого сообщения от кого-либо он завершает работу SObjectizer Run-Time. Из-за того, что агент `a_shutdowner` работает на самом низком приоритете, а основные агенты на более высоких, завершение работы примера осуществляется после того, как основные агенты выполнят свои действия.

8.2 Разбор файла main.cpp

В начале файла находится C++ описание класса агента, который не содержит ничего важного для SObjectizer, кроме структуры `msg_hello`:

```
class a_msg_owner_t
  :  public so_4::rt::agent_t
  {
    typedef so_4::rt::agent_t base_type_t;

  public :
    a_msg_owner_t()
      : base_type_t( agent_name() )
      {}

    virtual ~a_msg_owner_t()
      {}

    virtual const char *
    so_query_type() const;

    virtual void
    so_on_subscription()
      {}

    static std::string
    agent_name()
      {
        return "a_msg_owner";
      }

    // Сообщение, которым будут пользоваться агенты примера.
    struct  msg_hello
      {
        std::string m_sender;

        msg_hello() {}
        msg_hello(
          const std::string & sender )
          : m_sender( sender )
          {}

        static bool
        check( const msg_hello * msg )
          {
            return ( 0 != msg &&
            msg->m_sender.length() );
          }
```

```
    };
};
```

Этот класс будет использован для единственного агента, который будет выполнять роль владельца сообщений `msg_hello_to_all` и `msg_hello_to_you`.

Следует заострить внимание на следующих моментах в описании и реализации класса агента `a_msg_owner_t`:

- В отсутствии директив *using namespace* в некоторых компиляторах возникают проблемы вызова конструкторов базовых классов. Так, для класса *a_msg_owner_t* вызов конструктора класса *so_4::rt::agent_t* следовало записать так:

  ```
  a_msg_owner_t::a_msg_owner_t()
    :  so_4::rt::agent_t( agent_name() )
    {
      ...
    }
  ```

  Однако, не все компиляторы различают, что *so_4::rt::agent_t* в данном случае является именем базового класса, а не именем статического метода или функции в пространстве имен *so_4::rt*.

  Для обхода проблем с подобными компиляторами в описании класса *a_msg_owner_t* базовому классу дается псевдоним:

  ```
  typedef so_4::rt::agent_t base_type_t;
  ```

  Благодаря которому реализация конструктора приобретает понимаемый всеми компиляторами вид:

  ```
  a_msg_owner_t::a_msg_owner_t()
    :  base_type_t( agent_name() )
    {
      ...
    }
  ```

- Часто при использовании агентов, существующих в системе в единственном числе и только предоставляющих для использования собственные сообщения, возникает вопрос: как остальные агенты будут узнавать имя этого единственного агента?

  Практический опыт использования SObjectizer показал, что удачным ответом на этот вопрос является наличие в классе агента статического метода, возвращающего имя агента:

  ```
  class a_msg_owner_t :
    ...
    {
      ...
      static std::string
      agent_name();
      ...
    };
  ```

Описание класса агентов *a_msg_owner_t* для SObjectizer указывает, что каждый агент типа *a_msg_owner_t* будет обладать двумя сообщениями (msg_hello_to_all и msg_hello_to_you):

```
SOL4_CLASS_START( a_msg_owner_t )

  SOL4_MSG_START( msg_hello_to_all, a_msg_owner_t::msg_hello )
    SOL4_MSG_FIELD( m_sender )

    SOL4_MSG_CHECKER( a_msg_owner_t::msg_hello::check )
  SOL4_MSG_FINISH()

  SOL4_MSG_START( msg_hello_to_you, a_msg_owner_t::msg_hello )
    SOL4_MSG_FIELD( m_sender )

    SOL4_MSG_CHECKER( a_msg_owner_t::msg_hello::check )
  SOL4_MSG_FINISH()

SOL4_CLASS_FINISH()
```

Для SObjectizer сообщение описывается внутри макросов *SOL4_MSG_START(msg_name, msg_type)* и *SOL4_MSG_FINISH()*.

Макрос *SOL4_MSG_START* говорит, что агент обладает сообщением с именем msg_name, а структура данных этого сообщения определяется C++ типом *msg_type*. Типом сообщения может быть любой C++ тип, удовлетворяющий некоторым ограничениям. Пример сообщений msg_hello_to_all и msg_hello_to_you показывает, что имя типа сообщения не обязательно должно совпадать с именем сообщения. В данном случае тип *a_msg_owner_t::msg_hello* используется как для сообщения msg_hello_to_all, так и для сообщения msg_hello_to_you.

Структура данных для сообщения должна быть определена всегда, даже если в сообщении не передается никаких данных. Т.е. каждому сообщению должен соответствовать корректный C++ класс или структура. Если сообщение не содержит никаких данных, то соответствующий ей C++ тип может не содержать полей, как, например, это происходит для сообщения a_sobjectizer.msg_normal_shutdown:

```
struct  msg_normal_shutdown
  {};
```

Необходимость существования корректного C++ типа (класса или структуры) для сообщения возникает из-за того, что макросы *SOL4_MSG_START* и *SOL4_MSG_FINISH* строят специальный вспомогательный код, используемый SObjectizer для управления экземплярами сообщений. Этот вспомогательный код, среди прочего, содержит команды создания экземпляра сообщения (т.е. экземпляра данных сообщения) посредством оператора *new* и удаления экземпляра сообщения посредством оператора *delete*. Использование оператора *new* определяет главное ограничение на тип данных для сообщения: **в типе данных для сообщения должен быть доступный конструктор по умолчанию (конструктор без параметров)**.

Именно из-за этого ограничения в структуре *a_msg_owner_t::msg_hello* объявлено два конструктора, хотя по логике примера hello_all достаточно было бы конструктора с параметром sender.

Макрос *SOL4_MSG_FIELD* предназначен для описания полей сообщения, доступных для SOP-взаимодействия с внешним миром. Но этот вопрос не является ключевым для данного примера и здесь не рассматривается.

Чрезвычайно важную роль играет макрос *SOL4_MSG_CHECKER*. Он указывает SObjectizer функцию проверки корректности экземпляра сообщения.

Вся логика работы примера hello_all базируется на том, что в сообщениях передаются имена агентов. Т.е., что все экземпляры сообщений содержат реальные данные. Но средства отправки сообщений SObjectizer позволяют отправлять сообщения без данных. Кроме того, при программировании всегда легко допустить ошибку и отправить, например, сообщение `msg_hello_to_you` без имени агента-отправителя. Естественно, что получатель сообщений в своем событии должен учитывать возможность возникновения подобных ситуаций. Это означает, что код обработчика событий будет похож на:

```
void
a_msg_receiver_t::evt_hello_to_you(
  const a_msg_owner_t::msg_hello * cmd )
  {
    // Проверяем корректность данных.
    // Если данные были переданы...
    if( cmd )
      {
        // ... и имя агента указано...
        if( cmd->m_sender.length() )
          {
            // ... то выполняем что-то.
            ...
          }
      }
  }
```

Очевидно, что включение подобных проверок в каждый обработчик события имеет множество недостатков. Кроме того, если событие подписано на несколько разных инцидентов, то осуществить проверку для каждого из инцидентов в самом обработчике события может быть вообще невозможно. Поэтому SObjectizer предоставляет общий механизм для проверки корректности экземпляра сообщения.

Для каждого сообщения может быть указана функция или статический метод какого-либо класса формата:

```
bool
checker( const void * msg );
```

Эта функция должна возвращать *true*, если экземпляр сообщения является корректным, и *false* в противном случае. SObjectizer вызывает указанную в макросе *SOL4_MSG_CHECKER* функцию каждый раз, когда в системе появляется сообщение. Вызов функции проверки осуществляется непосредственно перед генерацией событий, подписанных на сообщение. Для отложенных и периодических сообщений это означает, что проверка будет осуществляться не в момент вызова *so_4::api::send_msg*, а в момент, когда наступает время возникновения сообщения.

Если функция проверки сообщения возвращает *false*, то сообщение игнорируется, и генерация подписанных на него событий не производится. Т.е., если для сообщения должным образом написана функция проверки, то в события не попадают заведомо некорректные экземпляры сообщений. А из этого следует важнейший вывод, подтвержденный практикой использования SObjectizer: **для каждого сообщения, в котором передаются данные, должна быть определена и описана в макросе *SOL4_MSG_CHECKER* функция проверки сообщения**.

Для типа *a_msg_owner_t* функция проверки сообщения реализована следующим образом:

```
bool
a_msg_owner_t::msg_hello::check(
  const msg_hello * msg )
  {
    return ( 0 != msg &&
      msg->m_sender.length() );
  }
```

Т.е. проверяется, чтобы в сообщении передавались данные *(0 != msg)* и в данных было указано имя агента – длина имени отправителя *(msg->m_sender)* отлична от нуля.

Класс *a_msg_owner_t* содержит пустое тело для метода *so_on_subscription*:

```
virtual void
so_on_subscription()
  {}
```

Класс агентов *a_msg_owner_t* не имеет событий, поэтому метод *so_on_subscription* пуст. Но определить и реализовать в классе *a_msg_owner_t* метод *so_on_subscription* необходимо, т.к. в базовом классе *so_4::rt::agent_t* этот метод является чистым виртульным методом.

После класса *a_msg_owner_t* описывается и реализуется класс *a_msg_receiver_t*. Начнем его рассмотрение с описания класса *a_msg_receiver_t* для SObjectizer:

```
SOL4_CLASS_START( a_msg_receiver_t )

  SOL4_EVENT( evt_start )
  SOL4_EVENT_STC(
    evt_hello_to_all,
    a_msg_owner_t::msg_hello )
  SOL4_EVENT_STC(
    evt_hello_to_you,
    a_msg_owner_t::msg_hello )

  SOL4_STATE_START( st_initial )
    SOL4_STATE_EVENT( evt_start )
    SOL4_STATE_EVENT( evt_hello_to_all )
    SOL4_STATE_EVENT( evt_hello_to_you )
  SOL4_STATE_FINISH()

SOL4_CLASS_FINISH()
```

Класс *a_msg_receiver_t* имеет три обработчика событий: evt_start, evt_hello_to_all, evt_hello_to_you. Метод *a_msg_receiver_t::evt_start*, который реализует событие evt_start, не имеет аргументов. Поэтому для описания события evt_start используется макрос *SOL4_EVENT*. Но методы *a_msg_receiver_t::evt_hello_to_all*, *a_msg_receiver_t::evt_hello_to_you* получают один аргумент: *const a_msg_owner_t::msg_hello & cmd*. Поэтому для описания событий evt_hello_to_all, evt_hello_to_you используется специальный макрос *SOL4_EVENT_STC*.

Макрос *SOL4_EVENT_STC* получает два аргумента: имя события (которое также является именем метода в C++ классе, реализующем агента) и имя C++ типа сообщения-инцидента события. Этот макрос позволяет:

- указать SObjectizer Run-Time, что данное событие может обрабатывать только инциденты, которые принадлежат указанному C++ типу. Если во время подписки события окажется, что инцидент принадлежит другому C++ типу, то подписка события не выполняется. В качестве имени C++ типа инцидента берется имя типа, указанное в макросе *SOL4_MSG_START* [1].

  Поскольку при отсылке сообщения посредством *so_4::api::send_msg* информация о реальном типе объекта сообщения теряется (т.к. *so_4::api::send_msg* получает *void*-указатель), то макрос *SOL4_EVENT_STC* является единственным механизмом проверки типов инцидентов;

- использовать в качестве обработчиков событий методы одного из следующих форматов:

```
void
evt_handler( const msg * );
```

или

```
void
evt_handler( const msg & );
```

или

```
void
evt_handler(
  const so_4::rt::event_data_t &,
  const msg * );
```

или

```
void
evt_handler(
  const so_4::rt::event_data_t &,
  const msg & );
```

[1] SObjectizer не знает про *typedef*. Поэтому, если одному C++ типу с помощью *typedef* дать два разных псевдонима, а затем указать имена псевдонимов в качестве имени C++ типа сообщения, то SObjectizer будет считать, что сообщения принадлежат разным C++ типам

Особенно удобны обработчики событий, которые получают в качестве аргумента ссылку на экземпляр сообщения-инцидента. В этом случае сам SObjectizer Run-Time проверяет, чтобы в качестве экземпляра сообщения-инцидента был указан реальный объект (т.е., чтобы в функцию *so_4::api::send_msg* был передан не нулевой указатель на данные сообщения).

Обработчики, которые получают указатель на экземпляр сообщения-инцидента, следует использовать, если не для всех экземпляров сообщений может быть указан объект-сообщение[2].

При подписке событий агентов типа *a_msg_receiver_t* указывается приоритет 1:

```
void
a_msg_receiver_t::so_on_subscription()
  {
    so_subscribe( "evt_start",
      so_4::rt::sobjectizer_agent_name(),
      "msg_start", 1 );

    so_subscribe( "evt_hello_to_all",
      a_msg_owner_t::agent_name(),
      "msg_hello_to_all", 1 );

    so_subscribe( "evt_hello_to_you",
      a_msg_owner_t::agent_name(),
      "msg_hello_to_you", 1 );
  }
```

В своем обработчике события `evt_start` агенты типа *a_msg_receiver_t* сначала отображают на стандартный поток вывода сообщение о своем старте, а затем делают широковещательную рассылку сообщения `msg_hello_to_all`:

```
void
a_msg_receiver_t::evt_start()
  {
    std::cout << so_query_name() << ".evt_start" << std::endl;

    so_4::api::send_msg_safely(
      a_msg_owner_t::agent_name(), "msg_hello_to_all",
      new a_msg_owner_t::msg_hello( so_query_name() ) );
  }
```

Функция *so_4::api::send_msg_safely* отвечает не только за отсылку сообщения, но и за решение проблемы уничтожения экземпляра сообщения, если сообщение не было отправлено.

Без использования *so_4::api::send_msg_safely* самым простым способом отсылки сообщения выглядит такое использование функции *so_4::api::send_msg*:

[2] У некоторых старых C++ компиляторов, например, Visual C++ 6.0, существуют проблемы с компиляцией обработчиков событий, получающих аргумент-ссылку. Поэтому, если необходима совместимость с такими компиляторами, то следует использовать обработчики событий с аргументом-указателем.

```
so_4::api::send_msg( a_msg_owner_t::agent_name(),
  "msg_hello_to_all", new a_msg_owner_t::msg_hello( so_query_name() ) );
```

Но за простотой скрывается потенциальная опасность расхода памяти в случае, если вызов *so_4::api::send_msg* завершится неудачно. При успешной отправке сообщения SObjectizer берет на себя задачу освобождения памяти, отведенной под данные сообщения. Если же сообщение не отправлено по каким-либо причинам, то *send_msg* возвращает код ошибки, и SObjectizer оставляет ответственность за освобождение памяти на программисте. Поэтому корректная отправка сообщения должна была выглядеть так:

```
a_msg_owner_t::msg_hello * msg =
  new a_msg_owner_t::msg_hello( so_query_name() );
if( so_4::api::send_msg( a_msg_owner_t::agent_name(),
  "msg_hello_to_all", msg ) )
  {
    // Ошибка доставки сообщения.
    delete msg;
  }
```

Очевидно, что если явно контролировать каждое обращение к *send_msg*, то код событий будет состоять из сплошных *if(...)*, затрудняющих понимание логики события. Поэтому в SObjectizer реализована шаблонная функция *so_4::api::send_msg_safely*, предназначенная для неявного контроля за успешностью отсылки сообщения: если отправка сообщения не удалась, то данные сообщения автоматически уничтожаются.

Ранее не случайно говорилось именно о динамически созданных экземплярах сообщений. Не трудно убедиться, что единственным универсальным и надежным способом доставки данных в сообщениях является динамическое создание объекта-сообщения. Действительно, сообщение может обрабатываться произвольным количеством агентов на произвольном количестве нитей различных диспетчеров. Время, которое потребуется для обработки сообщения, заранее не известно, более того, оно зависит от степени загрузки прикладной системы. Но все это время объект-сообщение должен существовать. Поэтому объект-сообщение не может быть автоматической переменной:

```
// Неправильно!
a_msg_owner_t::msg_hello msg( so_query_name() );
so_4::api::send_msg( a_msg_owner_t::agent_name(),
  "msg_hello_to_all", &msg );
```

поскольку указатель на автоматическую переменную оказывается недействительным после возврата из функции, в которой переменная объявлена (более точно: после выхода из блока, в котором переменная объявлена).

Использование статических переменных в качестве объектов-сообщений не позволяет одновременно существовать нескольким самостоятельным экземплярам сообщения с различными данными:

```
// Неправильно!
static a_msg_owner_t::msg_hello msg( so_query_name() );
so_4::api::send_msg( a_msg_owner_t::agent_name(),
  "msg_hello_to_all", &msg );
```

Поэтому SObjectizer спроектирован в предположении, что все объекты-сообщения являются динамически созданными объектами. Отсюда вытекает следующее требование к передаче данных в сообщениях: **если в сообщении пересылаются данные, то в *send_msg* (*send_msg_safely*) необходимо передавать указатель на объект, созданный посредством оператора *new***. Нарушение этого требования может привести к самым пагубным для прикладной системы последствиям!

Итак, агенты типа *a_msg_receiver_t* в своем событии `evt_start` широковещательно рассылают сообщение `msg_hello_to_all` агента `a_msg_owner_t::agent_name()`. Причем они же и подписаны на эти сообщения. Поэтому после отсылки сообщения `msg_hello_to_all` будут возникать события `evt_hello_to_all`, которые обрабатываются следующим образом:

```
void
a_msg_receiver_t::evt_hello_to_all(
  const a_msg_owner_t::msg_hello & cmd )
  {
    std::cout << so_query_name() << ".evt_hello_to_all: "
      << cmd.m_sender << std::endl;

    // Если приветствие слали не мы, то отошлем ответ.
    if( so_query_name() != cmd.m_sender )
      {
        so_4::api::send_msg_safely(
          a_msg_owner_t::agent_name(),
          "msg_hello_to_you",
          new a_msg_owner_t::msg_hello( so_query_name() ),
          cmd.m_sender );
      }
  }
```

На стандартный поток печатаются имена агентов: получателя сообщения и отправителя сообщения. Затем, если отправитель и получатель являются разными агентами (т.е. их имена не совпадают), отправителю в ответ отсылается сообщение `msg_hello_to_you`. Но отсылка осуществляется уже целенаправленно.

В обработчике события `evt_hello_to_you` просто печатаются имена агента-получателя и агента-отправителя сообщения.

```
void
a_msg_receiver_t::evt_hello_to_you(
  const a_msg_owner_t::msg_hello & cmd )
  {
    std::cout << so_query_name() << ".evt_hello_to_you: "
      << cmd.m_sender << std::endl;
  }
```

В примере также участвует агент типа `a_shutdowner_t`, задачей которого является завершение работы примера, как только для него произойдет событие `evt_shutdown`. Событие `evt_shutdown` подписывается на сообщение `sobjectizer.msg_start`, поэтому оно произойдет, как только агент `a_shutdowner` будет зарегистрирован в SObjectizer. Но

приоритет у события `evt_shutdown` нулевой, меньший, чем приоритет событий агентов типа `a_msg_receiver_t`. Поэтому событие `evt_shutdown` будет обработано после всех событий агентов типа `a_msg_receiver_t`.

В функции *main()* создается стартовая кооперация агентов и запускается SObjectizer Run-Time. В стартовую кооперацию включаются агенты:

- `a_msg_owner`, чьи сообщения будут использоваться агентами типа `a_msg_receiver_t`;
- `a_shutdowner`, чьей задачей является завершение работы примера, и
- агенты типа `a_msg_receiver_t`, которые выполняют основные действия примера.

Для того, чтобы создать кооперацию, в которую входит только один агент (как в примере hello_world, см. главу 7), используется конструктор класса *so_4::rt::agent_coop_t* с одним аргументом. Если же нужна кооперация из нескольких агентов, то используется конструктор класса *so_4::rt::agent_coop_t* формата[3]:

```
agent_coop_t(
  /*! Имя кооперации. */
  const std::string & coop_name,
  /*! Указатели на агентов кооперации. Каждый элемент
    должен содержать корректный указатель.

    После завершения работы конструктора сам вектор
    может быть уничтожен, но все агенты должны
    существовать дольше, чем объект agent_coop_t.
  */
  agent_t ** coop_agents,
  /*! Количество элементов в coop_agents. */
  size_t agent_count );
```

Поэтому код по формированию стартовой кооперации выглядит следующим образом:

```
a_msg_owner_t a_msg_owner;
a_shutdowner_t  a_shutdowner;

a_msg_receiver_t  a_receiver_one( "a_receiver_one" );
a_msg_receiver_t  a_receiver_two( "a_receiver_two" );
a_msg_receiver_t  a_receiver_three( "a_receiver_three" );

// Кооперация примера.
so_4::rt::agent_t * g_coop_agents[] =
  {
```

[3]При использовании конструктора *so_4::rt::agent_coop_t* с одним аргументом имя кооперации совпадает с именем единственного агента кооперации. Если необходимо дать кооперации отличное от имени агента имя, то следует использовать конструктор *so_4::rt::agent_coop_t* с несколькими аргументами.

```
      &a_msg_owner,
      &a_shutdowner,
      &a_receiver_one,
      &a_receiver_two,
      &a_receiver_three
    };

  so_4::rt::agent_coop_t  a_coop( "a_coop", g_coop_agents,
    sizeof( g_coop_agents ) / sizeof( g_coop_agents[ 0 ] ) );
```

8.3 Результат работы примера

В результате работы примера на стандартный поток вывода печатаются следующие сообщения:

```
a_receiver_one.evt_start
a_receiver_three.evt_start
a_receiver_two.evt_start
a_receiver_one.evt_hello_to_all: a_receiver_one
a_receiver_three.evt_hello_to_all: a_receiver_one
a_receiver_two.evt_hello_to_all: a_receiver_one
a_receiver_one.evt_hello_to_all: a_receiver_three
a_receiver_three.evt_hello_to_all: a_receiver_three
a_receiver_two.evt_hello_to_all: a_receiver_three
a_receiver_one.evt_hello_to_all: a_receiver_two
a_receiver_three.evt_hello_to_all: a_receiver_two
a_receiver_two.evt_hello_to_all: a_receiver_two
a_receiver_one.evt_hello_to_you: a_receiver_three
a_receiver_one.evt_hello_to_you: a_receiver_two
a_receiver_three.evt_hello_to_you: a_receiver_one
a_receiver_three.evt_hello_to_you: a_receiver_two
a_receiver_two.evt_hello_to_you: a_receiver_one
a_receiver_two.evt_hello_to_you: a_receiver_three
```

Всю результирующую печать можно разделить на три части. В первой части видно, как все три агента типа `a_msg_receiver_t` обрабатывают сообщение `msg_start`:

```
a_receiver_one.evt_start
a_receiver_three.evt_start
a_receiver_two.evt_start
```

Во второй части видно, как каждый из агентов типа `a_msg_receiver_t` обрабатывает сообщения `msg_start`, разосланные в обработчиках событий `evt_start`:

```
a_receiver_one.evt_hello_to_all: a_receiver_one
a_receiver_three.evt_hello_to_all: a_receiver_one
a_receiver_two.evt_hello_to_all: a_receiver_one
a_receiver_one.evt_hello_to_all: a_receiver_three
```

```
a_receiver_three.evt_hello_to_all: a_receiver_three
a_receiver_two.evt_hello_to_all: a_receiver_three
a_receiver_one.evt_hello_to_all: a_receiver_two
a_receiver_three.evt_hello_to_all: a_receiver_two
a_receiver_two.evt_hello_to_all: a_receiver_two
```

При этом видно, что в результате широковещательной рассылки сообщения получили все агенты, в том числе и отправители сообщений.

Также видно, что первым из обработавших событие `evt_start` был агент `a_receiver_one`, вторым — `a_receiver_three`, а третьим — `a_receiver_two`[4]. Следовательно, и отсылка сообщений `msg_hello_to_all` осуществлялась в этом же порядке. Это подтверждается порядком запуска обработчиков события `evt_hello_to_all`: сначала все три агента обрабатывают сообщение `msg_hello_to_all` от агента `a_receiver_one`, затем от агентов `a_receiver_three` и `a_receiver_two`.

В третьей части видно, как агенты обрабатывали сообщения `msg_hello_to_you` в результате целенаправленной рассылки:

```
a_receiver_one.evt_hello_to_you: a_receiver_three
a_receiver_one.evt_hello_to_you: a_receiver_two
a_receiver_three.evt_hello_to_you: a_receiver_one
a_receiver_three.evt_hello_to_you: a_receiver_two
a_receiver_two.evt_hello_to_you: a_receiver_one
a_receiver_two.evt_hello_to_you: a_receiver_three
```

Следует вспомнить про агента `a_shutdowner`, который также входил в стартовую кооперацию. Он также подписан на стартовое сообщение `msg_start`, следовательно, событие `evt_shutdown` для него было сгенерировано одновременно с событиями `evt_start` агентов типа `a_msg_receiver_t`. Но диспетчер запустил это событие на обработку только после того, как отработали все события агентов `a_msg_receiver_t`. Причем даже те события, которые были порождены уже после генерации события `evt_shutdown`! Произошло это из-за того, что приоритет события `evt_shutdown` был ниже приоритета событий агентов `a_msg_receiver_t`.

Важно понимать, что такая картина с очередностью вызовов обработчиков событий возможна только, если все обработчики событий агентов `a_msg_receiver_t` и агента `a_shutdowner` вызываются на одной и той же нити диспетчера. Что и происходит при применении простейшего диспетчера с одной рабочей нитью. Но, если для примера hello_all применить диспетчер с активными объектами и сделать агент `a_shutdowner` активным объектом (или агентов `a_msg_receiver_t` сделать активными агентами), то ситуация могла бы существенно измениться. Т.к. события различных агентов исполнялись бы на различных нитях, то все зависело бы от того, как операционная система выделяла бы время каждой из нитей диспетчера. Иногда агент `a_shutdowner` на своей нити отрабатывал бы быстрее, чем остальные агенты. Иногда наоборот.

Происходит это из-за того, что в штатных диспетчерах[5] SObjectizer нет общей очереди событий, элементы из которой извлекаются последовательно после окончания обработки очередного элемента. Вместо этого диспетчеры организуют очереди событий

[4] SObjectizer не определяет, в каком порядке агенты одной кооперации будут обрабатывать стартовое сообщение `msg_start`. Это определяется случайным образом в момент регистрации кооперации.

[5] На момент написания этих строк в состав SObjectizer входят:
- диспетчер с одной рабочей нитью (все агенты работают на одной нити);
- диспетчер с активными объектами (агенты, которые являются активными объектами работают на

на каждой из своих рабочих нитях. Поэтому в таких диспетчерах обеспечить гарантированный порядок запуска обработчиков событий можно только для агентов, которые работают на одной нити.

8.4 Резюме

- Если в системе должен существовать только один агент некоторого типа, то для хранения имени этого агента хорошо зарекомендовал себя следующий способ:

```
class some_agent_t
  : public so_4::rt::agent_t
  {
  public :
    ...
    static std::string
    agent_name()
      {
        return "some_name";
      }
    ...
  };
```

- Каждому сообщению в SObjectizer должен соответствовать реальный C++ тип (класс или структура). Если сообщение не содержит данных, то соответствующий C++ тип может не содержать ни одного поля:

```
struct  msg_normal_shutdown
  {};
```

- C++ тип, реализующий сообщение (соответствующий сообщению), должен обладать public-конструктором по умолчанию (конструктором без параметров).

- Все сообщения, которыми будут владеть агенты какого-либо типа, должны быть описаны в описании типа агента для SObjectizer (т.е. между макросами *SOL4_CLASS_START* и *SOL4_CLASS_FINISH*).

- Описание сообщения для SObjectizer-а начинается макросом *SOL4_MSG_START(msg_name, msg_type)* и заканчивается макросом *SOL4_MSG_FINISH()*.

своих нитях, остальные (пассивные) агенты — на одной общей нити);
- диспетчер с активными группами (агенты, составляющие активную группу, работают на одной нити, остальные агенты — на других нитях);
- диспетчер главной нити для Win32 (специально отмеченные агенты работают только на главной нити приложения);
- диспетчер главной нити для Qt (специально отмеченные агенты работают только на главной нити приложения).
Диспетчер с активными группами и диспетчеры главной нити могут использовать в качестве подчиненных диспетчер с активными объектами или диспетчер с одной рабочей нитью.

- Имя сообщения определяется аргументом *msg_name* макроса *SOL4_MSG_START*. Имя сообщения может не совпадать с именем C++ типа, реализующим сообщение. Один и тот же C++ тип может использоваться для реализации различных сообщений различных агентов одновременно.

- Для каждого сообщения можно задать функцию проверки экземпляра сообщения. Это функция или статический метод класса формата:

```
bool
checker( const void * msg );
```

  Функция проверки сообщения должна возвращать *true*, если сообщение корректно, и *false* в противном случае.

- Функция проверки сообщения описывается в сообщении посредством макроса *SOL4_MSG_CHECKER(fn)*.

- Если в сообщении пересылаются какие-либо данные, то для сообщения должна быть предоставлена функция проверки сообщения. Это позволит защитить обработчики сообщения от попадания в них заведомо некорректных данных или отсутствия данных вообще.

- Если в сообщении пересылаются данные, то в функцию *so_4::api::send_msg* нужно передать указатель на созданный посредством оператора new указатель на объект-сообщение. Нельзя передавать в *so_4::api::send_msg* указатели на локальные или глобальные автоматические или статические переменные.

- В случае успешной отправки сообщения уничтожением объекта-сообщения занимается SObjectizer. Объект-сообщение уничтожается посредством оператора *delete* после того, как сообщение будет обработано всеми подписчиками.

- Если сообщение не было отправлено (*so_4::api::send_msg* возвращает код ошибки), то ответственность за уничтожение объекта-сообщения лежит на прикладном программисте.

- Для облегчения контроля за успешностью отправки сообщения предназначена шаблонная функция *so_4::api::send_msg_safely*.

- Сообщения можно отправлять либо широковещательно (когда не указывается адресат сообщения, и сообщение получают все его подписчики), либо целенаправленно (когда указан адресат, и только адресат получает сообщение).

- Целенаправленная отправка сообщения адресату, который не подписан на данное сообщение, не считается ошибкой. Сообщение просто никем не обрабатывается, а SObjectizer уничтожает объект-сообщение.

- Обработчики событий могут иметь формат:

```
void
evt_handler( const msg * );
```

  или

```
void
evt_handler( const msg & );
```

или

```
void
evt_handler(
  const so_4::rt::event_data_t &,
  const msg * );
```

или

```
void
evt_handler(
  const so_4::rt::event_data_t &,
  const msg & );
```

такие обработчики должны описываться для SObjectizer с помощью макроса *SOL4_EVENT_STC*.

- Если обработчик события описан с помощью макроса *SOL4_EVENT_STC*, то SObjectizer Run-Time при подписке сообщения будет проверять совпадение имен типов сообщений, указанных в *SOL4_EVENT_STC* и *SOL4_MSG_START*.

- Если обработчик события получает аргумент-ссылку на объект сообщения, то SObjectizer сам будет контролировать, чтобы переданный в функцию *so_4::api::send_msg* указатель не был нулевым (в противном случае обработчик не будет вызван).

- Класс *so_4::rt::agent_coop_t* имеет два конструктора:

  ```
  agent_coop_t(
    agent_t & agent );
  agent_coop_t(
    const std::string & coop_name,
    agent_t ** coop_agents,
    size_t agent_count );
  ```

  Первый конструктор предназначен для случая, когда в SObjectizer Run-Time нужно зарегистрировать только одного агента. В этом случае именем кооперации является имя единственного агента в ней.

  Второй конструктор предназначен для случая, когда в кооперацию входит более одного агента. В этом случае кооперации нужно явно задать имя. Список агентов в кооперации задается при помощи вектора указателей на агентов и количества элементов в этом векторе.

- SObjectizer диспетчирует события с учетом их приоритета, но реальный порядок запуска обработчиков событий определяется конкретным типом диспетчера.

Глава 9

Пример hello_delay

Пример hello_delay демонстрирует использование отложенных сообщений. В примере работает один агент `a_hello`. В своем обработчике сообщения `a_sobjectizer.msg_start` он отсылает самому себе отложенное сообщение `msg_hello_time`. При получении сообщения `msg_hello_time` агент `a_hello` завершает работу SObjectizer.

Пример hello_delay состоит из одного файла `main.cpp` (стр. 199).

9.1 Разбор файла main.cpp

Главный агент примера `a_hello` владеет сообщением `msg_hello_time` и имеет два события: `evt_start` и `evt_hello_time`, которые обрабатываются в единственном состоянии `st_initial`. При подписке агент `a_hello` указывает в качестве инцидента события `evt_start` сообщение `msg_start` агента `a_sobjectizer`. А в качестве инцидента события `evt_hello_time` — собственное сообщение `msg_hello_time`:

```
void
a_hello_t::so_on_subscription()
  {
    so_subscribe( "evt_start",
      so_4::rt::sobjectizer_agent_name(), "msg_start" );

    so_subscribe( "evt_hello_time", "msg_hello_time" );
  }
```

В обработчике события `evt_start`:

```
void
a_hello_t::evt_start()
  {
    // Сообщаем, когда будет выдано приветствие.
    unsigned int sec = 2;
    time_t t = time( 0 );
    std::cout << so_query_name() << ": "
      << asctime( localtime( &t ) )
      << "hello after " << sec << " seconds" << std::endl;
```

```
    // Отсылаем себе отложенное сообщение.
    so_4::api::send_msg(
      so_query_name(),
      "msg_hello_time", 0, 0,
      sec * 1000 );
  }
```

на стандартный поток вывода печатается время старта обработчика, после чего отсылается отложенное сообщение msg_hello_time.

Для отсылки сообщений SObjectizer предоставляет несколько вариантов функции *so_4::api::send_msg*[1], в том числе и:

```
so_4::ret_code_t
send_msg(
  /*! Агент-владелец сообщения. */
  const std::string & agent_name,
  /*! Имя сообщения. */
  const std::string & msg_name,
  /*!
    Данные сообщения.

    Если msg_data != 0, то подразумевается, что
    это динамически созданный объект того типа,
    который был указан для сообщения в описании
    класса агента.
  */
  void * msg_data = 0,
  /*!
    Имя агента-получателя сообщения.

    Если receiver == 0, то производится широковещательная
    рассылка сообщения. Т.е. сообщение получают все
    подписанные на него агенты.

    Если receiver != 0, то производится целенаправленная
    отсылка сообщения. Т.е. сообщение получает только
    агент с именем receiver.
  */
  const char * receiver = 0,
  /*!
    Задержка перед первым появлением сообщения в
    миллисекундах.

    Точность отсчета времени определяется точностью
    используемой Run-time нити таймера.
```

[1] Формат функций *so_4::api::send_msg_safely* совпадает с форматом функций *so_4::api::send_msg*.

```
*/
unsigned int delay = 0,
/*!
  Период повторного появления периодического сообщения.

  Точность отсчета времени определяется точностью
  используемой Run-time нити таймера.
*/
unsigned int period = 0 );
```

Параметр delay определяет величину задержки в миллисекундах перед началом обработки сообщения. Если delay отличен от нуля, то *send_msg* только проверяет возможность отсылки сообщения, после чего сообщение передается диспетчеру для ожидания на нити таймера. Когда время задержки истечет, диспетчер возвратит сообщение SObjectizer Run-Time для дальнейшей обработки — проверки корректности экземпляра сообщения и генерации событий, подписанных на сообщение.

Отложенную отсылку можно осуществлять как широковещательно (как в данном примере), так и целенаправленно. При целенаправленной отложенной отсылке наличие агента-адресата проверяется только после истечения времени задержки. Т.е. существует возможность отсылки отложенного сообщения агенту, которого на данный момент времени не существует — он может быть зарегистрирован, пока отложенное сообщение ожидает начала своей обработки[2].

В обработчике события `evt_hello_time`:

```
void
a_hello_t::evt_hello_time()
  {
    time_t t = time( 0 );
    std::cout << so_query_name() << ": "
      << asctime( localtime( &t ) ) << std::flush;

    std::cout << "Hello, World!" << std::endl;

    // Отсылаем сообщение для завершения работы.
    so_4::api::send_msg(
      so_4::rt::sobjectizer_agent_name(),
      "msg_normal_shutdown", 0 );
  }
```

на стандартный поток вывода печатается время старта обработчика, после чего работа примера завершается.

9.2 Результат работы примера

В результате работы примера на стандартный поток вывода печатается:

[2] Но на момент отсылки сообщения должен быть зарегистрирован агент-владелец отсылаемого сообщения

```
a_hello: Tue Jun 15 08:43:45 2004
hello after 2 seconds
a_hello: Tue Jun 15 08:43:47 2004
Hello, World!
```

9.3 Резюме

- Для отправки отложенного сообщения необходимо в параметре delay функции *so_4::api::send_msg* указать время задержки в миллисекундах. Реальная обработка сообщения (проверка корректности экземпляра, генерация событий-подписчиков и т.д.) начнется только после истечения времени задержки.
- На момент отсылки отложенного сообщения должен быть зарегистрирован агент-владелец отсылаемого сообщения.

Глава 10

Пример hello_periodic

Пример hello_periodic демонстрирует использование периодических сообщений.

Пример hello_periodic состоит из одного файла `main.cpp` (стр. 202).

10.1 Что делает пример

В примере работает один агент `a_hello`. В своем обработчике события `evt_start` он организует периодическую отсылку двух сообщений: `msg_hello_no_data` и `msg_hello_with_data`. Сообщение `msg_hello_no_data` циркулирует без объекта-сообщения (т.е. используется *so_4::api::send_msg* с нулевым указателем на объект-сообщение).

Работа примера завершается после 5 циклов получения сообщения `msg_hello_no_data`.

10.2 Разбор файла main.cpp

В своем обработчике события `evt_start` агент `a_hello` определяет время задержки и период повторения периодических сообщений:

```
unsigned int no_data_delay = 2;
unsigned int no_data_period = 1;
unsigned int with_data_delay = 0;
unsigned int with_data_period = 2;

time_t t = time( 0 );

// Сообщаем, что и когда будет происходить.
std::cout << so_query_name() << ".evt_start: "
  << asctime( localtime( &t ) )
  << "\thello_no_data delay: " << no_data_delay
  << "\n\thello_no_data period: " << no_data_period
  << "\n\thello_with_data delay: " << with_data_delay
  << "\n\thello_with_data period: " << with_data_period
  << std::endl;
```

Для каждого из сообщений определяется два значения: задержка (delay) и период (period). Период не может быть нулевым, т.к. нулевой период означает, что это сообщение не периодическое, а однократное. Для периодического сообщения период в миллисекундах указывает время, через которое сообщение будет возникать вновь и вновь. Задержка в миллисекундах указывает время, через которое появится первый экземпляр периодического сообщения. Т.е. можно сделать так, чтобы первый экземпляр сообщения появился через десять секунд, а следующие экземпляры возникали через каждые две секунды. Или чтобы первый экземпляр появился сразу, а последующие через каждые десять секунд.

В данном случае сообщение `msg_hello_no_data` появится с задержкой в 2 секунды, после чего будет повторяться каждую секунду. Сообщение `msg_hello_with_data` появится сразу (без задержки), после чего будет повторяться каждые две секунды.

После определения задержки и периода сообщения отсылаются:

```
so_4::api::send_msg_safely(
  so_query_name(),
  "msg_hello_with_data",
  new msg_hello_with_data(
    std::string( "Sample started at " ) +
    asctime( localtime( &t ) ) ),
  "",
  with_data_delay * 1000, with_data_period * 1000 );

so_4::api::send_msg(
  so_query_name(), "msg_hello_no_data", 0, 0,
  no_data_delay * 1000, no_data_period * 1000 );
```

В качестве содержимого для сообщения `msg_hello_with_data` используется время старта обработчика события `evt_start`. Это сделано, чтобы продемонстрировать, что данные периодического сообщения не изменяются. Более того, в периодическом сообщении постоянно используется один и тот же экземпляр сообщения. Этот экземпляр будет уничтожен только при дерегистрации агента-владельца сообщения.

В обработчике события `evt_hello_no_data` контролируется количество выполненных примером циклов. Если все 5 циклов выполнены, то пример завершает работу:

```
void
a_hello_t::evt_hello_no_data()
  {
    --m_remaining;

    time_t t = time( 0 );
    std::cout << so_query_name() << ".evt_hello_no_data: "
      << asctime( localtime( &t ) )
      << "\tremaining: " << m_remaining << std::endl;

    if( !m_remaining )
      // Отсылаем сообщение для завершения работы.
      so_4::api::send_msg(
```

```
            so_4::rt::sobjectizer_agent_name(),
            "msg_normal_shutdown", 0 );
    }
```

Если бы это не было сделано, пример работал бы бесконечно.

Однажды сделанный периодическим экземпляр сообщения нельзя ни изъять, ни изменить его период. Если сообщение еще раз сделать периодическим, то в системе окажется два независимых периодически возникающих экземпляра сообщения (даже, если у них одинаковый период).

Единственный способ отменить периодическое сообщение — дерегистрировать агента-владельца периодического сообщения.

10.3 Результат работы примера

В результате работы примера на стандартный поток вывода печатается:

```
a_hello.evt_start: Thu Jun 17 08:25:16 2004
  hello_no_data delay: 2
  hello_no_data period: 1
  hello_with_data delay: 0
  hello_with_data period: 2
a_hello.evt_hello_with_data: Thu Jun 17 08:25:16 2004
  Sample started at Thu Jun 17 08:25:16 2004

a_hello.evt_hello_with_data: Thu Jun 17 08:25:18 2004
  Sample started at Thu Jun 17 08:25:16 2004

a_hello.evt_hello_no_data: Thu Jun 17 08:25:18 2004
  remaining: 4
a_hello.evt_hello_no_data: Thu Jun 17 08:25:19 2004
  remaining: 3
a_hello.evt_hello_with_data: Thu Jun 17 08:25:20 2004
  Sample started at Thu Jun 17 08:25:16 2004

a_hello.evt_hello_no_data: Thu Jun 17 08:25:20 2004
  remaining: 2
a_hello.evt_hello_no_data: Thu Jun 17 08:25:21 2004
  remaining: 1
a_hello.evt_hello_with_data: Thu Jun 17 08:25:22 2004
  Sample started at Thu Jun 17 08:25:16 2004

a_hello.evt_hello_no_data: Thu Jun 17 08:25:22 2004
  remaining: 0
```

Видно, как сообщение `msg_hello_with_data` появляется сразу после обработки события `evt_start`, а затем повторяется каждые две секунды. При этом в каждое событие `evt_hello_with_data` передаются одни и те же данные.

Сообщение `msg_hello_no_data` появляется через две секунды и повторяется каждую секунду.

10.4 Резюме

- Для отправки периодического сообщения необходимо в параметре *period* функции *so_4::api::send_msg* (*so_4::api::send_msg_safely*) указать время повторения сообщения в миллисекундах. Реальная обработка сообщения (проверка корректности экземпляра, генерация событий-подписчиков и т.д.) происходит только по истечении очередного периода времени.
- Параметр *delay* функции *so_4::api::send_msg* для периодического сообщения указывает время, через которое появится первый экземпляр сообщения.
- Периодическое сообщение повторяется все время, пока в SObjectizer существует агент-владелец сообщения. Однажды отправленное периодическое сообщение нельзя ни изъять, ни изменить период его генерации.
- Для периодического сообщения используется один и тот же экземпляр сообщения (т.е. данные периодического сообщения не изменяются).
- Повторные отсылки периодического сообщения будут приводить к тому, что в системе будут существовать несколько самостоятельных экземпляров сообщения (даже если у них одинаковый период).
- Единственный способ отменить периодическое сообщение — дерегистрировать агента-владельца периодического сообщения.

Глава 11

Пример dyn_reg

Пример dyn_reg демонстрирует:

- создание динамической кооперации;
- обработку сообщений SObjectizer о регистрации и дерегистрации кооперaций;
- переопределение метода *so_4::rt::agent_t::so_on_deregistration()*.

Пример dyn_reg состоит из одного файла `main.cpp` (стр. 207).

11.1 Что делает пример

Сначала стартует родительский агент `a_parent`, который посредством динамической кооперации создает дочернего агента `a_child`. С помощью сообщения SObjectizer `msg_coop_registered` родительский агент определяет момент регистрации кооперации дочернего агента.

После того, как дочерний агент будет зарегистрирован, родительский агент с помощью отложенного сообщения инициирует дерегистрацию дочернего агента. Через сообщение SObjectizer `msg_coop_deregistered` родительский агент определяет момент дерегистрации дочернего агента, после чего работа примера завершается.

11.2 Разбор файла main.cpp

В конструкторе и деструкторе класса дочернего агента *a_child_t* сделана отладочная печать:

```
a_child_t::a_child_t()
  :  base_type_t( child_name )
  {
    // Информируем о том, что мы созданы.
    std::cout << so_query_name() << " created" << std::endl;
  }

a_child_t::~a_child_t()
  {
```

```
    // Информируем о том, что мы уничтожены.
    std::cout << so_query_name() << " destroyed" << std::endl;
  }
```

С ее помощью будут видны моменты физического создания и уничтожения C++ объекта-агента.

В классе *a_child_t* перекрывается виртуальный метод *so_on_deregistration()*, унаследованный из базового класса *so_4::rt::agent_t*:

```
class a_child_t
  :  public so_4::rt::agent_t
  {
    ...
    virtual void
    so_on_deregistration();
    ...
  };

void
a_child_t::so_on_deregistration()
  {
    // Сообщаем о том, что нас дерегистрируют.
    std::cout << so_query_name() << " deregistered" << std::endl;
  }
```

В данном примере в этом методе осуществляется только отладочная печать для того, чтобы показать, в какой момент метод *so_on_deregistration* будет вызван.

В SObjectizer метод *so_on_deregistration* играет такую же роль для агента, как деструктор класса в C++. SObjectizer Run-Time вызывает этот метод для каждого зарегистрированного агента при дерегистрации. Причем как при обычной (ручной дерегистрации), которая инициируется вызовом *so_4::api::deregister_coop*, так и при автоматической дерегистрации во время завершения работы SObjectizer Run-Time. Т.е. метод *so_on_deregistration* вызывается у агента всегда, если он был зарегистрирован.

Основное назначение метода *so_on_deregistration* — сообщить агенту о том, что он уже дерегистрирован.

Агент может использовать метод *so_on_deregistration* для выполнения какой-либо деинициализации. Например, освобождения ресурсов (закрытие файлов, освобождение памяти, возвращение подключений к БД в пул подключений и т.д.) или информирования кого-либо о своем исчезновении. Т.е. *so_on_deregistration* похож на деструктор C++ класса, но с важным отличием: деструктор вызывается для уничтожаемого объекта и не может быть вызван для одного объекта дважды. Метод же *so_on_deregistration* вызывается для существующего C++ объекта. Более того, C++ объект-агент может остаться существовать и после того, как агент будет дерегистрирован. И даже может быть зарегистрирован вновь — в этом случае для агента опять будут вызваны методы *so_on_subscription* и *so_on_deregistration*.

Поэтому, если агент захватил какой-либо ресурс, который должен быть освобожден при дерегистрации агента, то следует использовать метод *so_on_deregistration*

для освобождения этого ресурса. Но практика использования SObjectizer показала, что агенты, которые повторно регистрируются в SObjectizer (при этом их C++ объекты не уничтожаются), являются крайне редким явлением. Как правило, используются динамически созданные агенты, которые уничтожаются сразу после дерегистрации. В таких случаях гораздо проще переложить ответственность за освобождение ресурсов на деструктор и вспомогательные классы, аналогичные *std::auto_ptr*.

Но метод *so_on_deregistration* сохраняет важность в случае, если агент должен проинформировать других агентов о своем исчезновении. Довольно часто это происходит при использовании идеи "подписчик-издатель", когда какой-то агент (издатель) целенаправленно рассылает свои сообщения всем агентам (подписчикам), которые явно сообщили ему о себе. Если агент-подписчик дерегистрируется и не сообщит об этом агенту-издателю, то издатель будет продолжать рассылать сообщения подписчику. Удобное решение этой проблемы заключается в использовании метода *so_on_deregistration* для информирования агента-издателя об исчезновении подписчика.

В отличие от метода *so_on_subscription*, метод *so_on_deregistration* в классе *so_4::rt::agent_t* не является чистым виртуальным методом. Поэтому его можно переопределять только в классах, которые нуждаются в этом методе.

Агент `a_child` владеет сообщением `msg_say_it_again`, которое используется, чтобы заставить агента `a_child` выполнить печать сообщения “Hello, World!” тогда, когда это потребуется агенту `a_parent`. В обработчике события `evt_say_it_again` агент `a_child` повторяет это сообщение сам себе, но с большой задержкой:

```
void
a_child_t::evt_say_it_again( const so_4::rt::event_data_t & data )
  {
    // Сообщаем, что мы получили сообщение.
    std::cout << so_query_name()
      << ": and again - Hello, World!" << std::endl;

    // Отошлем его самим себе с задержкой. Если нас за это
    // время дерегистрируют, то мы его не получим.
    so_4::api::send_msg( data.agent(), data.msg(), 0,
      so_query_name(), 15000 );
  }
```

Сделано это для того, чтобы показать, что все отложенные сообщения уничтожаются SObjectizer при дерегистрации агента. Т.е., если агент владел какими-либо сообщениями, экземпляры которых на момент дерегистрации агента были отложенными (или периодическими), то при дерегистрации агента все эти экземпляры уничтожаются. При этом не важно, кто вызывал *send_msg* для отсылки этих экземпляров — агент-владелец дерегистрируется, поэтому экземпляры уничтожаются.

Для выполнения своих действий агент `a_parent` использует два отложенных сообщения: `msg_reg_time` и `msg_dereg_time`. Первое отсылается `a_parent` самому себе после регистрации в SObjectizer. Второе отсылается после того, как `a_parent` узнает, что дочерняя кооперация зарегистрирована.

В обработчике события `evt_reg_time` агент `a_parent` создает дочернего агента с помощью т.н. *динамической кооперации*:

```
so_4::rt::dyn_agent_coop_helper_t helper(
  new so_4::rt::dyn_agent_coop_t( new a_child_t() ) );
```

В SObjectizer агенты разделяются на *статические* и *динамические*. SObjectizer ничего не знает о времени жизни статических агентов. Ответственность за создание и уничтожение таких агентов полностью лежит на прикладном программисте. Как правило, статические агенты реализуются переменными, которые можно объявить в функции *main* и обеспечить существование в течение всей работы приложения. Для регистрации статических агентов используются *статические* кооперации — объекты класса *so_4::rt::agent_coop_t*.

Динамические агенты должны создаваться при помощи оператора *new* и регистрироваться при помощи *динамических* кооперации — объектов класса *so_4::rt::dyn_agent_coop_t*, которые также должны быть созданы через *new*. SObjectizer сам уничтожает динамические агенты и их кооперации при дерегистрации.

Как правило, в приложении существуют несколько агентов, которые отвечают за старт и инициализацию приложения, и существуют все время работы приложения. Их проще создать в функции *main* и зарегистрировать как статических агентов. Затем в приложении может появляться и исчезать множество динамических агентов. Для многих из них прикладной программист точно знает, в какой момент они должны возникнуть. Поэтому достаточно просто создавать эти агенты с помощью *new*. Но вот определить, когда для созданного через *new* объекта нужно вызвать *delete*, намного сложнее. Только SObjectizer точно знает, когда агент дерегистрируется. И, поскольку после дерегистрации, как правило, в агенте нет необходимости, то SObjectizer можно поручить уничтожить созданных посредством *new* агентов. Именно это и происходит с динамическими кооперациями в SObjectizer.

Кооперации (статические и динамические) регистрируются в SObjectizer при помощи функции:

```
so_4::ret_code_t
register_coop(
  so_4::rt::agent_coop_t & coop );
```

определенной в пространстве имен *so_4::api*.

Вызов *so_4::api::register_coop* может завершиться неудачно. Для динамической кооперации это будет означать, что SObjectizer не уничтожит ни сам объект кооперации, ни одного агента из кооперации. Ответственность за уничтожение кооперации ложится на программиста. Следовательно, при использовании динамических кооперацией обращения к *so_4::api::register_coop* должны иметь вид:

```
so_4::rt::dyn_agent_coop_t * coop =
  new so_4::rt::dyn_agent_coop_t( ... );
if( so_4::api::register_coop( *coop ) )
  {
    // Кооперация не зарегистрирована и должна быть уничтожена явно.
    delete coop;
  }
```

что чревато ошибками, т.к. легко забыть сделать проверку и вызвать *delete*.

Для облегчения работы с динамическими кооперациями в SObjectizer существует вспомогательный класс *so_4::rt::dyn_agen_coop_helper_t*. Он получает в конструктор указатель на динамически созданный объект *so_4::rt::dyn_agent_coop_t* и пытается в конструкторе зарегистрировать кооперацию. Если это не удается, то кооперация уничтожается в деструкторе *so_4::rt::dyn_agen_coop_helper_t*. Если же кооперация зарегистрирована успешно, то ответственность за уничтожение кооперации перекладывается на SObjectizer.

Метод *result()* класса *so_4::rt::dyn_agen_coop_helper_t* сообщает код возврата функции *so_4::api::register_coop*:

```
if( helper.result() )
  {
    // Ошибка регистрации!
    std::cerr << "register_coop: " << helper.result() << std::endl;
    so_4::api::send_msg( so_4::rt::sobjectizer_agent_name(),
      "msg_alarm_shutdown", 0 );
  }
```

Не нулевой код возврата означает, что кооперация не была зарегистрирована. В этом случае пример должен быть завершен, для чего отсылается сообщение `msg_alarm_shutdown` агента `a_sobjectizer`. В отличие от `msg_normal_shutdown`, сообщение `msg_alarm_shutdown` обрабатывается агентом `a_sobjectizer` на высоком приоритете и приводит к моментальному[1] завершению работы SObjectizer Run-Time.

Когда SObjectizer успешно зарегистрирует кооперацию, он рассылает сообщение `msg_coop_registered`. После успешной дерегистрации SObjectizer отсылает сообщение `msg_coop_deregistered`. Владеет этими сообщениями агент `a_sobjectizer`. В обоих сообщениях есть поле *m_coop_name*, содержащее имя зарегистрированной/дерегистрированной кооперации.

Агент `a_parent` использует сообщение `msg_coop_registered` для определения момента регистрации кооперации с дочерним агентом. Строго говоря, обработка сообщения `msg_coop_registered` является избыточной. Дело в том, что на момент возврата из *so_4::api::register_coop* агент `a_parent` уже знает, зарегистрирована кооперация или нет, т.к. процесс регистрации кооперации в системном словаре SObjectizer происходит синхронно[2]. Поэтому агент `a_parent` получит сообщение `msg_coop_registered` уже после возврата из *so_4::api::register_coop*. Но, если бы кооперацию регистрировал какой-нибудь другой агент и `a_parent` работал бы на другой нити диспетчера, то он мог бы обработать сообщение `msg_coop_registered` еще до завершения *so_4::api::register_coop*.

Совершенно другая ситуация с дерегистрацией кооперации. Возврат из *so_4::api::deregister_coop* не означает, что кооперация дерегистрирована[3]. Поэтому единственным точным способом определения момента дерегистрации кооперации является обработка сообщения `msg_coop_deregistered`.

SObjectizer рассылает сообщения `msg_coop_registered` и `msg_coop_deregistered` одинаковым образом для всех кооперaций. Нельзя подписаться на сообщения

[1] Зависит от типа диспетчера.

[2] Точная последовательность действий при регистрации кооперации описана в разделе 4.5.2 на стр. 40.

[3] Подробнее см. в разделе 4.5.3 на стр. 40.

msg_coop_registered и msg_coop_deregistered, относящиеся только к одной конкретной кооперации. Поэтому в обработчиках событий evt_coop_registered и evt_coop_deregistered проверяется имя кооперации.

Когда агент a_parent обнаруживает, что кооперация с дочерним агентом зарегистрирована, он отсылает сам себе отложенное сообщение для дерегистрации дочернего агента. А также сообщение msg_say_it_again дочернему агенту для того, чтобы заставить дочернего агента выполнить нужные родительскому агенту действия.

Когда агент a_parent получает собственное отложенное сообщение msg_dereg_time, он дерегистрирует дочернего агента:

```
so_4::ret_code_t rc = so_4::api::deregister_coop( child_name );
if( rc )
  {
    std::cerr << "deregister_coop: " << rc << std::endl;
    so_4::api::send_msg( so_4::rt::sobjectizer_agent_name(),
      "msg_alarm_shutdown", 0 );
  }
```

Дерегистрация кооперации в SObjectizer осуществляется при помощи функции:

```
so_4::ret_code_t
deregister_coop(
  const std::string & coop_name );
```

определенной в пространстве имен *so_4::api*.

Дерегистрация кооперации в SObjectizer *всегда* выполняется по *имени кооперации*. В данном примере имя кооперации, как и имя дочернего агента, задается константой *child_name*.

Успешный код возврата функции *so_4::api::deregister_coop* сообщает о том, что SObjectizer нашел данную кооперацию среди зарегистрированных и начал процесс ее дерегистрации. Но это не означает, что на момент возврата из *so_4::api::deregister_coop* кооперация уже зарегистрирована. Быть уверенным в том, что кооперация полностью дерегистрирована, можно только при получении сообщения msg_coop_deregistered с ее именем.

В обработчике события evt_coop_deregistered агент a_parent проверяет, была ли дерегистрирована кооперация с дочерним агентом, и, если была, завершает работу примера.

11.3 Результат работы примера

В результате работы примера на стандартный поток вывода печатается:

```
a_parent: Cooperation registered: a_parent
a_parent: child coop will be registered after 1 sec
a_parent: It's time to register child coop
a_child created
a_parent: Cooperation registered: a_child
a_parent: child coop will be deregistered after 2 sec
```

```
a_child: Hello, World!
a_child: and again - Hello, World!
a_parent: It's time to deregister child coop
a_child deregistered
a_child destroyed
a_parent: Cooperation deregistered: a_child
a_parent: Work finished
```

Первая печать:

```
a_parent: Cooperation registered: a_parent
```

выполняется агентом `a_parent` в обработчике сообщения `msg_coop_registered`. Агент получил сообщение о том, что была зарегистрирована его собственная кооперация. Причем событие `evt_coop_registered` было обработано раньше события `evt_start`, т.к. у `evt_coop_registered` больший приоритет.

Следующая печать выполняется уже из обработчика события `evt_start`:

```
a_parent: child coop will be registered after 1 sec
```

а следующие две строки из обработчика `evt_reg_time`:

```
a_parent: It's time to register child coop
a_child created
```

При этом видно, когда агент `a_child` был физически создан.

Затем агент `a_parent` узнает, что дочерний агент был зарегистрирован:

```
a_parent: Cooperation registered: a_child
a_parent: child coop will be deregistered after 2 sec
```

После чего дочерний агент обрабатывает два своих события (`evt_start`, `evt_say_it_again`):

```
a_child: Hello, World!
a_child: and again - Hello, World!
```

Далее агент `a_parent` дерегистрирует кооперацию, а дочерний агент дерегистрируется и физически уничтожается до того, как агент `a_parent` узнает о дерегистрации кооперации:

```
a_parent: It's time to deregister child coop
a_child deregistered
a_child destroyed
a_parent: Cooperation deregistered: a_child
a_parent: Work finished
```

Результат работы примера подтверждает, что агент `a_child` обработал событие `evt_say_it_again` только один раз.

11.4 Резюме

- Для SObjectizer кооперации делятся на статические и динамические. За время жизни статических кооперация и агентов в них отвечает прикладной программист. Уничтожением динамических кооперация и агентов в них после дерегистрации занимается SObjectizer.
- Статические кооперации описываются с помощью объектов типа *so_4::rt::agent_coop_t*.
- Динамические кооперации описываются с помощью объектов типа *so_4::rt::dyn_agent_coop_t*, производного от *so_4::rt::agent_coop_t*.
- Все агенты в динамической кооперации должны быть созданы посредством оператора *new*.
- Все объекты типа *so_4::rt::dyn_agent_coop_t* должны создаваться посредством оператора *new*.
- Регистрация кооперации выполняется функцией *so_4::api::register_coop*:

```
so_4::ret_code_t
register_coop(
  so_4::rt::agent_coop_t & coop );
```

  Нулевой код возврата означает, что кооперация успешно зарегистрирована.
- Если динамическая кооперация не была успешно зарегистрирована, то за уничтожение объекта типа *so_4::rt::dyn_agent_coop_t* отвечает прикладной программист. Для облегчения контроля за успешностью регистрации динамических кооперация существует класс *so_4::rt::dyn_agent_coop_helper_t*, который пытается зарегистрировать динамическую кооперацию в конструкторе. Если это не удалось, то объект динамической кооперации будет уничтожен в деструкторе. Получить результат регистрации кооперации можно с помощью метода *result* этого класса.
- После окончательной регистрации кооперации SObjectizer рассылает сообщение `msg_coop_registered`, в атрибуте *m_coop_name* которого содержится имя зарегистрированной кооперации. Этим сообщением владеет агент `a_sobjectizer` (точное имя этого агента возвращается функцией *so_4::rt::sobjectizer_agent_name()*).
- Дерегистрация кооперации осуществляется при помощи функции *so_4::api::deregister_coop*:

```
so_4::ret_code_t
deregister_coop(
  const std::string & coop_name );
```

- Дерегистрация кооперации в SObjectizer *всегда* выполняется по *имени кооперации*.

- Успешный код возврата функции *so_4::api::deregister_coop* сообщает о том, что SObjectizer нашел данную кооперацию среди зарегистрированных и начал процесс ее дерегистрации. Но это не означает, что на момент возврата из *so_4::api::deregister_coop* кооперация уже зарегистрирована. Быть уверенным в том, что кооперация полностью дерегистрирована, можно только при получении сообщения `msg_coop_deregistered` с ее именем.

- В базовом классе *so_4::rt::agent_name_t* определен виртуальный метод *so_on_deregistration*, который автоматически вызывается SObjectizer для каждого агента при дерегистрации.

- Метод *so_on_deregistration* может быть переопределен агентом для выполнения действий по своей деинициализации.

- Метод *so_on_deregistration* не является чистым виртуальным методом, поэтому его не обязательно переопределять в каждом классе агента (в отличие от методов *so_query_type*, *so_on_subscription*).

- После окончательной дерегистрации кооперации SObjectizer рассылает сообщение `msg_coop_deregistered`, в атрибуте *m_coop_name* которого содержится имя дерегистрированной кооперации. Этим сообщением владеет агент `a_sobjectizer` (точное имя этого агента возвращается функцией *so_4::rt::sobjectizer_agent_name()*). Получение сообщения `msg_coop_deregistered` означает, что для всех агентов кооперации был вызван метод *so_on_deregistration*. Для динамических коопераций это также означает, что все агенты кооперации и сам объект кооперации уже физически уничтожены.

- SObjectizer рассылает сообщения `msg_coop_registered` и `msg_coop_deregistered` одинаковым образом для всех коопераций. Нельзя подписаться на сообщения `msg_coop_registered` и `msg_coop_deregistered`, относящиеся только к одной конкретной кооперации.

Глава 12

Пример chstate

Пример chstate демонстрирует:

- механизм смены состояний агента;
- поведение SObjectizer при обработке событий с одним инцидентом и разными приоритетами в одном состоянии агента;
- возможность использования обработчиков входа/выхода в/из состояния;
- возможность рассылки SObjectizer специальных сообщений при смене состояния агента и способ обработки этих сообщений;
- возможность отслеживания смены состояния агента при помощи т.н. “слушателей” состояния.

Пример chstate состоит из одного файла `main.cpp` (стр. 214).

12.1 Что делает пример

В примере создается один агент `a_main`, который владеет периодическим сообщением `msg_1` и имеет состояния: `st_1`, `st_2`, `st_3`, `st_4`, `st_shutdown`. В каждом из состояний агент обрабатывает сообщение `msg_1` с различными приоритетами, последовательно переводя агента из одного состояния в другое. После входа в состояние `st_shutdown` работа примера завершается.

Агент `a_main` использует механизм *обработчиков входа/выхода в/из состояния* для: a) информирования о смене состояния и b) для инициирования завершения работы примера при входе в состояние `st_shutdown`.

В состоянии `st_3` агент `a_main` пытается обработать сообщение `msg_1` двумя событиями с одинаковым приоритетом (`evt_msg_1_st_3_pri_1_one`, `evt_msg_1_st_3_pri_1_two`). SObjectizer рассматривает такое поведение как непредсказуемое и отказывается обрабатывать оба события.

Агент `a_main` заставляет SObjectizer рассылать специальное сообщение `so_msg_state` при смене своего состояния. Агент `a_main` сам обрабатывает это сообщение.

Агенту `a_main` назначается два *слушателя* состояния, которые печатают на стандартный поток вывода информацию о текущем состоянии агента.

12.2 Разбор файла main.cpp

В файле `main.cpp` находится код, затрагивающий разные стороны работы с состояниями агента:

- смена состояния агента, обработка событий с одним инцидентом в разных состояниях;
- обработчики входа/выхода в/из состояния;
- использование специального сообщения `so_msg_state`;
- использование слушателей состояния агента.

Все эти аспекты касаются механизма поддержки понятия состояния агента в SObjectizer[1], но являются достаточно самостоятельными. Поэтому рассмотрение кода в файле `main.cpp` ведется в четырех последующих подразделах независимо друг от друга.

12.2.1 Смена состояния агента и обработка событий в состояниях

Агент `a_main` реализуется классом агента `a_main_t`, в котором определены:

- сообщение `msg_1`;
- событие `evt_start`, которое запускает периодическое сообщение `msg_1`. Инцидентом события `evt_start` является сообщение `a_sobjectizer.msg_start`;
- события `evt_msg_1_st_1`, `evt_msg_1_st_2_pri_1`, `evt_msg_1_st_2_pri_0`, `evt_msg_1_st_3_pri_1_one`, `evt_msg_1_st_3_pri_1_two`, `evt_msg_1_st_3_pri_0`, `evt_msg_1_st_4`. Инцидентом этих событий является собственное сообщение `msg_1` агента `a_main`;
- состояния `st_1`, `st_2`, `st_3`, `st_4`, `st_shutdown`.

Также в классе `a_main_t` определено событие `evt_self_state_notify` и обработчики входа/выхода в состояние (методы *`on_enter_state`*, *`on_exit_state`*, *`on_enter_shutdown`*), которые подробно рассматриваются в следующих подразделах.

Итак, в классе агента `a_main_t` определено пять состояний: `st_1`, `st_2`, `st_3`, `st_4`, `st_shutdown`. Все они перечислены в описании класса агента для SObjectizer при помощи макросов *`SOL4_STATE_START`*, *`SOL4_STATE_FINISH`*. Это означает, что любой агент класса `a_main_t` может находиться в одном их этих состояний.

Но в каком состоянии окажется агент `a_main` после своей регистрации? SObjectizer использует для определения начального состояния следующие правила:

- если в классе агента использован макрос *`SOL4_INITIAL_STATE`*, то в качестве начального состояния используется указанное в этом макросе имя;
- если макрос *`SOL4_INITIAL_STATE`* не использовался (как в данном примере), то в качестве начального состояния берется первое из перечисленных в описании класса агента для SObjectizer состояние.

[1] См. 2.1.1 на стр. 14, 3.1.4 на стр. 29

В данном случае макрос *SOL4_INITIAL_STATE* не использовался, а первым было описано состояние `st_1`. Поэтому после регистрации агент `a_main` окажется в состоянии `st_1`.

Обработчик события `evt_start` запускает периодическое сообщение `msg_1`. Далее в примере работают события, демонстрирующие как SObjectizer обрабатывает события с одним инцидентом в разных состояниях.

Состояние агента в SObjectizer определяет набор событий, разрешенных к обработке в данном состоянии. При отсылке сообщения (возникновении очередного экземпляра периодического сообщения) SObjectizer формирует специальные заявки на запуск **всех** событий, подписанных на данное сообщение. Заявки передаются диспетчеру для выполнения, и диспетчер определяет, в какой момент времени какую заявку выполнить[2]. В данном случае при возникновении каждого экземпляра сообщения `msg_1` диспетчеру ставятся заявки на обработку событий `evt_msg_1_st_1`, `evt_msg_1_st_2_pri_1`, `evt_msg_1_st_2_pri_0`, `evt_msg_1_st_3_pri_1_one`, `evt_msg_1_st_3_pri_1_two`, `evt_msg_1_st_3_pri_0`, `evt_msg_1_st_4`.

Каждый диспетчер сам определяет, в какой момент времени он запустит обработчики событий на выполнение. Диспетчер с одной рабочей нитью просто поместит все заявки в очередь заявок и запустит их, когда до них дойдет очередь. Диспетчер, который использует отдельную нить для каждого приоритета, может распределить заявки по разным очередям в зависимости от их приоритета.

Но любой диспетчер перед запуском обработчика события обязан проверить, разрешена ли обработка события в текущем состоянии агента. И если обнаруживается, что событие запрещено к обработке в текущем состоянии, то заявка выбрасывается, и обработчик не запускается. Именно поэтому в состоянии `st_1` запускается только обработчик события `evt_msg_1_st_1`; в состоянии `st_2` — обработчики `evt_msg_1_st_2_pri_1`, `evt_msg_1_st_2_pri_0`; и т.д.

Ключевым моментом в механизме диспетчеризации SObjectizer является проверка состояния агента **перед** запуском заявки на выполнение. Это сделано из-за того, что в большинстве реальных приложений требуется выстраивать все заявки для одного агента в очередь и последовательно обрабатывать их. Поэтому вполне может случиться, что на момент появления заявки агент будет находиться в состоянии, которое запрещает обрабатывать событие. Но, пока до заявки дойдет очередь, агент может быть переведен предшествующими заявками в нужное состояние. Если бы проверка возможности обработки события осуществлялась в момент выставления заявки, то заявка была бы потеряна[3].

Если в каждом состоянии агента один инцидент приводит к возникновению только одного разрешенного к обработке события, то для диспетчера все просто и очевид-

[2] Подробнее см. 4.3 на стр. 36, 4.3.1 на стр. 36, 4.5.4 на стр. 41

[3] Применение проверки состояния агента в момент выставления заявки диспетчеру сильно затруднило бы программирование в SObjectizer. Если бы все события для агента отрабатывали мгновенно, то можно было бы использовать данный механизм. Но, т.к. заявки проходят через очереди, то при таком механизме нельзя реализовать простую задачу:
- агент `A` находится в состоянии `a1`;
- ему отсылаются два сообщения подряд. В результате обработки первого агент должен перейти в состояние `a2`, в котором разрешено к обработке второе сообщение.
Если бы обработка первого сообщения была выполнена мгновенно при отсылке, то корректно было бы обработано и второе сообщение. Но поскольку в большинстве случаев обработчику первого сообщения придется простоять некоторое время в очереди диспетчера, то второе сообщение вообще не будет обработано.

но — нужно выбрать только одно разрешенное событие и запустить его на обработку. Но что происходит, если в одном состоянии оказываются разрешенными к обработке два события, порожденных одним инцидентом? Если эти события имеют разные приоритеты, то проблем не возникает. Диспетчер вызывает их обработчики в порядке убывания приоритета. Именно это происходит с событиями `evt_msg_1_st_2_pri_1`, `evt_msg_1_st_2_pri_0`: сначала отрабатывает событие `evt_msg_1_st_2_pri_1`, поскольку у него больший, чем у `evt_msg_1_st_2_pri_0`, приоритет.

Поскольку события с разными приоритетами обрабатываются последовательно, то вполне может быть так, что обработчик события с более высоким приоритетом сменит состояние агента до того, как запустится обработчик с низким приоритетом. В этом случае второй обработчик не будет запущен — когда подойдет его очередь, то агент будет находиться уже в другом состоянии. Например, если обработчик `evt_msg_1_st_2_pri_1` сменит состояние агента на `st_3`, то событие `evt_msg_1_st_2_pri_0` не отработает[4].

Но что произойдет, если несколько событий с одним инцидентом имеют одинаковый приоритет? В этом случае SObjectizer считает систему непредсказуемой. Действительно, пусть есть два события: `e1` и `e2`, подписанные с одинаковым приоритетом на одно и то же сообщение и разрешенные к обработке в одном состоянии. Пусть обработчик события `e1` переводит агента в другое состояние. Если первым запустится обработчик события `e1`, то обработчик события `e2` не отработает. Если же первым запустится обработчик события `e2`, то затем отработает и `e1`.

Единственным критерием упорядочения обработчиков событий в SObjectizer является приоритет события. Если события имеют одинаковый приоритет, то SObjectizer не знает, в каком порядке их обрабатывать, а порядок может иметь для приложение важное значение. Поэтому, если SObjectizer встречает несколько равноприоритетных обработчиков событий с одним инцидентом в одном состоянии, то SObjectizer считает ситуацию неопределенной, а систему — непредсказуемой. И ни одно из событий не обрабатывается. Именно поэтому в состоянии `st_3` отрабатывает событие `evt_msg_1_st_3_pri_0`, а события `evt_msg_1_st_3_pri_1_one` и `evt_msg_1_st_3_pri_1_two` игнорируются.

Смена состояния агента выполняется в SObjectizer при помощи унаследованного из базового класса *`so_4::rt::agent_t`* метода *`so_change_state`*, имеющего формат:

```
so_4::ret_code_t
so_change_state(
  /*!
    Имя нового состояния агента.
    Имя должно быть определено в описании класса агента.
  */
  const std::string & state_name,
  /*!
    Нужно ли вызывать обработчики выхода/входа
    в состояние, если агент уже находится в этом
```

[4]Такое поведение можно гарантировать только на диспетчерах, которые выстраивают все заявки для одного агента в одну очередь. Но если бы использовался диспетчер, который расставляет все заявки в очереди по приоритетам независимо от агента, то ситуация может драматическим образом измениться. Например, событие с меньшим приоритетом может отработать раньше события с высоким приоритетом, если нить обработки заявок высокого приоритета сильно загружена. К счастью, необходимость использования таких диспетчеров возникает не часто и в специфических задачах (например, в системах реального времени).

```
    состоянии. По умолчанию -- не нужно.
  */
  bool always_call_handlers = false );
```

Данный метод может быть вызван только внутри обработчика какого-либо события этого агента. Проще говоря, только агент может изменить свое состояние и только во время обработки одного из своих событий. В параметре *state_name* задается имя нового состояния агента. Это состояние должно быть описано с помощью макросов *SOL4_STATE_START*, *SOL4_STATE_FINISH*. Параметр *always_call_handlers* подробнее рассматривается в 12.2.2.

Для получения имени текущего состояния агента предназначена функция:

```
so_4::ret_code_t
query_agent_state(
  /*! Имя агента, состояние которого необходимо узнать. */
  const std::string & agent_name,
  /*! Приемник имени состояния агента. */
  std::string & state_name );
```

описанная в пространстве имен *so_4::api*. В отличие от метода *so_4::rt::agent_t::so_change_state* функцию *query_agent_state* может вызывать не только сам агент. Но если ее использует не сам агент, то следует с осторожностью относиться к возвращаемому имени состояния: агент может перейти в другое состояние перед самым возвратом из *query_agent_state*.

12.2.2 Обработчики входа/выхода в/из состояния

Во многих случаях смена состояния агента означает нечто большее, чем изменение списка разрешенных к обработке событий. Как правило, смена состояния означает изменение поддерживаемой агентом функциональности. Например, после своей регистрации агент находится в состоянии `st_initial` и занимается только собственной инициализацией. В этом состоянии он не выполняет никакой полезной работы. Затем агент переходит в `st_normal` и начинает обрабатывать прикладные запросы. Если прикладных запросов накапливается слишком много, то агент может перейти в состояние `st_busy`, после чего вернуться в `st_normal`.

Часто при входе (или выходе) в состояние требуется выполнять какие-либо действия, касающиеся именно смены состояния. Например, когда агент входит в состояние `st_normal`, он может сообщить окружающим его агентам, что он готов к выполнению их запросов, для чего отсылает специальное сообщение. При переходе в состояние `st_busy` агенту нужно выставить флаг, ограничивающий функциональность агента, а при переходе из `st_busy` в любое другое состояние — снять этот флаг.

Для упрощения выполнения действий, связанных с переходом из одного состояния в другое, SObjectizer поддерживает т.н. *обработчики входа/выхода в/из состояние*. Обработчиком может быть открытый (*public*) нестатический метод класса агента формата:

```
void
enter_exit_handler(
  /*! Имя состояния агента. */
  const std::string & state_name );
```

Обработчик входа в состояние агента задается при помощи макроса *SOL4_STATE_ON_ENTER(method_name)*, а обработчик выхода — при помощи макроса *SOL4_STATE_ON_EXIT(method_name)*. Эти макросы указываются в описании состояния агента между макросами *SOL4_STATE_START*, *SOL4_STATE_FINISH*:

```
SOL4_STATE_START( st_shutdown )
  SOL4_STATE_EVENT( evt_self_state_notify )

  SOL4_STATE_ON_ENTER( on_enter_state )
  SOL4_STATE_ON_ENTER( on_enter_st_shutdown )
SOL4_STATE_FINISH()
```

Один и тот же метод может быть назначен как обработчик входа/выхода в несколько состояний агента. Более того, один обработчик может быть назначен для одного состояния как обработчик входа, а в другом состоянии как обработчик выхода (хотя вряд ли в этом есть какой-либо смысл).

В одном состоянии может быть определено несколько обработчиков входа/выхода. В этом случае SObjectizer не определяет порядка, в котором обработчики будут вызваны. Гарантируется только, что они будут вызваны все. Поэтому, если при входе/выходе требуется выполнить несколько действий в строгом порядке, следует написать отдельного обработчика и назначить его состоянию.

В 4.5.5 на стр. 42 описывается точная процедура смены состояния агента. В соответствии с ней при вызове *so_change_state* для агента атомарно изменяется имя текущего состояния, и только затем вызываются обработчики. Сначала вызываются обработчики выхода из старого состояния, затем обработчики входа в новое состояние. Поэтому не рекомендуется в обработчике входа/выхода изменять состояние агента. Например, пусть есть состояние `s0`, в котором задан обработчик *exit_s0*, состояние `s1`, в котором заданы обработчики *enter_s1* и *exit_s1*, и есть состояние `s2` с обработчиками *enter_s2*, *exit_s2*. При смене состояния агента с `s0` на `s1` обработчик *exit_s0* переводит агента в состояние `s2`. Тогда окажется, что обработчики будут вызваны в следующем порядке:

1. *exit_s0* (агент в состоянии `s1`);
2. *exit_s1* (агент в состоянии `s2`);
3. *enter_s2* (агент в состоянии `s2`);
4. *enter_s1* (агент в состоянии `s2`);

т.е. обработчик выхода из `s1` будет вызван до обработчика входа в `s1`. Происходит это из-за того, что все обработчики вызываются внутри *so_change_state* (срабатывает рекурсия обращений к *so_change_state*).

При вызове обработчика выхода из состояния в параметре *state_name* передается имя старого состояния. При вызове обработчика входа в состояние в параметре *state_name* передается имя нового состояния агента.

Важно отметить, что при регистрации агента обработчики входа в начальное состояние агента **не вызываются**. Например, в агенте `a_hello` задан обработчик входа в состояние `st_1`. При регистрации агента `a_hello` этот обработчик вызван не будет. Но он будет обычным образом вызван, если агент сменит свое состояние на `st_1` уже в процессе своей работы.

SObjectizer допускает, чтобы при вызове *so_change_state* было показано имя текущего состояния агента. В этом случае состояние агента не изменяется, а обработчики входа/выхода вызываются, если параметр *always_call_handlers* равен *true*. По умолчанию он равен *false*, поэтому, если при вызове *so_change_state* с именем текущего состояния агента параметр *always_call_handlers* опущен, обработчики входа/выхода не вызываются.

В примере chstate для агента a_main определены три обработчика: *on_enter_state*, *on_exit_state* и *on_enter_st_shutdown*. Два первых обработчика служат только для демонстрационных целей. Их главная задача состоит в печати значения параметра *state_name*.

Обработчик *on_enter_st_shutdown* выполняет реальную работу — завершает пример, как только агент a_main переходит в состояние st_shutdown. При этом обработчик *on_enter_st_shutdown* демонстрирует основное назначение обработчиков входа/выхода в SObjectizer: выполнение действий, связанных с состоянием, вне зависимости от того, как произошел переход в это состояние. Действительно, обработчик *on_enter_st_shutdown* будет вызываться при входе в st_shutdown всегда, из какого бы состояния не был совершен переход.

12.2.3 Специальное сообщение so_msg_state

Иногда возникает необходимость "прослушивать" состояние некоторого агента. Например, для мониторинговых или отладочных целей может потребоваться контролировать моменты смены состояния агента. SObjectizer предоставляет для этих целей два механизма. Первый, описанный в данном подразделе, основан на том, что SObjectizer может рассылать специальное сообщение об изменении состояния от имени агента, чье состояние изменилось. Второй механизм, подробно описываемый в следующем подразделе, использует специальные объекты-слушатели (listeners).

Если в описании класса агента указать макрос *SOL4_CHANGE_STATE_NOTIFY()*, то SObjectizer неявно добавляет к классу агента сообщение so_msg_state, которое реализуется типом:

```
struct so_msg_state
  {
    so_msg_state();
    so_msg_state(
      const std::string & agent_name,
      const std::string & state_name );
    ~so_msg_state();

    //! Имя агента, чье состояние изменилось.
    std::string m_agent;
    //! Имя нового состояния агента.
    std::string m_state;

    static bool
    check( const so_msg_state * msg );
  };
```

В примере chstate показана обработка данного сообщения с помощью события `evt_self_state_notify`.

Данный механизм был самым первым механизмом уведомлений о смене состояния агента в SObjectizer. Он уже устарел и оставлен в SObjectizer из соображений совместимости. Для мониторинга смены состояния агента рекомендуется использовать механизм слушателей, подробно описываемый в следующем подразделе.

12.2.4 "Слушатели" состояния агента

В версии 4.2.6 в SObjectizer было добавлено понятие *слушателя (listener) состояния агента*. Слушатель — это объект, реализующий интерфейс *so_4::rt::agent_state_listener_t*:

```
class agent_state_listener_t
  : private cpp_util_2::nocopy_t
  {
  public :
    virtual ~agent_state_listener_t();

    //! Вызывается после занесения слушателя в список
    //! слушателей агента.
    /*! В базовом классе ничего не делает. */
    virtual void
    stored( agent_t & agent );

    //! Вызывается после успешной смены состояния агента.
    virtual void
    changed(
      //! Агент, чье состояние изменилось.
      agent_t & agent,
      //! Имя текущего состояния агента.
      const std::string & state_name ) = 0;
  };
```

Объекты-слушатели назначаются агенту при помощи унаследованных из *so_4::rt::agent_t* методов:

```
void
so_add_nondestroyable_listener(
  agent_state_listener_t & l );

void
so_add_destroyable_listener(
  agent_state_listener_t * l );
```

Метод *so_add_nondestroyable_listener* добавляет к агенту слушателя, время жизни которого агент не контролирует. Т.е. за уничтожение такого слушателя ответственность несет прикладной программист. Метод *so_add_destroyable_listener* используется, если время жизни этого слушателя будет контролироваться агентом. Слушатель

будет уничтожен либо при дерегистрации агента, либо в деструкторе агента, если агент не был зарегистрирован.

Пример chstate демонстрирует, что для создания своего слушателя необходимо создать класс, производный от *so_4::rt::agent_state_listener_t*:

```
class sample_listener_t
  : public so_4::rt::agent_state_listener_t
  {
  private :
    // Имя данного слушателя.
    std::string m_name;
  public :
    sample_listener_t(
      const std::string & name )
      : m_name( name )
      {}
    virtual ~sample_listener_t()
      {
        // Отладочная печать покажет, когда слушатель будет уничтожен.
        std::cout << "listener destroyed: " << m_name << std::endl;
      }

    virtual void
    changed(
      so_4::rt::agent_t & agent,
      const std::string & state_name )
      {
        std::cout << m_name << ": state of '"
        << agent.so_query_name() << "' is '"
        << state_name << "'" << std::endl;
      }
  };
```

затем создать объект слушателя и связать его с агентом. В примере chstate создаются два слушателя:

```
sample_listener_t l( "static_listener" );
a_main.so_add_nondestroyable_listener( l );

a_main.so_add_destroyable_listener(
  new sample_listener_t( "dynamic_listener" ) );
```

Сделано это для того, чтобы показать, когда будет уничтожен статический слушатель (время жизни которого не контролируется SObjectizer), и когда будет уничтожен динамический слушатель (время жизни которого контролируется SObjectizer).

Когда агенту назначается слушатель, агент вызывает у объекта-слушателя метод *stored()*, но не сообщает имя своего текущего состояния. Если слушателю это необходимо, то он может узнать имя текущего состояния агента с помощью *so_4::api::query_agent_state*. Но здесь есть тонкий момент: если слуша-

тель назначается агенту, когда агент еще не зарегистрирован в системе, то слушатель не сможет узнать имя начального состояния агента, т.к. в момент вызова *agent_state_listener_t::stored()* состояние агенту еще не назначено. А при регистрации агента в системе процедура смены состояния не выполняется — SObjectizer просто связывает с агентом имя его начального состояния.

После того, как слушатель назначен агенту, метод *changed* вызывается после смены состояния агента. Причем вызывается после запуска всех обработчиков входа/выхода. И в метод *changed* передается имя состояния, в котором агент оказался в результате всех переходов. В случае, если какой-либо обработчик еще раз сменит состояние агента, это приведет к тому, что слушатель даже не узнает об этой смене состояния. Например, если агент переходит в состояние `A`, но обработчик входа в состояние `A` сменит его на `B`, то слушателю будет сообщено, что агент находится в состоянии `B`. Причем сделано это будет дважды — первый раз при вызове *so_change_state("B")* в обработчике входа в состояние `A`. Второй раз — при вызове *so_change_state("A")*, но уже после того, как отработает обработчик входа в состояние `A`. К этому моменту агент будет находиться в состоянии `B`. Такое поведение выбрано специально — обработчики входа/выхода в/из состояние имеют приоритет над слушателями состояний, поэтому сначала запускаются именно обработчики состояния и только затем, когда состояние агента будет установлено, слушатели.

Агенту может быть назначено любое количество слушателей. SObjectizer не определяет порядок, в котором слушатели будут вызываться при смене состояния.

Список слушателей агента очищается после каждой дерегистрации агента. Сделано это для того, чтобы слушателей могли устанавливать *свойства объекта*[5] в своем методе *agent_traits_t::init()*. В этом случае свойство, которое уже нельзя изъять у агента, при каждой регистрации агента будет добавлять слушателя, не заботясь о том, что такой слушатель агенту уже был добавлен ранее. Ведь агент при дерегистрации удалит всех слушателей.

12.3 Результат работы примера

В результате работы примера на стандартный поток вывода печатается:

```
a_main.evt_start
a_main.evt_msg_1_st_1
        exit state: st_1
        enter state: st_2
static_listener: state of 'a_main' is 'st_2'
dynamic_listener: state of 'a_main' is 'st_2'
so_msg_state: agent: a_main, state: st_2
a_main.evt_msg_1_st_2_pri_1
a_main.evt_msg_1_st_2_pri_0
        exit state: st_2
        enter state: st_3
static_listener: state of 'a_main' is 'st_3'
dynamic_listener: state of 'a_main' is 'st_3'
```

[5] Понятие свойства агента и класс *so_4::rt::agent_traits_t* подробнее рассматриваются в примере destroyable_traits (см. 20 на стр. 184).

```
so_msg_state: agent: a_main, state: st_3
so_4\rt\impl\event_data_only_one_of.cpp:116: System is not predictable!
        Agent: a_main
        State: st_3
        Incident: a_main.msg_1
a_main.evt_msg_1_st_3_pri_0
        exit state: st_3
        enter state: st_4
static_listener: state of 'a_main' is 'st_4'
dynamic_listener: state of 'a_main' is 'st_4'
so_msg_state: agent: a_main, state: st_4
a_main.evt_msg_1_st_4
        exit state: st_4
System is shutting down...
        enter state: st_shutdown
static_listener: state of 'a_main' is 'st_shutdown'
dynamic_listener: state of 'a_main' is 'st_shutdown'
so_msg_state: agent: a_main, state: st_shutdown
listener destroyed: dynamic_listener
successful finish
listener destroyed: static_listener
```

Следует обратить внимание на следующие моменты:

- печать, выполняемую обработчиками входа/выхода в/из состояние. Во-первых, нет печати о входе агента в состояние `st_1`, поскольку это начальное состояние агента. Во-вторых, обработчики входа/выхода отрабатывают до того, как запустятся слушатели состояний и обработчик события `evt_self_state_notify`;
- обработчик `evt_msg_1_st_2_pri_1` запускается до обработчика `evt_msg_1_st_2_pri_0`, поскольку у события `evt_msg_1_st_2_pri_1` более высокий приоритет;
- вместо запуска обработчиков `evt_msg_1_st_3_pri_1_one` и `evt_msg_1_st_3_pri_1_two` SObjectizer печатает сообщение об ошибке:

  ```
  so_4\rt\impl\event_data_only_one_of.cpp:116: System is not predictable!
          Agent: a_main
          State: st_3
          Incident: a_main.msg_1
  ```

- динамический слушатель был уничтожен при дерегистрации агента перед возвратом из *so_4::api::start*. Статический слушатель был уничтожен при возврате из функции *main*.

12.4 Резюме

- Для определения состояния, в котором агент окажется после регистрации, SObjectizer использует следующие правила:

  - если в классе агента использован макрос *SO_4_INITIAL_STATE*, то в качестве начального состояния используется указанное в этом макросе имя;
  - если макрос *SO_4_INITIAL_STATE* не использовался (как в данном примере), то в качестве начального состояния берется первое из перечисленных в описании класса агента для SObjectizer состояние.

- При появлении любого сообщения SObjectizer генерирует диспетчеру заявки на запуск обработчиков **всех** событий, у которых это сообщение является инцидентом вне зависимости от того, в каком состоянии находится агент сейчас. Диспетчер проверяет, разрешено ли событие к обработке в текущем состоянии агента **непосредственно перед** запуском очередной заявки на обработку.

- Если два события одного агента имеют одного инцидента и разрешены к обработке в одном состоянии, то:

  - если события имеют разные приоритеты, обработчики событий запускаются диспетчером в порядке убывания приоритета. Если при этом обработчик с более высоким приоритетом сменит состояние агента, то второй обработчик будет запущен только, если он разрешен к обработке в новом состоянии агента;
  - если события имеют одинаковый приоритет, SObjectizer печатает сообщение о непредсказуемости системы и игнорирует оба события.

- Смена состояния агента осуществляется унаследованным из *so_4::rt::agent_t* методом:

```
so_4::ret_code_t
so_change_state(
  const std::string & state_name,
  bool always_call_handlers = false );
```

  который может быть вызван только агентом и только при обработке своего события.

- Определить текущее состояние агента можно с помощью функции:

```
so_4::ret_code_t
query_agent_state(
  /*! Имя агента, состояние которого необходимо узнать. */
  const std::string & agent_name,
  /*! Приемник имени состояния агента. */
  std::string & state_name );
```

  которая описана в пространстве имен *so_4::api*. В отличие от метода *so_4::rt::agent_t::so_change_state* функцию *query_agent_state* может вызывать не только сам агент. Но если ее использует не сам агент, то следует с осторожностью относиться к возвращаемому имени состояния: агент может перейти в другое состояние перед самым возвратом из *query_agent_state*.

- Состоянию могут быть назначены *обработчики входа/выхода в/из состояние*. Обработчики запускаются SObjectizer вне зависимости от того, из какого состояния в какое состояние осуществляется переход.

- Обработчик входа/выхода должен быть открытым (*public*) нестатическим методом класса агента формата:

```
void
enter_exit_handler(
  /*! Имя состояния агента. */
  const std::string & state_name );
```

- Обработчик входа в состояние агента задается при помощи макроса *SOL4_STATE_ON_ENTER(method_name)*, а обработчик выхода — при помощи макроса *SOL4_STATE_ON_EXIT(method_name)*. Эти макросы указываются в описании состояния агента между макросами *SOL4_STATE_START*, *SOL4_STATE_FINISH*:

```
SOL4_STATE_START( st_shutdown )
  SOL4_STATE_EVENT( evt_self_state_notify )

  SOL4_STATE_ON_ENTER( on_enter_state )
  SOL4_STATE_ON_ENTER( on_enter_st_shutdown )
SOL4_STATE_FINISH()
```

- Один и тот же метод может быть назначен как обработчик входа/выхода в несколько состояний агента. Более того, один обработчик может быть назначен для одного состояния как обработчик входа, а в другом состоянии как обработчик выхода.

- В одном состоянии может быть определено несколько обработчиков входа/выхода. В этом случае SObjectizer не определяет порядка, в котором обработчики будут вызваны. Гарантируется только, что они будут вызваны все.

- При вызове обработчика выхода из состояния в параметре *state_name* передается имя старого состояния. При вызове обработчика входа в состояние в параметре *state_name* передается имя нового состояния агента.

- При регистрации агента SObjectizer не вызывает обработчик входа в начальное состояние.

- Не рекомендуется изменять состояние агента в обработчике входа/выхода, т.к. это может привести к неожиданному порядку запуска обработчиков входа/выхода и даже к их бесконечной рекурсии.

- В случае необходимости мониторинга текущего состояния агента следует использовать т.н. *слушателей состояния*. Слушателем состояния является объект класса, реализующего интерфейс *so_4::rt::agent_state_listener_t*, назначенный агенту с помощью унаследованных методов:

```
void
so_add_nondestroyable_listener(
  agent_state_listener_t & l );

void
so_add_destroyable_listener(
  agent_state_listener_t * l );
```

- Слушатели состояния делятся на статические (время жизни которых не контролируется агентом) и динамические (за их уничтожение отвечает агент). Статические слушатели назначаются агенту методом *so_add_nondestroyable_listener*, а динамические — *so_add_destroyable_listener*.

- При смене состояния агента у слушателя вызывается метод *changed()*, которому передается имя нового состояния агента.

- Метод *changed()* у слушателей вызывается после того, как отработают обработчики входа/выхода.

- Агенту может быть назначено произвольное количество слушателей. В этом случае SObjectizer не определяет порядок вызова слушателей.

- Список слушателей очищается агентом при дерегистрации агента. Поэтому перед повторной регистрацией агенту нужно назначить новых слушателей.

- Когда агенту назначается очередной слушатель, у объекта-слушателя вызывается метод *stored*. Этот метод может использоваться слушателями для определения имени текущего состояния агента с помощью функции *so_4::api::query_agent_state*.

Глава 13

Пример inheritance

Пример inheritance демонстрирует наследование классов агентов. Описание примера требует понимания идеи наследования агентов в SObjectizer, которая была подробно освещена в 2.1.4 на стр. 18, 2.2.3 на стр. 24 и 3.1.5 на стр. 31, поскольку в данной главе внимание уделяется техническим деталям поддержки наследования классов агентов в SObjectizer.

Пример inheritance состоит из одного файла `main.cpp` (стр. 221).

13.1 Что делает пример

Пример имитирует работу некоторой условной транзакции, обслуживаемой специальным агентом. Транзакция начинается посредством создания агента-обработчика транзакции и отсылки ему сообщения `msg_start`. Начатая транзакция может быть подтверждена (сообщение `msg_commit`) или отменена (сообщение `msg_rollback`). Для реализации транзакций с разными особенностями поведения используется наследование классов агентов.

В примере описывается четыре класса агента:

`a_trx_t` — базовый класс для всех транзакций, который является владельцем сообщений `msg_start`, `msg_commit`, `msg_rollback`;

`a_concrete_trx_t` — производный от `a_trx_t` класс “конкретной” транзакции, который определяет реальные состояния и события для обработки унаследованных сообщений;

`a_timed_trx_t` — производный от `a_trx_t` класс транзакции, которая может быть отменена по истечении тайм-аута. Предназначен для того, чтобы быть примесью (mixin) при множественном наследовании в других классах транзакций;

`a_concrete_timed_trx_t` — производный от `a_concrete_trx_t` и `a_timed_trx_t` класс “конкретной” транзакции, которая может быть отменена по истечении тайм-аута.

В SObjectizer регистрируется агент `a_concrete_timed_trx` класса `a_concrete_timed_trx_t`, который при получении сообщения `a_sobjectizer.msg_start` начинает транзакцию. Подтверждение или откат транзакции выполняется при получении отложенных сообщений `msg_commit`, `msg_rollback`. Время, на которое эти сообщения откладываются, задается в конструкторе агента.

Т.о., варьируя значения параметров конструктора агента, можно заставлять агента либо подтверждать, либо откатывать транзакцию.

Также в конструкторе агенту `a_concrete_timed_trx` указывается максимальное время жизни транзакции. Если по истечении этого времени транзакция не завершается каким-либо образом (т.е. не поступает ни `msg_commit`, ни `msg_rollback`), то транзакция откатывается из-за тайм-аута. Для этого используется унаследованная из класса `a_timed_trx_t` функциональность.

13.2 Разбор файла main.cpp

Класс `a_trx_t` примечателен тем, что в его описании для SObjectizer указываются только сообщения, но нет ни событий, ни состояний:

```
SOL4_CLASS_START( a_trx_t )

  SOL4_MSG_START( msg_start, a_trx_t::msg_start )
  SOL4_MSG_FINISH()

  SOL4_MSG_START( msg_commit, a_trx_t::msg_commit )
  SOL4_MSG_FINISH()

  SOL4_MSG_START( msg_rollback, a_trx_t::msg_rollback )
  SOL4_MSG_FINISH()

SOL4_CLASS_FINISH()
```

Это абсолютно корректное описание класса агента для SObjectizer. Более того, агент такого класса может быть зарегистрирован. В этом случае агент будет находиться в *пустом состоянии* и не сможет выполнять никаких действий. Но зато будет владельцем своих сообщений, которые могут использоваться другими агентами[1].

Реальные события и состояния определяет производный от `a_trx_t` класс агента `a_concrete_trx_t`.

Наследование классов агентов в SObjectizer необходимо выразить как на уровне наследования C++ классов, так и на уровне описания класса агента для SObjectizer. На уровне C++ в качестве базового класса вместо *so_4::rt::agent_t* необходимо указать имя C++ класса-предка. В данном случае для *a_concrete_trx_t* указывается имя *a_trx_t*:

```
class a_concrete_trx_t
  :  public virtual a_trx_t
```

В примере inheritance используется множественное наследование классов, которые унаследованы от *so_4::rt::agent_t*. Поэтому необходимо применять виртуальное наследование от *so_t::rt::agent_t*. Если бы в примере использовалось только одиночное наследование, то достаточно было бы применять простое наследование от *so_t::rt::agent_t*.

[1] Аналогичным образом агенты используют сообщения глобальных агентов, но сообщения глобальных агентов автоматически рассылаются через SOP (см. 5 на стр. 45).

На уровне SObjectizer наследование классов агентов выражается с помощью макроса *SOL4_SUPER_CLASS*, который необходимо разместить между макросами описания класса агента *SOL4_CLASS_START*, *SOL4_CLASS_FINISH*:

```
SOL4_CLASS_START( a_concrete_trx_t )
  // Указываем наследование.
  SOL4_SUPER_CLASS( a_trx_t )

  // Поскольку есть наследование, нужно
  // сразу определить начальное состояние.
  SOL4_INITIAL_STATE( st_not_started )

  ...
SOL4_CLASS_FINISH()
```

Макрос *SOL4_SUPER_CLASS* должен быть использован для каждого базового класса-агента[2].

Ключевым моментом в описании класса-наследника для SObjectizer является применение макроса *SOL4_INITIAL_STATE*. Если начальное состояние для класса-наследника не задано явно, то SObjectizer не сможет выбрать начальное состояние самостоятельно и откажется регистрировать агентов этого класса.

При подписке своих событий агент класса `a_concrete_trx_t` использует тот факт, что он является владельцем сообщений `msg_start`[3], `msg_commit`, `msg_rollback`. Хотя эти сообщения и не были описаны в классе `a_concrete_trx_t`, агент действительно владеет ими, т.к. он унаследовал данные сообщения из своего базового класса `a_trx_t`.

Класс агента `a_timed_trx_t` также наследуется от `a_trx_t`. В отличие от `a_concrete_trx_t` класс `a_timed_trx_t` не определяет собственных состояний, но определяет сообщение `msg_lifetime_left` и событие для его обработки `evt_lifetime_left`. Т.о., класс `a_timed_trx_t` не предполагает, что будут существовать агенты его класса. Вместо этого класс `a_timed_trx_t` будет участвовать в классах конкретных транзакций в качестве базового.

На уровне C++ наследования класс *a_timed_trx_t* определяет метод-обработчик входа в состояние *on_enter_appropriate_state*. Производные классы должны включить этот обработчик в то состояние, в котором они начинают обработку транзакции.

Класс агента `a_concrete_timed_trx_t` производен и от `a_concrete_trx_t`, и от `a_timed_trx_t`. При этом класс `a_concrete_timed_trx_t` не определяет ни одного собственного события, используется только унаследованная из базовых классов функциональность. Все, что требуется от `a_concrete_timed_trx_t` — это связать возможность выполнения транзакции и отката транзакции по тайм-ауту, что достигается переопределением состояния `st_started`.

Если производный класс переопределяет состояние, унаследованное из базового класса, то производный класс должен сделать полное описание этого состояния. Т.е.

[2]Макрос *SOL4_SUPER_CLASS* должен содержать имена классов агентов, которые где-то описаны с помощью макросов *SOL4_CLASS_START*, *SOL4_CLASS_FINISH*.

[3]Можно увидеть, что агент класса `a_concrete_trx_t` подписывается на два сообщения с именем `msg_start`. Но это не вызывает проблем, т.к. сообщения принадлежат разным агентам, т.е. их полные имена, которыми оперирует SObjectizer, не совпадают.

перечислить в нем все разрешенные к обработке события и описать обработчики входа/выхода в состояние, в том числе и унаследованные из базового класса. Если этого не сделать, то SObjectizer будет считать, что в состоянии разрешены к обработке только описанные в производном классе события. Например, если бы в классе `a_concrete_timed_trx_t` состояние `st_started` было описано так:

```
SOL4_STATE_START( st_started )
  SOL4_STATE_EVENT( evt_lifetime_left )

  SOL4_STATE_ON_ENTER( on_enter_appropriate_state )
SOL4_STATE_FINISH()
```

то SObjectizer разрешал бы обработку только события `evt_lifetime_left`[4]. Поэтому состояние `st_started` описывается следующим образом:

```
SOL4_STATE_START( st_started )
  SOL4_STATE_MERGE( a_concrete_trx_t, st_started )

  SOL4_STATE_EVENT( evt_lifetime_left )

  SOL4_STATE_ON_ENTER( on_enter_appropriate_state )
SOL4_STATE_FINISH()
```

Классу `a_concrete_timed_trx_t` необходимо, чтобы в состоянии `st_started` обрабатывались те же события, что и в базовом классе `a_concrete_trx_t`. Поэтому используется макрос *SOL4_STATE_MERGE*, который предписывает SObjectizer включить в описание состояния все события, разрешенные к обработке в указанном состоянии указанного класса. В данном примере вместо макроса *SOL4_STATE_MERGE* можно было бы написать:

```
SOL4_STATE_START( st_started )
  SOL4_STATE_EVENT( evt_trx_commit )
  SOL4_STATE_EVENT( evt_trx_rollback )

  SOL4_STATE_EVENT( evt_lifetime_left )

  SOL4_STATE_ON_ENTER( on_enter_appropriate_state )
SOL4_STATE_FINISH()
```

но макрос *SOL4_STATE_MERGE* гораздо удобнее при сопровождении. Например, если в `a_concrete_trx_t` список разрешенных к обработке событий в состоянии `st_started` изменится, то это автоматически будет учтено SObjectizer благодаря *SOL4_STATE_MERGE*. Без использования *SOL4_STATE_MERGE* при таком изменении пришлось бы вручную модифицировать описание класса `a_concrete_timed_trx_t`.

Также класс `a_concrete_timed_trx_t` добавляет в состояние `st_started` событие `evt_lifetime_left` и обработчик *on_enter_appropriate_state*, унаследованные

[4]Такое поведение SObjectizer было выбрано намеренно, чтобы дать возможность в производных классах полностью переопределить унаследованное состояние.

из a_timed_trx_t. Благодаря этому в состоянии st_state объединяется функциональность классов a_concrete_trx_t и a_timed_trx_t.

В функции *main* создается кооперация из одного агента a_concrete_timed_trx, которая указывается SObjectizer в качестве стартовой. В *main* также показано, что приложение само может контролировать время жизни объектов таймерной нити и диспетчера:

```
std::auto_ptr< so_4::timer_thread::timer_thread_t >
  timer_ptr( so_4::timer_thread::simple::create_timer_thread() );

std::auto_ptr< so_4::rt::dispatcher_t >
  disp_ptr( so_4::disp::one_thread::create_disp( *timer_ptr ) );
```

Созданные таким образом объекты таймерной нити и диспетчера будут уничтожены не при завершении работы SObjectizer Run-Time, а при выходе из *main*.

13.3 Результаты работы примера

Пример inheritance показывает разные результаты в зависимости от значений, переданных в конструктор агента a_concrete_timed_trx_t. Если указать значения 2500, 2000, 1500 (т.е. сообщение msg_commit отсылается через 2500 миллисекунд после начала транзакции, сообщение msg_rollback — через 2000 миллисекунд, а сообщение msg_lifetime_left — через 1500 миллисекунд), то результатом работы будет:

```
commit_timeout: 2500
rollback_timeout: 2000
lifetime: 1500
trx started
no time left
trx rollbacked
```

т.е. транзакция откатывается из-за тайм-аута.

Если же указать значения 1000, 2000, 1500, то результат изменится:

```
commit_timeout: 1000
rollback_timeout: 2000
lifetime: 1500
trx started
trx commited
```

Соответствующий результат будет получен и при указании значений 2500, 1000, 1500:

```
commit_timeout: 2500
rollback_timeout: 1000
lifetime: 1500
trx started
trx rollbacked
```

13.4 Резюме

- Наследование классов агентов в SObjectizer необходимо выразить как на уровне наследования C++ классов, так и на уровне описания класса агента для SObjectizer. На уровне C++ в качестве базового класса вместо *so_4::rt::agent_t* необходимо указать имя C++ класса-предка. На уровне SObjectizer наследование классов агентов выражается с помощью макроса *SOL4_SUPER_CLASS*, который необходимо разместить между макросами описания класса агента *SOL4_CLASS_START*, *SOL4_CLASS_FINISH*:

```
SOL4_CLASS_START( derived_t )

  SOL4_SUPER_CLASS( base_t )
  ...
SOL4_CLASS_FINISH()
```

- Макрос *SOL4_SUPER_CLASS* должен содержать имена классов агентов, которые где-то описаны с помощью макросов *SOL4_CLASS_START*, *SOL4_CLASS_FINISH*.

- Если планируется использовать множественное наследование классов агентов, то каждый базовый класс-агент должен быть виртуально унаследован от *so_4::rt::agent_t*. Для простого одиночного наследования этого не требуется.

- В описании класса-наследника для SObjectizer имя начального состояния агента должно быть задано явно с помощью макроса *SOL4_INITIAL_STATE*.

```
SOL4_CLASS_START( derived_t )

  SOL4_SUPER_CLASS( base_t )
  ...
  SOL4_INITIAL_STATE( st_some_state )
  ...
SOL4_CLASS_FINISH()
```

  Если начальное состояние для класса-наследника не задано явно, то SObjectizer не сможет выбрать начальное состояние самостоятельно и откажется регистрировать агентов этого класса.

- Если производный класс переопределяет состояние, унаследованное из базового класса, то производный класс должен сделать полное описание этого состояния. Т.е. перечислить в нем все разрешенные к обработке события и описать обработчики входа/выхода в состояние, в том числе и унаследованные из базового класса. Если этого не сделать, то SObjectizer будет считать, что в состоянии разрешены к обработке только описанные в производном классе события.

- Для удобства переопределения состояния в производном классе рекомендуется использовать макрос *SOL4_STATE_MERGE*, который предписывает SObjectizer включить в описание состояния все события, разрешенные к обработке в указанном состоянии указанного класса. Использование этого макроса облегчает сопровождение, т.к. изменение списка событий в состоянии базового класса автоматически подхватывается макросом *SOL4_STATE_MERGE*.

```
SOL4_CLASS_START( derived_t )

  SOL4_SUPER_CLASS( base_t )
  ...
  SOL4_INITIAL_STATE( st_some_state )
  ...
  SOL4_STATE_START( st_some_state )
    SOL4_STATE_MERGE( base_t, st_some_state )
    ...
  SOL4_STATE_FINISH()
  ...
SOL4_CLASS_FINISH()
```

Глава 14

Пример subscr_hook

Пример демонстрирует использование т.н. *hook-ов подписки* и диспетчера с *активными объектами*.

Пример subscr_hook состоит из одного файла `main.cpp` (стр. 232).

14.1 Что такое hook подписки

Прежде, чем приступить к обсуждению самого примера, необходимо сделать небольшое введение в понятие *hook-а подписки*.

Практика применения SObjectizer показала, что существует проблема *упреждающей подписки*, которая проявляется, например, в следующем случае:

- некоторый агент `A` для поддержки TCP/IP соединения начинает свою работу при получении сообщения `a_sobjectizer.msg_start`. Он пытается установить TCP/IP соединение и сразу же сообщает результат посредством своих сообщений `msg_connection_ok` или `msg_connection_fail`;
- существует агент `B`, которому необходимо TCP/IP соединение. Агент `B` создает подчиненную кооперацию с агентом `A`, после чего подписывается на сообщения `A.msg_connection_ok` и `A.msg_connection_fail`.

Проблема в том, что при использовании некоторых типов диспетчеров (с активными объектами, например) агент `A` успеет получить сообщение `msg_start`, создать TCP/IP соединение и отослать результат соединения до того, как агент `B` получит управление после вызова *`so_4::api::register_coop()`*.

Этой проблемы не было бы, если бы агент `B` мог подписаться на сообщение агента `A` еще до создания агента `A`. Но в SObjectizer подписаться можно только на сообщения уже зарегистрированных агентов.

Для преодоления этого противоречия в SObjectizer существует понятие т.н. *hook-ов подписки*. Hook подписки – это объект, реализующий интерфейс *so_4::rt::subscr_hook_t*:

```
class subscr_hook_t
  : private cpp_util_2::nocopy_t
  {
    public :
```

```
        virtual ~subscr_hook_t();

        virtual void
        hook() = 0;
    };
```

Объект-hook должен быть связан с конкретной кооперацией агентов посредством метода:

```
void
add(
  subscr_hook_t * hook );
```

класса *so_4::rt::agent_coop_t*. При регистрации кооперации непосредственно перед вызовом *so_on_subscription* у агентов зарегистрированной кооперации[1] SObjectizer Run-Time вызывает метод *so_4::rt::subscr_hook_t::hook()* у объекта-hook-а.

Для приведенного примера с агентами `A` и `B` агент `B` должен создать hook-и для подписки своих событий на сообщения агента `A` и связать эти hook-и с подчиненной кооперацией. В результате агент `B` окажется подписанным на сообщения агента `A` еще до того, как агент `A` подпишется на сообщение `a_sobjectizer.msg_start`. Причем вне зависимости от типа используемого диспетчера.

14.2 Что делает пример

В примере используются два агента: `a_child` и `a_owner`.

Агент `a_child` владеет сообщениями `msg_hello`, `msg_bye` и обрабатывает сообщение `a_sobjectizer.msg_start` посредством события `evt_start`. В обработчике события `evt_start` сначала отсылается сообщение `msg_hello`, затем — `msg_bye`.

Агент `a_owner` владеет событиями: `evt_start`, `evt_hello`, `evt_bye`. В обработчике события `evt_start` агента `a_owner` создает агента `a_child` и подписывается на его сообщения с помощью hook-ов подписки. Получив сообщение `a_child.msg_bye` агент `a_owner` завершает работу примера.

14.3 Разбор файла main.cpp

14.3.1 Использование hook-ов подписки

Hook-и подписки используются в коде метода *a_owner_t::evt_start*:

```
void
evt_start()
  {
    // Создаем подчиненную кооперацию.
    a_child_t * child = new a_child_t( "a_child" );
    so_4::rt::dyn_agent_coop_t * child_coop =
      new so_4::rt::dyn_agent_coop_t( child );
```

[1] Но уже после того, как агенты зарегистрированы в системном словаре SObjectizer.

```
    // И подписываемся на сообщения дочернего агента.
    so_4::rt::def_subscr_hook( *child_coop,
      // Подписываем родительского агента.
      *this, "evt_hello",
      // На сообщения дочернего агента.
      // В этом случае через указатель на агента.
      *child, "msg_hello" );
    so_4::rt::def_subscr_hook( *child_coop,
      // Подписываем родительского агента.
      *this, "evt_bye",
      // На сообщения дочернего агента.
      // В этом случае через имя агента (для примера).
      "a_child", "msg_bye" );

    // Регистрируем кооперацию.
    so_4::rt::dyn_agent_coop_helper_t child_coop_helper(
      child_coop );
  }
```

Как было сказано выше, hook подписки — это объект, реализующий интерфейс *so_4::rt::subscr_hook_t*. Поскольку, в большинстве случаев, hook подписки используется для подписки событий одного агента на сообщения другого агента, то SObjectizer предоставляет уже готовую реализацию интерфейса *so_4::rt::subscr_hook_t*. В данном случае используются перегруженные функции *so_4::rt::def_subscr_hook*, которые создают объект-hook для подписки одного события одного агента на одно сообщение второго агента. Эти функции имеют формат:

```
void
def_subscr_hook(
  //! Кооперация, в которую должен быть помещен hook.
  so_4::rt::agent_coop_t & coop,
  //! Агент, у которого нужно вызывать so_subscribe().
  so_4::rt::agent_t & agent_to_subscribe,
  //! Имя события, которое нужно подписать.
  const std::string & evt_to_subscribe,
  //! Имя агента-владельца сообщения.
  const std::string & msg_owner,
  //! Имя сообщения инцидента.
  const std::string & msg_name,
  //! Приоритет события.
  int priority = 0,
  //! Поток для вывода сообщений об ошибках подписки.
  std::ostream * err = &std::cerr,
  //! Тип диспетчеризации события.
  const evt_subscr_t::dispatching_t & dispatching =
    evt_subscr_t::normal_dispatching );
```

```
void
def_subscr_hook(
  //! Кооперация, в которую должен быть помещен hook.
  so_4::rt::agent_coop_t & coop,
  //! Агент, у которого нужно вызывать so_subscribe().
  so_4::rt::agent_t & agent_to_subscribe,
  //! Имя события, которое нужно подписать.
  const std::string & evt_to_subscribe,
  //! Агент-владелец сообщения.
  so_4::rt::agent_t & msg_owner,
  //! Имя сообщения инцидента.
  const std::string & msg_name,
  //! Приоритет события.
  int priority = 0,
  //! Поток для вывода сообщений об ошибках подписки.
  std::ostream * err = &std::cerr,
  //! Тип диспетчеризации события.
  const evt_subscr_t::dispatching_t & dispatching =
    evt_subscr_t::normal_dispatching );
```

Работа функций *def_subscr_hook* состоит в следующем:

- создается объект-hook;
- объекту-hook-у указываются полные имена сообщения-инцидента и события-реципиента (т.е. имена, включающие имена агентов);
- объект-hook связывается с указанной кооперацией.

Благодаря использованию функций *def_subscr_hook* агент a_owner гарантирует себе подписку на сообщения создаваемого агента a_child до возврата из *register_coop*.

Следует отметить, что предоставляемые SObjectizer готовые hook-и подписки имеют примитивные средства обработки ошибок подписки. Максимум, что они могут сделать в случае возникновения ошибки — это отобразить ее описание в указанный *std::ostream*. Поэтому, если приложению требуется от hook-а подписки более развитая система обработки ошибок или большая функциональность (например, подписка одного события сразу на несколько инцидентов), прикладному программисту необходимо реализовать собственный класс hook-а, производный от *so_4::rt::subscr_hook_t*.

14.3.2 Использование диспетчера с активными объектами

Диспетчер с *активными объектами* является одним из штатных диспетчеров SObjectizer. Этот диспетчер делит всех зарегистрированных агентов на две категории:

Активные объекты. Для каждого активного объекта (агента) диспетчер создает отдельную нить и обрабатывает события этого агента только на данной нити[2].

[2]За исключением т.н. *insend-событий*, которые обсуждаются в примере parent_insend (см. 18 на стр. 173).

Пассивные объекты. Все агенты, не объявленные активными объектами, автоматически считаются пассивными объектами. Для запуска обработчиков событий таких агентов используется одна общая нить (т.н. *нить пассивных объектов*). Для пассивных объектов диспетчер с активными объектами практически ничем не отличается от диспетчера с одной активной нитью.

Для использования диспетчера с активными объектами необходимо:

1. Создать объект-диспетчер и указать его при вызове *so_4::api::start*:

```
so_4::ret_code_t rc = so_4::api::start(
  // Диспетчер будет уничтожен при выходе из start().
  so_4::disp::active_obj::create_disp(
    // Таймер будет уничтожен диспетчером.
    so_4::timer_thread::simple::create_timer_thread(),
    so_4::auto_destroy_timer ),
  so_4::auto_destroy_disp,
  &coop );
```

   Прототип функции *so_4::disp::active_obj::create_disp* описан в заголовочном файле `so_4/disp/active_obj/h/pub.hpp`.

2. Специальным образом пометить агентов, которые являются активными объектами. Осуществляется это путем назначения агенту специального свойства (*traits*):

```
so_add_traits(
  so_4::disp::active_obj::query_active_obj_traits() );
```

   Вызов унаследованного из *so_4::rt::agent_t* метода *so_add_traits* для назначения свойства активного объекта нужно выполнить **до** регистрации агента.

В данном примере агент `a_child` сам объявляет себя активным объектом. Но, вообще говоря, в SObjectizer можно сделать активным объектом любого агента, если он не привязан жестко к конкретному типу диспетчера. В составе SObjectizer есть ряд агентов, которые не предъявляют никаких требований к механизму диспетчеризации и поэтому способны работать на разных диспетчерах. Но некоторых из них, например, *so_4::rt::comm::a_srv_channel_t*, удобно делать активными объектами, если приложение использует диспетчер с активными объектами. В этом случае в приложении нужно просто создать агента типа *so_4::rt::comm::a_srv_channel_t* и вызвать у него перед регистрацией метод *so_add_traits* (что демонстрируется в примере filter[3]).

Если же агента объявить активным объектом, но не использовать диспетчер с активными объектами, то агент будет работать в SObjectizer как обычный (пассивный) объект.

14.4 Результаты работы примера

В результате работы примера на стандартный поток вывода отображается:

[3] См. 15 на стр. 130.

```
hello!
bye!
```

что подтверждает факт запуска событий `evt_hello` и `evt_bye` агента `a_owner`.

14.5 Резюме

- Hook подписки – это объект, реализующий интерфейс *so_4::rt::subscr_hook_t*:

  ```
  class subscr_hook_t
    : private cpp_util_2::nocopy_t
    {
      public :
        virtual ~subscr_hook_t();

        virtual void
        hook() = 0;
    };
  ```

  Объект-hook должен быть связан с конкретной кооперацией агентов посредством метода:

  ```
  void
  add(
    subscr_hook_t * hook );
  ```

  класса *so_4::rt::agent_coop_t*.

- Объект-hook должен быть связан с кооперацией **перед** регистрацией кооперации.

- С кооперацией может быть связано любое количество объектов-hook-ов. Один объект-hook должен быть связан только с одной кооперацией и только один раз. Объясняется это тем, что объекты-hook-и должны быть динамически созданными объектами, ответственность за уничтожение которых берет на себя объект-кооперация.

- При регистрации кооперации непосредственно перед вызовом *so_on_subscription* у агентов зарегистрированной кооперации (но уже после того, как агенты зарегистрированы в системном словаре SObjectizer) SObjectizer Run-Time вызывает метод *so_4::rt::subscr_hook_t::hook()* у всех связанных с кооперацией объектов-hook-ов.

- В большинстве случаев hook подписки используется для подписки событий одного агента на сообщения другого объекта.

- SObjectizer предоставляет готовую реализацию hook-а, который подписывает событие агента на сообщение другого объекта. Этот hook создается и связывается с конкретной кооперацией посредством функций *so_4::rt::def_subscr_hook*.

- Предоставляемые SObjectizer готовые hook-и подписки имеют ограниченную функциональность и примитивные средства обработки ошибок подписки. Поэтому, если приложению требуется от hook-а подписки более развитая система обработки ошибок или большие возможности (например, подписка одного события сразу на несколько инцидентов), прикладному программисту необходимо реализовать собственный класс hook-а, производный от *so_4::rt::subscr_hook_t*.

- Диспетчер с активными объектами создается с помощью функции *so_4::disp::active_obj::create_disp*, описанной в заголовочном файле so_4/disp/active_obj/h/pub.hpp.

- Диспетчер с активными объектами делит всех агентов на две категории: активные и пассивные объекты. Для каждого активного объекта диспетчером создается отдельная нить, на контексте которой обрабатываются события этого агента. События всех пассивных агентов обрабатываются на одной нити пассивных агентов.

- Для того, чтобы сделать агента активным объектом, необходимо перед регистрацией агента назначить ему специальное свойство (traits):

```
so_add_traits(
  so_4::disp::active_obj::query_active_obj_traits() );
```

- Любого агента можно сделать активным объектом, если только сам агент не привязывается к конкретному диспетчеру.

- Если же агента объявить активным объектом, но не использовать диспетчер с активными объектами, то агент будет работать в SObjectizer как обычный (пассивный) объект.

Глава 15

Пример filter

Пример filter демонстрирует возможность построения распределенного приложения с использованием коммуникационных возможностей SObjectizer. В этом примере один сервер общается с двумя клиентами через TCP/IP соединения. Для каждого клиента используется свой интерфейс (глобальный агент). Описание примера требует понимания SObjectizer Protocol (см. 5 на стр. 45), поскольку в данной главе внимание уделяется только техническим деталям создания распределенных приложений с использованием SOP.

Также в примере показывается:

- создание интерактивных приложений, в которых главная нить используется для диалога с пользователем, а SObjectizer запускается на отдельной нити;
- назначение фильтров для SOP-каналов;
- назначение агентам свойства активного объекта;
- повторная регистрация и дерегистрация кооперaций.

Пример filter состоит из следующих файлов:

c1i.hpp, c1i.cpp — глобальный агент, чьими сообщениями обмениваются сервер и первый клиент (стр. 235, стр. 237).

c2i.hpp, c2i.cpp — глобальный агент, чьими сообщениями обмениваются сервер и второй клиент (стр. 238, стр. 239).

c1.cpp — исходный код первого клиента (стр. 240).

c2.cpp — исходный код второго клиента (стр. 246).

server.cpp — исходный код сервера (стр. 252).

15.1 Что делает пример

В пример входят три приложения: сервер и два клиента, которые называются “первый” и “второй” клиенты.

Сначала должен быть запущен сервер, которому указывается TCP/IP адрес для создания серверного сокета. Затем запускаются клиенты, которые будут подключаться к этому сокету.

Первый клиент позволяет оператору:

- зарегистрировать главную кооперацию клиента. В эту кооперацию входит транспортный агент для поддержки клиентского подключения к серверу. Когда главная кооперация будет запущена, транспортный агент начнет осуществлять попытки подключения к серверу;
- дерегистрировать главную кооперацию клиента. Транспортный агент дерегистрируется, и соединение с сервером, если было установлено, разрывается;
- отослать запрос на сервер.

Второй клиент позволяет оператору:

- зарегистрировать главную кооперацию клиента. В эту кооперацию входит транспортный агент для поддержки клиентского подключения к серверу. Когда главная кооперация будет запущена, транспортный агент начнет осуществлять попытки подключения к серверу;
- дерегистрировать главную кооперацию клиента. Транспортный агент дерегистрируется, и соединение с сервером, если было установлено, разрывается;

Сервер создает транспортного агента для поддержки серверного TCP/IP канала.

У каждого клиента для общения с сервером определен свой интерфейс: первый агент использует глобального агента `a_c1i`, второй — глобального агента `a_c2i`. Каждый из глобальных агентов владеет сообщениями `msg_request` и `msg_reply`.

Когда сервер получает запрос от первого клиента, представленный сообщением `msg_request` глобального агента `a_c1i`, то сервер инициирует запрос ко второму клиенту с помощью сообщения `msg_request` глобального агента `a_c2i`. Получив от второго клиента ответ в виде сообщения `a_c2i.msg_reply` сервер отвечает первому клиенту сообщением `a_c1i.msg_reply`. Т.о., сервер отвечает первому клиенту только после того, как проведет диалог со вторым клиентом. Поэтому, если в момент получения сообщения от первого клиента у сервера не было соединения со вторым клиентом, первый клиент не получит ответа от сервера.

15.2 Разбор файлов c1i.hpp, c1i.cpp, c2i.hpp, c2i.cpp

Файлы `c1i.hpp`, `c1i.cpp`, `c2i.hpp`, `c2i.cpp` содержат описания классов *c1i_t* и *c2i_t*, которые будут использоваться в качестве классов агентов `a_c1i` и `a_c2i`.

Следует обратить внимание только на один момент: унаследованный метод *so_on_subscription* оставлен чистым виртуальным методом. Сделано это специально, т.к. для глобального агента нет объекта-агента. Наличие чистого виртуального метода в классе *c1i_t* (*c2i_t*) не позволит по ошибке создать в программе объект этого типа. А т.к. невозможно будет создать объект-агент, то невозможно будет и попытаться зарегистрировать глобального агента через кооперацию и функцию *so_4::api::register_coop* — для регистрации глобального агента должна использоваться функция *so_4::api::make_global_agent*.

15.3 Разбор файлов c1.cpp, c2.cpp

Файлы `c1.cpp`, `c2.cpp` содержат реализацию первого и второго клиента. Реализации являются практически близнецами, за исключением того, что:

- первый клиент отсылает сообщение-запрос (`a_c1i.msg_request`) и ожидает ответа (`a_c1i.msg_reply`), а второй клиент, наоборот, ожидает запрос и отвечает на него (сообщения `a_c2i.msg_request`, `a_c2i.msg_reply`);
- первый и второй клиенты используют разных глобальных агентов и поэтому устанавливают разные фильтры на свои исходящие соединения.

Во всем остальном оба клиента являются копией друг друга. Поэтому далее рассматривается код из файла `c1.cpp`, и все сказанное в той же мере относится к содержимому `c2.cpp`.

15.3.1 Подписка агента a_cln

Агент `a_cln` должен быть подписан на сообщения трех агентов: сообщение `a_sobjectizer.msg_start`, свое собственное сообщение `msg_send_request` и сообщение `msg_reply` глобального агента `a_c1i`. Подписка на сообщение `a_sobjectizer.msg_start` и собственное `msg_send_request` осуществляется обычным образом — в методе *so_on_subscription*. Но для подписки на `a_c1i.msg_reply` требуется больше усилий.

Дело в том, что глобальный агент `a_c1i` не появляется автоматически. Его нужно явно зарегистрировать при помощи функции *so_4::api::make_global_agent*. Поэтому на момент вызова *so_on_deregistration* агента `a_c1i` еще нет, и попытка подписаться на его сообщение приведет к ошибке. Из-за этого агент `a_cln` регистрирует глобального агента и подписывается на его сообщение в своем событии `evt_start`:

```
void
a_cln_t::evt_start()
  {
    so_4::ret_code_t rc = so_4::api::make_global_agent(
      c1i_t::agent_name(),
      c1i_t::agent_type() );
    if( rc )
      std::cerr << rc << std::endl;
    else
      so_subscribe( "evt_server_reply",
        c1i_t::agent_name(), "msg_reply" );
  }
```

Вообще говоря, это же можно было сделать в методе *so_on_subscription* (на практике так чаще всего и делается). Но в данном примере подписка события `evt_server_reply` выполняется именно так, чтобы показать, что изменить подписку события можно в любой момент, а не только в *so_on_subscription*.

Регистрация глобального агента осуществляется с помощью функции:

```
so_4::ret_code_t
```

```
make_global_agent(
  /*! Имя глобального агента. */
  const std::string & agent_name,
  /*! Тип глобального агента. */
  const std::string & agent_type );
```

определенной в пространстве имен *so_4::api*. Параметр *agent_name* задает имя агента, параметр *agent_type* должен содержать имя типа агента, которое было указано при описании типа глобального агента в макросе *SOL4_CLASS_START*.

Функцию *make_global_agent* можно вызывать неоднократно для одного и того же глобального агента. Важно только, чтобы во всех случаях было одинаковое имя типа агента.

15.3.2 Регистрация и дерегистрация главной кооперации

В примере filter клиенты стартуют SObjectizer без указания стартовой кооперации. Главная кооперация клиента создается (регистрируется) и уничтожается (дерегистрируется) уже после запуска SObjectizer по командам оператора. Регистрация главной кооперации осуществляется с помощью функции:

```
void
create_coop(
  const char * sock_addr )
  {

  ...

    so_4::rt::agent_t * agents[] =
      {
        a_cln, a_sock
      };
    so_4::rt::dyn_agent_coop_helper_t coop_helper(
      new so_4::rt::dyn_agent_coop_t(
        "client_coop", agents,
        sizeof( agents ) / sizeof( agents[ 0 ] ) ) );

    if( coop_helper.result() )
      std::cerr << "register_coop:\n"
        << coop_helper.result() << std::endl;
  }
```

В функции *create_coop* контролируется успешность регистрации кооперации с помощью метода *result()* класса *so_4::rt::dyn_agent_coop_helper_t*. Эта проверка здесь явно не лишняя: поскольку кооперация регистрируется по команде оператора, то оператор может дать повторную команду на регистрацию кооперации, предварительно не дерегистрировав ранее зарегистрированную кооперацию. В этом случае пример отобразит на стандартный поток вывода сообщение вида:

```
register_coop:
so_4\rt\impl\sys_dict.cpp:684: 1 [client_coop: name used by cooperation]
```

Дерегистрация кооперации в примере также производится по команде оператора, и выполняется дерегистрация с помощью функции:

```
void
destroy_coop()
  {
    so_4::ret_code_t rc =
      so_4::api::deregister_coop( "client_coop" );
    if( rc )
      std::cerr << "deregister_coop:\n" << rc << std::endl;
  }
```

Результат завершения *so_4::api::deregister_coop* контролируется, и, если дерегистрация завершилась неудачно, описание ошибки помещается на стандартный поток ошибок. Например, если оператор попытается дерегистрировать кооперацию, которая не была зарегистрирована, то увидит сообщение вида:

```
deregister_coop:
so_4\rt\impl\sys_dict.cpp:876: 2 [client_coop: cooperation not found]
```

15.3.3 Транспортный агент

Большую часть кода функции *create_coop* в примере занимает создание и настройка транспортного агента:

```
// Создание клиентского сокета для взаимодействия по SOP.
// Фильтр, который допускает только сообщения агента c1i.
so_4::sop::std_filter_t * filter =
  so_4::sop::create_std_filter();
filter->insert( c1i_t::agent_name() );

so_4::rt::comm::a_cln_channel_t * a_sock =
  new so_4::rt::comm::a_cln_channel_t(
    "a_sock",
    so_4::socket::channels::create_client_factory( sock_addr ),
    // Назначаем фильтр.
    filter,
    // Назначаем обработчик разрывов связи.
    so_4::rt::comm::a_cln_channel_base_t::
      create_def_disconnect_handler( 5000, 0 )
  );
// Сокет также будет активным агентом.
a_sock->so_add_traits( so_4::disp::active_obj::
  query_active_obj_traits() );
// Поскольку трафик предполагается небольшой, то устанавливаем
// порог так, чтобы сообщения уходили сразу же.
a_sock->set_out_threshold( so_4::rt::comm::threshold_t( 1, 1 ) );
```

В двух словах этот фрагмент делает следующее:

- сначала создается фильтр для данного соединения;
- затем создается транспортный агент, который будет использовать этот фильтр. транспортный агент будет использовать клиентское подключение к серверу по протоколу TCP/IP на адрес, указанный в параметре *sock_addr*;
- транспортный агент объявляется активным объектом, и
- устанавливаются минимальные пороги ввода-вывода.

Ниже все эти действия рассматриваются более подробно.

15.3.4 Фильтр

Особенностью взаимодействия по SOP с помощью сообщений глобальных агентов является то, что при отсылке сообщения глобального агента в каком-то модуле неизвестно, кто подписан на это сообщение в других модулях. Поэтому сообщение глобального агента отсылается во **все** существующие соединения. Естественно, что при таком подходе клиенту будут поступать сообщения не только известных клиенту глобальных агентов, но и всех глобальных агентов, про которых знает только сервер и другие клиенты. Поскольку клиента это вряд ли будет устраивать, в SObjectizer реализован механизм фильтров для коммуникационных каналов.

Фильтр — это объект, который определяет, разрешена ли передача клиенту сообщений конкретного глобального агента. Фильтры предназначены для двух целей:

1. Ограничение трафика. Клиенту со стороны сервера и серверу со стороны клиента отсылаются сообщения только тех глобальных агентов, которые разрешены фильтром.
2. Разграничение доступа. Фильтр разрешает транспорт ограниченного числа сообщений и делает невозможным доставку сообщения агента, который запрещен фильтром.

В SObjectizer различаются два вида фильтров: исходящие и входящие. Оба вида реализуются одними и теми же C++ классами. Различие состоит в том, что исходящий фильтр задается для исходящего (клиентского) соединения. Т.е. исходящий фильтр указывается при создании агентов типа `so_4::rt::comm::a_cln_channel_t`. На стороне клиента исходящий фильтр является единственным фильтром для канала.

Когда клиент подключается к серверу, SObjectizer инициирует отсылку фильтра на сервер. Сервер получает фильтр клиента и сохраняет его у себя. В дальнейшем этот фильтр используется для контроля поступающих от клиента сообщений. Т.е. исходящее от клиента сообщение фильтруется дважды: сначала на стороне отсылающего сообщения клиента, а затем на стороне получившего сообщение сервера. Сделано это для того, чтобы клиент не мог сначала отослать серверу один фильтр, а затем начать присылать не удовлетворяющие фильтру сообщения.

Входящий фильтр назначается входящему (серверному) соединению. Т.е. входящий фильтр указывается при создании агентов типа `so_4::rt::comm::a_srv_channel_t`. На стороне сервера исходящий и входящий фильтры комбинируются: поступающие от клиента сообщения сначала пропускаются через входящий фильтр сервера, а затем через исходящий фильтр клиента.

На C++ фильтры реализуются в виде сериализуемых[1] объектов, производных от типа:

```
class filter_t
  :  public oess_1::stdsn::serializable_t
  {
  OESS_SERIALIZER_EX( filter_t, SO_4_TYPE )
  public :
    virtual ~filter_t();

    //! Создание копии объекта-фильтра.
    /*!
      Должен использоваться вместо оператора
      копирования.

      \return Указатель на динамически созданный
      объект, который должен быть уничтожен
      посредством оператора delete.
    */
    virtual filter_t *
    clone() const = 0;

    //! Проверка возможности обработки сообщения
    //! указанного глобального агента.
    /*!
      \return true, если обработка разрешена.
      false в противном случае.
    */
    virtual bool
    is_enabled(
      const std::string & agent_name ) const = 0;
  };
```

определенного в пространстве имен *so_4::sop*.

На данный момент SObjectizer предоставляет два готовых фильтра:

- разрешающий все сообщения. Создается функцией *so_4::sop::create_all_enable_filter*;
- "штатный" фильтр, разрешающий только сообщения указанных агентов. Штатный фильтр создается функцией *so_4::sop::create_std_filter* и реализует интерфейс:

```
class std_filter_t
  :  public filter_t
  {
  OESS_SERIALIZER_EX( std_filter_t, SO_4_TYPE )
```

[1] Для сериализации используется проект ObjESSty.

```
public :
  virtual ~std_filter_t();

  virtual filter_t *
  clone() const = 0;

  virtual bool
  is_enabled(
    const std::string & agent_name ) const = 0;

  //! Добавить имя агента в фильтр.
  virtual void
  insert(
    const std::string & agent_name ) = 0;

  //! Добавить имена указанных агентов в фильтр.
  virtual void
  insert(
    //! Вектор указателей на имена агентов.
    const char ** agent_names,
    //! Количество элементов в векторе.
    size_t count ) = 0;

  //! Изъять имя агента из фильтра.
  virtual void
  erase(
    const std::string & agent_name ) = 0;

  //! Изъять имена указанных агентов из фильтра.
  virtual void
  erase(
    //! Вектор указателей на имена агентов.
    const char ** agent_names,
    //! Количество элементов в векторе.
    size_t count ) = 0;
};
```

В примере filter используется штатный фильтр, который разрешает сообщения только одного агента:

```
so_4::sop::std_filter_t * filter =
  so_4::sop::create_std_filter();
filter->insert( c1i_t::agent_name() );
```

15.3.5 Канал ввода-вывода

Транспортные агенты SObjectizer (например, *so_4::rt::comm::a_cln_channel_t*, *so_4::rt::comm::a_srv_channel_t*) предназначены для работы с разными типами соединений (например, сокетами, pipe, разделяемой памятью). Для того, чтобы это было

возможно, в SObjectizer выделено отдельное понятие: *канал ввода-вывода (io_ channel)*. Транспортные агенты работают с каналами ввода-вывода через объекты, реализующие интерфейс *so_4::rt::comm::io_channel_t*:

```
class io_channel_t
  :  private cpp_util_2::nocopy_t
  {
  public :
    //! Режим закрытия канала связи.
    enum  close_mode_t
     {
        //! Нормальный режим.
        /*!
          Должны быть выполнены все действия, которые
          необходимы для нормального закрытия канала
          связи.

          Например, для SSL соединения необходимо
          выдать команду shutdown, после чего закрыть
          соответствующий физический канал.
        */
        normal_close,

        //! Режим деструктора.
        /*!
          Означает, что закрытие канала связи осуществляется
          в деструкторе объекта, владеющего каналом связи.
          В некоторых случаях, например, для SSL, в этом
          режиме проблематично корректно закрывать SSL
          соединение (т.к. команда SSL_shutdown требует
          нескольких обменов). Но можно корректно закрыть
          соответствующий физический канал (сокет).
        */
        destructor_close,

        //! Режим закрытия после обнаружения ошибки.
        /*!
          Означает, что закрытие канала осуществляется
          после обнаружения ошибок ввода/вывода канала.
          Как правило, в этом случае требуется только
          освободить ресурсы, выделенные данному каналу.
        */
        error_close
      };

    //! Виртуальный деструктор.
    /*!
      Деструктор производных классов должен
```

```
  осуществлять закрытие коммуникационного
  канала в режиме destructor_close.
*/
virtual ~io_channel_t();

//! Доступна ли следующая порция данных
//! в канале без блокировки.
/*!
  \return true, если последующее обращение
  к read() не приведет к блокированию вызвавшей
  метод нити.
*/
virtual bool
is_readable() = 0;

//! Прочитать данные из канала.
/*!
  Должен возвращать success, если чтение не
  сопровождалось ошибками, даже если из соединения
  прочитано 0 байт (такое может происходить в
  SSL-соединениях).

  Если обнаруживается, что соединение закрыто на
  другой стороне, то должен возвращать
  connection_closed.
*/
virtual so_4::ret_code_t
read(
  //! Приемник данных.
  void * buf,
  //! Размер приемника.
  unsigned int buf_size,
  //! Количество реально помещенных в
  //! приемник байт.
  unsigned int & bytes_read ) = 0;

//! Доступен ли канал для записи.
/*!
  \return true, если канал может принимать
  данные.
*/
virtual bool
is_writeable() = 0;

//! Записать данные в канал.
virtual so_4::ret_code_t
write(
  //! Буфер с данными для записи.
```

```
      const void * buf,
      //! Количество байт для записи.
      unsigned int buf_size,
      //! Сколько байт реально было записано.
      unsigned int & bytes_written ) = 0;

    //! Закрытие канала.
    virtual void
    close(
      //! В каком режиме закрывается канал.
      close_mode_t close_mode ) = 0;
  };
```

Реализации интерфейса *so_4::rt::comm::io_channels_t* предназначены для работы по уже установленному физическому соединению. Для установления физического соединения в SObjectizer определены еще два интерфейса. Интерфейс *so_4::rt::comm::client_factory_t* предназначен для инициирования исходящего соединения со стороны клиента:

```
  class client_factory_t
    :  private cpp_util_2::nocopy_t
    {
    public :
      //! Виртуальный деструктор.
      /*!
        Деструкторы производных классов не должны
        уничтожать созданные ими объекты io_channel_t.
        Каждый объект io_channel_t должен быть
        уничтожен отдельно.
      */
      virtual ~client_factory_t();

      //! Создать новое клиентское соединение.
      /*!
        \return success, если клиентское соединение
        успешно создано.
      */
      virtual so_4::ret_code_t
      connect(
        //! Приемник указателя на объект-канал.
        /*!
          Возвращенный объект должен быть полностью
          готов к работе.

          Должен быть возвращен указатель на динамически
          созданный объект, который будет удален
          посредством delete.
        */
```

```
        io_channel_t * & client ) = 0;
    };
```

Интерфейс *so_4::rt::comm::server_channel_t* предназначен для обслуживания серверных соединений:

```
class server_channel_t
  :  private cpp_util_2::nocopy_t
  {
  public :
    //! Виртуальный деструктор.
    /*!
      Деструкторы производных классов не должны
      уничтожать созданные ими объекты io_channel_t.

      Если это требуется коммуникационными средствами,
      то серверный канал может закрыть в своем
      деструкторе все физические каналы,
      с которыми работают созданные объекты io_channel_t.
      Но сами объекты io_channel_t должны остаться
      существовать.
    */
    virtual ~server_channel_t();

    //! Создать серверный канал.
    virtual so_4::ret_code_t
    create() = 0;

    //! Закрыть серверный канал.
    /*!
      Закрыть серверный канал так, чтобы последующим
      вызовом create() можно было создать канал.

      Созданные объекты io_channel_t не должны
      уничтожаться.
    */
    virtual void
    close() = 0;


    //! Проверить существование новых подключений,
    //! для которых можно создать объекты io_channel_t.
    /*!
      \return success, если есть новые подключения.
      no_new_connections, если новых подключений нет,
      но сам серверный канал находится в нормальном
      состоянии.
    */
```

```
    virtual so_4::ret_code_t
    check_new_connections() = 0;

    //! Создать объект io_channel_t для нового подключения.
    virtual so_4::ret_code_t
    accept(
      //! Приемник указателя на объект-канал.
      /*!
        Возвращенный объект должен быть полностью
        готов к работе.

        Должен быть возвращен указатель на динамически
        созданный объект, который будет удален
        посредством delete.
      */
      io_channel_t * & client ) = 0;
};
```

Идея клиентских и серверных каналов ввода-вывода в SObjectizer аналогична идее сокетов: серверный сокет сначала переводится в режим ожидания подключения (listening), а затем происходит ожидание подключения нового клиента (accepting). Когда серверный сокет принимает новое подключение, на стороне сервера появляется еще один сокет, не зависящий от исходного серверного сокета. Взаимодействие с клиентом осуществляется через этот новый сокет.

В SObjectizer для каждого типа канала (сокеты, именованные каналы (pipe), разделяемая память и др.) необходимо реализовать:

- интерфейс *so_4::rt::comm::io_channel_t* для передачи данных по уже установленному соединению;
- интерфейс *so_4::rt::comm::server_channel_t* для создания канала, к которому могут подключиться клиенты, определения момента подключения новых клиентов и создания независимых объектов *io_channel* для каждого из подключившихся клиентов (так, чтобы из *io_channel* можно было извлекать только данные, относящиеся к конкретному клиенту);
- интерфейс *so_4::rt::comm::client_factory_t* для подключения к каналу, созданному на удаленной стороне с помощью реализации интерфейса *so_4::rt::comm::server_channel_t*. Для каждого успешного подключения должен создаваться новый независимый объект *io_channel*, который будет использоваться только в рамках одной сессии (т.е. от момента установления соединения до момента разрыва соединения).

В состав SObjectizer уже входят средства для поддержки TCP/IP соединений (реализованные в пространстве имен *so_4::socket::channels*). Средства для поддержки других типов соединений, например, SSL, доступны в виде дополнительных библиотек.

В примере filter используется TCP/IP канал и штатные средства SObjectizer по поддержке TCP/IP соединений:

```
so_4::rt::comm::a_cln_channel_t * a_sock =
  new so_4::rt::comm::a_cln_channel_t(
    "a_sock",
    so_4::socket::channels::create_client_factory( sock_addr ),
    // Назначаем фильтр.
    filter,
    // Назначаем обработчик разрывов связи.
    so_4::rt::comm::a_cln_channel_base_t::
      create_def_disconnect_handler( 5000, 0 )
  );
```

Функция *create_client_factory* из пространства имен *so_4::socket::channels* создает объект, реализующий интерфейс *so_4::rt::comm::client_factory_t* и обеспечивающий подключение по TCP/IP к серверу с заданным в аргументе *sock_addr* адресом.

15.3.6 Обработчик разрывов связи

Агент типа `so_4::rt::comm::a_cln_channel_t` выполняет попытку установления соединения сразу после своей регистрации в SObjectizer. Если эта попытка завершилась успешно, то отсылается сообщение `msg_success`, реализуемое C++ классом *so_4::rt::comm::a_cln_channel_base_t::msg_success*. Если же соединение установить не удалось, то отсылается сообщение `msg_fail` класса *so_4::rt::comm::a_cln_channel_base_t::msg_fail*. Если распределенное приложение рассчитывает на то, что связь между его компонентами будет существовать всегда, то такому приложению достаточно обработать сообщения `msg_success` (для того, чтобы начать свою работу) и `msg_fail` (для того, чтобы завершить свою работу).

Но в реальной жизни рассчитывать на постоянное, гарантированное наличие связи не приходится. Более реалистичные сценарии работы заключаются в том, чтобы повторять попытки установления соединения, используя какие-то правила (политики). Самое простое правило — повторение попытки через определенные тайм-ауты. Заставить агент типа `so_4::rt::comm::a_cln_channel_t` повторить попытку установления соединения можно, отослав ему сообщение `msg_connect` (которое реализуется C++ типом *so_4::rt::comm::a_cln_channel_base_t::msg_connect*).

Для реализации простейшей политики повторений попыток установления соединения необходимо:

- подписаться на сообщения `msg_fail`;
- получив сообщение `msg_fail` отослать агенту типа `so_4::rt::comm::a_cln_channel_t` отложенное сообщение `msg_connect`.

Но задача усложняется еще и тем, что связь может рваться уже после того, как она будет установлена. Для этого нужно подписываться на сообщения `msg_client_disconnected`, которыми владеет агент-коммуникатор.

Очевидно, что реализация даже простой политики восстановления соединения требует выполнения некоторого количества пусть и несложных, но необходимых действий. Как минимум нужно выделить агента, который будет отслеживать состояние связи

(т.е. обрабатывать сообщения `msg_success`, `msg_fail`, `msg_client_disconnected`). Либо нужно включить эту функциональность в какого-то прикладного агента.

Для упрощения задачи управления попытками установления соединения в SObjectizer существует понятие т.н. *обработчика разрывов связи (disconnect_handler)*. Обработчик разрывов связи — это объект, реализующий интерфейс[2]:

```
class disconnect_handler_t
  : private cpp_util_2::nocopy_t
  {
  public :
    virtual ~disconnect_handler_t();

    //! Обработчик неудачного установления соединения.
    /*!
      Вызывается, если не удалось установить
      соединения с серверным сокетом.
    */
    virtual void
    on_connection_fail(
      //! Агент, который пытался установить соединение.
      const std::string & agent ) = 0;

    //! Обработчик разрыва соединения.
    /*!
      Вызывается при обнаружении разрыва ранее
      установленного соединения.
    */
    virtual void
    on_connection_lost(
      //! Агент, который поддерживал соединение.
      const std::string & agent ) = 0;
  };
```

Объект-обработчик назначается транспортному агенту в конструкторе. Когда транспортный агент обнаруживает, что попытка установления соединения завершилась неудачно, то вызывает у обработчика метод *on_connection_fail*. Если соединение было установлено, а затем потеряно, то у обработчика вызывается метод *on_connection_lost*.

SObjectizer предоставляет "штатную" реализацию обработчика разрывов связи. Она создается при помощи статического метода *create_def_disconnect_handler* класса *so_4::rt::comm::a_cln_channel_base_t*. Этот метод получает в качестве аргументов величины двух тайм-аутов: первый тайм-аут задает время, через которое транспортному агенту будет отослано сообщение `msg_connect` после неудачной попытки установления соединения (у обработчика вызван метод *on_connection_fail*), а второй тайм-аут задает время, через которое будет отослано сообщение `msg_connect` в случае разрыва связи (у обработчика вызван метод *on_connection_lost*).

[2]Полное имя: *so_4::rt::comm::a_cln_channel_base_t::disconnect_handler_t*

В примере filter транспортному агенту назначается "штатный" обработчик разрывов связи, который инициирует повторную попытку установления соединения через 5 секунд (5000 миллисекунд) после очередной неудачной попытки. Если же было потеряно уже установленное соединение, то повторная попытка установления соединения выполняется сразу же (задержка 0 миллисекунд):

```
so_4::rt::comm::a_cln_channel_t * a_sock =
  new so_4::rt::comm::a_cln_channel_t(
    "a_sock",
    so_4::socket::channels::create_client_factory( sock_addr ),
    // Назначаем фильтр.
    filter,
    // Назначаем обработчик разрывов связи.
    so_4::rt::comm::a_cln_channel_base_t::
      create_def_disconnect_handler( 5000, 0 )
  );
```

15.3.7 Транспортный агент как активный объект

Работа транспортных агентов построена на постоянном опросе состояния канала ввода-вывода. Для этого транспортные агенты используют специальные периодические сообщения, которые генерируются несколько раз в секунду. Если транспортный агент обнаруживает в канале готовые для чтения данные или нуждается в записи данных в канал, то темп генерации сообщений транспортного агента может увеличиться (т.е. агент будет отсылать их самому себе без задержки до окончания операции чтения/записи данных).

Естественно, что такой режим работы транспортного агента может оказывать негативное влияние на производительность всего приложения, если события транспортных агентов запускаются на обработку на одной нити диспетчера с другими агентами. Более того, могут быть случаи, когда такая диспетчеризация может вызвать продолжительные паузы в работе приложения. Например, при установке SSL-соединения операции SSL_connect, SSL_accept могут остановить вызвавшую их нить на длительное время. Если событие транспортного агента было запущено на единственной рабочей нити диспетчера, то на такое же время будет остановлена работа всего приложения.

Поэтому при использовании транспортных агентов желательно применять диспетчеры, позволяющие выделить транспортному агенту отдельную рабочую нить. Например, диспетчер с активными агентами или диспетчер с активными группами.

В данном примере используется диспетчер с активными агентами, и сразу после создания транспортному агенту назначается свойство, делающее его активным объектом:

```
a_sock->so_add_traits( so_4::disp::active_obj::
  query_active_obj_traits() );
```

Это возможно благодаря тому, что агент `so_4::rt::comm::a_cln_channel_t` не привязывается сам к конкретным типам диспетчеров, поэтому его можно сделать либо активным объектом (в случае диспетчера с активными объектами), либо членом активной группы (в случае диспетчера с активными группами).

15.3.8 Пороги ввода-вывода

Транспортные агенты осуществляют буферизированный ввод/вывод. Это означает, что как исходящие, так и входящие данные перед обработкой сначала накапливаются во внутреннем буфере агента. При этом используется механизм т.н. *порогов* для оптимизации работы. Более точно при выполнении операций ввода-вывода происходит следующее:

Входящие данные извлекаются из канала по мере их обнаружения в канале. Извлеченные данные записываются в *буфер входящих* данных, после чего производится попытка их разбора на SOP-пакеты. Успешно разобранные SOP-пакеты из буфера входящих данных изымаются и с помощью специальных сообщений отсылаются агенту-коммуникатору для обработки. Т.о., для SOP-каналов в буфере входящих данных находится только последний неполный SOP-пакет.

Но транспортный агент подсчитывает количество разобранных и отосланных коммуникатору SOP-пакетов и их суммарный объем. Как только количество пакетов или их объем превышает некоторый *порог входящих данных*, агент блокирует канал ввода-вывода. Т.е. данные из канала не извлекаются до тех пор, пока канал не будет разблокирован.

В случае SOP-каналов разблокированием канала занимается агент-коммуникатор — получив уведомление о том, что канал заблокирован, агент-коммуникатор дает команду на разблокирование канала. Но происходит это только после того, как агент-коммуникатор обработает все поступившие от транспортного агента SOP-пакеты. Т.е., если система сильно загружена, и заявки на обработку SOP-пакетов скапливаются где-то в очередях диспетчера, то транспортный агент заблокирует канал, и канал будет оставаться заблокированным до тех пор, пока система не “разгрузится”, т.е. пока диспетчер не обработает все заявки агента-коммуникатора.

Если канал находится в заблокированном состоянии слишком долго, то транспортный агент сам закрывает канал ввода-вывода, тем самым инициируя разрыв соединения. Это оправдано, т.к. не имеет смысла держать соединение, если система не успевает обрабатывать поступающие в соединение данные.

Транспортный агент также разрывает соединение, если его входящий буфер оказывается переполненным. В случае с SOP-каналами это означает, что осуществляется попытка передать очень большой SOP-пакет (более 1Mb).

Исходящие данные перед отправкой помещаются в *буфер исходящих* данных и находятся там до наступления следующей операции записи. Операция записи инициируется либо при поступлении очередного периодического сообщения транспортного агента, либо если количество пакетов в буфере исходящих данных или их суммарный объем превышает некоторый *порог исходящих данных*.

Такая буферизация сделана из-за того, что во многих сетевых коммуникациях, например, в TCP/IP, более эффективным является редкий обмен большими порциями данных, чем частый обмен небольшими порциями. Поэтому периодическое сообщение транспортного агента обеспечивает некоторую паузу между помещением пакета в буфер исходящих данных и отправкой этого пакета в канал. В течение этой паузы в буфер исходящих данных может быть помещено еще несколько пакетов, которые будут отправлены в канал одной операцией ввода-вывода.

T.o., при небольшом исходящем трафике максимальное время пребывания данных в буфере исходящих данных[3] будет равно периоду повторения специального сообщения транспортного агента (на момент написания этих строк период составлял приблизительно 250 миллисекунд).

При увеличении трафика происходит превышение порога исходящих данных: либо увеличивается количество пакетов, которые были помещены в буфер исходящих данных после последней операции записи, либо их суммарный объем превышает заданные величины. Если порог исходящих данных превышен, транспортный агент инициирует внеочередную операцию записи данных в канал. В этом случае максимальное время ожидания пакета в буфере исходящих данных может быть существенно меньше периода повторения специального сообщения транспортного агента.

Перед выполнением операции записи транспортный агент проверяет возможность записи данных в канал (посредством метода *io_channel_t::is_writeable*). Если запись в канал невозможна, то канал переходит в специальное состояние: *nonwriteable*. Если канал находится в nonwriteable-состоянии слишком долго, то транспортный агент закрывает канал и инициирует разрыв соединения.

Также транспортный агент закрывает канал, если объем данных в буфере исходящих данных становится слишком большим (более 1Mb). Как правило, эта ситуация возникает при большом исходящем трафике, когда физический транспортный канал не справляется с передачей данных.

Буфера входящих и исходящих данных являются сессионными, т.е. они действительны только для своего канала ввода-вывода. Как только канал ввода-вывода закрывается, содержимое буферов сбрасывается.

Для представления значений порогов ввода-вывода в SObjectizer предназначен тип:

```
class threshold_t
{
  public :
    threshold_t();
    threshold_t(
      unsigned int package_count,
      unsigned int traffic_bulk );

    threshold_t &
    operator+=( unsigned int package_size );

    bool
    is_exceeded( const threshold_t & o ) const;

    static threshold_t
    infinite();

  private :
```

[3] При отсутствии проблем с каналом ввода-вывода.

```
        //! Разрешенное количество прикладных пакетов.
        unsigned int  m_package_count;
        //! Разрешенный объем прикладных пакетов.
        unsigned int  m_traffic_bulk;
    };
```

Задать пороги ввода-вывода для канала можно при помощи методов:

```
void
set_in_threshold(
  const threshold_t & value );

void
set_out_threshold(
  const threshold_t & value );
```

которые определены в *so_4::rt::comm::a_cln_channel_base_t* и в *so_4::rt::comm::a_srv_channel_base_t*. Соответственно, для получения текущих порогов ввода-вывода предназначены методы:

```
const threshold_t &
in_threshold() const;

const threshold_t &
out_threshold() const;
```

В примере filter предусматривается небольшой исходящий трафик, поэтому для увеличения скорости отсылки сообщений на сервер для транспортного клиента устанавливается минимально возможный порог исходящих данных:

```
a_sock->set_out_threshold( so_4::rt::comm::threshold_t( 1, 1 ) );
```

Значения $(1, 1)$ указывают транспортному агенту инициировать внеочередную операцию записи в канал после помещения первого же пакета в буфер исходящих данных.

15.3.9 Запуск SObjectizer на отдельной нити

Функция *so_4::api::start()* блокирует вызвавшую ее нить до завершения работы SObjectizer Run-Time. Поскольку в примере filter выполняется интерактивный диалог с пользователем, то нельзя запустить SObjectizer на главной нити приложения из функции *main*. Выход в том, чтобы запустить SObjectizer на отдельной нити, которая стартует из функции *main* и завершается при возврате из *so_4::api::start()*. Такая вспомогательная нить реализуется в примере filter с помощью класса *thread_t* из библиотеки threads_1[4]:

```
class sobj_thread_t
  :  public threads_1::thread_t
  {
```

[4]Эта библиотека используется для поддержки многопоточности самим SObjectizer.

```
  public :
    sobj_thread_t();
    virtual ~sobj_thread_t();

  protected :
    virtual void
    body();
  };

sobj_thread_t::sobj_thread_t()
  {}

sobj_thread_t::~sobj_thread_t()
  {}

void
sobj_thread_t::body()
  {
    std::auto_ptr< so_4::timer_thread::timer_thread_t >
      timer_ptr( so_4::timer_thread::simple::create_timer_thread() );

    std::auto_ptr< so_4::rt::dispatcher_t >
      disp_ptr( so_4::disp::active_obj::create_disp( *timer_ptr ) );

    so_4::ret_code_t rc = so_4::api::start( *disp_ptr, 0 );
    if( rc )
      {
        std::cerr << "start: " << rc << std::endl;
      }
  }
```

Главную сложность при создании вспомогательной нити для запуска SObjectizer представляет определение момента, когда SObjectizer стартует. Сейчас SObjectizer не может проинформировать о том, что он успешно стартовал или не смог стартовать. Поэтому в функции *main* просто организована задержка:

```
sobj_thread_t thread;
thread.start();

// Засыпаем, чтобы дать стартовать SObjectizer.
// Это самый простой способ синхронизации с sobj_thread_t.
threads_1::sleep_thread( 1000 );
```

с расчетом на то, что пока главная нить будет "спать", произойдет старт вспомогательной нити и успешный запуск SObjectizer.

Гораздо проще с определением момента завершения работы SObjectizer — достаточно просто дождаться завершения вспомогательной нити:

```
thread.wait();
```

SObjectizer не накладывает ограничений на то, какие средства поддержки многопоточности использует прикладное приложение. SObjectizer не требует, чтобы применялась именно библиотека threads_1. Приложение может выбирать то, что ему удобно. Например, если приложению требуется использовать функцию *_begin_thread*, то пример filter мог бы содержать такие фрагменты:

```
void
sobj_thread( void * params )
  {
    bool * finished = reinterpret_cast< bool * >( params );

    std::auto_ptr< so_4::timer_thread::timer_thread_t >
      timer_ptr( so_4::timer_thread::simple::create_timer_thread() );

    std::auto_ptr< so_4::rt::dispatcher_t >
      disp_ptr( so_4::disp::active_obj::create_disp( *timer_ptr ) );

    so_4::ret_code_t rc = so_4::api::start( *disp_ptr, 0 );
    if( rc )
      {
        std::cerr << "start: " << rc << std::endl;
      }

    *finished = true;
  }

int
main( int argc, char ** argv )
  {
    ...
        bool sobj_finished = false;

        _begin_thread( sobj_thread, 8192, &sobj_finished );
        Sleep( 1000 );

        ...

        // Ожидаем завершения SObjectizer.
        while( !sobj_finished )
          ;
    ...
  }
```

Либо с использованием *QThread* из Qt[5]:

```
class SObjThread
  :  public QThread
```

[5] http://www.trolltech.com

```
  {
  protected :
    virtual void
    run()
    {
      std::auto_ptr< so_4::timer_thread::timer_thread_t >
        timer_ptr( so_4::timer_thread::simple::create_timer_thread() );

      std::auto_ptr< so_4::rt::dispatcher_t >
        disp_ptr( so_4::disp::active_obj::create_disp( *timer_ptr ) );

      so_4::ret_code_t rc = so_4::api::start( *disp_ptr, 0 );
      if( rc )
        {
          std::cerr << "start: " << rc << std::endl;
        }
    }
  };

int
main( int argc, char ** argv )
  {
    ...
        SObjThread sobjThread;

        sobjThread.start();
        QThread::sleep( 1 );

        ...

        // Ожидаем завершения SObjectizer.
        sobjThread.wait();
    ...
  }
```

15.3.10 Интерактивный диалог с пользователем

После того, как SObjectizer запущен на вспомогательной нити, главная нить приложения оказывается свободной. В примере filter это используется для организации на контексте главной нити интерактивного диалога с пользователем. В данном случае диалог организован самым тривиальным образом: в бесконечном цикле с консоли у пользователя запрашивается номер одного из доступных ему вариантов. Затем ввод пользователя обрабатывается. Завершение примера является одним из доступных пользователю вариантов.

15.4 Разбор файла server.cpp

По сравнению с исходным кодом первого и второго клиентов, код сервера не содержит ничего сложного. Отметить можно только то, что основной агент примера `a_srv` подписывает свои события `evt_c1_request` и `evt_c2_reply` на сообщения двух разных глобальных агентов. Подробнее же имеет смысл остановиться на создании транспортного агента для серверного входа и на обсуждении того, как именно сообщения глобальных агентов позволяют серверу общаться с обоими клиентами.

15.4.1 Транспортный агент для серверного входа

Как показано в 15.3.5 на стр. 137, каналы ввода-вывода в SObjectizer делятся на клиентские и серверные. Клиентские SOP-каналы в SObjectizer обслуживаются с помощью агентов типа `so_4::rt::comm::a_cln_channel_t`, а серверные SOP-каналы — с помощью агентов типа `so_4::rt::comm::a_srv_channel_t`.

В задачи агентов `so_4::rt::comm::a_srv_channel_t` входит:

- создание серверного канала связи. Осуществляется это с помощью переданного транспортному агенту объекта, реализующего интерфейс *so_4::rt::comm::server_channel_t*. Когда транспортный агент стартует, он предпринимает попытку создания серверного канала. Если это удается, то отсылается сообщение `msg_success`, которое имеет тип *so_4::rt::comm::a_srv_channel_base_t::msg_success*. В противном случае отсылается сообщение `msg_fail` типа *so_4::rt::comm::a_srv_channel_base_t::msg_fail*.

  В отличие от клиентских каналов для серверных транспортных агентов понятие обработчика разрывов связи не применимо, поскольку невозможность создания серверного канала (например, серверного TCP/IP сокета или отображаемого в разделяемую память файла) обычно должна рассматриваться как серьезная ошибка конфигурирования приложения, а не мелкая проблема времени исполнения;

- отслеживание подключения новых клиентов. При обнаружении нового подключения транспортный агент создает для клиента отдельный канал ввода-вывода (экземпляр *so_4::rt::comm::io_channel_t*), а также буфер входящих и исходящих данных. Созданный канал и соответствующие буфера используются затем для обмена данными с клиентом;

- обмен данными с подключившимися клиентами. Для обмена данными используется тот же механизм, который был описан в 15.3.8 на стр. 146;

- отслеживание отключения клиентов. При обнаружении отключившегося клиента транспортный агент уничтожает созданный для клиента канал ввода-вывода и буфера входящих и исходящих данных.

В примере filter при создании серверного транспортного агента используется "штатная" реализация интерфейса *so_4::rt::comm::server_channel_t* для TCP/IP каналов:

```
so_4::rt::comm::a_srv_channel_t * a_socksrv =
  new so_4::rt::comm::a_srv_channel_t(
    "a_srv_channel",
    so_4::socket::channels::create_server_channel(
      argv[ 1 ] ) );
```

Функция *create_server_channel* из пространства имен *so_4::socket::channels* создает объект, реализующий интерфейс *so_4::rt::comm::server_channel_t* и обеспечивающий создание серверного TCP/IP сокета с заданным адресом.

15.4.2 Взаимодействие сервера с клиентами

Главный агент сервера `a_srv` выполняет очень простые действия: получает сообщение `a_c1i.msg_request` и отсылает сообщение `a_c2i.msg_request`, затем получает `a_c2i.msg_reply` и отсылает `a_c1i.msg_reply`. При этом его не волнует, какой клиент и через какой канал связи подключен. Но сообщения все равно поступают туда, куда нужно — от клиентов к серверу, от сервера к клиентам и обратно.

Происходит это благодаря механизму глобальных агентов SObjectizer и фильтров (см. 15.3.4 на стр. 135). Когда первый клиент отсылает сообщение `a_c1i.msg_request`, SObjectizer на стороне клиента знает про существование канала связи, фильтр которого пропускает сообщения агента `a_c1i`. SObjectizer отсылает сообщение в этот канал, и сообщение попадает на сервер. Агент `a_srv` получает его и отсылает `a_c2i.msg_request`. SObjectizer на стороне сервера знает, что у него есть канал, который пропускает сообщения агента `a_c2i`, и SObjectizer передает сообщение в этот канал. Так сообщение `a_c2i.msg_request` приходит ко второму клиенту. С ответными сообщениями происходит та же самая история.

Если же в момент появления сообщения глобального агента SObjectizer не обнаруживает ни одного канала, фильтр которого пропускал бы сообщения этого глобального агента, то сообщение за рамки SObjectizer не выпускается, т.е. теряется. Именно это происходит, например, если к серверу подключен только первый клиент, и сервер получает `a_c1i.msg_request` — сообщение `a_c2i.msg_request` просто теряется, т.к. нет соединения со вторым клиентом.

Отсутствие гарантированной доставки сообщений глобальных агентов на удаленную сторону является фундаментальной особенностью текущих версий SObjectizer. Механизм глобальных агентов и коммуникационных каналов является базой, на основе которой можно строить различные решения проблемы гарантированной доставки. Но сам SObjectizer не предоставляет их, во-первых, чтобы оставаться простым и минималистическим инструментом, и, во-вторых, для предоставления возможностей построения или использования специализированных средств, которые могут поддерживать гарантированную однократную доставку, балансировку нагрузки, отказоустойчивость, восстанавливаемость и пр. Реализация подобных механизмов в виде штатных средств SObjectizer существенно усложнит SObjectizer Run-Time и, возможно, сделает невозможным совместное использование SObjectizer с конкурирующими продуктами, более приспособленными к той или иной задаче.

15.5 Резюме

- Если в C++ классе глобального агента оставить метод *so_on_subscription* чистым виртуальным методом, то нельзя будет создать и зарегистрировать через *register_coop* обычного агента этого типа. Но можно будет использовать этот тип для создания глобальных агентов.

- Глобальные агенты регистрируются в SObjectizer при помощи функции:

```
so_4::ret_code_t
make_global_agent(
  /*! Имя глобального агента. */
  const std::string & agent_name,
  /*! Тип глобального агента. */
  const std::string & agent_type );
```

- Функцию *make_global_agent* можно вызывать неоднократно для одного и того же глобального агента. Важно только, чтобы во всех случаях было одинаковое имя типа агента.

- Глобальные агенты не возникают автоматически. Поэтому, если какой-то агент хочет подписаться на сообщения глобального агента, он должен позаботиться о предварительном создании глобального агента. Например, это можно сделать, вызвав *make_global_agent* в методе *so_on_subscription* непосредственно перед подпиской события.

- SObjectizer различает клиентские и серверные соединения. Клиентские соединения обслуживаются транспортными агентами типа `so_4::rt::comm::a_cln_channel_t`, а серверные — `so_4::rt::comm::a_cln_channel_t`.

- При отсылке сообщения глобального агента в каком-то модуле неизвестно, кто подписан на это сообщение в других модулях. Поэтому сообщение глобального агента отсылается во **все** существующие соединения, фильтры которых разрешают сообщения данного глобального агента.

- Фильтр — это объект, который определяет, разрешена ли передача клиенту сообщений конкретного глобального агента. Фильтры предназначены для двух целей:

  1. Ограничение трафика. Клиенту со стороны сервера и серверу со стороны клиента отсылаются сообщения только тех глобальных агентов, которые разрешены фильтром.
  2. Разграничение доступа. Фильтр разрешает транспорт ограниченного числа сообщений и делает невозможным доставку сообщения агента, который запрещен фильтром.

- Фильтры в SObjectizer разделяются на входящие (назначаются серверным соединением) и исходящие (назначаются клиентским соединением), но реализуются одними и теми же C++ классами.

- Исходящее от клиента сообщение фильтруется дважды: сначала на стороне отсылающего сообщение клиента, а затем на стороне получившего сообщение сервера.

- На стороне сервера исходящий и входящий фильтры комбинируются: поступающие от клиента сообщения сначала пропускаются через входящий фильтр сервера, а затем через исходящий фильтр клиента.

- На данный момент SObjectizer предоставляет два готовых фильтра:

  – разрешающий все сообщения. Создается функцией *so_4::sop::create_all_enable_filter*;

  – "штатный" фильтр, разрешающий только сообщения указанных агентов. Штатный фильтр создается функцией *so_4::sop::create_std_filter*. Он реализует интерфейс *so_4::sop::std_filter_t* и позволяет задать имена глобальных агентов, чьи сообщения разрешены к передаче.

- Различные типы физических соединений в SObjectizer абстрагированы понятием *канала ввода-вывода*. Транспортные агенты работают с каналами ввода-вывода через объекты, реализующие интерфейс *so_4::rt::comm::io_channel_t*.

- Каналы ввода-вывода в SObjectizer делятся на клиентские и серверные. Для установления физического соединения на клиентском канале транспортному агенту `so_4::rt::comm::a_cln_channel_t` должен быть указан объект, реализующий интерфейс *so_4::rt::comm::client_factory_t*. Для создания и обслуживания серверного канала транспортному агенту `so_4::rt::comm::a_srv_channel_t` должен быть указан объект, реализующий интерфейс *so_4::rt::comm::server_channel_t*.

- Идея клиентских и серверных каналов ввода-вывода в SObjectizer аналогична идее сокетов: серверный канал сначала переводится в режим ожидания подключения (listening), а затем происходит ожидание подключения нового клиента (accepting). Оба этих действия должны выполняться реализацией интерфейса *so_4::rt::comm::server_channel_t*. Когда серверный канал принимает новое подключение, на стороне сервера появляется еще один объект-канал (экземпляр *so_4::rt::comm::io_channel_t*), не зависящий от исходного серверного канала. Взаимодействие с клиентом осуществляется через этот новый канал.

- В состав SObjectizer входят штатные средства для поддержки TCP/IP соединений. Так, реализация интерфейса *client_factory_t* создается функцией *so_4::socket::channels::create_client_factory*, а реализация интерфейса *server_channel_t* — функцией *so_4::socket::channels::create_server_channel*.

- Транспортный агент `so_4::rt::comm::a_cln_channel_t` владеет сообщениями `msg_success`, `msg_fail`, `msg_client_connected`, `msg_client_disconnected`, которые рассылаются при изменении состояния соединения.

- Транспортный агент `so_4::rt::comm::a_cln_channel_t` пытается создать соединение сразу после начала работы в SObjectizer. Если соединение установить не удалось, или уже установленное соединение было потеряно, то транспортный агент не пытается его восстановить самостоятельно. Если приложению необходимо автоматически восстанавливать разорванные соединения, то приложение может:

  - само обрабатывать сообщения транспортного агента и при потере соединения заставлять транспортного агента восстановить соединение (для этого транспортный агент владеет сообщением `msg_connect`), либо
  - назначить транспортному агенту *обработчик разрывов связи* — объект, реализующий интерфейс *so_4::rt::comm::a_cln_channel_base_t::disconnect_handler_t*. Транспортный агент сам вызывает методы этого объекта при возникновении проблем с соединением, что позволяет обработчику выполнять необходимые действия по восстановлению соединения.

- SObjectizer предоставляет штатную реализацию обработчика разрывов связи, которая создается с помощью статического метода *create_def_disconnect_handler* класса *so_4::rt::comm::a_cln_channel_base_t*

- Транспортный агент `so_4::rt::comm::a_srv_channel_t` владеет сообщениями `msg_success`, `msg_fail`, `msg_client_connected`, `msg_client_disconnected`, которые рассылаются при изменении состояния соединения. Но понятие обработчика разрывов связи для серверного соединения не применимо.

- Транспортные агенты выгодно делать активными агентами или членами активных групп агентов. В этом случае диспетчер выделяет транспортным агентам отдельную нить, что позволяет им выполнять длительные операции ввода-вывода, не оказывая влияния на других агентов.

- Транспортные агенты используют буферизированный ввод-вывод. Входящие данные сначала поступают в *буфер входящих данных*, затем в этом буфере данные разбираются на отдельные SOP-пакеты, после чего SOP-пакеты передаются агенту-коммуникатору на обработку. Исходящие данные сначала поступают в *буфер исходящих данных*, содержимое которого периодически передается в канал ввода-вывода.

- SObjectizer использует понятие *порогов ввода-вывода*. Порог – это пара значений: (количество пакетов, суммарный объем пакетов). Порог считается превышенным, если превышено хотя бы одно значение из этой пары.

- Для канала превышение *порога входящих данных* приводит к блокировке канала. Данные из заблокированного канала не извлекаются. Если канал находится в заблокированном состоянии слишком долго, он закрывается.

- Заблокированные SOP-каналы разблокируются агентом-коммуникатором автоматически.

- Превышение *порога исходящих данных* приводит к инициированию внеочередной записи содержимого буфера исходящих данных в канал.

- Если канал не готов для записи, он переводится в *nonwriteable*-состояние, и запись в него не производится, данные остаются в буфере исходящих данных. Если канал находится в состоянии *nonwriteable* слишком долго, то он закрывается.

- Буфера входящих и исходящих данных являются сессионными и уничтожаются вместе с соответствующим каналом связи.

- Для установки значений порогов ввода-вывода транспортные агенты предоставляют методы *set_in_threshold*, *set_out_threshold*. Получить текущие значения порогов ввода-вывода можно с помощью методов *in_threshold*, *out_threshold*.

- Функция *so_4::api::start* блокирует вызвавшую ее нить до завершения работы SObjectizer Run-Time. Если необходимо освободить главную нить, например, для выполнения интерактивного диалога с пользователем, то вызов *so_4::api::start* можно выполнить на контексте специально созданной для этого вспомогательной нити.

- Текущие версии SObjectizer не гарантируют надежной доставки сообщений. Т.е., если в момент отсылки сообщения глобального агента нет ни одного соединения, фильтр которого пропускает сообщения этого агента, то сообщение просто теряется.

Глава 16

Пример high_traffic

Пример high_traffic демонстрирует возможность передачи данных в распределенных приложениях при помощи сообщений глобальных агентов. В этом примере один сервер общается с произвольным числом клиентов через TCP/IP соединения. Для всех клиентов используется один и тот же интерфейс (глобальный агент), но для организации peer-to-peer взаимодействия используются идентификаторы коммуникационных каналов. Описание примера требует понимания SObjectizer Protocol (см. 5 на стр. 45) и примера filter (см. 15 на стр. 130), поскольку в данной главе внимание уделяется только техническим деталям передачи данных в SOP-соединениях.

Также в примере демонстрируется поведение транспортных агентов SObjectizer при большом трафике в коммуникационном канале.

Пример high_traffic состоит из следующих файлов:

common.ddl, common.cpp — глобальный агент, чьими сообщениями обмениваются сервер и клиенты (стр. 256, стр. 257).

client.cpp — исходный код клиента (стр. 259).

server.cpp — исходный код сервера (стр. 265).

16.1 Что делает пример

В пример входят два приложения: клиент и сервер. Сначала должен быть запущен сервер, которому указывается TCP/IP адрес для создания серверного сокета. Затем запускаются клиенты, которые будут подключаться к этому сокету.

Подключившись к серверу клиент начинает отсылать запросы на сервер. Каждый запрос представлен одним сообщением `msg_request` глобального агента `a_common`. Каждый запрос имеет уникальный идентификатор. Сервер получает от клиента запросы и отвечает на каждый запрос ответом с тем же самым идентификатором. Ответ — это сообщение `msg_reply` глобального агента `a_common`. Клиент продолжает отсылать запросы на сервер до тех пор, пока не получит ответы на все свои запросы.

Количество запросов и размер одного запроса указываются клиенту в виде аргументов командной строки. Там же клиенту указывается количество запросов, которые клиент может отослать на сервер за один раз (размер группы) и тайм-аут между отсылками групп. В течение этого тайм-аута клиент ожидает ответов сервера.

При отсылке очередной группы запросов клиент запоминает идентификаторы отосланных запросов. После истечения тайм-аута клиент повторяет те запросы, на которые от сервера не поступило ответов, и, если таких запросов было меньше размера группы, отсылает на сервер новые запросы. При поступлении ответа от сервера клиент вычеркивает идентификатор запроса из списка сохраненных идентификаторов. Когда клиент получает ответы на все свои отосланные запросы, он завершает работу.

Сервер для обработки запросов использует агента `a_receiver`, который подписывается на сообщение `msg_request` глобального агента `a_common`. Получив сообщение `a_common.msg_request` агент `a_receiver` просто отвечает сообщением `a_common.msg_reply`. Но если бы агент `a_receiver` просто отсылал сообщение `msg_reply`, то оно уходило бы всем подключенным в данный момент клиентам (т.е. была бы осуществлена т.н. broadcast рассылка). Агенту же `a_receiver` необходимо, чтобы ответное сообщение было доставлено именно тому клиенту, запрос которого обрабатывается (т.е. необходимо поддержать peer-to-peer взаимодействие). Для этого агент `a_receiver` отсылает сообщение `msg_reply` в тот канал, из которого поступило сообщение `msg_request`.

16.2 Разбор файла common.cpp

Файл `common.cpp` содержит реализацию глобального агента `a_common` и его сообщений `msg_request` и `msg_reply`.

Рассмотрим подробнее сообщения `msg_request` и `msg_reply`. Они реализуются с помощью C++ типов:

```
// Объекты этого типа передаются в сообщениях глобального агента.
class data_t
  : public oess_1::stdsn::serializable_t
  {
    OESS_SERIALIZER( data_t )
    public :
      data_t()
        {}

      data_t(
        // Размер данных в байтах.
        unsigned int size )
        : m_data( size, '0' )
        {}

      virtual ~data_t()
        {}

    private :
      // Размер и значение задается в инициализирующем конструкторе.
      std::string m_data;
  };

class a_common_t
```

```
: public  so_4::rt::agent_t
{
  public :
    ...
    // Тип сообщения-запроса к серверу.
    struct  msg_request
      {
        // Порядковый номер запроса.
        unsigned int  m_uid;
        // Данные запроса.
        data_t  m_data;

        msg_request()
          {}
        msg_request(
          unsigned int uid,
          unsigned int size )
          : m_uid( uid )
          , m_data( size )
          {}

        static bool
        check( const msg_request * cmd )
          {
            return ( 0 != cmd );
          }
      };

    // Тип сообщения-ответа от сервера.
    struct  msg_reply
      {
        // Порядковый номер запроса.
        unsigned int  m_uid;

        msg_reply()
          {}
        msg_reply(
          unsigned int uid )
          : m_uid( uid )
          {}

        static bool
        check( const msg_reply * cmd )
          {
            return ( 0 != cmd );
          }
      };
    ...
```

```
};
```

В структурах *a_common_t::msg_request* и *a_common_t::msg_reply* есть поле *m_uid*, в котором передается идентификатор запроса. Также в структуре *a_common_t::msg_request* есть поле *m_data* типа *data_t*, в котором передается содержимое запроса. При этом тип *data_t* сериализуется с помощью ObjESSty.

Поля *msg_request.m_uid*, *msg_request.m_data* и *msg_reply.m_uid* являются теми данными, которые должны передаваться в сообщениях глобального агента `a_common`. Для этого программисту требуется описать данные поля в описании сообщений `msg_request`, `msg_reply` для SObjectizer при помощи макросов *SOL4_MSG_FIELD*:

```
SOL4_MSG_START( msg_request, a_common_t::msg_request )
  SOL4_MSG_FIELD( m_uid )
  SOL4_MSG_FIELD( m_data )
  SOL4_MSG_CHECKER( a_common_t::msg_request::check )
SOL4_MSG_FINISH()

SOL4_MSG_START( msg_reply, a_common_t::msg_reply )
  SOL4_MSG_FIELD( m_uid )
  SOL4_MSG_CHECKER( a_common_t::msg_reply::check )
SOL4_MSG_FINISH()
```

Такое описание указывает SObjectizer, что при передаче сообщения `msg_request` нужно сериализовать/десериализовать поля *m_uid* и *m_data*. А при передаче сообщения `msg_reply` — только поле *m_uid*.

В макросах *SOL4_MSG_FIELD* можно указывать имена полей, которые удовлетворяют следующим условиям:

- являются доступными (public) полями;
- принадлежат либо примитивным C++ типам (*char*, *short*, *int* и их *unsigned*-аналогам, *float*, *double*), либо типу *std::string*, либо сериализуемому через ObjESSty типу;
- являются одиночными объектами (т.е. не указателями и не векторами).

Если в сообщении требуется передать поле-вектор, то оно должно описываться с помошью макроса *SOL4_MSG_FIELD_ARRAY*:

```
struct  msg_set_key
  {
    unsigned char m_key[ 8 ];
    ...
  };

SOL4_MSG_START( msg_set_key, msg_set_key )
  SOL4_MSG_FIELD_ARRAY( m_key )
  ...
SOL4_MSG_FINISH()
```

16.3 Разбор файла client.cpp

При разборе файла `client.cpp` остановиться следует только на одном моменте: обработке сообщений агента-коммуникатора `msg_client_connected`, `msg_client_disconnected`.

Агент-коммуникатор владеет сообщениями `msg_client_connected`, `msg_client_disconnected`, которые отсылаются в момент установления и разрыва любого SOP-соединения. При этом не важно, отсылается ли это сообщение в приложении-клиенте, которое подключилось к серверу, или в приложении-сервере, которое обнаружило подключение нового клиента. Как только соединение было обнаружено, отсылается сообщение `msg_client_connected`. При обнаружении разрыва соединения отсылается сообщение `msg_client_disconnected`.

Сообщения `msg_client_connected`, `msg_client_disconnected`, которыми владеет агент-коммуникатор, рассылаются при возникновении/разрыве любого SOP-соединения вне зависимости от того, какой транспортный агент обслуживает это соединение. В данном примере существует всего один транспортный агент и всего одно соединение в один момент времени. Поэтому клиент предполагает, что возникновение сообщения `msg_client_connected` означает наличие связи с сервером, а возникновение `msg_client_disconnected` — исчезновение связи.

В приложениях же, в которых существуют несколько транспортных агентов, обработка сообщений `msg_client_connected`, `msg_client_disconnected`, которыми владеет агент-коммуникатор, требует осторожности, т.к. эти сообщения могут относиться к каналам, обслуживаемым разными транспортными агентами. Для того, чтобы определить, к какому каналу относится сообщение, необходимо проанализировать идентификатор канала, который содержится в поле *`m_channel`*.

Более удобным способом контроля соединений, обслуживаемых конкретным транспортным агентом, является подписка на сообщения `msg_client_connected`, `msg_client_disconnected`, которыми владеет сам транспортный агент. Исторически[1] сложилось так, что в SObjectizer при создании нового SOP-соединения отсылается два сообщения `msg_client_connected`: одно принадлежит агенту-коммуникатору, а второе транспортному агенту. Аналогично и с сообщением `msg_client_disconnected` — оно также рассылается от имени двух агентов.

Главный агент клиента `a_sender` использует обработку сообщений `msg_client_connected`, `msg_client_disconnected` для того, чтобы сохранить у себя признак наличия соединения. Этот признак затем используется только при выдаче очередной порции статистики отосланных запросов и полученных ответов. Но агента `a_sender` можно было бы реализовать так, чтобы при отсутствии связи он не предпринимал попыток отсылки сообщений.

16.4 Разбор файла server.cpp

Главный агент сервера `a_receiver` так же, как и агент `a_sender` клиента, обрабатывает сообщения `msg_client_connected`, `msg_client_disconnected`, которыми владеет транспортный агент. Обработка этих сообщений на сервере носит исключительно демонстрационный характер. Обработчики сообщений `msg_client_connected`,

[1] Транспортные агенты для SOP-соединений стали владеть сообщениями `msg_client_connected`, `msg_client_disconnected` начиная с SObjectizer v.4.2.7.

`msg_client_disconnected` просто печатают на стандартный поток ошибок состояние соединения. Эта печать позволяет мониторить состояние подключений клиентов к серверу.

Главное, на что следует обратить внимание в коде сервера — это отсылка ответного сообщения клиенту:

```
so_4::api::send_msg_safely(
  data.channel(),
  a_common_t::agent_name(),
  "msg_reply",
  new a_common_t::msg_reply( cmd->m_uid ) );
```

Здесь используется вариант функции *so_4::api::send_msg_safely*, который получает первым аргументом идентификатор коммуникационного канала, из которого было получено сообщение-инцидент. Именно с помощью этого идентификатора организуется peer-to-peer взаимодействие: ответное сообщение уходит в тот канал, из которого поступил запрос.

Каждому коммуникационному каналу в SObjectizer соответствует уникальный идентификатор канала (см. так же 5.4 на стр. 48). Идентификатор канала представляется в SObjectizer объектами типа *so_4::rt::comm_channel_t* [2].

Для каждого события в SObjectizer известно, из какого канала поступило сообщение-инцидент данного события. Идентификатор канала сообщения-инцидента доступен через метод *channel()* класса *so_4::rt::event_data_t*. Если же сообщение было порождено внутри SObjectizer-приложения, то метод *channel()* будет возвращать специальное значение: *localhost* [3]. Значение *localhost* также можно указывать как идентификатор канала в функциях *so_4::api::send_msg_safely*, *so_4::api::send_msg* — в этом случае сообщение будет обрабатываться как обычное сообщение.

16.5 Парность сообщений msg_client_connected, msg_client_disconnected

Исходя из того, что сообщение `msg_client_connected` отсылается при установлении SOP-соединения, а сообщение `msg_client_disconnect` при разрыве, можно предположить, что для каждого канала эти сообщения рассылаются попарно. На самом деле это не всегда так. Могут быть случаи, когда сообщение `msg_client_connected` отсутствует, а возникает только сообщение `msg_client_disconnected`.

Происходит это потому, что для установления SOP-соединения два SObjectizer-приложения после физического соединения выполняют процедуру *handshake*: устанавливают трансформаторы трафика (например, zip-ование) и обмениваются фильтрами. SOP-соединение считается успешно установленным только после успешного завершения процедуры handshake, и только в этом случае рассылаются сообщения `msg_client_connected`.

[2] Немного подробнее структура идентификатора канала рассматривается в примере raw_channel. Для полной информации о типе *so_4::rt::comm_channel_t* следует обратиться к SObjectizer On-line Reference Manual.

[3] Оно возвращается методом *localhost()* класса *so_4::rt::comm_channel_t*.

Сообщение `msg_client_disconnected` рассылается всегда при разрыве физического соединения вне зависимости от того, завершилась ли процедура handshake успешно или нет. Поэтому, если в момент выполнения handshake связь между SObjectizer-приложениями разорвалась, сообщение `msg_client_disconnected` будет отослано для канала, для которого не отсылалось сообщение `msg_client_connected`.

16.6 Результат работы примера

Несмотря на свою простоту пример high_traffic показывает интересные результаты, которые зависят от многих факторов. В частности, можно увидеть:

- поведение клиента в отсутствие сервера. Клиент повторяет попытки подключения к серверу самостоятельно;
- влияние порогов ввода-вывода на трафик. Можно увидеть, что сервер получает запросы не по одному, а порциями. При этом размер порции связан с порогом исходящих данных клиента. Можно так подобрать размер одного запроса, что каждый запрос будет поступать на сервер отдельно;
- автоматическое закрытие и восстановление соединения при слишком большом трафике. Подобрав не очень большой размер запроса и большой размер группы при маленьком тайм-ауте можно увидеть, как клиент разрывает соединение с сервером из-за того, что сервер не успевает считывать все запросы из канала;
- автоматическое закрытие канала при слишком больших объемах ожидающих отправки данных. Достичь этого можно, выбрав большой размер запроса.

Можно также создать ситуацию, когда клиент будет повторять одни и те же запросы, сервер будет их получать и отсылать ответы, но ответы не будут доходить до клиента из-за большого встречного трафика.

К сожалению, привести след работы примера для всех этих ситуаций не представляется возможным, т.к. статический след не передает динамику взаимодействия клиента и сервера.

16.7 Резюме

- Поля сообщений, которые должны быть доступны к передаче по SOP-соединению, необходимо описывать для SObjectizer с помощью макросов *SOL4_MSG_FIELD*, *SOL4_MSG_FIELD_ARRAY*:

```
SOL4_MSG_START( msg_request, a_common_t::msg_request )
  SOL4_MSG_FIELD( m_uid )
  SOL4_MSG_FIELD( m_data )
  SOL4_MSG_FIELD_ARRAY( m_key )
SOL4_MSG_FINISH()
```

  Макросы *SOL4_MSG_FIELD* используются для полей, являющихся одиночными объектами (не указателями и не векторами). Макросы *SOL4_MSG_FIELD_ARRAY* используются для полей-векторов.

- Для передачи по SOP-соединениям могут использоваться поля типов *char*, *short*, *int* и их *unsigned*-аналоги, *float*, *double*, *std::string*, сериализуемых через ObjESSty типов.
- Поля, которые указываются в макросах *SOL4_MSG_FIELD*, *SOL4_MSG_FIELD_ARRAY*, должны быть public-полями.
- При установлении любого SOP-соединения отсылаются два сообщения `msg_client_connected`: одно из них принадлежит коммуникационному агенту, второе — транспортному агенту, который обслуживает это соединение.
- При разрыве любого SOP-соединения отсылаются два сообщения `msg_client_disconnected`: одно из них принадлежит коммуникационному агенту, второе — транспортному агенту, который обслуживает это соединение.
- Сообщение `msg_client_disconnected` может рассылаться для каналов, для которых не было отослано `msg_client_connected`. Происходит это в случае, когда физическое соединение разрывается прежде, чем будет установлено логическое SOP-соединение.
- Каждый коммуникационный канал в SObjectizer имеет уникальный идентификатор, представляемый типом *so_4::rt::comm_channel_t*. Идентификатор канала передается в поле *m_channel* сообщений `msg_client_connected`, `msg_client_disconnected`.
- С помощью метода *so_4::rt::event_data_t::channel()* можно получить идентификатор канала, из которого было получено сообщение-инцидент.
- SObjectizer предоставляет варианты функций *so_4::api::send_msg*, *so_4::api::send_msg_safely*, которые позволяют отослать сообщение в конкретный коммуникационный канал. Эти функции совместно с методом *channel()* класса *so_4::rt::event_data_t* позволяют организовать peer-to-peer взаимодействие SObjectizer-приложений.

Глава 17

Пример raw_channel

Пример raw_channel демонстрирует применение коммуникационных средств SObjectizer для работы с т.н. *raw*-соединениями. Описание примера требует понимания примеров filter (см. 15 на стр. 130), high_traffic (см. 16 на стр. 158), поскольку в данной главе используются понятия коммуникационных каналов, идентификаторов коммуникационных каналов, порогов ввода-вывода.

Пример raw_channel состоит из следующих файлов:

tcp_cln.cpp — исходный код клиента (стр. 269).

tcp_srv.cpp — исходный код сервера (стр. 275).

17.1 Что такое raw-канал

Понятие *raw*-канала было введено в SObjectizer v.4.2.4 для того, чтобы дать возможность приложениям использовать коммуникационные возможности SObjectizer (интерфейсы *so_4::rt::comm::io_channel_t*, *so_4::rt::comm::client_factory_t*, *so_4::rt::comm::server_channel_t*) для организации собственных типов взаимодействия. Например, для применения TCP/IP соединений в реализации HTTP- или SMTP-клиентов.

Интерфейс *so_4::rt::comm::io_channel_t* предназначен для того, чтобы добавлять в SObjectizer возможность работы по разным типам соединений, например, таким, как именованные каналы (pipes) и разделяемая память. Причем *so_4::rt::comm::io_channel_t* используется только для передачи и приема сырых (raw), необработанных данных, а их преобразование в SOP-пакеты осуществляют транспортные агенты (*so_4::rt::comm::a_cln_channel_t*, *so_4::rt::comm::a_srv_channel_t*). Идея raw-каналов состоит в том, чтобы дать возможность приложению обработать весь трафик самостоятельно — дать доступ к raw-данным. При этом приложения получают возможность использовать те же реализации интерфейса *io_channel_t*, которые применяются для SOP-соединений.

Для этого в SObjectizer реализовано два класса агентов: *so_4::rt::comm::a_raw_cln_channel_t*, *so_4::rt::comm::a_raw_srv_channel_t*. Они являются аналогами агентов *so_4::rt::comm::a_cln_channel_t*, *so_4::rt::comm::a_srv_channel_t*, более того, они используют общие базовые классы. Но, в отличие от транспортных агентов для SOP-каналов, агенты `a_raw_cln_channel_t`

и a_raw_srv_channel_t не занимаются каким-либо анализом или преобразованием трафика.

Т.о., *raw-канал* — это канал, который создан с использованием интерфейсов *so_4::rt::comm::io_channel_t*, *so_4::rt::comm::client_factory_t* или *so_4::rt::comm::server_channel_t*, и для работы с которым применяются агенты *so_4::rt::comm::a_raw_cln_channel_t* или *so_4::rt::comm::a_raw_srv_channel_t*.

Агент типа a_raw_cln_channel_t, как и агент типа a_cln_channel_t, используется для реализации клиентского соединения. Агенты a_raw_cln_channel_t владеют такими же сообщениями msg_success, msg_fail, msg_client_connected, msg_client_disconnected. Кроме того, агенты типа a_raw_cln_channel_t могут использовать обработчики разрывов соединений.

Агент типа a_raw_srv_channel_t, как и агент типа a_srv_channel_t, используется для реализации серверного соединения. Агенты a_raw_srv_channel_t владеют такими же сообщениями msg_success, msg_fail, msg_client_connected, msg_client_disconnected.

Агенты для raw-соединений используют ту же самую схему буферизации данных, что и транспортные агенты.

Для raw-соединений также назначаются идентификаторы каналов, которые также представляются объектами типа *so_4::rt::comm_channel_t*. Но идентификатор raw-канала бесполезно использовать для отсылки сообщения функциями *so_4::api::send_msg* — сообщение никуда не уйдет, т.к. этот канал не контролируется агентом-коммуникатором.

Вообще, между транспортными агентами и агентами для SOP-соединений есть очень много общего, т.к. они используют одинаковую реализацию, унаследованную из общих базовых типов. Главное отличие состоит в том, что транспортные агенты все входящие данные отсылают напрямую агенту-коммуникатору и исходящие данные получают от агента-коммуникатора. Агенты для SOP-соединений рассылают входящие данные при помощи своих сообщений msg_raw_package и получают исходящие данные в виде своих сообщений msg_send_package. При этом в сообщении msg_raw_package указывается идентификатор канала, из которого данные были получены. А в сообщении msg_send_package должен быть указан идентификатор канала, в который данные должны быть записаны[1].

Еще одним важным отличием raw-соединений от SOP-соединений является то, что сообщения msg_client_connected, msg_client_disconnected для raw-канала всегда отсылаются парами. Т.е. для raw-канала не может появиться сообщение msg_client_disconnected, если предварительно не было сообщения msg_client_connected для этого канала.

17.2 Структура идентификатора коммуникационного канала

Идентификатор коммуникационного канала в SObjectizer состоит из двух частей: имени агента, который обслуживает данный канал, и имени этого канала в рамках агента. При этом агент должен гарантировать, что имя будет уникальным.

[1] Это особенно актуально для агентов типа a_raw_srv_channel_t, которые в один момент времени могут обслуживать несколько каналов одновременно.

Благодаря такой структуре обеспечивается уникальность идентификаторов коммуникационных каналов — уникальность имен агентов гарантируется SObjectizer, а уникальность имен внутри агента гарантирует сам агент.

Класс *so_4::rt::comm_channel_t* предоставляет метод *comm_agent()* для получения имени агента, который обслуживает коммуникационный канал, и метод *client()* для получения имени канала в рамках агента. Оба эти метода обычно применяются при отсылке сообщения `msg_send_package`: этим сообщением владеет агент, который обслуживает коммуникационный канал. Поэтому сообщение `msg_send_package` нужно отсылать от имени обслуживающего канал агента, для чего используется метод *comm_agent()*. В самом же сообщении `msg_send_package` должно быть указано имя канала в рамках агента, и для указания этого имени применяется метод *client()*. Именно такой способ отсылки исходящих данных показан в файле `tcp_cln.cpp`.

17.3 Что делает пример

В состав примера входят два приложения: tcp_cln (файл `tcp_cln.cpp`) и tcp_srv (файл `tcp_srv.cpp`), которые демонстрируют работу с клиентскими и серверными TCP/IP соединениями соответственно. Они могут использоваться как совместно (т.е. tcp_cln будет подключаться к tcp_srv), так и по отдельности. Например, tcp_srv может использоваться для получения HTTP-запросов, которые отсылаются web-браузером. А приложение tcp_cln может использоваться для генерации HTTP-запросов к web-серверам.

Приложение tcp_cln делает одну попытку подключения к удаленной стороне. Если это удается, то tcp_cln отсылает в виде отдельных пакетов все строки, заданные tcp_cln в командной строке. После чего tcp_cln переходит в режим чтения входящих данных. Завершение tcp_cln происходит либо после неудачной попытки установки соединения, либо после разрыва соединения удаленной стороной.

Приложение tcp_srv работает только в режиме ожидания входящих данных: вся поступающая информация отображается на стандартный поток вывода, но в ответ ничего не отсылается, и соединения не закрываются.

17.4 Разбор файлов tcp_cln.cpp и tcp_srv.cpp

В файлах `tcp_cln.cpp` и `tcp_srv.cpp` имеет смысл обратить внимание лишь на то, как исходящие данные отсылаются в raw-канал, и как входящие данные поступают из raw-канала.

Для отсылки исходящих данных в raw-канал необходимо отослать сообщение `msg_send_package` агенту, который обслуживает этот канал:

```
so_4::api::send_msg_safely(
  cmd->m_channel.comm_agent(), "msg_send_package",
  new so_4::rt::comm::msg_send_package(
    cmd->m_channel.client(),
    data ) );
```

Главная сложность здесь в том, что нужно знать, во-первых, имя обслуживающего канал агента, и, во-вторых, нужно знать имя канала в рамках этого агента. В примере raw_channel эта проблема решается за счет того, что исходящие данные

отсылаются во время обработки сообщения `msg_client_connected`, а в этом сообщении находится идентификатор канала. Поэтому имя обслуживающего канал агента и имя канала в рамках агента извлекаются из идентификатора канала с помощью методов *`comm_agent()`* и *`client()`*. Аналогичным решением можно воспользоваться и при отсылке исходящих данных в ответ на входящие данные во время обработки `msg_raw_package`, т.к. в этом сообщении также содержится идентификатор канала.

Но что делать, если исходящие данные требуется отослать в произвольный момент времени после того, как соединение будет установлено? Единственным решением в этом случае будет предварительная обработка сообщения `msg_client_connected` и сохранение идентификатора канала из него для того, чтобы воспользоваться сохраненным идентификатором при отсылке `msg_send_package`.

Обработка входящих данных требует внимания всего к двум моментам: входящие данные доставляются посредством сообщения `msg_raw_channel`, которым владеет обслуживающий канал агент, и программисту нужно самому заботиться о регулярном разблокировании канала. Проще говоря, агенту, который обрабатывает входные данные из raw-канала, необходимо: 1) подписаться на сообщение `msg_raw_channel` соответствующего агента `a_raw_cln_channel_t` и 2) при обработке сообщения `msg_raw_channel` вызывать метод *`msg_raw_channel::unblock_channel`*. Именно так поступает главный агент приложения tcp_cln:

```
void
a_main_t::so_on_subscription()
  {
    ...

    so_subscribe( "evt_incoming_data", tcp_agent_name(),
      "msg_raw_package" );
  }

void
a_main_t::evt_incoming_data(
  const so_4::rt::comm::msg_raw_package * cmd )
  {
    std::cout << so_query_name() << ".evt_incoming_data: "
      << cmd->m_channel.comm_agent() << ", "
      << cmd->m_channel.client()
      << "\n\tdata size: " << cmd->m_package.size()
      << "\n\tis channel blocked: " << cmd->m_is_blocked
      << std::endl;

    std::string v(
      (const char *)cmd->m_package.ptr(),
      cmd->m_package.size() );
    std::cout << v << std::endl;

    // Если канал оказался заблокированным, то разблокируем его.
    cmd->unblock_channel();
  }
```

Как показано в 15.3.8 на стр. 146, коммуникационный канал блокируется, когда объем входящих данных превышает порог входящих данных. При этом чтение новых данных из заблокированного канала прекращается до тех пор, пока канал не будет разблокирован. Для SOP-соединений за разблокирование канала отвечает агент-коммуникатор. А для raw-соединений за разблокирование канала должен отвечать прикладной программист.

Агент, обслуживающий raw-канал, сигнализирует о блокировке канала посредством поля *msg_raw_package::m_is_blocked*. Оно имеет значение *true*, если канал заблокирован, или *false* в противном случае. Для разблокировки канала необходимо отослать сообщение `msg_unblock_channel`, которым владеет обслуживающий канал агент. Обычно для контроля поля *m_is_blocked* и отсылки `msg_unblock_channel` достаточно использовать метод *unblock_channel* сообщения `msg_raw_package`. Но, если программиста по каким-то причинам этот способ не устраивает, он может отослать `msg_unblock_channel` самостоятельно.

В приложении tcp_cln для raw-соединения устанавливаются минимальные пороги входящих и исходящих данных. Это означает, что все исходящие пакеты будут сразу же отосланы в канал, а сам канал будет оказываться заблокированным после каждого полученного входящего пакета данных.

17.5 Результат работы примера

Если запустить приложение tcp_cln, указав в командной строке:

```
sample_raw_channel_tcp_cln :80 "GET / HTTP/1.1\nHost: localhost"
```

(т.е. запросить по HTTP у локального web-сервера стартовую страницу), то результатом работы примера может быть:

```
a_main.evt_client_connected: a_tcp_srvsock, {client_channel_client 1}
sending data
a_main.evt_success
a_main.evt_incoming_data: a_tcp_srvsock, {client_channel_client 1}
  data size: 1932
  is channel blocked: 1
HTTP/1.1 200 OK
Date: Thu, 15 Jul 2004 03:47:02 GMT
Server: Apache/2.0.43 (Win32)
Content-Location: index.html.ru.cp866
Vary: negotiate,accept-language,accept-charset
TCN: choice
Last-Modified: Fri, 28 Sep 2001 07:55:50 GMT
ETag: "da90-60d-400cd80;da95-9c0-1af2e600"
Accept-Ranges: bytes
Content-Length: 1549
Content-Type: text/html; charset=cp866
Content-Language: ru
```

```
<!doctype html public "-//w3c//dtd html 4.0 transitional//en">
<html>
...
</html>

a_main.evt_client_disconnected: a_tcp_srvsock, {client_channel_client 1}
```

Если запустить приложение tcp_srv, задав в качестве адреса серверного сокета ":80", а затем в web-браузере запросить страницу по URL http://localhost, то можно получить следующий результат:

```
a_main.evt_success
a_main.evt_client_connected: a_tcp_srvsock, {server_channel_client 1}
a_main.evt_incoming_data: a_tcp_srvsock, {server_channel_client 1}
  data size: 453
GET / HTTP/1.1
User-Agent: Opera/7.51 (Windows NT 5.1; U)  [en]
Host: localhost
Accept: text/html, application/xml;q=0.9,
  application/xhtml+xml, image/png, image/jpeg,
  image/gif, image/x-xbitmap, */*;q=0.1
Accept-Language: ru;q=1.0,en;q=0.9
Accept-Charset: windows-1251, utf-8, utf-16, iso-8859-1;q=0.6, *;q=0.1
Accept-Encoding: deflate, gzip, x-gzip, identity, *;q=0
Connection: Keep-Alive, TE
TE: deflate, gzip, chunked, identity, trailers


a_main.evt_client_disconnected: a_tcp_srvsock, {server_channel_client 1}
```

17.6 Резюме

- *raw-канал* — это канал, который создан с использованием интерфейсов *so_4::rt::comm::io_channel_t*, *so_4::rt::comm::client_factory_t* или *so_4::rt::comm::server_channel_t*, и для работы с которым применяются агенты *so_4::rt::comm::a_raw_cln_channel_t* или *so_4::rt::comm::a_raw_srv_channel_t*.
- Агенты для raw-соединений во многом аналогичны транспортным агентам SOP-соединений, т.к. в обоих случаях используется одна и та же реализация, унаследованная из общего базового типа.
- Агент типа `a_raw_cln_channel_t` используется для реализации клиентского соединения. Агенты `a_raw_cln_channel_t` владеют сообщениями `msg_success`, `msg_fail`, `msg_client_connected`, `msg_client_disconnected`. Агенты типа `a_raw_cln_channel_t` могут использовать обработчики разрывов соединений.
- Агент типа `a_raw_srv_channel_t` используется для реализации клиентского соединения. Агенты `a_raw_srv_channel_t` владеют сообщениями `msg_success`, `msg_fail`, `msg_client_connected`, `msg_client_disconnected`.

- Агенты для raw-соединений используют ту же самую схему буферизации данных, что и транспортные агенты.
- Для raw-соединений также назначаются идентификаторы каналов, которые также представляются объектами типа *so_4::rt::comm_channel_t*. Идентификатор raw-канала нельзя использовать для отсылки сообщения функциями *so_4::api::send_msg*, т.к. этот канал не контролируется агентом-коммуникатором.
- Агенты для SOP-соединений рассылают входящие данные при помощи своих сообщений `msg_raw_package` и получают исходящие данные в виде своих сообщений `msg_send_package`. При этом в сообщении `msg_raw_package` указывается идентификатор канала, из которого данные были получены. А в сообщении `msg_send_package` должен быть указан идентификатор канала, в который данные должны быть записаны.
- Для raw-соединений (в отличие от SOP-соединений) сообщения `msg_client_connected`, `msg_client_disconnected` всегда отсылаются парами. Т.е. для raw-канала не может появиться сообщение `msg_client_disconnected`, если предварительно не было сообщения `msg_client_connected` для этого канала.
- Идентификатор коммуникационного канала в SObjectizer состоит из двух частей: имени агента, который обслуживает данный канал, и имени этого канала в рамках агента. При этом агент должен гарантировать, что имя будет уникальным.
- Класс *so_4::rt::comm_channel_t* предоставляет метод *comm_agent()* для получения имени агента, который обслуживает коммуникационный канал, и метод *client()* для получения имени канала в рамках агента.
- Для передачи исходящих данных в raw-канал необходимо отослать сообщение `msg_send_package` агенту, который обслуживает этот канал. При обработке сообщений `msg_client_connected` или `msg_raw_package` имя агента и имя канала в агенте можно извлечь из идентификатора канала, содержащегося в данных сообщениях. Если сообщение `msg_send_package` необходимо отослать в произвольный момент времени, то предварительно нужно обработать сообщение `msg_client_connected` (либо `msg_raw_package`) и сохранить содержащийся в нем идентификатор канала.
- Входящие данные доставляются посредством сообщения `msg_raw_package`, которым владеет обслуживающий канал агент.
- Поле *m_is_blocked* сообщения `msg_raw_package` содержит *true*, если канал оказался заблокированным из-за превышения порога входящих данных. В этом случае чтение данных из канала будет остановлено до разблокировки канала. Для разблокирования канала необходимо отослать сообщение `msg_unblock_channel`, которым владеет обслуживающий канал агент.
- Самым простым способом контроля блокированности канала является вызов метода *unblock_channel* у экземпляра сообщения `msg_raw_package` при получении каждого сообщения `msg_raw_package`. Этот метод сам проверит значение поля *m_is_blocked* и, если нужно, отошлет сообщение `msg_unblock_channel`.

Глава 18

Пример parent_insend

Пример parent_insend демонстрирует использование родственных отношений между кооперациями и применение *insend*-событий. Несмотря на небольшой объем примера для его понимания необходимо знакомство со всеми особенностями SObjectizer, описанных в предыдущих примерах. Также необходимо знать о взаимоотношениях "родитель-потомок" между кооперациями (3.3.1 на стр. 34) и об insend-событиях (4.6 на стр. 42).

Пример parent_insend состоит из одного файла `main.cpp` (стр. 281).

18.1 Что делает пример

Приложение-пример работает в интерактивном режиме. Оператор указывает момент регистрации и дерегистрации кооперации маршрутизатора (агента с именем `a_router`). Агент-маршрутизатор в своем событии `evt_start` создает дочернюю кооперацию с агентом серверного сокета (агента `a_server`). При этом агент `a_router` является динамическим агентом и регистрируется при помощи динамической кооперации. Но агент `a_server` является атрибутом агента `a_router`, т.е. статическим агентом, и регистрируется при помощи статической кооперации (т.е. посредством класса *so_4::rt::agent_coop_t*).

Далее агент-маршрутизатор отслеживает сообщения о подключениях новых клиентов. Для каждого нового клиента создается динамический агент-обработчик (слушатель). Все приходящие от клиента данные пересылаются маршрутизатором агентам-обработчикам.

Агент серверного сокета и агенты-обработчики являются активными агентами. Агент-маршрутизатор является пассивным агентом, но он использует insend-события.

18.2 Разбор файла main.cpp

Первое, на что нужно обратить внимание в примере parent_insend — это то, что агент `a_server` и кооперация для него являются атрибутами агента `a_router`:

```
class a_router_t
  : public so_4::rt::agent_t
  {
  private :
    so_4::rt::comm::a_raw_srv_channel_t m_srv_channel;
```

```
    so_4::rt::agent_coop_t  m_srv_channel_coop;

    ...
  public :
    a_router_t(
      const std::string & ip )
      : base_type_t( router_agent_name )
      , m_srv_channel( server_agent_name,
          so_4::socket::channels::create_server_channel( ip ) )
      , m_srv_channel_coop( m_srv_channel )
      , m_child_counter( 0 )
      { ... }
    ...
  };
```

При этом сам агент `a_router` регистрируется в SObjectizer как динамический агент при помощи динамической кооперации:

```
void
create_coop( const std::string & ip )
  {
    so_4::rt::dyn_agent_coop_helper_t helper(
      new so_4::rt::dyn_agent_coop_t(
        new a_router_t( ip ) ) );
  }
```

Когда же агент `a_router` начинает работать в SObjectizer (т.е. при обработке сообщения `msg_start` агента `a_sobjectizer`), он регистрирует агента `a_server` как статического агента:

```
so_4::api::register_coop( m_srv_channel_coop );
```

Т.е. получается, что агент `a_router` определяет время жизни агента `a_server`. А это означает, что агент `a_router` должен гарантировать, что агент `a_server` будет дерегистрирован раньше, чем `a_router`. Причем вне зависимости от того, когда и как инициируется дерегистрация `a_router`. Но в примере вообще нет кода по дерегистрации агента `a_server`!

Все дело в том, что используется возможность SObjectizer отслеживать взаимоотношения “родитель-потомок” между кооперациями. Вместо того, чтобы самому заниматься контролем времени регистрации агента `a_server` в SObjectizer, агент `a_router` просто объявляет SObjectizer, что кооперация с `a_server` является дочерней для кооперации с агентом `a_router`:

```
m_srv_channel_coop.set_parent_coop_name(
  so_query_coop()->query_name() );
so_4::api::register_coop( m_srv_channel_coop );
```

После чего сам SObjectizer обеспечивает автоматическую дерегистрацию кооперации с агентом `a_server`, когда по какой-то причине дерегистрируется агент `a_router`.

Аналогичным образом агент `a_router` поступает и с агентами-обработчиками, которые создаются для каждого из подключившихся клиентов:

```
// Создаем для этого канала новую кооперацию.
std::string agent_name( "a_listener_" +
  cpp_util_2::slexcast( ++m_child_counter ) );
so_4::rt::dyn_agent_coop_t * coop(
  new so_4::rt::dyn_agent_coop_t(
    new a_listener_t( agent_name ) ) );
// Указываем, что мы являемся родителями этой кооперации.
coop->set_parent_coop_name(
  so_query_coop()->query_name() );

// Регистрируем новую кооперацию.
so_4::rt::dyn_agent_coop_helper_t h( coop );
```

Отличие от ситуации с агентом `a_server` лишь в том, что при разрыве TCP/IP соединения агент `a_router` сам инициирует дерегистрацию созданного для канала агента-обработчика:

```
void
evt_client_disconnected(
  const so_4::rt::comm::msg_client_disconnected & cmd )
  {
    client_map_t::iterator it( m_clients.find( cmd.m_channel ) );
    if( it != m_clients.end() )
      {
        // Такой клиент нам известен. Нужно уничтожить прикладного
        // агента для этого клиента.
        so_4::api::deregister_coop( it->second );
        m_clients.erase( it );
      }
  }
```

Когда дерегистрируется агент `a_router`, но остаются TCP/IP соединения и созданные для них агенты-обработчики, то агенты-обработчики дерегистрируются и уничтожаются SObjectizer до того, как будет уничтожен агент `a_router`.

Второй важной особенностью примера parent_insend является то, что агент `a_router` обрабатывает сообщения агента `a_server` с помощью insend-событий.

Внешне обработчики событий `evt_channel_success`, `evt_channel_fail`, `evt_client_connected`, `evt_client_disconnected`, `evt_channel_data` выглядят обычными обработчиками событий. Более того, таковыми они и являются. Становятся эти обработчики insend-событиями благодаря подписке (в этом примере используется подписка через hook-и подписки, но на этом внимание будет заострено чуть позже):

```
so_4::rt::def_subscr_hook( m_srv_channel_coop,
  *this, event, m_srv_channel, msg, 0, &std::cerr,
  so_4::rt::evt_subscr_t::insend_dispatching );
```

Все дело в константе *so_4::rt::evt_subscr_t::insend_dispatching*. Как раз она и сообщает, что подписываемое событие является insend-событием. Если бы вместо этой константы использовалась константа *so_4::rt::evt_subscr_t::normal_dispatching*,

либо этот аргумент для функции *def_subscr_hook* был опущен, то событие было бы normal-событием.

Также следует обратить внимание на то, что агент a_router не использует никаких механизмов по собственной защите от многопоточности внутри insend-событий. В данном примере такая защита была бы избыточной: insend-события агента a_router запускаются только на нити агента a_server.

Третьей особенностью примера parent_insend является использование hook-ов подписки[1] для того, чтобы подписать события агента a_router на сообщения a_server. На самом деле, это единственная возможность корректной подписки для агента a_router: т.к. агенты a_router и a_server входят в разные кооперации, то агент a_router не может выполнить подписку в своем методе *so_on_subscription*, поскольку агент a_server еще не зарегистрирован. Также нельзя выполнять подписку после возврата из *register_coop*, т.к. можно потерять часть сообщений — будучи активным агентом a_server может начать работать еще до завершения *register_coop*. Поэтому остается только один вариант: hook-и подписки, при которых подписка выполняется непосредственно внутри вызова *register_coop*.

Если бы агенты a_router и a_server входили в состав одной кооперации, то агент a_router для подписки своих insend-событий должен был бы использовать унаследованный метод:

```
void
so_subscribe_insend_event(
  //! Имя подписываемого события.
  const std::string & event,
  //! Имя агента-владельца инцидента.
  const std::string & owner,
  //! Имя сообщения-инцидента.
  const std::string & msg,
  //! Приоритет события.
  int priority = 0,
  //! Поток для отображения сообщений об ошибках
  //! подписки. Если равен 0, то сообщения об
  //! ошибках игнорируются.
  std::ostream * err = &std::cerr );
```

а метод *so_on_subscription* мог бы выглядеть так:

```
so_subscribe_insend_event( "evt_srv_channel_success",
  server_agent_name, "msg_success" );
so_subscribe_insend_event( "evt_srv_channel_fail",
  server_agent_name, "msg_fail" );
so_subscribe_insend_event( "evt_client_connected",
  server_agent_name, "msg_client_connected" );
so_subscribe_insend_event( "evt_client_disconnected",
  server_agent_name, "msg_client_disconnected" );
so_subscribe_insend_event( "evt_channel_data",
  server_agent_name, "msg_raw_package" );
```

[1] Подробнее см. 14 на стр. 123

18.3 Результат работы примера

Если запустить пример, указав в командной строке:

```
sample_parent_insend :80
```

после чего:

- дать команду на создание `a_router`;
- попробовать из нескольких web-браузеров обратиться по URL http://localhost (для браузера пример будет выступать в роли web-сервера по этому URL);
- дать команду на уничтожение `a_router`;

то можно получить следующий результат:

```
a_router created
server channel created
client connected: {a_server {server_channel_client 1}}
a_listener_1 created
agent: a_listener_1, channel: {a_server {server_channel_client 1}},
  received: 388 byte(s)
client connected: {a_server {server_channel_client 2}}
a_listener_2 created
agent: a_listener_2, channel: {a_server {server_channel_client 2}},
  received: 388 byte(s)
a_listener_1 destroyed
a_listener_2 destroyed
a_router destroyed
```

Печать:

```
a_router created
server channel created
```

появляется, когда стартуют агент `a_router` и агент серверного сокета.

Затем, при подключении очередного браузера, возникают печати:

```
client connected: {a_server {server_channel_client 1}}
a_listener_1 created
agent: a_listener_1, channel: {a_server {server_channel_client 1}},
  received: 388 byte(s)
```

Наконец, когда дается команда на уничтожение `a_router`, осуществляется печать:

```
a_listener_1 destroyed
a_listener_2 destroyed
a_router destroyed
```

18.4 Резюме

- Перед регистрацией кооперации можно назначить имя родительской кооперации. Осуществляется это посредством метода:

```
void
set_parent_coop_name(
  const std::string & coop_name );
```

  класса *so_4::rt::agent_coop_t*.

- На момент регистрации дочерней кооперации родительская кооперация уже должна быть зарегистрирована.

- SObjectizer гарантирует, что все дочерние кооперации будут дерегистрированы до своих родительских кооперацией. Вне зависимости от того, как и когда дерегистрируется родительская кооперация.

- Один и тот же обработчик события может использоваться как для normal-, так и для insend-событий. Тип события, в роли которого будет использоваться обработчик, определяется в момент подписки события.

- Для подписки insend-события необходимо использовать либо унаследованные из *so_4::rt::agent_t* методы *so_subscribe_insend_event*, либо указывать константу *so_4::rt::evt_subscr_t::insend_dispatching* в качестве аргумента *dispatching* при обращении к функциям *so_4::rt::def_subscr_hook* и *so_4::api::subscribe_event*.

- Подписать insend-событие с помощью макросов *SOL4_SUBSCR_EVENT_START()*, *SOL4_SUBSCR_EVENT_FINISH()* нельзя.

Глава 19

Пример dyn_coop_controlled

Пример dyn_coop_contolled демонстрирует, как можно контролировать время жизни любых динамически созданных объектов при помощи динамической кооперации.

Пример dyn_coop_controlled состоит из одного файла `main.cpp` (стр. 289).

19.1 Зачем динамической кооперации что-либо контролировать

SObjectizer предоставляет возможность сохранить в динамической кооперации указатель на любой динамически созданный объект, который будет уничтожен при уничтожении кооперации после того, как будут уничтожены все входящие в кооперацию объекты. Зачем это нужно?

Пусть несколько агентов, входящих в кооперацию, нуждаются в одном объекте *db_connection*, через который происходит работа с базой данных. Причем агенты обращаются к объекту *db_connection* в деструкторе, например, для освобождения выделенных в БД ресурсов.

В случае статических агентов SObjectizer никак не контролирует время жизни агентов. Поэтому SObjectizer не должен заботиться о том, когда будут вызваны деструкторы агентов и будет ли в этот момент объект *db_connection* существовать. За это несет ответственность прикладной программист.

Но в случае динамических агентов только SObjectizer знает, когда он вызовет деструкторы агентов. И здесь сложно требовать от прикладного программиста обеспечения нужного времени жизни объекта *db_connection*. Особенно, если *db_connection* является динамически созданным объектом, который должен быть уничтожен сразу после уничтожения всех использовавших его агентов.

В общем случае такая задача решается с помощью “умного указателя” на объект *db_connection*, который подсчитывает количество ссылок, например, класса *shared_ptr* из Boost (`http://www.boost.org`) или класса *SmartPtr* из библиотеки Loki (Alexandrescu, Andrei. "Modern C++ Design: Generic Programming and Design Patterns Applied". Copyright (c) 2001. Addison-Wesley). Но есть несколько причин, по которым использование “умных указателей” может быть невозможным:

- ни Boost, ни Loki не являются стандартными библиотеками языка C++ на данный момент. Стандарт C++ сейчас вообще не определяет никаких “умных указателей”, кроме *auto_ptr*. А т.к. Boost и Loki не являются стандартными классами C++,

то могут быть административные препятствия к их использованию в конкретном проекте. Например, нежелание привязывать проект к такой большой библиотеке, как Boost, только из-за одного класса *shared_ptr*;

- для использования умных указателей может потребоваться серьезная переработка уже существующего кода, для которой может не быть ни времени, ни ресурсов. Например, в случае, когда агенты сначала были спроектированы и реализованы как статические, а затем их стали использовать в качестве динамических. В реальном проекте может просто не быть времени на модификацию и повторное тестирование кода уже работающих агентов;

- использование "умных указателей" связывает код агентов с конкретным проектным решением: объект *db_connection* уничтожается, как только будет уничтожен последний из использующих его агентов. Получается, что агенты, которые не знают, как и когда создается объект *db_connection*, будут тесно связаны с моментом уничтожения этого объекта. Такая связь может породить проблемы, если в дальнейшем окажется, что время жизни объекта *db_connection* и момент его уничтожения будут определяться по другому.

Именно для случаев, когда использование каких-либо умных указателей невозможно или нежелательно, SObjectizer предоставляет возможность сделать ответственной за уничтожение динамического объекта динамическую кооперацию. В примере с *db_connection* использование этой возможности могло бы выглядеть, например, так:

1. Создается и инициализируется объект *db_connection*.
2. Создаются объекты динамических агентов, которым в конструкторе указывается ссылка на *db_connection*.
3. Создается объект динамической кооперации.
4. Динамической кооперации назначается указатель на *db_connection*.
5. Динамическая кооперация регистрируется.

19.2 Что делает пример

В примере создаются три агента: `a_1`, `a_2` типа `a_my_t` и агент `a_shutdowner` типа `a_shutdowner_t`.

Агентам `a_1`, `a_2` назначаются ссылки на два объекта типа *my_obj_t*. В своем обработчике `msg_start` агенты `a_1`, `a_2` печатают имена назначенных им объектов типа *my_obj_t*.

Агент `a_shutdowner` инициирует завершение работы примера в своем обработчике сообщения `msg_start`.

Объекты типа *my_obj_t* создаются в примере динамически, а время их жизни определяется динамической кооперацией. Отладочные печати в деструкторах объектов типа *my_obj_t* и агентов типа `a_my_t` показывают, что агенты уничтожаются до того, как будут уничтожены указанные им объекты типа *my_obj_t*.

19.3 Разбор файла main.cpp

В файле `main.cpp` в пояснении нуждается только фрагмент функции *main()*, в котором динамически создаются объекты типа *my_obj_t*, и происходит связывание этих объектов с агентами и динамической кооперацией:

```
my_obj_t * a = new my_obj_t( "a" );
my_obj_t * b = new my_obj_t( "b" );
my_obj_t * c = new my_obj_t( "c" );

a_my_t * a_1 = new a_my_t( "a_1", *a, *b );
a_my_t * a_2 = new a_my_t( "a_2", *b, *c );
a_shutdowner_t * a_shutdowner = new a_shutdowner_t();

so_4::rt::agent_t * agents[] = { a_1, a_2, a_shutdowner };
so_4::rt::dyn_agent_coop_t * coop = new so_4::rt::dyn_agent_coop_t(
  "sample", agents, sizeof( agents ) / sizeof( agents[ 0 ] ) );
// Заставляем динамическую кооперацию контролировать
// вспомогательные объекты.
so_4::rt::dyn_coop_controlled( *coop, a );
so_4::rt::dyn_coop_controlled( *coop, b );
so_4::rt::dyn_coop_controlled( *coop, c );
```

Все объекты и агенты создаются как динамические объекты, но агентам `a_1`, `a_2` передаются ссылки на объекты типа *my_obj_t*. Поэтому агенты `a_1`, `a_2` даже не знают, созданы ли используемые ими объекты динамически, или они существуют статически.

Связывание объектов и кооперации происходит при помощи функции:

```
template< class T >
inline void
dyn_coop_controlled(
  //! Кооперация, которая должна контролировать объект.
  so_4::rt::dyn_agent_coop_t & to,
  //! Динамически созданный контролируемый объект.
  T * what )
```

определенной в пространстве имен *so_4::rt*.

Самое важное, о чем следует помнить при использовании *dyn_coop_controlled*, это то, что на момент вызова *dyn_coop_controlled* объект-кооперация уже должна быть создана. Т.е. уже должны быть созданы все входящие в кооперацию агенты. Т.е. уже должны быть созданы все объекты, время которых должно контролироваться кооперацией. Это особенно важно, если между созданием контролируемого кооперацией объекта и его связыванием с кооперацией могут быть порождены исключения. В этом случае следует позаботиться о корректном уничтожении динамических агентов.

Например, допустим, что в приведенном выше фрагменте создание объектов типа *a_my_t* может привести к порождению исключения, и нужно корректно уничтожить все динамически созданные объекты при возникновении исключения. Для такой защиты можно воспользоваться стандартным C++ классом *auto_ptr*:

```
std::auto_ptr< my_obj_t > a( new my_obj_t( "a" ) );
std::auto_ptr< my_obj_t > b( new my_obj_t( "b" ) );
std::auto_ptr< my_obj_t > c( new my_obj_t( "c" ) );

std::auto_ptr< a_my_t > a_1( new a_my_t( "a_1", *a, *b ) );
std::auto_ptr< a_my_t > a_2( new a_my_t( "a_2", *b, *c ) );
std::auto_ptr< a_shutdowner_t > a_shutdowner( new a_shutdowner_t() );

so_4::rt::agent_t * agents[] =
  {
    a_1.release()
  , a_2.release()
  , a_shutdowner.release()
  };
so_4::rt::dyn_agent_coop_t * coop = new so_4::rt::dyn_agent_coop_t(
  "sample", agents, sizeof( agents ) / sizeof( agents[ 0 ] ) );

// Заставляем динамическую кооперацию контролировать
// вспомогательные объекты.
so_4::rt::dyn_coop_controlled( *coop, a.release() );
so_4::rt::dyn_coop_controlled( *coop, b.release() );
so_4::rt::dyn_coop_controlled( *coop, c.release() );
```

19.4 Результат работы примера

В результате работы примера на стандартный поток вывода отображается:

```
a_1:
  a
  b
a_2:
  b
  c
~a_my_t: a_1
~a_my_t: a_2
~my_obj_t: a
~my_obj_t: b
~my_obj_t: c
```

По этим печатям видно, что сначала уничтожаются входящие в кооперацию агенты `a_1`, `a_2`, а затем — контролируемые кооперацией объекты типа *my_obj_t*.

19.5 Резюме

- SObjectizer позволяет сохранить в динамической кооперации указатель на любой динамически-созданный объект, который будет уничтожен при уничтожении кооперации после того, как будут уничтожены все входящие в кооперацию объекты.

- Связывание объектов и кооперации происходит при помощи функции:

```
template< class T >
inline void
dyn_coop_controlled(
  //! Кооперация, которая должна контролировать объект.
  so_4::rt::dyn_agent_coop_t & to,
  //! Динамически созданный контролируемый объект.
  T * what )
```

  определенной в пространстве имен *so_4::rt*.

- На момент вызова *dyn_coop_controlled* объект-кооперация уже должна быть создана, т.е. уже должны быть созданы все входящие в кооперацию агенты и все объекты, время которых должно контролироваться кооперацией.

Глава 20

Пример destroyable_traits

Пример destroyable_traits демонстрирует создание собственных свойств (traits) агентов и назначение агенту этих свойств в виде динамически созданных объектов.

Пример destroyable_traits состоит из одного файла `main.cpp` (стр. 293).

20.1 Что такое свойства агента

Свойство (trait) агента — это объект, реализующий интерфейс:

```
class agent_traits_t
  {
  public :
    virtual ~agent_traits_t();

    /*! Этот метод будет вызван до того, как
         для агента будет осуществлена подписка. */
    virtual void
    init( agent_t & agent ) = 0;

    /*! Этот метод будет вызван сразу после
         вызова у агента метода so_on_deregistration. */
    virtual void
    deinit( agent_t & agent ) = 0;
  };
```

определенный в пространстве имен *so_4::rt*.

Свойства агентов предназначены для реализации любых особенностей, которые не были заложены изначально в возможности класса *so_4::rt::agent_t*. Например, класс *so_4::rt::agent_t* никак не описывает тип диспетчеризации, который должен использоваться для событий агента. Вместо этого используются свойства агента, реализуемые конкретным диспетчером. Так, чтобы сделать агента активным объектом, необходимо назначить ему соответствующее свойство:

```
a_my_agent_t * agent = new a_my_agent_t( ... );
agent->so_add_traits(
  so_4::disp::active_obj::query_active_obj_traits() );
```

Идея использования свойств состоит в том, что агент владеет списком назначенных ему свойств. При регистрации кооперации перед тем, как у агентов кооперации будет вызван метод *so_on_subscription*, у всех свойств всех агентов вызывается метод *init*. Свойство должно использовать этот метод для обработки факта успешной регистрации агента в SObjectizer. При дерегистрации кооперации непосредственно после вызова у очередного агента кооперации метода *so_on_deregistration* у всех свойств этого агента вызывается метод *deinit*. Свойство должно использовать этот метод для обработки успешной дерегистрации в SObjectizer. Например, свойство, делающее агента активным объектом, в своем методе *init* сообщает диспетчеру активных объектов о том, что появился еще один агент. Поскольку метод *init* вызывается еще до того, как агенту будет отослано хотя бы одно сообщение, диспетчер получает возможность включить этого агента в свои списки и создать для агента рабочую нить. В методе *deinit* это свойство сообщает диспетчеру, что агент дерегистрирован, и диспетчер завершает рабочую нить агента.

Свойства агентов успешно используются в случаях, когда требуется выполнить какие-то действия при регистрации и дерегистрации агента, но нет возможности включения этих действий в код методов *so_on_deregistration* и *so_on_deregistration*. Например, в одном из проектов свойства используются для создания специальных мониторинговых объектов, которые автоматически следят за состоянием агента и рассылают сообщения при изменении состояния агента. В другом случае на базе свойств агентов реализованы специальные “почтальоны”, которые преобразуют объекты из специализированной подсистемы синхронной доставки уведомлений в “нормальные” асинхронные сообщения агента SObjectizer. Такие почтальоны используют методы *init*, *deinit*, чтобы зарегистрироваться и дерегистрироваться в подсистеме доставки уведомлений, а также почтальоны сами подписывают своего агента на доставляемые ими сообщения. В SObjectizer на базе свойств агентов организован выбор схемы диспетчеризации для агентов: чтобы привязать агента к конкретному диспетчеру необходимо назначить агенту свойство, которое реализует данный диспетчер.

При работе со свойствами агентов необходимо учитывать следующие особенности:

- т.к. метод *init* свойства вызывается перед методом *so_on_subscription* агента, то назначать свойства агенту имеет смысл только до регистрации агента. Если назначить агенту свойство после регистрации, то метод *init* у свойства вызван не будет;
- SObjectizer не определяет порядка, в котором у свойств будут вызываться методы *init*, *deinit*;
- однажды назначенное агенту свойство изъять у агента нельзя.

Для SObjectizer свойства агента делятся на статические и динамические. За уничтожение статических свойств отвечает прикладной программист. При этом прикладной программист должен гарантировать, что ссылка на объект-свойство остается корректной в течение всего времени жизни объекта. Статические свойства назначаются агенту посредством метода:

```
void
so_add_traits(
  /*! Объект-свойство агента. Указатель на этот объект
```

```
    сохраняется в объекте-агенте в течение всей жизни
    объекта-агента. */
  agent_traits_t & traits );
```

Поскольку статические свойства не уничтожаются агентами, то можно использовать один объект-свойство для нескольких агентов. Так, например, происходит со свойством, возвращаемым функцией *so_4::disp::active_obj::query_active_obj_traits*.

Динамические свойства — это динамически созданные объекты, реализующие интерфейс *so_4::rt::agent_traits_t*, за уничтожение которых отвечает сам агент. Динамические свойства назначаются агенту при помощи метода:

```
void
so_add_destroyable_traits(
  /*! Объект-свойство агента. Указатель на этот объект
    сохраняется в объекте-агенте в течение всей жизни
    объекта-агента.

    Должен быть указателем на динамически созданный объект. */
  agent_traits_t * traits );
```

Динамические свойства уничтожаются в деструкторе *so_4::rt::agent_t*.

20.2 Что делает пример

В примере создаются три агента: `a_1`, `a_2` типа `a_my_t` и агент `a_shutdowner` типа `a_shutdowner_t`.

Агентам `a_1`, `a_2` назначаются динамически созданные свойства типа *my_traits_t*. В своем обработчике `msg_start` агенты `a_1`, `a_2` печатают свои имена.

Агент `a_shutdowner` инициирует завершение работы примера в своем обработчике сообщения `msg_start`.

Отладочные печати в объектах типа *my_traits_t* и агентах типа `a_my_t` показывают, в какой момент выполняется вызов методов *init* и *deinit*, в какой момент у агентов вызываются методы *so_on_subscription* и *so_on_deregistration*, в какой момент происходит уничтожение агентов и их свойств.

20.3 Разбор файла main.cpp

В примере destroyable_traits используется своя реализация интерфейса *so_4::rt::agent_traits_t*:

```
class my_traits_t
  : public so_4::rt::agent_traits_t
  {
  private :
    // Имя агента, которому принадлежит свойство.
    std::string m_agent;

  public :
```

```
    my_traits_t( const std::string & agent )
      : m_agent( agent )
      {}
    ~my_traits_t()
      {
        // Печать в деструкторе покажет, в какое время будет
        // уничтожен объект-свойство.
        std::cout << "~my_traits_t: " << m_agent << std::endl;
      }

    // Реализация основных методов свойства.
    virtual void
    init( so_4::rt::agent_t & agent )
      {
        std::cout << "my_traits_t::init(): " << m_agent << std::endl;
      }

    virtual void
    deinit( so_4::rt::agent_t & agent )
      {
        std::cout << "my_traits_t::deinit(): " << m_agent << std::endl;
      }
  };
```

Свойству *my_traits_t* передается имя агента — для того, чтобы по отладочным печатям можно было определить, какому агенту принадлежит свойство.

Реализация свойства *my_traits_t* тривиальна, но она показывает принцип создания собственных свойств: необходимо всего лишь унаследоваться от *so_4::rt::agent_traits_t* и переопределить два виртуальных метода.

Главные агенты примера, принадлежащие типу `a_my_t`, назначают себе свойства *my_traits_t* в своем конструкторе:

```
  class a_my_t
    : public so_4::rt::agent_t
    {
      typedef so_4::rt::agent_t base_type_t;
    public :
      a_my_t(
        const std::string & self_name )
        : base_type_t( self_name )
        {
          // После создания свойство живет своей жизнью, никто
          // про него не знает и не может получить указатель на
          // свойство. Но свойство будет уничтожено вместе с агентом.
          so_add_destroyable_traits( new my_traits_t( self_name ) );
        }
    ...
```

Унаследованные из класса *so_4::rt::agent_t* методы являются public-методами. Поэтому свойства агенту можно назначать где угодно и когда угодно. В данном примере свойство удобно назначить непосредственно в конструкторе заинтересованного в свойстве агента. Но, скажем, в примерах filter (см. 15 на стр. 130), high_traffic (см. 16 на стр. 158), raw_channel (см. 17 на стр. 166) показано, как свойства устанавливаются для агентов, которые ничего про это не подозревают.

Реализация функции *main* в примере destroyable_traits также тривиальна: все участвующие в примере агенты создаются и регистрируются в виде одной динамической кооперации:

```
// Создаем кооперацию из 3 агентов.
a_my_t * a_1 = new a_my_t( "a_1" );
a_my_t * a_2 = new a_my_t( "a_2" );
a_shutdowner_t * a_shutdowner = new a_shutdowner_t();

so_4::rt::agent_t * agents[] = { a_1, a_2, a_shutdowner };
so_4::rt::dyn_agent_coop_t * coop = new so_4::rt::dyn_agent_coop_t(
  "sample", agents, sizeof( agents ) / sizeof( agents[ 0 ] ) );

so_4::ret_code_t rc = so_4::api::start(
  // Диспетчер будет уничтожен при выходе из start().
  so_4::disp::one_thread::create_disp(
    // Таймер будет уничтожен диспетчером.
    so_4::timer_thread::simple::create_timer_thread(),
    so_4::auto_destroy_timer ),
  so_4::auto_destroy_disp,
  coop );
if( rc )
  {
    std::cerr << "start: " << rc << std::endl;
  }
else
  std::cout << "successful finish" << std::endl;
```

Самую важную роль здесь играет печать:

```
std::cout << "successful finish" << std::endl;
```

В примере используется динамическая кооперация, поэтому сама кооперация и все входящие в нее агенты должны быть уничтожены при завершении работы SObjectizer Run-Time. Т.е. все отладочные печати в деструкторах агентов и их свойств должны быть выполнены до того, как будет осуществлена печать “successful finish”.

20.4 Результат работы примера

В результате работы примера на стандартный поток вывода отображается:

```
my_traits_t::init(): a_1
```

```
my_traits_t::init(): a_2
a_my_t::so_on_subscription(): a_1
a_my_t::so_on_subscription(): a_2
a_1
a_2
a_my_t::so_on_deregistration(): a_1
my_traits_t::deinit(): a_1
a_my_t::so_on_deregistration(): a_2
my_traits_t::deinit(): a_2
~a_my_t: a_1
~my_traits_t: a_1
~a_my_t: a_2
~my_traits_t: a_2
successful finish
```

По отладочным печатям видно, что метод *init* вызывается у свойств до того, как у их агентов будет вызван метод *so_on_subscription*. Также видно, что метод *deinit* вызывается у свойства после вызова *so_on_deregistration* у соответствующего агента.

Печать, которая осуществляется в деструкторах агентов и свойств, показывает, что агенты действительно были уничтожены при завершении работы SObjectizer Run-Time перед возвратом из *so_4::api::start*. Деструкторы свойств отрабатывают позже деструкторов владеющих ими агентов, потому что печать "~a_my_t:..." осуществляется в деструкторе производного от *so_4::rt::agent_t* класса, а объекты-свойства уничтожаются в деструкторе базового класса *so_4::rt::agent_t*.

20.5 Резюме

- Свойство агента — это объект, реализующий интерфейс *so_4::rt::agent_traits_t*.
- Свойства агента делятся на статические (время жизни объекта-свойства контролируется прикладным программистом) и динамические (уничтожаются в деструкторе *so_4::rt::agent_t*).
- Статические свойства назначаются агенту посредством унаследованного из *so_4::rt::agent_t* метода:

```
void
so_add_traits(
  agent_traits_t & traits );
```

- Динамические свойства назначаются агенту при помощи унаследованного из *so_4::rt::agent_t* метода:

```
void
so_add_destroyable_traits(
  agent_traits_t * traits );
```

- При регистрации агента перед вызовом *so_on_subscription* у всех его свойств вызывается метод *init*. SObjectizer не определяет порядка, в котором будет вызываться метод *init*.
- При дерегистрации агента после вызова *so_on_deregistration* у всех его свойств вызывается метод *deinit*. SObjectizer не определяет порядка, в котором будет вызываться метод *deinit*.
- Поскольку метод *init* свойства вызывается перед методом *so_on_subscription* агента, то назначать свойства агенту имеет смысл только до регистрации агента. Если назначить агенту свойство после регистрации, то метод *init* у свойства вызван не будет.
- Однажды назначенное агенту свойство изъять у агента нельзя.
- Методы *so_add_traits*, *so_add_destroyable_traits* являются public-методами, поэтому свойства можно назначать любому агенту.
- Для создания собственного свойства необходимо унаследоваться от *so_4::rt::agent_traits_t* и переопределить виртуальные методы *init*, *deinit*.

Приложение А

Исходные тексты примеров

А.1 Исходный код примера hello_world

А.1.1 Файл main.cpp

```
/*
  Пример простейшего агента.
*/

#include <iostream>

// Загружаем основные заголовочные файлы SObjectizer.
#include <so_4/rt/h/rt.hpp>
#include <so_4/api/h/api.hpp>

// Загружаем описание нити таймера и диспетчера.
#include <so_4/timer_thread/simple/h/pub.hpp>
#include <so_4/disp/one_thread/h/pub.hpp>

// Номер версии SObjectizer-a.
// Нужен только для отображения в строке приветствия.
#include <so_4/h/version.hpp>

// C++ описание класса агента.
class a_hello_t
  : public so_4::rt::agent_t
  {
    // Псевдоним для базового типа.
    typedef so_4::rt::agent_t base_type_t;
  public :
    a_hello_t()
      :
      // Сразу задаем имя агента.
      base_type_t( "a_hello" )
      {}
    virtual ~a_hello_t()
```

```
      {}

    virtual const char *
    so_query_type() const;

    virtual void
    so_on_subscription()
      {
        // Нужно подписать наше единственное событие.
        so_subscribe( "evt_start",
         so_4::rt::sobjectizer_agent_name(),
         "msg_start" );
      }

    // Обработка начала работы агента в системе.
    void
    evt_start()
      {
        std::cout << "Hello, world! This is SObjectizer v.4."
         << std::hex << __SO_4_VERSION__ << std::dec << std::endl;

        // Завершаем работу примера.
        so_4::api::send_msg(
         so_4::rt::sobjectizer_agent_name(),
         "msg_normal_shutdown", 0 );
      }
  };

// Описание класса агента для SObjectizer-а.
SOL4_CLASS_START( a_hello_t )

  // Одно событие.
  SOL4_EVENT( evt_start )

  // И одно состояние.
  SOL4_STATE_START( st_normal )
    // С одним событием.
    SOL4_STATE_EVENT( evt_start )
  SOL4_STATE_FINISH()

SOL4_CLASS_FINISH()

int
main()
  {
    // Наш агент.
    a_hello_t a_hello;
    // И кооперация для него.
```

```
    so_4::rt::agent_coop_t a_hello_coop( a_hello );

    // Запускаем SObjectizer Run-Time.
    so_4::ret_code_t rc = so_4::api::start(
      so_4::disp::one_thread::create_disp(
        so_4::timer_thread::simple::create_timer_thread(),
        so_4::auto_destroy_timer ),
      so_4::auto_destroy_disp,
      &a_hello_coop );
    if( rc )
      {
        // Запустить SObjectizer Run-Time не удалось.
        std::cerr << "start: " << rc << std::endl;
      }

    return int( rc );
  }
```

A.2 Исходный код примера hello_all

A.2.1 Файл main.cpp

```
/*
  Демострация целенаправленной и широковещательной рассылки сообщений.
*/

#include <iostream>

#include <so_4/rt/h/rt.hpp>
#include <so_4/api/h/api.hpp>

#include <so_4/timer_thread/simple/h/pub.hpp>
#include <so_4/disp/one_thread/h/pub.hpp>

//
// a_msg_owner_t
//

// Класс агента, единственной задачей которого является
// владение сообщениями.
class a_msg_owner_t
  :  public so_4::rt::agent_t
  {
    typedef so_4::rt::agent_t base_type_t;

  public :
    a_msg_owner_t()
      : base_type_t( agent_name() )
```

```
      {}

    virtual ~a_msg_owner_t()
      {}

    virtual const char *
    so_query_type() const;

    virtual void
    so_on_subscription()
      {}

    static std::string
    agent_name()
      {
        return "a_msg_owner";
      }

    // Сообщение, которым будут пользоваться агенты примера.
    struct  msg_hello
      {
        std::string m_sender;

        msg_hello() {}
        msg_hello(
          const std::string & sender )
          : m_sender( sender )
          {}

        static bool
        check( const msg_hello * msg )
          {
            return ( 0 != msg &&
            msg->m_sender.length() );
          }
      };
  };

SOL4_CLASS_START( a_msg_owner_t )

  SOL4_MSG_START( msg_hello_to_all, a_msg_owner_t::msg_hello )
    SOL4_MSG_FIELD( m_sender )

    SOL4_MSG_CHECKER( a_msg_owner_t::msg_hello::check )
  SOL4_MSG_FINISH()

  SOL4_MSG_START( msg_hello_to_you, a_msg_owner_t::msg_hello )
    SOL4_MSG_FIELD( m_sender )
```

```
    SOL4_MSG_CHECKER( a_msg_owner_t::msg_hello::check )
  SOL4_MSG_FINISH()

SOL4_CLASS_FINISH()

//
// a_msg_receiver_t
//

// Класс агента, который получает сообщения.
class a_msg_receiver_t
  :  public so_4::rt::agent_t
  {
    typedef so_4::rt::agent_t base_type_t;

  public :
    a_msg_receiver_t(
      const std::string & agent_name );
    virtual ~a_msg_receiver_t();

    virtual const char *
    so_query_type() const;

    virtual void
    so_on_subscription();

    // Отсылаем всем приветствие от своего имени.
    void
    evt_start();

    void
    evt_hello_to_all(
      const a_msg_owner_t::msg_hello & cmd );

    void
    evt_hello_to_you(
      const a_msg_owner_t::msg_hello & cmd );
  };

SOL4_CLASS_START( a_msg_receiver_t )

  SOL4_EVENT( evt_start )
  SOL4_EVENT_STC(
    evt_hello_to_all,
    a_msg_owner_t::msg_hello )
  SOL4_EVENT_STC(
    evt_hello_to_you,
```

```
    a_msg_owner_t::msg_hello )

  SOL4_STATE_START( st_initial )
    SOL4_STATE_EVENT( evt_start )
    SOL4_STATE_EVENT( evt_hello_to_all )
    SOL4_STATE_EVENT( evt_hello_to_you )
  SOL4_STATE_FINISH()

SOL4_CLASS_FINISH()

//
// a_msg_receiver_t
//
a_msg_receiver_t::a_msg_receiver_t(
  const std::string & agent_name )
  :
  base_type_t( agent_name )
  {}

a_msg_receiver_t::~a_msg_receiver_t()
  {}

void
a_msg_receiver_t::so_on_subscription()
  {
    so_subscribe( "evt_start",
      so_4::rt::sobjectizer_agent_name(),
      "msg_start", 1 );

    so_subscribe( "evt_hello_to_all",
      a_msg_owner_t::agent_name(),
      "msg_hello_to_all", 1 );

    so_subscribe( "evt_hello_to_you",
      a_msg_owner_t::agent_name(),
      "msg_hello_to_you", 1 );
  }

void
a_msg_receiver_t::evt_start()
  {
    std::cout << so_query_name() << ".evt_start" << std::endl;

    so_4::api::send_msg_safely(
      a_msg_owner_t::agent_name(), "msg_hello_to_all",
      new a_msg_owner_t::msg_hello( so_query_name() ) );
  }
```

```
void
a_msg_receiver_t::evt_hello_to_all(
  const a_msg_owner_t::msg_hello & cmd )
  {
    std::cout << so_query_name() << ".evt_hello_to_all: "
      << cmd.m_sender << std::endl;

    // Если привествие слали не мы, то отошлем ответ.
    if( so_query_name() != cmd.m_sender )
      {
        so_4::api::send_msg_safely(
          a_msg_owner_t::agent_name(),
          "msg_hello_to_you",
          new a_msg_owner_t::msg_hello( so_query_name() ),
          cmd.m_sender );
      }
  }

void
a_msg_receiver_t::evt_hello_to_you(
  const a_msg_owner_t::msg_hello & cmd )
  {
    std::cout << so_query_name() << ".evt_hello_to_you: "
      << cmd.m_sender << std::endl;
  }

//
// a_shutdowner_t
//

// Агент для завершения работы примера.
class a_shutdowner_t
  :  public so_4::rt::agent_t
  {
    typedef so_4::rt::agent_t base_type_t;

  public :
    a_shutdowner_t();
    virtual ~a_shutdowner_t();

    virtual const char *
    so_query_type() const;

    virtual void
    so_on_subscription();

    static std::string
    agent_name();
```

```
    void
    evt_shutdown();
  };

SOL4_CLASS_START( a_shutdowner_t )

  SOL4_EVENT( evt_shutdown )

  SOL4_STATE_START( st_initial )
    SOL4_STATE_EVENT( evt_shutdown )
  SOL4_STATE_FINISH()

SOL4_CLASS_FINISH()

//
// a_shutdowner_t
//
a_shutdowner_t::a_shutdowner_t()
  :  base_type_t( agent_name() )
  {}

a_shutdowner_t::~a_shutdowner_t()
  {}

void
a_shutdowner_t::so_on_subscription()
  {
    so_subscribe( "evt_shutdown",
      so_4::rt::sobjectizer_agent_name(),
      "msg_start" );
  }

std::string
a_shutdowner_t::agent_name()
  {
    return "a_shutdowner";
  }

void
a_shutdowner_t::evt_shutdown()
  {
    so_4::api::send_msg( so_4::rt::sobjectizer_agent_name(),
    "msg_normal_shutdown" );
  }


int
```

```
main()
  {
    // Агенты примера.
    a_msg_owner_t a_msg_owner;
    a_shutdowner_t  a_shutdowner;

    a_msg_receiver_t  a_receiver_one( "a_receiver_one" );
    a_msg_receiver_t  a_receiver_two( "a_receiver_two" );
    a_msg_receiver_t  a_receiver_three( "a_receiver_three" );

    // Кооперация примера.
    so_4::rt::agent_t * g_coop_agents[] =
      {
        &a_msg_owner,
        &a_shutdowner,
        &a_receiver_one,
        &a_receiver_two,
        &a_receiver_three
      };

    so_4::rt::agent_coop_t  a_coop( "a_coop", g_coop_agents,
      sizeof( g_coop_agents ) / sizeof( g_coop_agents[ 0 ] ) );

    so_4::ret_code_t rc = so_4::api::start(
      // Диспетчер будет уничтожен при выходе из start().
      so_4::disp::one_thread::create_disp(
        // Таймер будет уничтожен диспетчером.
        so_4::timer_thread::simple::create_timer_thread(),
        so_4::auto_destroy_timer ),
      so_4::auto_destroy_disp,
      &a_coop );
    if( rc )
      {
        std::cerr << "start: " << rc << std::endl;
      }

    return int( rc );
  }
```

A.3 Исходный код примера hello_world

A.3.1 Файл main.cpp

```
/*
  Пример простейшего агента, который использует собственное
  отложенное сообщение.
*/
```

```
#include <iostream>

#include <time.h>

// Загружаем основные заголовочные файлы SObjectizer.
#include <so_4/rt/h/rt.hpp>
#include <so_4/api/h/api.hpp>

// Загружаем описание нити таймера и диспетчера.
#include <so_4/timer_thread/simple/h/pub.hpp>
#include <so_4/disp/one_thread/h/pub.hpp>

// C++ описание класса агента.
class a_hello_t
  : public so_4::rt::agent_t
  {
    // Псевдоним для базового типа.
    typedef so_4::rt::agent_t base_type_t;
  public :
    a_hello_t();
    virtual ~a_hello_t();

    virtual const char *
    so_query_type() const;

    virtual void
    so_on_subscription();

    // Сообщение о необходимости выдать приветствие.
    struct msg_hello_time {};

    // Начало работы агента в системе.
    void
    evt_start();

    // Пора выдать приветствие.
    void
    evt_hello_time();
  };

// Описание класса агента для SObjectizer-а.
SOL4_CLASS_START( a_hello_t )

  // Описание сообщений.
  SOL4_MSG_START( msg_hello_time, a_hello_t::msg_hello_time )
  SOL4_MSG_FINISH()

  // Описание событий.
```

```
  SOL4_EVENT( evt_start )
  SOL4_EVENT( evt_hello_time )

  // Описание единственного состояния.
  SOL4_STATE_START( st_initial )
    SOL4_STATE_EVENT( evt_start )
    SOL4_STATE_EVENT( evt_hello_time )
  SOL4_STATE_FINISH()

SOL4_CLASS_FINISH()

a_hello_t::a_hello_t()
  :
  base_type_t( "a_hello" )
  {}

a_hello_t::~a_hello_t()
  {}

void
a_hello_t::so_on_subscription()
  {
    so_subscribe( "evt_start",
      so_4::rt::sobjectizer_agent_name(), "msg_start" );

    so_subscribe( "evt_hello_time", "msg_hello_time" );
  }

void
a_hello_t::evt_start()
  {
    // Сообщаем когда будет выдано приветствие.
    unsigned int sec = 2;
    time_t t = time( 0 );
    std::cout << so_query_name() << ": "
      << asctime( localtime( &t ) )
      << "hello after " << sec << " seconds" << std::endl;

    // Отсылаем себе отложенное сообщение.
    so_4::api::send_msg(
      so_query_name(),
      "msg_hello_time", 0, 0,
      sec * 1000 );
  }

void
a_hello_t::evt_hello_time()
  {
```

```
    time_t t = time( 0 );
    std::cout << so_query_name() << ": "
      << asctime( localtime( &t ) ) << std::flush;

    std::cout << "Hello, World!" << std::endl;

    // Отсылаем сообщение для завершения работы.
    so_4::api::send_msg(
      so_4::rt::sobjectizer_agent_name(),
      "msg_normal_shutdown", 0 );
  }

int
main()
{
  // Наш агент.
  a_hello_t a_hello;
  // И кооперация для него.
  so_4::rt::agent_coop_t a_hello_coop( a_hello );

  // Запускаем SObjectizer Run-Time.
  so_4::ret_code_t rc = so_4::api::start(
    so_4::disp::one_thread::create_disp(
      so_4::timer_thread::simple::create_timer_thread(),
      so_4::auto_destroy_timer ),
    so_4::auto_destroy_disp,
    &a_hello_coop );
  if( rc )
    {
      // Запустить SObjectizer Run-Time не удалось.
      std::cerr << "start: " << rc << std::endl;
    }

  return int( rc );
}
```

A.4 Исходный код примера hello_periodic

A.4.1 Файл main.cpp

```
/*
  Пример простейшего агента, который использует собственные
  переодические сообщения.
*/

#include <iostream>

#include <time.h>
```

```
// Загружаем основные заголовочные файлы SObjectizer.
#include <so_4/rt/h/rt.hpp>
#include <so_4/api/h/api.hpp>

// Загружаем описание нити таймера и диспетчера.
#include <so_4/timer_thread/simple/h/pub.hpp>
#include <so_4/disp/one_thread/h/pub.hpp>

// C++ описание класса агента.
class a_hello_t
  : public so_4::rt::agent_t
  {
    // Псевдоним для базового типа.
    typedef so_4::rt::agent_t base_type_t;
  public :
    a_hello_t();
    virtual ~a_hello_t();

    virtual const char *
    so_query_type() const;

    virtual void
    so_on_subscription();

    // Переодичесткое сообщение без данных.
    struct msg_hello_no_data {};

    // Переодическое сообщение, которое содержит данные.
    struct msg_hello_with_data
      {
        std::string m_msg;

        msg_hello_with_data() {}
        msg_hello_with_data(
          const std::string & msg )
          : m_msg( msg )
          {}

        // Не разрешает хождение сообщения без данных.
        static bool
        check( const msg_hello_with_data * msg )
          {
            return ( 0 != msg );
          }
    };

    // Начало работы агента в системе.
```

```
    void
    evt_start();

    // Реакция на сообщение без данных.
    void
    evt_hello_no_data();

    // Реакция на сообщение с данными.
    void
    evt_hello_with_data(
      const msg_hello_with_data & cmd );

  private :
    // Оставшеся количество циклов до завершения работы.
    unsigned int m_remaining;
  };

// Описание класса агента для SObjectizer-а.
SOL4_CLASS_START( a_hello_t )

  // Описание сообщений.
  SOL4_MSG_START( msg_hello_no_data, a_hello_t::msg_hello_no_data )
  SOL4_MSG_FINISH()

  SOL4_MSG_START( msg_hello_with_data, a_hello_t::msg_hello_with_data )
    SOL4_MSG_CHECKER( a_hello_t::msg_hello_with_data::check )
  SOL4_MSG_FINISH()

  // Описание событий.
  SOL4_EVENT( evt_start )
  SOL4_EVENT( evt_hello_no_data )
  SOL4_EVENT_STC(
    evt_hello_with_data,
    a_hello_t::msg_hello_with_data )

  // Описание единственного состояния.
  SOL4_STATE_START( st_initial )
    SOL4_STATE_EVENT( evt_start )
    SOL4_STATE_EVENT( evt_hello_no_data )
    SOL4_STATE_EVENT( evt_hello_with_data )
  SOL4_STATE_FINISH()

SOL4_CLASS_FINISH()

a_hello_t::a_hello_t()
  :  base_type_t( "a_hello" )
  ,m_remaining( 5 )
  {}
```

```
a_hello_t::~a_hello_t()
  {}

void
a_hello_t::so_on_subscription()
  {
    so_subscribe( "evt_start",
      so_4::rt::sobjectizer_agent_name(), "msg_start" );

    so_subscribe( "evt_hello_no_data", "msg_hello_no_data" );
    so_subscribe( "evt_hello_with_data", "msg_hello_with_data" );
  }

void
a_hello_t::evt_start()
  {
    unsigned int no_data_delay = 2;
    unsigned int no_data_period = 1;
    unsigned int with_data_delay = 0;
    unsigned int with_data_period = 2;

    time_t t = time( 0 );

    // Сообщаем, что и когда будет происходить.
    std::cout << so_query_name() << ".evt_start: "
      << asctime( localtime( &t ) )
      << "\thello_no_data delay: " << no_data_delay
      << "\n\thello_no_data period: " << no_data_period
      << "\n\thello_with_data delay: " << with_data_delay
      << "\n\thello_with_data period: " << with_data_period
      << std::endl;

    // Запускаем переодические сообщения.
    so_4::api::send_msg_safely(
      so_query_name(),
      "msg_hello_with_data",
      new msg_hello_with_data(
        std::string( "Sample started at " ) +
        asctime( localtime( &t ) ) ),
      "",
      with_data_delay * 1000, with_data_period * 1000 );

    so_4::api::send_msg(
      so_query_name(), "msg_hello_no_data", 0, 0,
      no_data_delay * 1000, no_data_period * 1000 );
  }
```

```
void
a_hello_t::evt_hello_no_data()
  {
    --m_remaining;

    time_t t = time( 0 );
    std::cout << so_query_name() << ".evt_hello_no_data: "
      << asctime( localtime( &t ) )
      << "\tremaining: " << m_remaining << std::endl;

    if( !m_remaining )
      // Отсылаем сообщение для завершения работы.
      so_4::api::send_msg(
        so_4::rt::sobjectizer_agent_name(),
        "msg_normal_shutdown", 0 );
  }

void
a_hello_t::evt_hello_with_data(
  const msg_hello_with_data & cmd )
  {
    time_t t = time( 0 );
    std::cout << so_query_name() << ".evt_hello_with_data: "
      << asctime( localtime( &t ) )
      << "\t" << cmd.m_msg << std::endl;
  }

int
main()
  {
    // Наш агент.
    a_hello_t a_hello;
    // И кооперация для него.
    so_4::rt::agent_coop_t a_hello_coop( a_hello );

    // Запускаем SObjectizer Run-Time.
    so_4::ret_code_t rc = so_4::api::start(
      so_4::disp::one_thread::create_disp(
        so_4::timer_thread::simple::create_timer_thread(),
        so_4::auto_destroy_timer ),
      so_4::auto_destroy_disp,
      &a_hello_coop );
    if( rc )
      {
        // Запустить SObjectizer Run-Time не удалось.
        std::cerr << "start: " << rc << std::endl;
      }
```

```
    return int( rc );
  }
```

A.5 Исходный код примера dyn_reg

A.5.1 Файл main.cpp

```
/*
  Пример использования:
    - динамической кооперации;
    - обработки сообщений SObjectizer-а о регистрации и
    дерегистрации кооперации;
    - переопределения метода so_4::rt::agent_t::so_on_deregistration().
*/

#include <iostream>

// Загружаем основные заголовочные файлы SObjectizer.
#include <so_4/rt/h/rt.hpp>
#include <so_4/api/h/api.hpp>

// Загружаем описание нити таймера и диспетчера.
#include <so_4/timer_thread/simple/h/pub.hpp>
#include <so_4/disp/one_thread/h/pub.hpp>

// Имя дочернего агента и динамической кооперации.
const std::string child_name( "a_child" );

// Класс агента, который будет входить в динамическую кооперацию.
class a_child_t
  :  public so_4::rt::agent_t
  {
    typedef so_4::rt::agent_t base_type_t;

  public :
    a_child_t();
    virtual ~a_child_t();

    virtual const char *
    so_query_type() const;

    virtual void
    so_on_subscription();

    virtual void
    so_on_deregistration();

    // Сообщение, которым обладает данный агент.
```

```
    struct msg_say_it_again {};

    void
    evt_start();

    void
    evt_say_it_again( const so_4::rt::event_data_t & data );
  };

SOL4_CLASS_START( a_child_t )

  SOL4_MSG_START( msg_say_it_again, a_child_t::msg_say_it_again )
  SOL4_MSG_FINISH()

  SOL4_EVENT( evt_start )

  SOL4_EVENT( evt_say_it_again )

  SOL4_STATE_START( st_initial )
    SOL4_STATE_EVENT( evt_start )
    SOL4_STATE_EVENT( evt_say_it_again )
  SOL4_STATE_FINISH()

SOL4_CLASS_FINISH()

a_child_t::a_child_t()
  :  base_type_t( child_name )
  {
    // Информируем о том, что мы созданы.
    std::cout << so_query_name() << " created" << std::endl;
  }

a_child_t::~a_child_t()
  {
    // Информируем о том, что мы уничтожены.
    std::cout << so_query_name() << " destroyed" << std::endl;
  }

void
a_child_t::so_on_subscription()
  {
    so_subscribe( "evt_start",
      so_4::rt::sobjectizer_agent_name(), "msg_start" );

    so_subscribe( "evt_say_it_again", "msg_say_it_again" );
  }

void
```

```
a_child_t::so_on_deregistration()
  {
    // Сообщаем о том, что нас дерегистрируют.
    std::cout << so_query_name() << " deregistered" << std::endl;
  }

void
a_child_t::evt_start()
  {
    // Сообщаем о том, что мы стартовали.
    std::cout << so_query_name() << ": Hello, World!" << std::endl;
  }

void
a_child_t::evt_say_it_again( const so_4::rt::event_data_t & data )
  {
    // Сообщаем, что мы получили сообщение.
    std::cout << so_query_name()
      << ": and again - Hello, World!" << std::endl;

    // Отошлем его самим себе с задержкой. Если нас за это
    // время дерегистрируют, то мы его не получим.
    so_4::api::send_msg( data.agent(), data.msg(), 0,
      so_query_name(), 15000 );
  }

//
// Класс агента, который создает и уничтожает динамическую кооперацию.
//
class a_parent_t
  : public so_4::rt::agent_t
  {
    typedef so_4::rt::agent_t base_type_t;
  public :
    a_parent_t();
    virtual ~a_parent_t();

    virtual const char *
    so_query_type() const;

    virtual void
    so_on_subscription();

    // При поступлении этого сообщения кооперация будет создана.
    struct msg_reg_time {};

    // При поступлении этого сообщения кооперация
    // будет дерегистрирована.
```

```
    struct msg_dereg_time {};

    void
    evt_start();

    void
    evt_reg_time();

    void
    evt_dereg_time();

    void
    evt_coop_registered(
    const so_4::rt::msg_coop_registered & cmd );

    void
    evt_coop_deregistered(
    const so_4::rt::msg_coop_deregistered & cmd );
  };

SOL4_CLASS_START( a_parent_t )

  SOL4_MSG_START( msg_reg_time, a_parent_t::msg_reg_time )
  SOL4_MSG_FINISH()

  SOL4_MSG_START( msg_dereg_time, a_parent_t::msg_dereg_time )
  SOL4_MSG_FINISH()

  SOL4_EVENT( evt_start )

  SOL4_EVENT( evt_reg_time )

  SOL4_EVENT( evt_dereg_time )

  SOL4_EVENT_STC(
    evt_coop_registered,
    so_4::rt::msg_coop_registered )

  SOL4_EVENT_STC(
    evt_coop_deregistered,
    so_4::rt::msg_coop_deregistered )

  SOL4_STATE_START( st_initial )
    SOL4_STATE_EVENT( evt_start )
    SOL4_STATE_EVENT( evt_reg_time )
    SOL4_STATE_EVENT( evt_dereg_time )
    SOL4_STATE_EVENT( evt_coop_registered )
    SOL4_STATE_EVENT( evt_coop_deregistered )
```

```
  SOL4_STATE_FINISH()

SOL4_CLASS_FINISH()

a_parent_t::a_parent_t()
  :  base_type_t( "a_parent" )
  {}

a_parent_t::~a_parent_t()
  {}

void
a_parent_t::so_on_subscription()
  {
    so_subscribe( "evt_start",
      so_4::rt::sobjectizer_agent_name(), "msg_start" );

    so_subscribe( "evt_reg_time", "msg_reg_time" );

    so_subscribe( "evt_dereg_time", "msg_dereg_time" );

    // Если приоритит у этого события больше, чем приоритет
    // evt_start у дочернего агента, то мы должны отработать
    // быстрее, чем дочерний агент (для приоритеных диспетчеров).
    so_subscribe( "evt_coop_registered",
      so_4::rt::sobjectizer_agent_name(), "msg_coop_registered", 1 );

    so_subscribe( "evt_coop_deregistered",
      so_4::rt::sobjectizer_agent_name(), "msg_coop_deregistered" );
  }

void
a_parent_t::evt_start()
  {
    // Указываем, когда дочерняя кооперация будет зарегистрирована.
    unsigned int sec = 1;
    std::cout << so_query_name()
      << ": child coop will be registered after "
      << sec << " sec" << std::endl;

    so_4::api::send_msg( so_query_name(), "msg_reg_time", 0,
      so_query_name(), sec * 1000 );
  }

void
a_parent_t::evt_reg_time()
  {
    std::cout << so_query_name()
```

```
      << ": It's time to register child coop"
      << std::endl;

    // Регистрируем дочернюю кооперацию.
    so_4::rt::dyn_agent_coop_helper_t helper(
      new so_4::rt::dyn_agent_coop_t( new a_child_t() ) );
    if( helper.result() )
      {
        // Ошибка регистрации!
        std::cerr << "register_coop: " << helper.result() << std::endl;
        so_4::api::send_msg( so_4::rt::sobjectizer_agent_name(),
          "msg_alarm_shutdown", 0 );
      }
  }

void
a_parent_t::evt_dereg_time()
  {
    std::cout << so_query_name()
      << ": It's time to deregister child coop" << std::endl;

    so_4::ret_code_t rc = so_4::api::deregister_coop( child_name );
    if( rc )
      {
        std::cerr << "deregister_coop: " << rc << std::endl;
        so_4::api::send_msg( so_4::rt::sobjectizer_agent_name(),
          "msg_alarm_shutdown", 0 );
      }
  }

void
a_parent_t::evt_coop_registered(
  const so_4::rt::msg_coop_registered & cmd )
  {
    std::cout << so_query_name()
      << ": Cooperation registered: "
      << cmd.m_coop_name << std::endl;

    if( child_name == cmd.m_coop_name )
      {
        // Это наша дочерняя кооперация, которую нужно
        // дерегистрировать.

        unsigned int sec = 2;
        std::cout << so_query_name()
          << ": child coop will be deregistered after "
          << sec << " sec" << std::endl;
```

```
        so_4::api::send_msg(
          so_query_name(), "msg_dereg_time", 0,
          so_query_name(), sec * 1000 );

        // Заставляем дочернего агента еще раз сказать Hello...
        so_4::api::send_msg( child_name, "msg_say_it_again" );
      }
  }

void
a_parent_t::evt_coop_deregistered(
  const so_4::rt::msg_coop_deregistered & cmd )
  {
    std::cout << so_query_name()
      << ": Cooperation deregistered: "
      << cmd.m_coop_name << std::endl;

    if( child_name == cmd.m_coop_name )
      {
        // Дочерняя кооперация дерегистрирована. Можно завершать работу.
        std::cout << so_query_name()
          << ": Work finished" << std::endl;

        so_4::api::send_msg(
          so_4::rt::sobjectizer_agent_name(),
          "msg_normal_shutdown", 0 );
    }
  }

int
main()
  {
    // Родительский агент.
    a_parent_t a_parent;
    // И кооперация для него.
    so_4::rt::agent_coop_t a_parent_coop( a_parent );

    // Запускаем SObjectizer Run-Time.
    so_4::ret_code_t rc = so_4::api::start(
      so_4::disp::one_thread::create_disp(
        so_4::timer_thread::simple::create_timer_thread(),
        so_4::auto_destroy_timer ),
      so_4::auto_destroy_disp,
      &a_parent_coop );
    if( rc )
      {
        // Запустить SObjectizer Run-Time не удалось.
        std::cerr << "start: " << rc << std::endl;
```

```
    }

  return int( rc );
 }
```

A.6 Исходный код примера chstate

A.6.1 Файл main.cpp

```
/*
 Пример агента, изменяющего свое состояние.

 Агент обладает состояниями: st_1, st_2, st_3, st_4, st_shutdown.
 В каждом из состояний с разным приоритетом обрабатывается
 сообщение msg_1.

 Показывается, как назначение нескольких обработчиков
 одного сообщения в одно состояние с одинаковым приоритетом
 приводит к игнорированию обоих событий.

 Показывается возможность автоматической рассылки сообщений
 об изменении состояния агента и обработка этого сообщения.

 Показывается назначение обработчиков входа и выхода
 в состояние.

 Показывается назначение "слушателей" состояния агента.
*/

#include <iostream>

// Загружаем основные заголовочные файлы SObjectizer.
#include <so_4/rt/h/rt.hpp>
#include <so_4/api/h/api.hpp>

// Загружаем описание нити таймера и диспетчера.
#include <so_4/timer_thread/simple/h/pub.hpp>
#include <so_4/disp/one_thread/h/pub.hpp>

// Слушатель состояния агента.
class sample_listener_t
  : public so_4::rt::agent_state_listener_t
  {
  private :
    // Имя данного слушателя.
    std::string m_name;
  public :
    sample_listener_t(
```

```
      const std::string & name )
      : m_name( name )
      {}
    virtual ~sample_listener_t()
      {
        // Отладочная печать покажет, когда слушатель будет уничтожен.
        std::cout << "listener destroyed: " << m_name << std::endl;
      }

    virtual void
    changed(
      so_4::rt::agent_t & agent,
      const std::string & state_name )
      {
        std::cout << m_name << ": state of '"
        << agent.so_query_name() << "' is '"
        << state_name << "'" << std::endl;
      }
  };

// C++ описание класса агента.
class a_main_t
  : public so_4::rt::agent_t
  {
  typedef so_4::rt::agent_t base_type_t;

  public :
    a_main_t();
    virtual ~a_main_t();

    virtual const char *
    so_query_type() const;

    virtual void
    so_on_subscription();

    struct msg_1 {};

    void
    evt_start();

    void
    evt_msg_1_st_1();

    void
    evt_msg_1_st_2_pri_1();

    void
```

```
    evt_msg_1_st_2_pri_0();

    void
    evt_msg_1_st_3_pri_1_one();

    void
    evt_msg_1_st_3_pri_1_two();

    void
    evt_msg_1_st_3_pri_0();

    void
    evt_msg_1_st_4();

    // Возникает при смене собственного состояния.
    void
    evt_self_state_notify(
    const so_4::rt::so_msg_state * cmd );

    void
    on_enter_state(
    const std::string & state_name );

    void
    on_exit_state(
    const std::string & state_name );

    void
    on_enter_st_shutdown(
    const std::string & state_name );
  };

SOL4_CLASS_START( a_main_t )

  SOL4_MSG_START( msg_1, a_main_t::msg_1 )
  SOL4_MSG_FINISH()

  SOL4_EVENT( evt_start )
  SOL4_EVENT( evt_msg_1_st_1 )
  SOL4_EVENT( evt_msg_1_st_2_pri_1 )
  SOL4_EVENT( evt_msg_1_st_2_pri_0 )
  SOL4_EVENT( evt_msg_1_st_3_pri_1_one )
  SOL4_EVENT( evt_msg_1_st_3_pri_1_two )
  SOL4_EVENT( evt_msg_1_st_3_pri_0 )
  SOL4_EVENT( evt_msg_1_st_4 )

  SOL4_EVENT_STC(
    evt_self_state_notify,
```

```
  so_4::rt::so_msg_state )

SOL4_STATE_START( st_1 )
  SOL4_STATE_EVENT( evt_start )
  SOL4_STATE_EVENT( evt_msg_1_st_1 )

  SOL4_STATE_EVENT( evt_self_state_notify )

  SOL4_STATE_ON_ENTER( on_enter_state )
  SOL4_STATE_ON_EXIT( on_exit_state )
SOL4_STATE_FINISH()

SOL4_STATE_START( st_2 )
  SOL4_STATE_EVENT( evt_msg_1_st_2_pri_1 )
  SOL4_STATE_EVENT( evt_msg_1_st_2_pri_0 )

  SOL4_STATE_EVENT( evt_self_state_notify )

  SOL4_STATE_ON_ENTER( on_enter_state )
  SOL4_STATE_ON_EXIT( on_exit_state )
SOL4_STATE_FINISH()

SOL4_STATE_START( st_3 )
  // Назначение этих событий в одно состояние
  // приводит к их игнорированию из-за
  // непредсказуемости системы.
  SOL4_STATE_EVENT( evt_msg_1_st_3_pri_1_one )
  SOL4_STATE_EVENT( evt_msg_1_st_3_pri_1_two )

  SOL4_STATE_EVENT( evt_msg_1_st_3_pri_0 )

  SOL4_STATE_EVENT( evt_self_state_notify )

  SOL4_STATE_ON_ENTER( on_enter_state )
  SOL4_STATE_ON_EXIT( on_exit_state )
SOL4_STATE_FINISH()

SOL4_STATE_START( st_4 )
  SOL4_STATE_EVENT( evt_msg_1_st_4 )

  SOL4_STATE_EVENT( evt_self_state_notify )

  SOL4_STATE_ON_ENTER( on_enter_state )
  SOL4_STATE_ON_EXIT( on_exit_state )
SOL4_STATE_FINISH()

SOL4_STATE_START( st_shutdown )
  SOL4_STATE_EVENT( evt_self_state_notify )
```

```
    SOL4_STATE_ON_ENTER( on_enter_state )
    SOL4_STATE_ON_ENTER( on_enter_st_shutdown )
  SOL4_STATE_FINISH()

  // Указание SObjectizer-у рассылать сообщения об
  // изменении состояния агента.
  SOL4_CHANGE_STATE_NOTIFY()

SOL4_CLASS_FINISH()

//
// a_main_t
//
a_main_t::a_main_t()
  :
  base_type_t( "a_main" )
  {}

a_main_t::~a_main_t()
  {}

void
a_main_t::so_on_subscription()
  {
    so_subscribe( "evt_start",
      so_4::rt::sobjectizer_agent_name(), "msg_start" );

    so_subscribe( "evt_msg_1_st_1", "msg_1" );

    so_subscribe( "evt_msg_1_st_2_pri_1", "msg_1", 1 );

    so_subscribe( "evt_msg_1_st_2_pri_0", "msg_1" );

    so_subscribe( "evt_msg_1_st_3_pri_1_one", "msg_1", 1 );

    so_subscribe( "evt_msg_1_st_3_pri_1_two", "msg_1", 1 );

    so_subscribe( "evt_msg_1_st_3_pri_0", "msg_1" );

    so_subscribe( "evt_msg_1_st_4", "msg_1" );

    so_subscribe( "evt_self_state_notify", "so_msg_state", 10 );
  }

void
a_main_t::evt_start()
  {
```

```
    std::cout << so_query_name() << ".evt_start" << std::endl;

    // Сообщение msg_1 должно возникать переодически.
    so_4::api::send_msg( so_query_name(), "msg_1", 0, "", 500, 1000 );
  }

void
a_main_t::evt_msg_1_st_1()
  {
    std::cout << so_query_name() << ".evt_msg_1_st_1" << std::endl;

    so_change_state( "st_2" );
  }

void
a_main_t::evt_msg_1_st_2_pri_1()
  {
    std::cout << so_query_name() << ".evt_msg_1_st_2_pri_1" << std::endl;
  }

void
a_main_t::evt_msg_1_st_2_pri_0()
  {
    std::cout << so_query_name() << ".evt_msg_1_st_2_pri_0" << std::endl;

    so_change_state( "st_3" );
  }

void
a_main_t::evt_msg_1_st_3_pri_1_one()
  {
    std::cout << so_query_name()
      << ".evt_msg_1_st_3_pri_1_one" << std::endl;
  }

void
a_main_t::evt_msg_1_st_3_pri_1_two()
  {
    std::cout << so_query_name()
      << ".evt_msg_1_st_3_pri_1_two" << std::endl;
  }

void
a_main_t::evt_msg_1_st_3_pri_0()
  {
    std::cout << so_query_name() << ".evt_msg_1_st_3_pri_0" << std::endl;

    so_change_state( "st_4" );
```

```
  }

void
a_main_t::evt_msg_1_st_4()
  {
    std::cout << so_query_name() << ".evt_msg_1_st_4" << std::endl;

    so_change_state( "st_shutdown" );
  }

void
a_main_t::evt_self_state_notify(
  const so_4::rt::so_msg_state * cmd )
  {
    std::cout << "so_msg_state: agent: " << cmd->m_agent
    << ", state: " << cmd->m_state << std::endl;
  }

void
a_main_t::on_enter_state(
  const std::string & state_name )
  {
    std::cout << "\tenter state: " << state_name <<std::endl;
  }

void
a_main_t::on_exit_state(
  const std::string & state_name )
  {
    std::cout << "\texit state: " << state_name <<std::endl;
  }

void
a_main_t::on_enter_st_shutdown(
  const std::string & state_name )
  {
    std::cout << "System is shutting down..." << std::endl;
    so_4::api::send_msg( so_4::rt::sobjectizer_agent_name(),
      "msg_normal_shutdown" );
  }

int
main()
  {
    // Наш агент.
    a_main_t a_main;
    // Статический слушатель состояний.
    sample_listener_t l( "static_listener" );
```

```
    a_main.so_add_nondestroyable_listener( 1 );
    // И динамический слушатель.
    // Т.к. динамический слушатель назначается агенту до
    // регистрации агента, то он не сможет определить начальное
    // состояние агента в своем методе stored.
    a_main.so_add_destroyable_listener(
      new sample_listener_t( "dynamic_listener" ) );

    // И кооперация для агента.
    so_4::rt::agent_coop_t a_main_coop( a_main );

    // Запускаем SObjectizer Run-Time.
    so_4::ret_code_t rc = so_4::api::start(
      so_4::disp::one_thread::create_disp(
        so_4::timer_thread::simple::create_timer_thread(),
        so_4::auto_destroy_timer ),
      so_4::auto_destroy_disp,
      &a_main_coop );
    if( rc )
      {
        // Запустить SObjectizer Run-Time не удалось.
        std::cerr << "start: " << rc << std::endl;
      }
    else
      std::cerr << "successful finish" << std::endl;

    return int( rc );
  }
```

A.7 Исходный код примера inheritance

A.7.1 Файл main.cpp

```
/*
  Демонстрация наследования агентов.

  Определяется базовый класс транзакции. От него
  производятся классы конкретной транзакции и
  прерываемой по таймеру транзакции. От классов
  конкретной транзакции и прерываемой по таймеру
  транзакции производится класс конкретной,
  прерываемой по таймеру транзакции.
*/

#include <iostream>
#include <memory>

#include <so_4/rt/h/rt.hpp>
```

```
#include <so_4/api/h/api.hpp>

#include <so_4/timer_thread/simple/h/pub.hpp>
#include <so_4/disp/one_thread/h/pub.hpp>

/*
  Базовый класс для транзакции.

  Играет роль интерфейса, т.к. только определяет основные
  сообщения и состояния. Но не выполняет никаких действий.

  Виртуально наследуется от so_4::rt::agent_t, т.к.
  впоследствии будет использоваться во множественном
  наследовании.
*/
class a_trx_t
  :  public virtual so_4::rt::agent_t
  {
    typedef so_4::rt::agent_t agent_type_t;
  public :
    a_trx_t();
    virtual ~a_trx_t();

    virtual const char *
    so_query_type() const;

    virtual void
    so_on_subscription();

    // Сообщение о начале транзакции.
    struct  msg_start {};

    // Сообщение о необходимости завершить
    // транзакцию с сохранением всех изменений.
    struct  msg_commit {};

    // Сообщение о необходимости откатить
    // транзакцию в начальное состояние.
    struct  msg_rollback {};

  protected :
    // Реальная подписка.
    /*
      Ничего не делает, но введен для того,
      чтобы при множественном наследовании
      избежать ошибки множественного
      вызова a_trx_t::so_on_subscription().
    */
```

```
    void
    do_subscription();
  };

SOL4_CLASS_START( a_trx_t )

  SOL4_MSG_START( msg_start, a_trx_t::msg_start )
  SOL4_MSG_FINISH()

  SOL4_MSG_START( msg_commit, a_trx_t::msg_commit )
  SOL4_MSG_FINISH()

  SOL4_MSG_START( msg_rollback, a_trx_t::msg_rollback )
  SOL4_MSG_FINISH()

SOL4_CLASS_FINISH()

a_trx_t::a_trx_t()
  :  agent_type_t( "a_trx" )
  {}

a_trx_t::~a_trx_t()
  {}

void
a_trx_t::so_on_subscription()
  {
    do_subscription();
  }

void
a_trx_t::do_subscription()
  {}


/*
  Класс конкретной транзакции, которая выполняет
  какую-то работу.

  Виртуально производится от a_trx_t, т.к. впоследствии
  будет использоваться во множественном наследовании.
*/
class a_concrete_trx_t
  :  public virtual a_trx_t
  {
    typedef so_4::rt::agent_t agent_type_t;
  public :
    a_concrete_trx_t(
```

```
    // Время (в миллисекундах) через которое
    // будет имитировано сообщение msg_commit.
    unsigned int commit_timeout,
    // Время (в миллисекундах) через которое
    // будет имитировано сообщение msg_rollback.
    unsigned int rollback_timeout );
  virtual ~a_concrete_trx_t();

  virtual const char *
  so_query_type() const;

  virtual void
  so_on_subscription();

  /*
    Реакция на появление агента в системе.

    Для имитации работы начинает транзакцию.
  */
  void
  evt_start();

  /*
    Начинает транзакцию.
  */
  void
  evt_trx_start();

  /*
    Подтверждает транзакцию.
  */
  void
  evt_trx_commit();

  /*
    Откатывает транзакцию.
  */
  void
  evt_trx_rollback();

protected :
  // Реальная подписка.
  void
  do_subscription();

private :
  // Время (в миллисекундах) через которое
  // будет имитировано сообщение msg_commit.
```

```
    unsigned int m_commit_timeout;
    // Время (в миллисекундах) через которое
    // будет имитировано сообщение msg_rollback.
    unsigned int m_rollback_timeout;
  };

SOL4_CLASS_START( a_concrete_trx_t )
  // Указываем наследование.
  SOL4_SUPER_CLASS( a_trx_t )

  // Поскольку есть наследование, нужно
  // сразу определить начальное состояние.
  SOL4_INITIAL_STATE( st_not_started )

  SOL4_EVENT( evt_start )
  SOL4_EVENT( evt_trx_start )
  SOL4_EVENT( evt_trx_commit )
  SOL4_EVENT( evt_trx_rollback )

  SOL4_STATE_START( st_not_started )
    SOL4_STATE_EVENT( evt_start )
    SOL4_STATE_EVENT( evt_trx_start )
  SOL4_STATE_FINISH()

  SOL4_STATE_START( st_started )
    SOL4_STATE_EVENT( evt_trx_commit )
    SOL4_STATE_EVENT( evt_trx_rollback )
  SOL4_STATE_FINISH()

  // Конечное состояние, в которое осуществляется
  // переход по подтверждению транзакции.
  SOL4_STATE_START( st_commited )
  SOL4_STATE_FINISH()

SOL4_CLASS_FINISH()

a_concrete_trx_t::a_concrete_trx_t(
  unsigned int commit_timeout,
  unsigned int rollback_timeout )
  : agent_type_t( "a_concrete_trx" )
  , m_commit_timeout( commit_timeout )
  , m_rollback_timeout( rollback_timeout )
  {
    std::cout << "commit_timeout: " << commit_timeout
      << "\nrollback_timeout: " << rollback_timeout
      << std::endl;
  }
```

```
a_concrete_trx_t::~a_concrete_trx_t()
  {}

void
a_concrete_trx_t::so_on_subscription()
  {
    a_trx_t::do_subscription();
    do_subscription();
  }

void
a_concrete_trx_t::evt_start()
  {
    // Имитируем начало транзакции.
    so_4::api::send_msg( so_query_name(), "msg_start" );

    // Имитируем успешное завершение транзакции.
    so_4::api::send_msg( so_query_name(), "msg_commit", 0,
      so_query_name(), m_commit_timeout );
    // Имитируем откат транзакции.
    so_4::api::send_msg( so_query_name(), "msg_rollback", 0,
      so_query_name(), m_rollback_timeout );
  }

void
a_concrete_trx_t::evt_trx_start()
  {
    // Начинаем транзакцию.
    so_change_state( "st_started" );

    std::cout << "trx started" << std::endl;
  }

void
a_concrete_trx_t::evt_trx_commit()
  {
    // Подтверждаем транзакцию.
    so_change_state( "st_commited" );

    std::cout << "trx commited" << std::endl;

    so_4::api::send_msg(
      so_4::rt::sobjectizer_agent_name(),
      "msg_normal_shutdown" );
  }

void
a_concrete_trx_t::evt_trx_rollback()
```

```
  {
    // Откатываем транзакцию.
    so_change_state( "st_not_started" );

    std::cout << "trx rollbacked" << std::endl;

    so_4::api::send_msg(
      so_4::rt::sobjectizer_agent_name(),
      "msg_normal_shutdown" );
  }

void
a_concrete_trx_t::do_subscription()
  {
    so_subscribe( "evt_start",
      so_4::rt::sobjectizer_agent_name(), "msg_start" );

    so_subscribe( "evt_trx_start", "msg_start" );

    so_subscribe( "evt_trx_commit", "msg_commit" );

    so_subscribe( "evt_trx_rollback", "msg_rollback" );
  }

/*
  Класс прерываемой по таймеру транзакции. Не
  выполняет никаких действий и предназначен
  быть примесью (mixin-ом) для конкретных
  классов транзакций.

  Виртуально производится от a_trx_t, т.к. впоследствии
  будет использоваться во множественном наследовании.
*/
class a_timed_trx_t
  : public virtual a_trx_t
  {
    typedef so_4::rt::agent_t agent_type_t;
  public :
    a_timed_trx_t(
      // Время (в миллисекундах) через которое
      // будет работа транзакции будет прервана.
      unsigned int lifetime );
    virtual ~a_timed_trx_t();

    virtual const char *
    so_query_type() const;

    virtual void
```

```
    so_on_subscription();

    // Сообщение об истечении времени работы
    // транзакции.
    struct  msg_lifetime_left {};

    /*
      Реакция на истечение времени работы
      транзакции. Отсылает самому себе
      сообщение msg_rollback.
    */
    void
    evt_lifetime_left(
      const msg_lifetime_left * );

    /*
      Обработчик входа в состояние, означающее
      начало транзакции. Производный класс
      должен либо указать его в качестве обработчика
      входа в состояние, либо вызвать из собственного
      обработчика.
    */
    void
    on_enter_appropriate_state(
      const std::string & );

  protected :
    // Реальная подписка.
    void
    do_subscription();

  private :
    // Время (в миллисекундах) через которое
    // будет работа транзакции будет прервана.
    unsigned int m_lifetime;
  };

SOL4_CLASS_START( a_timed_trx_t )
  // Указываем наследование.
  SOL4_SUPER_CLASS( a_trx_t )

  // Начальное состояние не указывается,
  // т.к. мы не вводим здесь своих состояний.

  SOL4_MSG_START( msg_lifetime_left,
    a_timed_trx_t::msg_lifetime_left )
  SOL4_MSG_FINISH()
```

```
  SOL4_EVENT_STC( evt_lifetime_left,
    a_timed_trx_t::msg_lifetime_left )

SOL4_CLASS_FINISH()

a_timed_trx_t::a_timed_trx_t(
  unsigned int lifetime )
  : agent_type_t( "a_timed_trx" )
  , m_lifetime( lifetime )
  {
    std::cout << "lifetime: " << lifetime
      << std::endl;
  }

a_timed_trx_t::~a_timed_trx_t()
  {}

void
a_timed_trx_t::so_on_subscription()
  {
    a_trx_t::do_subscription();
    do_subscription();
  }

void
a_timed_trx_t::evt_lifetime_left(
  const msg_lifetime_left * )
  {
    // Транзакцию пора прерывать.
    so_4::api::send_msg( so_query_name(), "msg_rollback" );

    std::cout << "no time left" << std::endl;
  }

void
a_timed_trx_t::on_enter_appropriate_state(
  const std::string & )
  {
    // Начинаем отсчет времени работы транзакции.
    so_4::api::send_msg( so_query_name(), "msg_lifetime_left", 0,
      so_query_name(), m_lifetime );
  }

void
a_timed_trx_t::do_subscription()
  {
    so_subscribe( "evt_lifetime_left", "msg_lifetime_left" );
  }
```

```
/*
  Класс конкретной, прерываемой по таймеру транзакции,
  которая выполняет какую-то работу.
*/
class a_concrete_timed_trx_t
  : public virtual a_concrete_trx_t
  , public virtual a_timed_trx_t
  {
    typedef so_4::rt::agent_t agent_type_t;
  public :
    a_concrete_timed_trx_t(
      // Время (в миллисекундах) через которое
      // будет имитировано сообщение msg_commit.
      unsigned int commit_timeout,
      // Время (в миллисекундах) через которое
      // будет имитировано сообщение msg_rollback.
      unsigned int rollback_timeout,
      // Время (в миллисекундах) через которое
      // будет работа транзакции будет прервана.
      unsigned int lifetime );
    virtual ~a_concrete_timed_trx_t();

    virtual const char *
    so_query_type() const;

    virtual void
    so_on_subscription();

  protected :
    // Реальная подписка.
    /*
      Ничего не делает. Введен для единообразия
      и с расчетом на возможное наполнение в
      будущем.
    */
    void
    do_subscription();
  };

SOL4_CLASS_START( a_concrete_timed_trx_t )
  // Указываем наследование.
  SOL4_SUPER_CLASS( a_concrete_trx_t )
  SOL4_SUPER_CLASS( a_timed_trx_t )

  // Поскольку есть наследование, нужно
  // сразу определить начальное состояние.
  SOL4_INITIAL_STATE( st_not_started )
```

```
  // Своих событий и состояний не вводится,
  // но событие st_started нужно преопределить
  // с учетом требований класса a_timed_trx_t.

  SOL4_STATE_START( st_started )
    SOL4_STATE_MERGE( a_concrete_trx_t, st_started )

    SOL4_STATE_EVENT( evt_lifetime_left )

    SOL4_STATE_ON_ENTER( on_enter_appropriate_state )
  SOL4_STATE_FINISH()

SOL4_CLASS_FINISH()

a_concrete_timed_trx_t::a_concrete_timed_trx_t(
  unsigned int commit_timeout,
  unsigned int rollback_timeout,
  unsigned int lifetime )
  : agent_type_t( "a_concrete_timed_trx" )
  , a_concrete_trx_t( commit_timeout, rollback_timeout )
  , a_timed_trx_t( lifetime )
  {}

a_concrete_timed_trx_t::~a_concrete_timed_trx_t()
  {}

void
a_concrete_timed_trx_t::so_on_subscription()
  {
    a_trx_t::do_subscription();
    a_concrete_trx_t::do_subscription();
    a_timed_trx_t::do_subscription();
    do_subscription();
  }

void
a_concrete_timed_trx_t::do_subscription()
  {}

int
main()
  {
    std::auto_ptr< so_4::timer_thread::timer_thread_t >
      timer_ptr( so_4::timer_thread::simple::create_timer_thread() );

    std::auto_ptr< so_4::rt::dispatcher_t >
      disp_ptr( so_4::disp::one_thread::create_disp( *timer_ptr ) );
```

```
    // Эти значения нужно варьировать, чтобы получать
    // различные результаты работы примера.
    a_concrete_timed_trx_t agent( 2500, 2000, 1500 );
    so_4::rt::agent_coop_t coop( agent );

    so_4::ret_code_t rc = so_4::api::start( *disp_ptr, &coop );
    if( rc )
      {
        std::cerr << "start: " << rc << std::endl;
      }

    return int( rc );
  }
```

A.8 Исходный код примера subscr_hook

A.8.1 Файл main.cpp

```
/*
  Демонстрация hook-ов подписки.
*/

#include <iostream>

#include <so_4/api/h/api.hpp>
#include <so_4/rt/h/rt.hpp>

#include <so_4/timer_thread/simple/h/pub.hpp>
#include <so_4/disp/active_obj/h/pub.hpp>

// После начала работы отсылает сообщения msg_hello и msg_bye.
class a_child_t
  : public so_4::rt::agent_t
  {
    typedef so_4::rt::agent_t base_type_t;
  public :
    a_child_t(
      const std::string & self_name )
      : base_type_t( self_name )
      {
        so_add_traits(
          so_4::disp::active_obj::query_active_obj_traits() );
      }
    virtual ~a_child_t()
      {}

    struct msg_hello {};
```

```
    struct msg_bye {};

    virtual const char *
    so_query_type() const;

    virtual void
    so_on_subscription()
      {
        so_subscribe( "evt_start",
          so_4::rt::sobjectizer_agent_name(), "msg_start" );
      }

    void
    evt_start()
      {
        so_4::api::send_msg( so_query_name(), "msg_hello", 0 );
        so_4::api::send_msg( so_query_name(), "msg_bye", 0 );
      }
  };

SOL4_CLASS_START( a_child_t )

  SOL4_MSG_START( msg_hello, a_child_t::msg_hello )
  SOL4_MSG_FINISH()

  SOL4_MSG_START( msg_bye, a_child_t::msg_bye )
  SOL4_MSG_FINISH()

  SOL4_EVENT( evt_start )

  SOL4_STATE_START( st_normal )
    SOL4_STATE_EVENT( evt_start )
  SOL4_STATE_FINISH()

SOL4_CLASS_FINISH()

// При начале работы создает подчиненную кооперацию с
// агентом a_child. Получив от a_child сообщение msg_bye
// завершает работу примера.
class a_owner_t
  : public so_4::rt::agent_t
  {
    typedef so_4::rt::agent_t base_type_t;
  public :
    a_owner_t()
      : base_type_t( "a_owner" )
      {}
    virtual ~a_owner_t()
```

```
  {}

virtual const char *
so_query_type() const;

virtual void
so_on_subscription()
  {
    so_subscribe( "evt_start",
      so_4::rt::sobjectizer_agent_name(), "msg_start" );
  }

void
evt_start()
  {
    // Создаем подчиненную кооперацию.
    a_child_t * child = new a_child_t( "a_child" );
    so_4::rt::dyn_agent_coop_t * child_coop =
      new so_4::rt::dyn_agent_coop_t( child );

    // И подписываемся на сообщения дочернего агента.
    so_4::rt::def_subscr_hook( *child_coop,
      // Подписываем родительского агента.
      *this, "evt_hello",
      // На сообщения дочернего агента.
      // В этом случае через указатель на агента.
      *child, "msg_hello" );
    so_4::rt::def_subscr_hook( *child_coop,
      // Подписываем родительского агента.
      *this, "evt_bye",
      // На сообщения дочернего агента.
      // В этом случае через имя агента (для примера).
      "a_child", "msg_bye" );

    // Регистрируем кооперацию.
    so_4::rt::dyn_agent_coop_helper_t child_coop_helper(
      child_coop );
  }

void
evt_hello()
  {
    std::cout << "hello!" << std::endl;
  }

void
evt_bye()
  {
```

```
        std::cout << "bye!" << std::endl;

        so_4::api::send_msg( so_4::rt::sobjectizer_agent_name(),
        "msg_normal_shutdown", 0 );
      }
  };

SOL4_CLASS_START( a_owner_t )

  SOL4_EVENT( evt_start )
  SOL4_EVENT( evt_hello )
  SOL4_EVENT( evt_bye )

  SOL4_STATE_START( st_normal )
    SOL4_STATE_EVENT( evt_start )
    SOL4_STATE_EVENT( evt_hello )
    SOL4_STATE_EVENT( evt_bye )
  SOL4_STATE_FINISH()

SOL4_CLASS_FINISH()

int
main()
  {
    a_owner_t a_owner;
    so_4::rt::agent_coop_t coop( a_owner );

    so_4::ret_code_t rc = so_4::api::start(
      // Диспетчер будет уничтожен при выходе из start().
      so_4::disp::active_obj::create_disp(
        // Таймер будет уничтожен диспетчером.
        so_4::timer_thread::simple::create_timer_thread(),
        so_4::auto_destroy_timer ),
      so_4::auto_destroy_disp,
      &coop );
    if( rc )
      {
        std::cerr << "start: " << rc << std::endl;
      }

    return int( rc );
  }
```

A.9 Исходный код примера filter

A.9.1 Файл c1i.hpp

```
/*
```

```
  Интерфейс сервера для клиента #1.
*/

#if !defined( _SAMPLE__FILTER__C1I_HPP_ )
#define _SAMPLE__FILTER__C1I_HPP_

#include <so_4/rt/h/rt.hpp>

/*
  Глобальный агент, сообщениями которого взаимодействуют
  сервер и клиент #1.
*/
class c1i_t
  : public so_4::rt::agent_t
  {
    typedef so_4::rt::agent_t base_type_t;
  public :
    c1i_t();
    virtual ~c1i_t();

    virtual const char *
    so_query_type() const;

    virtual void
    so_on_subscription() = 0;

    // Имя единственного агента этого типа.
    static const std::string &
    agent_name();

    // Тип этого агента. Нужен для обращения
    // к so_4::api::make_global_agent.
    static const std::string &
    agent_type();

    // Запрос. Отсылается от клиента к серверу.
    struct  msg_request
      {};

    // Ответ. Отсылается сервером клиенту.
    struct  msg_reply
      {};
  };

#endif
```

A.9.2 Файл c1i.cpp

```
/*
  Интерфейс сервера для клиента #1.
*/

#include <so_4/h/std_incl.hpp>

#include "c1i.hpp"

// Описание агента для SObjectizer.
SOL4_CLASS_START( c1i_t )

  SOL4_MSG_START( msg_request, c1i_t::msg_request )
  SOL4_MSG_FINISH()

  SOL4_MSG_START( msg_reply, c1i_t::msg_reply )
  SOL4_MSG_FINISH()

SOL4_CLASS_FINISH()

// Реализация агента.
c1i_t::c1i_t()
  :
  // Сразу назначаем имя агента.
  base_type_t( agent_name() )
  {}

c1i_t::~c1i_t()
  {}

const std::string &
c1i_t::agent_name()
  {
    // Имя глобального агента.
    static std::string name( "a_c1i" );
    return name;
  }

const std::string &
c1i_t::agent_type()
  {
    // Тип глобального агента. В точности тоже имя,
    // которое указано в SOL4_CLASS_START.
    static std::string type_name( "c1i_t" );
    return type_name;
  }
```

A.9.3 Файл c2i.hpp

```
/*
  Интерфейс сервера для клиента #2.
*/

#if !defined( _SAMPLE__FILTER__C2I_HPP_ )
#define _SAMPLE__FILTER__C2I_HPP_

#include <so_4/rt/h/rt.hpp>

/*
  Глобальный агент, сообщениями которого взаимодействуют
  сервер и клиент #2.
*/
class c2i_t
  : public so_4::rt::agent_t
  {
    typedef so_4::rt::agent_t base_type_t;
  public :
    c2i_t();
    virtual ~c2i_t();

    virtual const char *
    so_query_type() const;

    virtual void
    so_on_subscription() = 0;

    // Имя единственного агента этого типа.
    static const std::string &
    agent_name();

    // Тип этого агента. Нужен для обращения
    // к so_4::api::make_global_agent.
    static const std::string &
    agent_type();

    // Запрос. Сервером клиенту.
    struct  msg_request
      {};

    // Ответ. Клиентом серверу.
    struct  msg_reply
      {};
  };

#endif
```

A.9.4 Файл c2i.cpp

```
/*
  Интерфейс сервера для клиента #2.
*/

#include <so_4/h/std_incl.hpp>

#include "c2i.hpp"

// Описание агента для SObjectizer.
SOL4_CLASS_START( c2i_t )

  SOL4_MSG_START( msg_request, c2i_t::msg_request )
  SOL4_MSG_FINISH()

  SOL4_MSG_START( msg_reply, c2i_t::msg_reply )
  SOL4_MSG_FINISH()

SOL4_CLASS_FINISH()

// Реализация агента.
c2i_t::c2i_t()
  :
  // Сразу назначаем имя агента.
  base_type_t( agent_name() )
  {}

c2i_t::~c2i_t()
  {}

const std::string &
c2i_t::agent_name()
  {
    // Имя глобального агента.
    static std::string name( "a_c2i" );
    return name;
  }

const std::string &
c2i_t::agent_type()
  {
    // Тип глобального агента. В точности тоже имя,
    // которое указано в SOL4_CLASS_START.
    static std::string type_name( "c2i_t" );
    return type_name;
  }
```

A.9.5 Файл c1.cpp

```
/*
  Клиент #1.

  Отсылает запросы серверу и ждет ответа от сервера.

  Демонстрирует возможность запуска SObjectizer-а на вспомогательной
  нити и проведения диалога с пользователем на контексте главной нити.
*/

#include <iostream>
#include <memory>

#include <threads_1/h/threads.hpp>

#include <so_4/rt/h/rt.hpp>
#include <so_4/rt/h/msg_auto_ptr.hpp>

#include <so_4/socket/channels/h/channels.hpp>
#include <so_4/rt/comm/h/a_cln_channel.hpp>

#include <so_4/api/h/api.hpp>

#include <so_4/api/h/api.hpp>

#include <so_4/timer_thread/simple/h/pub.hpp>
#include <so_4/disp/active_obj/h/pub.hpp>

#include "c1i.hpp"

//
// Класс агента клиента #1.
//
class a_cln_t
  :  public so_4::rt::agent_t
  {
    typedef so_4::rt::agent_t base_type_t;
  public :
    a_cln_t();
    virtual ~a_cln_t();

    virtual const char *
    so_query_type() const;

    virtual void
    so_on_subscription();
```

```
    // Имя единственного агента этого типа в приложении.
    static const std::string &
    agent_name();

    // Сообщение, которое указывает клиенту отослать
    // запрос серверу.
    struct  msg_send_request
      {};

    // Реакция на появление агента в системе.
    void
    evt_start();

    // Реакция на команду отсылки запроса серверу.
    void
    evt_send_request(
      const msg_send_request * );

    // Реакция на ответ сервера.
    void
    evt_server_reply(
      const c1i_t::msg_reply * );
  };

SOL4_CLASS_START( a_cln_t )

  SOL4_MSG_START( msg_send_request,
    a_cln_t::msg_send_request )
  SOL4_MSG_FINISH()

  SOL4_EVENT( evt_start )

  SOL4_EVENT_STC(
    evt_send_request,
    a_cln_t::msg_send_request )

  SOL4_EVENT_STC(
    evt_server_reply,
    c1i_t::msg_reply )

  SOL4_STATE_START( st_normal )
    SOL4_STATE_EVENT( evt_start )
    SOL4_STATE_EVENT( evt_send_request )
    SOL4_STATE_EVENT( evt_server_reply )
  SOL4_STATE_FINISH()

SOL4_CLASS_FINISH()
```

```
a_cln_t::a_cln_t()
  :  base_type_t( agent_name().c_str() )
  {
    // Делаем агента активным объектом.
    so_add_traits( so_4::disp::active_obj::
      query_active_obj_traits() );
  }

a_cln_t::~a_cln_t()
  {}

void
a_cln_t::so_on_subscription()
  {
    // Подписываем те события, инциденты для которых
    // в системе уже существуют.

    so_subscribe( "evt_start",
      so_4::rt::sobjectizer_agent_name(), "msg_start" );

    so_subscribe( "evt_send_request", "msg_send_request" );
  }

const std::string &
a_cln_t::agent_name()
  {
    static std::string name( "a_cln" );
    return name;
  }

void
a_cln_t::evt_start()
  {
    // Агент стартовал.

    // Нужно зарегистрировать глобального агента и
    // подписаться на его сообщение.
    so_4::ret_code_t rc = so_4::api::make_global_agent(
      c1i_t::agent_name(),
      c1i_t::agent_type() );
    if( rc )
      std::cerr << rc << std::endl;
    else
      so_subscribe( "evt_server_reply",
        c1i_t::agent_name(), "msg_reply" );
  }

void
```

```
a_cln_t::evt_send_request(
  const msg_send_request * )
  {
    so_4::api::send_msg( c1i_t::agent_name(),
      "msg_request", 0 );
  }

void
a_cln_t::evt_server_reply(
  const c1i_t::msg_reply * )
  {
    // Ответ сервера получен.
    std::cout << "Reply from server received" << std::endl;
  }

// Создание кооперации, в которую входит главный агент
// клиента #1 и коммуникационный агент (клиентский сокет).
void
create_coop(
  // Адрес серверного сокета.
  const char * sock_addr )
  {
    a_cln_t * a_cln = new a_cln_t();

    // Создание клиентского сокета для взаимодействия по SOP.
    // Фильтр, который допускает только сообщения агента c1i.
    so_4::sop::std_filter_t * filter =
      so_4::sop::create_std_filter();
    filter->insert( c1i_t::agent_name() );

    so_4::rt::comm::a_cln_channel_t * a_sock =
      new so_4::rt::comm::a_cln_channel_t(
        "a_sock",
        so_4::socket::channels::create_client_factory( sock_addr ),
        // Назначаем фильтр.
        filter,
        // Назначаем обработчик разрывов связи.
        so_4::rt::comm::a_cln_channel_base_t::
          create_def_disconnect_handler( 5000, 0 )
      );
    // Сокет так же будет активным агентом.
    a_sock->so_add_traits( so_4::disp::active_obj::
      query_active_obj_traits() );
    // Поскольку трафик предполагается небольшой, то устанавливаем
    // порог так, чтобы сообщения уходили сразу же.
    a_sock->set_out_threshold( so_4::rt::comm::threshold_t( 1, 1 ) );

    so_4::rt::agent_t * agents[] =
```

```
      {
        a_cln, a_sock
      };
    so_4::rt::dyn_agent_coop_helper_t coop_helper(
      new so_4::rt::dyn_agent_coop_t(
        "client_coop", agents,
        sizeof( agents ) / sizeof( agents[ 0 ] ) ) );

    if( coop_helper.result() )
      std::cerr << "register_coop:\n"
        << coop_helper.result() << std::endl;
  }

// Уничтожение клиентской кооперации.
void
destroy_coop()
  {
    so_4::ret_code_t rc =
      so_4::api::deregister_coop( "client_coop" );
    if( rc )
      std::cerr << "deregister_coop:\n" << rc << std::endl;
  }

//
// Нить, на которой будет происходить запуск SObjectizer-а
//
class sobj_thread_t
  :  public threads_1::thread_t
  {
  public :
    sobj_thread_t();
    virtual ~sobj_thread_t();

  protected :
    virtual void
    body();
  };

sobj_thread_t::sobj_thread_t()
  {}

sobj_thread_t::~sobj_thread_t()
  {}

void
sobj_thread_t::body()
  {
    std::auto_ptr< so_4::timer_thread::timer_thread_t >
```

```
      timer_ptr( so_4::timer_thread::simple::create_timer_thread() );

    std::auto_ptr< so_4::rt::dispatcher_t >
      disp_ptr( so_4::disp::active_obj::create_disp( *timer_ptr ) );

    so_4::ret_code_t rc = so_4::api::start( *disp_ptr, 0 );
    if( rc )
      {
        std::cerr << "start: " << rc << std::endl;
      }
  }

int
main( int argc, char ** argv )
  {
    if( 2 == argc )
      {
        sobj_thread_t thread;
        thread.start();

        // Засыпаем, чтобы дать стартовать SObjectizer.
        // Это самый простой способ синхронизации с sobj_thread_t.
        threads_1::sleep_thread( 1000 );

        bool is_continue = true;
        while( is_continue )
          {
            std::string choice;

            std::cout << "Choose action:\n"
              "\t0 - quit\n"
              "\t1 - create client coop\n"
              "\t2 - destroy client coop\n"
              "\t3 - send request to server\n> "
              << std::flush;

            std::cin >> choice;

            if( choice == "0" )
              {
                // Завершаем работу.
                so_4::api::send_msg(
                  so_4::rt::sobjectizer_agent_name(),
                  "msg_normal_shutdown", 0,
                  so_4::rt::sobjectizer_agent_name() );
                is_continue = false;
              }
            else if( choice == "1" )
```

```
            // Создаем кооперацию клиента.
            create_coop( argv[ 1 ] );
          else if( choice == "2" )
            // Уничтожаем кооперацию клиента.
            destroy_coop();
          else if( choice == "3" )
            {
              // Отсылаем запрос серверу.
              so_4::api::send_msg(
                a_cln_t::agent_name(),
                "msg_send_request", 0 );
            }
        }

      // Ожидаем завершения SObjectizer.
      thread.wait();

      return 0;
    }
  else
    {
      std::cerr << "sample_filter_c1 <server_sock_addr>"
        << std::endl;

      return -1;
    }
}
```

A.9.6 Файл c2.cpp

```
/*
  Клиент #2.

  Получает запросы сервера и отсылает серверу ответы.

  Демонстрирует возможность запуска SObjectizer-а на вспомогательной
  нити и проведения диалога с пользователем на контексте главной нити.
*/

#include <iostream>
#include <memory>

#include <threads_1/h/threads.hpp>

#include <so_4/rt/h/rt.hpp>
#include <so_4/rt/h/msg_auto_ptr.hpp>

#include <so_4/socket/channels/h/channels.hpp>
```

```
#include <so_4/rt/comm/h/a_cln_channel.hpp>

#include <so_4/api/h/api.hpp>

#include <so_4/api/h/api.hpp>

#include <so_4/timer_thread/simple/h/pub.hpp>
#include <so_4/disp/active_obj/h/pub.hpp>

#include "c2i.hpp"

//
// Класс агента клиента #2.
//
class a_cln_t
  :  public so_4::rt::agent_t
  {
    typedef so_4::rt::agent_t base_type_t;
  public :
    a_cln_t();
    virtual ~a_cln_t();

    virtual const char *
    so_query_type() const;

    virtual void
    so_on_subscription();

    // Имя единственного агента этого типа в приложении.
    static const std::string &
    agent_name();

    // Реакция на появление агента в системе.
    void
    evt_start();

    // Реакция на запрос сервера.
    void
    evt_server_request(
      const c2i_t::msg_request * );
  };

SOL4_CLASS_START( a_cln_t )

  SOL4_EVENT( evt_start )

  SOL4_EVENT_STC(
    evt_server_request,
```

```
    c2i_t::msg_request )

  SOL4_STATE_START( st_normal )
    SOL4_STATE_EVENT( evt_start )
    SOL4_STATE_EVENT( evt_server_request )
  SOL4_STATE_FINISH()

SOL4_CLASS_FINISH()

a_cln_t::a_cln_t()
  :  base_type_t( agent_name().c_str() )
  {
    // Делаем агента активным объектом.
    so_add_traits( so_4::disp::active_obj::
      query_active_obj_traits() );
  }

a_cln_t::~a_cln_t()
  {}

void
a_cln_t::so_on_subscription()
  {
    // Подписываем те события, инциденты для которых
    // в системе уже существуют.

    so_subscribe( "evt_start",
      so_4::rt::sobjectizer_agent_name(), "msg_start" );

  }

const std::string &
a_cln_t::agent_name()
  {
    static std::string name( "a_cln" );
    return name;
  }

void
a_cln_t::evt_start()
  {
    // Агент стартовал.

    // Нужно зарегистрировать глобального агента и
    // подписаться на его сообщение.
    so_4::ret_code_t rc = so_4::api::make_global_agent(
      c2i_t::agent_name(),
      c2i_t::agent_type() );
```

```
    if( rc )
      std::cerr << rc << std::endl;
    else
      so_subscribe( "evt_server_request",
        c2i_t::agent_name(), "msg_request" );
  }

void
a_cln_t::evt_server_request(
  const c2i_t::msg_request * )
  {
    std::cout << "Request from server" << std::endl;
    so_4::api::send_msg( c2i_t::agent_name(),
      "msg_reply", 0 );
  }

// Создание кооперации, в которую входит главный агент
// клиента #2 и коммуникационный агент (клиентский сокет).
void
create_coop(
  // Адрес серверного сокета.
  const char * sock_addr )
  {
    a_cln_t * a_cln = new a_cln_t();

    // Создание клиентского сокета для взаимодействия по SOP.
    // Фильтр, который допускает только сообщения агента c2i.
    so_4::sop::std_filter_t * filter =
      so_4::sop::create_std_filter();
    filter->insert( c2i_t::agent_name() );

    so_4::rt::comm::a_cln_channel_t * a_sock =
      new so_4::rt::comm::a_cln_channel_t(
        "a_sock",
        so_4::socket::channels::create_client_factory( sock_addr ),
        // Назначаем фильтр.
        filter,
        // Назначаем обработчик разрывов связи.
        so_4::rt::comm::a_cln_channel_base_t::
          create_def_disconnect_handler( 5000, 0 )
      );
    // Сокет так же будет активным агентом.
    a_sock->so_add_traits( so_4::disp::active_obj::
      query_active_obj_traits() );
    // Поскольку трафик предполагается небольшой, то устанавливаем
    // порог так, чтобы сообщения уходили сразу же.
    a_sock->set_out_threshold(
      so_4::rt::comm::threshold_t( 1, 1 ) );
```

```
    so_4::rt::agent_t * agents[] =
      {
        a_cln, a_sock
      };
    so_4::rt::dyn_agent_coop_helper_t coop_helper(
      new so_4::rt::dyn_agent_coop_t(
        "client_coop", agents,
        sizeof( agents ) / sizeof( agents[ 0 ] ) ) );

    if( coop_helper.result() )
      std::cerr << "register_coop:\n"
        << coop_helper.result() << std::endl;
  }

// Уничтожение клиентской кооперации.
void
destroy_coop()
  {
    so_4::ret_code_t rc =
      so_4::api::deregister_coop( "client_coop" );
    if( rc )
      std::cerr << "deregister_coop:\n" << rc << std::endl;
  }

//
// Нить, на которой будет происходить запуск SObjectizer-а
//
class sobj_thread_t
  :  public threads_1::thread_t
  {
  public :
    sobj_thread_t();
    virtual ~sobj_thread_t();

  protected :
    virtual void
    body();
  };

sobj_thread_t::sobj_thread_t()
  {}

sobj_thread_t::~sobj_thread_t()
  {}

void
sobj_thread_t::body()
```

```
  {
    std::auto_ptr< so_4::timer_thread::timer_thread_t >
      timer_ptr( so_4::timer_thread::simple::create_timer_thread() );

    std::auto_ptr< so_4::rt::dispatcher_t >
      disp_ptr( so_4::disp::active_obj::create_disp( *timer_ptr ) );

    so_4::ret_code_t rc = so_4::api::start( *disp_ptr, 0 );
    if( rc )
      {
        std::cerr << "start: " << rc << std::endl;
      }
  }

int
main( int argc, char ** argv )
  {
    if( 2 == argc )
      {
        sobj_thread_t thread;
        thread.start();

        // Засыпаем, чтобы дать стартовать SObjectizer.
        // Это самый простой способ синхронизации с sobj_thread_t.
        threads_1::sleep_thread( 1000 );

        bool is_continue = true;
        while( is_continue )
          {
            std::string choice;

            std::cout << "Choose action:\n"
              "\t0 - quit\n"
              "\t1 - create client coop\n"
              "\t2 - destroy client coop\n> "
              << std::flush;

            std::cin >> choice;

            if( choice == "0" )
              {
                // Завершаем работу.
                so_4::api::send_msg(
                  so_4::rt::sobjectizer_agent_name(),
                  "msg_normal_shutdown", 0,
                  so_4::rt::sobjectizer_agent_name() );
                is_continue = false;
              }
```

```
            else if( choice == "1" )
              // Создаем кооперацию клиента.
              create_coop( argv[ 1 ] );
            else if( choice == "2" )
              // Уничтожаем кооперацию клиента.
              destroy_coop();
          }

        // Ожидаем завершения SObjectizer.
        thread.wait();

        return 0;
      }
    else
      {
        std::cerr << "sample_filter_c2 <server_sock_addr>"
          << std::endl;

        return -1;
      }
  }
```

A.9.7 Файл server.cpp

```
/*
  Сервер.

  Получает запросы от клиента #1 и передает их клиенту #2.
  Получает ответы от клиента #2 и передает их клиенту #1.
*/

#include <iostream>
#include <memory>

#include <so_4/rt/h/rt.hpp>
#include <so_4/rt/h/msg_auto_ptr.hpp>

#include <so_4/socket/channels/h/channels.hpp>
#include <so_4/rt/comm/h/a_srv_channel.hpp>

#include <so_4/api/h/api.hpp>

#include <so_4/timer_thread/simple/h/pub.hpp>
#include <so_4/disp/active_obj/h/pub.hpp>

#include <so_4/sop/h/filter.hpp>

#include "c1i.hpp"
```

```
#include "c2i.hpp"

//
// Класс главного агента серверной части.
//
class a_srv_t
  :  public so_4::rt::agent_t
  {
    typedef so_4::rt::agent_t base_type_t;
  public :
    a_srv_t();
    virtual ~a_srv_t();

    virtual const char *
    so_query_type() const;

    virtual void
    so_on_subscription();

    // Имя единственного агента этого типа в приложении.
    static const std::string &
    agent_name();

    // Реакция на появление агента в системе.
    void
    evt_start();

    // Реакция на запрос клиента #1.
    void
    evt_c1_request(
      const c1i_t::msg_request * );

    // Реакция на ответ клиента #2.
    void
    evt_c2_reply(
      const c2i_t::msg_reply * );
  };

SOL4_CLASS_START( a_srv_t )

  SOL4_EVENT( evt_start )

  SOL4_EVENT_STC(
    evt_c1_request,
    c1i_t::msg_request )

  SOL4_EVENT_STC(
    evt_c2_reply,
```

```
    c2i_t::msg_reply )

  SOL4_STATE_START( st_normal )
    SOL4_STATE_EVENT( evt_start )
    SOL4_STATE_EVENT( evt_c1_request )
    SOL4_STATE_EVENT( evt_c2_reply )
  SOL4_STATE_FINISH()

SOL4_CLASS_FINISH()

a_srv_t::a_srv_t()
  : base_type_t( agent_name() )
  {}

a_srv_t::~a_srv_t()
  {}

void
a_srv_t::so_on_subscription()
  {
    // Подписываем те события, инциденты для которых
    // в системе уже существуют.
    so_subscribe( "evt_start",
      so_4::rt::sobjectizer_agent_name(), "msg_start" );
  }

const std::string &
a_srv_t::agent_name()
  {
    static std::string name( "a_srv" );
    return name;
  }

void
a_srv_t::evt_start()
  {
    so_4::ret_code_t rc = so_4::api::make_global_agent(
      c1i_t::agent_name(),
      c1i_t::agent_type() );
    if( rc )
      std::cerr << rc << std::endl;
    else
      {
        rc = so_4::api::make_global_agent(
          c2i_t::agent_name(),
          c2i_t::agent_type() );
        if( rc )
          std::cerr << rc << std::endl;
```

```
        else
          {
            // Подписываемся на сообщения глобальных агентов.
            so_subscribe( "evt_c1_request",
              c1i_t::agent_name(), "msg_request" );

            so_subscribe( "evt_c2_reply",
              c2i_t::agent_name(), "msg_reply" );
          }
      }
  }

void
a_srv_t::evt_c1_request(
  const c1i_t::msg_request * )
  {
    std::cout << "Request from client #1" << std::endl;

    // Нужно отослать запрос клиенту #2.
    so_4::api::send_msg( c2i_t::agent_name(),
      "msg_request", 0 );
  }

void
a_srv_t::evt_c2_reply(
  const c2i_t::msg_reply * )
  {
    std::cout << "Reply from client #2" << std::endl;

    // Нужно отослать ответ клиенту #1.
    so_4::api::send_msg( c1i_t::agent_name(),
      "msg_reply", 0 );
  }

int
main( int argc, char ** argv )
  {
    if( 2 == argc )
      {
        // Готовимся запустить диспетчер.
        std::auto_ptr< so_4::timer_thread::timer_thread_t >
          timer_ptr( so_4::timer_thread::simple::create_timer_thread() );

        std::auto_ptr< so_4::rt::dispatcher_t >
          disp_ptr( so_4::disp::active_obj::create_disp( *timer_ptr ) );

        // Агент серверного сокета.
        so_4::rt::comm::a_srv_channel_t * a_socksrv =
```

```
        new so_4::rt::comm::a_srv_channel_t(
          "a_srv_channel",
          so_4::socket::channels::create_server_channel(
            argv[ 1 ] ) );

      // Должен быть активным объектом.
      a_socksrv->so_add_traits( so_4::disp::active_obj::
        query_active_obj_traits() );

      // Агент-сервер.
      a_srv_t * a_server = new a_srv_t();

      // Подготавливаем стартовую кооперацию, которая и будет
      // выполнять основную работу приложения.
      so_4::rt::agent_t * agents[] =
        {
          a_socksrv,
          a_server
        };
      so_4::rt::dyn_agent_coop_t * coop = new
        so_4::rt::dyn_agent_coop_t(
          "a_srv", agents,
          sizeof( agents ) / sizeof( agents[ 0 ] ) );

      // Запускаемся на работу.
      so_4::ret_code_t rc = so_4::api::start( *disp_ptr, coop );
      if( rc )
        {
          std::cerr << "start: " << rc << std::endl;
        }

      return int( rc );
    }
  else
    {
      std::cerr << "sample_filter_server <sock_addr>"
        << std::endl;

      return -1;
    }
}
```

A.10 Исходный код примера high_traffic

A.10.1 Файл common.ddl

```
{type data_t
  {attr m_data  {of std::string}}
```

```
}
```

A.10.2 Файл common.cpp

```
#include <string>

#include <stdio.h>
#include <time.h>

#include <oess_1/stdsn/h/serializable.hpp>
#include <oess_1/stdsn/h/inout_templ.hpp>

//
// Общая часть, необходимая как серверу, так и клиенту.
// Включена непосредственно в код для того, чтобы не нужно
// было создавать дополнительных библиотек.
//

// Объекты этого типа передаются в сообщениях глобального агента.
class data_t
  : public oess_1::stdsn::serializable_t
  {
    OESS_SERIALIZER( data_t )
    public :
      data_t()
        {}

      data_t(
        // Размер данных в байтах.
        unsigned int size )
        : m_data( size, '0' )
        {}

      virtual ~data_t()
        {}

    private :
      // Размер и значение задается в инициализирующем конструкторе.
      std::string m_data;
  };

#include "common.ddl.cpp"

// Глобальный агент, сообщения которого используются для
// взаимодействия клиента и сервера.
class a_common_t
  : public  so_4::rt::agent_t
  {
```

```
typedef so_4::rt::agent_t base_type_t;
public :
  a_common_t()
    : base_type_t( agent_name() )
    {}
  virtual ~a_common_t()
    {}

  virtual const char *
  so_query_type() const;

  static std::string
  agent_name()
    {
      return "a_common";
    }

  static std::string
  agent_type()
    {
      return "a_common_t";
    }

  // Тип сообщения-запроса к серверу.
  struct  msg_request
    {
      // Порядковый номер запроса.
      unsigned int  m_uid;
      // Данные запроса.
      data_t  m_data;

      msg_request()
        {}
      msg_request(
        unsigned int uid,
        unsigned int size )
        : m_uid( uid )
        , m_data( size )
        {}

      static bool
      check( const msg_request * cmd )
        {
          return ( 0 != cmd );
        }
    };

  // Тип сообщения-ответа от сервера.
```

```
      struct  msg_reply
        {
          // Порядковый номер запроса.
          unsigned int  m_uid;

          msg_reply()
            {}
          msg_reply(
            unsigned int uid )
            : m_uid( uid )
            {}

          static bool
          check( const msg_reply * cmd )
            {
              return ( 0 != cmd );
            }
        };
  };

SOL4_CLASS_START( a_common_t )

  SOL4_MSG_START( msg_request, a_common_t::msg_request )
    SOL4_MSG_FIELD( m_uid )
    SOL4_MSG_FIELD( m_data )
    SOL4_MSG_CHECKER( a_common_t::msg_request::check )
  SOL4_MSG_FINISH()

  SOL4_MSG_START( msg_reply, a_common_t::msg_reply )
    SOL4_MSG_FIELD( m_uid )
    SOL4_MSG_CHECKER( a_common_t::msg_reply::check )
  SOL4_MSG_FINISH()

SOL4_CLASS_FINISH()
```

A.10.3 Файл client.cpp

```
/*
  Тестирование поведения коммуникационных агентов при
  большом трафике.

  Клиентская часть.
*/

#include <iostream>
#include <set>

#include <stdio.h>
```

```
#include <so_4/api/h/api.hpp>
#include <so_4/rt/h/rt.hpp>

#include <so_4/rt/comm/h/a_cln_channel.hpp>
#include <so_4/socket/channels/h/channels.hpp>

#include <so_4/timer_thread/simple/h/pub.hpp>
#include <so_4/disp/active_obj/h/pub.hpp>

#include "common.cpp"

// Класс тестового агента, который отсылает сообщения.
class a_sender_t
  : public so_4::rt::agent_t
  {
    typedef so_4::rt::agent_t base_type_t;
    private :
      // Размер одного запроса.
      unsigned int  m_data_size;

      // Количество запросов, которые нужно отослать на сервер.
      unsigned int  m_request_count;
      // Количество полученных от сервера ответов.
      unsigned int  m_reply_received;

      // Количество запросов в группе.
      unsigned int  m_group_size;
      // Тайм-аут между отсылкой групп.
      unsigned int  m_timeout;

      // Счетчик идентификаторов для отсылаемых запросов.
      unsigned int  m_uid;

      // Множество идентификаторов отосланных запросов,
      // на которые еще не были полученны ответы.
      std::set< unsigned int >  m_no_reply_uids;

      // Признак того, что есть соединение с сервером.
      bool  m_is_connected;

      void
      show_stat() const
        {
          double percents = double( m_reply_received ) /
            double( m_request_count ) * 100.0;
          std::cout << "*** "
            << ( m_is_connected ? "connected" : "not connected" )
```

```
          << ", " << percents << "% ("
          << m_no_reply_uids.size() << ")\r"
          << std::flush;
      }

    void
    send( unsigned int uid,
      bool insert_to_no_reply_uids = true )
      {
        so_4::api::send_msg_safely(
          a_common_t::agent_name(),
          "msg_request",
          new a_common_t::msg_request( uid, m_data_size ) );
        if( insert_to_no_reply_uids )
          m_no_reply_uids.insert( uid );
      }

  public :
    a_sender_t(
      unsigned int data_size,
      unsigned int request_count,
      unsigned int group_size,
      unsigned int timeout )
      : base_type_t( "a_receiver" )
      , m_data_size( data_size )
      , m_request_count( request_count )
      , m_reply_received( 0 )
      , m_group_size( group_size )
      , m_timeout( timeout )
      , m_uid( 0 )
      , m_is_connected( false )
      {}
    virtual ~a_sender_t()
      {}

    struct  msg_timeout {};

    virtual const char *
    so_query_type() const;

    virtual void
    so_on_subscription()
      {
        // Агент a_common должен быть глобальным.
        so_4::api::make_global_agent(
          a_common_t::agent_name(),
          a_common_t::agent_type() );
```

```
    so_subscribe( "evt_start",
      so_4::rt::sobjectizer_agent_name(), "msg_start" );

    so_subscribe( "evt_client_connected",
      so_4::rt::comm::communicator_agent_name(),
      "msg_client_connected" );

    so_subscribe( "evt_client_disconnected",
      so_4::rt::comm::communicator_agent_name(),
      "msg_client_disconnected" );

    so_subscribe( "evt_reply",
      a_common_t::agent_name(),
      "msg_reply" );

    so_subscribe( "evt_timeout", "msg_timeout" );
  }

void
evt_start(
  const so_4::rt::event_data_t & )
  {
    so_4::api::send_msg( so_query_name(), "msg_timeout", 0,
      so_query_name(), 1000, 1000 * m_timeout );
  }

void
evt_client_connected(
  const so_4::rt::event_data_t & data,
  const so_4::rt::comm::msg_client_connected * cmd )
  {
    m_is_connected = true;
    show_stat();
  }

void
evt_client_disconnected(
  const so_4::rt::event_data_t & data,
  const so_4::rt::comm::msg_client_disconnected * cmd )
  {
    m_is_connected = false;
    show_stat();
  }

void
evt_reply(
  const so_4::rt::event_data_t &,
  const a_common_t::msg_reply * cmd )
```

```
      {
        if( m_no_reply_uids.find( cmd->m_uid ) !=
          m_no_reply_uids.end() )
          {
            m_no_reply_uids.erase( cmd->m_uid );
            ++m_reply_received;
            show_stat();

            if( m_reply_received == m_request_count )
              {
                so_4::api::send_msg(
                  so_4::rt::sobjectizer_agent_name(),
                  "msg_normal_shutdown", 0 );
              }
          }
      }

    void
    evt_timeout(
      const so_4::rt::event_data_t & )
      {
        unsigned int sent = 0;

        // Сначала отсылаем повторно те запросы,
        // на которые не получено ответов.
        for( std::set< unsigned int >::iterator
            it = m_no_reply_uids.begin(),
            it_end = m_no_reply_uids.end();
          it != it_end && sent != m_group_size;
          ++it,
          ++sent )
          {
            send( *it, false );
          }

        // Затем доведем группу до нужного размера
        // новыми запросами.
        for( unsigned int count = 0;
          sent + count != m_group_size &&
          m_uid != m_request_count;
          ++count, ++m_uid )
          {
            send( m_uid );
          }
      }
  };

SOL4_CLASS_START( a_sender_t )
```

```
  SOL4_MSG_START( msg_timeout, a_sender_t::msg_timeout )
  SOL4_MSG_FINISH()

  SOL4_EVENT( evt_start )
  SOL4_EVENT_STC(
    evt_client_connected,
    so_4::rt::comm::msg_client_connected )
  SOL4_EVENT_STC(
    evt_client_disconnected,
    so_4::rt::comm::msg_client_disconnected )
  SOL4_EVENT_STC(
    evt_reply,
    a_common_t::msg_reply )
  SOL4_EVENT( evt_timeout )

  SOL4_STATE_START( st_normal )
    SOL4_STATE_EVENT( evt_start )
    SOL4_STATE_EVENT( evt_client_connected )
    SOL4_STATE_EVENT( evt_client_disconnected )
    SOL4_STATE_EVENT( evt_reply )
    SOL4_STATE_EVENT( evt_timeout )
  SOL4_STATE_FINISH()

SOL4_CLASS_FINISH()

int
main( int argc, char ** argv )
  {
    if( 6 == argc )
      {
        unsigned int data_size = 0;
        sscanf( argv[ 2 ], "%u", &data_size );

        unsigned int count = 0;
        sscanf( argv[ 3 ], "%u", &count );

        unsigned int group_size = 0;
        sscanf( argv[ 4 ], "%u", &group_size );

        unsigned int timeout = 0;
        sscanf( argv[ 5 ], "%u", &timeout );


        so_4::sop::std_filter_t * filter =
          so_4::sop::create_std_filter();
        filter->insert( a_common_t::agent_name() );
```

```
      so_4::rt::comm::a_cln_channel_t a_channel(
        "a_channel",
        so_4::socket::channels::create_client_factory( argv[ 1 ] ),
        filter,
        so_4::rt::comm::a_cln_channel_t::
          create_def_disconnect_handler( 5000, 0 ) );
      so_4::disp::active_obj::make_active( a_channel );

      std::cout << "Client threshold: in "
        << a_channel.in_threshold()
        << ", out " << a_channel.out_threshold()
        << std::endl;

      a_sender_t a_sender(
        data_size, count, group_size, timeout );

      so_4::rt::agent_t * agents[] =
        {
          &a_channel, &a_sender
        };
      so_4::rt::agent_coop_t coop( "server",
        agents, sizeof( agents ) / sizeof( agents[ 0 ] ) );

      so_4::ret_code_t rc = so_4::api::start(
        // Диспетчер будет уничтожен при выходе из start().
        so_4::disp::active_obj::create_disp(
          // Таймер будет уничтожен диспетчером.
          so_4::timer_thread::simple::create_timer_thread(),
          so_4::auto_destroy_timer ),
        so_4::auto_destroy_disp,
        &coop );

      if( rc )
        {
          std::cerr << "start: " << rc << std::endl;
        }

      return int( rc );
    }

  std::cerr << "sample_high_traffic_client <[ip]:port> "
    "data_size count group_size timeout" << std::endl;
  return 1;
}
```

A.10.4 Файл server.cpp

```
/*
```

```
  Тестирование поведения коммуникационных агентов при
  большом трафике.

  Серверная часть.
*/

#include <iostream>

#include <stdio.h>

#include <so_4/api/h/api.hpp>
#include <so_4/rt/h/rt.hpp>

#include <so_4/rt/comm/h/a_srv_channel.hpp>
#include <so_4/socket/channels/h/channels.hpp>

#include <so_4/timer_thread/simple/h/pub.hpp>
#include <so_4/disp/one_thread/h/pub.hpp>

#include "common.cpp"

// Класс тестового агента, который получает сообщения.
class a_receiver_t
  : public so_4::rt::agent_t
  {
    typedef so_4::rt::agent_t base_type_t;
    public :
      a_receiver_t()
        : base_type_t( "a_receiver" )
        {}
      virtual ~a_receiver_t()
        {}

      virtual const char *
      so_query_type() const;

      virtual void
      so_on_subscription()
        {
          // Агент a_common должен быть глобальным.
          so_4::api::make_global_agent(
            a_common_t::agent_name(),
            a_common_t::agent_type() );

          so_subscribe( "evt_request",
            a_common_t::agent_name(),
            "msg_request" );
```

```
        so_subscribe( "evt_client_connected",
          "a_channel", "msg_client_connected" );

        so_subscribe( "evt_client_disconnected",
          "a_channel", "msg_client_disconnected" );
      }

    void
    evt_request(
      const so_4::rt::event_data_t & data,
      const a_common_t::msg_request * cmd )
      {
        std::cout << cmd->m_uid << " ";
        // Сразу отвечаем ответным сообщением.
        so_4::api::send_msg_safely(
          data.channel(),
          a_common_t::agent_name(),
          "msg_reply",
          new a_common_t::msg_reply( cmd->m_uid ) );
      }

    void
    evt_client_connected(
      const so_4::rt::event_data_t & data,
      const so_4::rt::comm::msg_client_connected * cmd )
      {
        std::cout << "\nclient connected: " << cmd->m_channel
          << std::endl;
      }

    void
    evt_client_disconnected(
      const so_4::rt::event_data_t & data,
      const so_4::rt::comm::msg_client_disconnected * cmd )
      {
        std::cout << "\nclient disconnected: " << cmd->m_channel
          << std::endl;
      }
  };

SOL4_CLASS_START( a_receiver_t )

  SOL4_EVENT_STC(
    evt_request,
    a_common_t::msg_request )
  SOL4_EVENT_STC(
    evt_client_connected,
    so_4::rt::comm::msg_client_connected )
```

```
  SOL4_EVENT_STC(
    evt_client_disconnected,
    so_4::rt::comm::msg_client_disconnected )

  SOL4_STATE_START( st_normal )
    SOL4_STATE_EVENT( evt_request )
    SOL4_STATE_EVENT( evt_client_connected )
    SOL4_STATE_EVENT( evt_client_disconnected )
  SOL4_STATE_FINISH()

SOL4_CLASS_FINISH()

int
main( int argc, char ** argv )
  {
    if( 2 == argc )
      {
        so_4::rt::comm::a_srv_channel_t a_channel(
          "a_channel",
          so_4::socket::channels::create_server_channel( argv[ 1 ] ) );

        std::cout << "Server thresholds: in " << a_channel.in_threshold()
          << ", out " << a_channel.out_threshold()
          << std::endl;

        a_receiver_t a_receiver;

        so_4::rt::agent_t * agents[] =
          {
            &a_channel, &a_receiver
          };
        so_4::rt::agent_coop_t coop( "server",
          agents, sizeof( agents ) / sizeof( agents[ 0 ] ) );

        so_4::ret_code_t rc = so_4::api::start(
          // Диспетчер будет уничтожен при выходе из start().
          so_4::disp::one_thread::create_disp(
            // Таймер будет уничтожен диспетчером.
            so_4::timer_thread::simple::create_timer_thread(),
            so_4::auto_destroy_timer ),
          so_4::auto_destroy_disp,
          &coop );

        if( rc )
          {
            std::cerr << "start: " << rc << std::endl;
          }
```

```
        return int( rc );
      }

    std::cerr << "sample_high_traffic_server <[ip]:port>" << std::endl;
    return 1;
  }
```

A.11 Исходный код примера raw_channel

A.11.1 Файл tcp_cln.cpp

```
/*
  Пример, демонстрирующий работу с серверным raw-соединением.

  Производится попытка подключения к указанному адресу,
  после чего в соединение отсылаются все необязательные
  аргументы командной строки. После этого осуществляется
  переход в режим чтения и отображения данных.
*/

#include <iostream>

// Базовые заголовочные файлы SObjectizer.
#include <so_4/rt/h/rt.hpp>
#include <so_4/api/h/api.hpp>

#include <so_4/timer_thread/simple/h/pub.hpp>
#include <so_4/disp/active_obj/h/pub.hpp>

// Описание агента, который обслуживает серверный
// raw-канал и средств для создания каналов.
#include <so_4/rt/comm/h/a_raw_cln_channel.hpp>
#include <so_4/socket/channels/h/channels.hpp>

//
// Класс агента, который будет работать с серверным соединением.
//
class a_main_t
  : public so_4::rt::agent_t
  {
    typedef so_4::rt::agent_t base_type_t;
  public :
    a_main_t(
      int argc,
      char ** argv );
    virtual ~a_main_t();

    virtual const char *
```

```
  so_query_type() const;

  virtual void
  so_on_subscription();

  // Имя главного тестового агента, который должен
  // присутствовать в приложении в единственном числе.
  static std::string &
  agent_name();

  // Имя агента клиентского соединения, который должен
  // присутствовать в приложении в единственном числе.
  static std::string &
  tcp_agent_name();

  // Реакция на успешное создание клиентского сокета.
  void
  evt_success(
    const so_4::rt::comm::a_cln_channel_base_t::msg_success * );

  // Реакция на неудачное создание клиентского сокета.
  void
  evt_fail(
    const so_4::rt::comm::a_cln_channel_base_t::msg_fail * );

  // Реакция на подключение клиента.
  void
  evt_client_connected(
    const so_4::rt::comm::msg_client_connected * cmd );

  // Реакция на отключение клиента.
  void
  evt_client_disconnected(
    const so_4::rt::comm::msg_client_disconnected * cmd );

  // Реакция на поступление данных в канал.
  void
  evt_incoming_data(
    const so_4::rt::comm::msg_raw_package * cmd );

private :
  char ** m_argv;
  int m_argc;

  // Завершение работы примера.
  void
  shutdown()
    {
```

```
        so_4::api::send_msg(
          so_4::rt::sobjectizer_agent_name(),
          "msg_normal_shutdown", 0 );
      }
  };

SOL4_CLASS_START( a_main_t )

  SOL4_EVENT_STC(
    evt_success,
    so_4::rt::comm::a_cln_channel_base_t::msg_success )
  SOL4_EVENT_STC(
    evt_fail,
    so_4::rt::comm::a_cln_channel_base_t::msg_fail )
  SOL4_EVENT_STC(
    evt_client_connected,
    so_4::rt::comm::msg_client_connected )
  SOL4_EVENT_STC(
    evt_client_disconnected,
    so_4::rt::comm::msg_client_disconnected )
  SOL4_EVENT_STC(
    evt_incoming_data,
    so_4::rt::comm::msg_raw_package )

  SOL4_STATE_START( st_normal )
    SOL4_STATE_EVENT( evt_success )
    SOL4_STATE_EVENT( evt_fail )
    SOL4_STATE_EVENT( evt_client_connected )
    SOL4_STATE_EVENT( evt_client_disconnected )
    SOL4_STATE_EVENT( evt_incoming_data )
  SOL4_STATE_FINISH()

SOL4_CLASS_FINISH()

a_main_t::a_main_t(
  int argc,
  char ** argv )
  : base_type_t( agent_name().c_str() )
  , m_argc( argc )
  , m_argv( argv )
  {}

a_main_t::~a_main_t()
  {}

void
a_main_t::so_on_subscription()
  {
```

```
    so_subscribe( "evt_success", tcp_agent_name(), "msg_success" );

    so_subscribe( "evt_fail", tcp_agent_name(), "msg_fail" );

    so_subscribe( "evt_client_connected", tcp_agent_name(),
      "msg_client_connected" );

    so_subscribe( "evt_client_disconnected", tcp_agent_name(),
      "msg_client_disconnected" );

    so_subscribe( "evt_incoming_data", tcp_agent_name(),
      "msg_raw_package" );
  }

std::string &
a_main_t::agent_name()
  {
    static std::string name( "a_main" );

    return name;
  }

std::string &
a_main_t::tcp_agent_name()
  {
    static std::string name( "a_tcp_srvsock" );

    return name;
  }

void
a_main_t::evt_success(
  const so_4::rt::comm::a_cln_channel_base_t::msg_success * )
  {
    std::cout << so_query_name() << ".evt_success" << std::endl;
  }

void
a_main_t::evt_fail(
  const so_4::rt::comm::a_cln_channel_base_t::msg_fail * cmd )
  {
    std::cout << so_query_name() << ".evt_fail: "
      << cmd->m_ret_code << std::endl;

    shutdown();
  }

// Замена последовательности "\n" на \r\n.
```

```
const std::string &
replace_escaped_lf( std::string & what )
  {
    std::string::size_type where = 0;
    while( std::string::npos !=
      ( where = what.find( "\\n", where ) ) )
      {
        what.replace( where, 2, "\r\n" );
        where += 2;
      }

    return what;
  }

void
a_main_t::evt_client_connected(
  const so_4::rt::comm::msg_client_connected * cmd )
  {
    std::cout << so_query_name() << ".evt_client_connected: "
      << cmd->m_channel.comm_agent() << ", "
      << cmd->m_channel.client()
      << std::endl;

    // Нужно отсылать данные в соединение.
    std::cout << "sending data" << std::endl;
    for( int i = 0; i != m_argc; ++i )
      {
        std::string what( m_argv[ i ] );
        replace_escaped_lf( what );

        // Данные нужно подготовить в comm_buf.
        so_4::rt::comm_buf_t data;
        data.insert( 0, what.data(), what.size() );
        data.insert( data.size(), "\r\n\r\n", 4 );

        // Отсылаем данные.
        so_4::api::send_msg_safely(
          cmd->m_channel.comm_agent(), "msg_send_package",
          new so_4::rt::comm::msg_send_package(
            cmd->m_channel.client(),
            data ) );
      }
  }

void
a_main_t::evt_client_disconnected(
  const so_4::rt::comm::msg_client_disconnected * cmd )
  {
```

```
    std::cout << so_query_name() << ".evt_client_disconnected: "
      << cmd->m_channel.comm_agent() << ", "
      << cmd->m_channel.client()
      << std::endl;

    shutdown();
  }

void
a_main_t::evt_incoming_data(
  const so_4::rt::comm::msg_raw_package * cmd )
  {
    std::cout << so_query_name() << ".evt_incoming_data: "
      << cmd->m_channel.comm_agent() << ", "
      << cmd->m_channel.client()
      << "\n\tdata size: " << cmd->m_package.size()
      << "\n\tis channel blocked: " << cmd->m_is_blocked
      << std::endl;

    std::string v(
      (const char *)cmd->m_package.ptr(),
      cmd->m_package.size() );
    std::cout << v << std::endl;

    // Если канал оказался заблокированным, то разблокируем его.
    cmd->unblock_channel();
  }

// Создание главной кооперации примера.
so_4::rt::agent_coop_t *
create_coop( const char * ip_address,
  int argc,
  char ** argv )
  {
    a_main_t * a_main = new a_main_t( argc, argv );
    so_4::rt::comm::a_raw_cln_channel_t * a_tcp_clnsock =
      new so_4::rt::comm::a_raw_cln_channel_t(
        a_main_t::tcp_agent_name(),
        so_4::socket::channels::
          create_client_factory( ip_address ) );
    // Сокет должен быть активным объектом.
    a_tcp_clnsock->so_add_traits(
      so_4::disp::active_obj::query_active_obj_traits() );
    // Чтение следуюшего буфера должно осуществляться только,
    // если мы успели обработать предыдущий пакет.
    a_tcp_clnsock->set_in_threshold(
      so_4::rt::comm::threshold_t( 1, 1 ) );
    // Запись должна осуществляться для каждого исходящего пакета.
```

```
    a_tcp_clnsock->set_out_threshold(
      so_4::rt::comm::threshold_t( 1, 1 ) );

    so_4::rt::agent_t * agents[] =
      {
        a_main, a_tcp_clnsock
      };

    return new so_4::rt::dyn_agent_coop_t( "srvsock1",
        agents, sizeof( agents ) / sizeof( agents[ 0 ] ) );
  }

int
main( int argc, char ** argv )
  {
    if( 2 <= argc )
      {
        // Создаем таймер, диспетчер. Затем запускаем
        // run-time SObjectizer-a.
        so_4::ret_code_t rc = so_4::api::start(
          // Диспетчер будет уничтожен при выходе из start().
          so_4::disp::active_obj::create_disp(
            // Таймер будет уничтожен диспетчером.
            so_4::timer_thread::simple::create_timer_thread(),
            so_4::auto_destroy_timer ),
          so_4::auto_destroy_disp,
          create_coop( argv[ 1 ], argc - 2, &(argv[ 2 ]) ) );
        if( rc )
          {
            // Ошибка старта.
            std::cerr << "start: " << rc << std::endl;
          }

        return rc;
      }
    else
      std::cerr << "sample_raw_channel_tcp_cln "
        "<ip-address> [data_to_send]"
        << std::endl;

    return 0;
  }
```

A.11.2 Файл tcp_srv.cpp

```
/*
  Пример, демонстрирующий работу с серверным raw-соединением.
```

```
  Создается серверное соединение и отображаются события,
  которые происходят с этим соединением.
*/

#include <iostream>

// Базовые заголовочные файлы SObjectizer.
#include <so_4/rt/h/rt.hpp>
#include <so_4/api/h/api.hpp>

// Используем простой диспетчер с одной рабочей нитью.
#include <so_4/timer_thread/simple/h/pub.hpp>
#include <so_4/disp/one_thread/h/pub.hpp>

// Описание агента, который обслуживает серверный
// raw-канал и средств для создания каналов.
#include <so_4/rt/comm/h/a_raw_srv_channel.hpp>
#include <so_4/socket/channels/h/channels.hpp>

//
// Класс агента, который будет работать с серверным соединением.
//
class a_main_t
  : public so_4::rt::agent_t
  {
    typedef so_4::rt::agent_t base_type_t;

  public :
    a_main_t();
    virtual ~a_main_t();

    virtual const char *
    so_query_type() const;

    virtual void
    so_on_subscription();

    // Имя главного тестового агента, который должен
    // присутствовать в приложении в единственном числе.
    static std::string &
    agent_name();

    // Имя агента серверного соединения, который должен
    // присутствовать в приложении в единственном числе.
    static std::string &
    tcp_agent_name();

    // Реакция на успешное создание серверного сокета.
```

```
    void
    evt_success(
      const so_4::rt::comm::a_srv_channel_base_t::msg_success & );

    // Реакция на неудачное создание серверного сокета.
    void
    evt_fail(
      const so_4::rt::comm::a_srv_channel_base_t::msg_fail & );

    // Реакция на подключение клиента.
    void
    evt_client_connected(
      const so_4::rt::comm::msg_client_connected & cmd );

    // Реакция на отключение клиента.
    void
    evt_client_disconnected(
      const so_4::rt::comm::msg_client_disconnected & cmd );

    // Реакция на поступление данных в канал.
    void
    evt_incoming_data(
      const so_4::rt::comm::msg_raw_package & cmd );

  };

SOL4_CLASS_START( a_main_t )

  SOL4_EVENT_STC(
    evt_success,
    so_4::rt::comm::a_srv_channel_base_t::msg_success )
  SOL4_EVENT_STC(
    evt_fail,
    so_4::rt::comm::a_srv_channel_base_t::msg_fail )
  SOL4_EVENT_STC(
    evt_client_connected,
    so_4::rt::comm::msg_client_connected )
  SOL4_EVENT_STC(
    evt_client_disconnected,
    so_4::rt::comm::msg_client_disconnected )
  SOL4_EVENT_STC(
    evt_incoming_data,
    so_4::rt::comm::msg_raw_package )

  SOL4_STATE_START( st_normal )
    SOL4_STATE_EVENT( evt_success )
    SOL4_STATE_EVENT( evt_fail )
    SOL4_STATE_EVENT( evt_client_connected )
```

```
    SOL4_STATE_EVENT( evt_client_disconnected )
    SOL4_STATE_EVENT( evt_incoming_data )
  SOL4_STATE_FINISH()

SOL4_CLASS_FINISH()

a_main_t::a_main_t()
  : base_type_t( agent_name().c_str() )
  {}

a_main_t::~a_main_t()
  {}

void
a_main_t::so_on_subscription()
  {
    so_subscribe( "evt_success", tcp_agent_name(), "msg_success" );

    so_subscribe( "evt_fail", tcp_agent_name(), "msg_fail" );

    so_subscribe( "evt_client_connected", tcp_agent_name(),
      "msg_client_connected" );

    so_subscribe( "evt_client_disconnected", tcp_agent_name(),
      "msg_client_disconnected" );

    so_subscribe( "evt_incoming_data", tcp_agent_name(),
      "msg_raw_package" );
  }

std::string &
a_main_t::agent_name()
  {
    static std::string name( "a_main" );

    return name;
  }

std::string &
a_main_t::tcp_agent_name()
  {
    static std::string name( "a_tcp_srvsock" );

    return name;
  }

void
a_main_t::evt_success(
```

```
  const so_4::rt::comm::a_srv_channel_base_t::msg_success & )
  {
    std::cout << so_query_name() << ".evt_success" << std::endl;
  }

void
a_main_t::evt_fail(
  const so_4::rt::comm::a_srv_channel_base_t::msg_fail & cmd )
  {
    std::cout << so_query_name() << ".evt_fail: "
      << cmd.m_ret_code << std::endl;
  }

void
a_main_t::evt_client_connected(
  const so_4::rt::comm::msg_client_connected & cmd )
  {
    std::cout << so_query_name() << ".evt_client_connected: "
      << cmd.m_channel.comm_agent() << ", "
      << cmd.m_channel.client()
      << std::endl;
  }

void
a_main_t::evt_client_disconnected(
  const so_4::rt::comm::msg_client_disconnected & cmd )
  {
    std::cout << so_query_name() << ".evt_client_disconnected: "
      << cmd.m_channel.comm_agent() << ", "
      << cmd.m_channel.client()
      << std::endl;
  }

void
a_main_t::evt_incoming_data(
  const so_4::rt::comm::msg_raw_package & cmd )
  {
    std::cout << so_query_name() << ".evt_incoming_data: "
      << cmd.m_channel.comm_agent() << ", "
      << cmd.m_channel.client()
      << "\n\tdata size: " << cmd.m_package.size() << std::endl;

    std::string v(
      (const char *)cmd.m_package.ptr(),
      cmd.m_package.size() );
    std::cout << v << std::endl;

    // Если канал оказался заблокированным, то разблокируем его.
```

```
    cmd.unblock_channel();
  }

// Создание главной кооперации примера.
so_4::rt::agent_coop_t *
create_coop( const char * ip_address )
  {
    a_main_t * a_main = new a_main_t();
    so_4::rt::comm::a_raw_srv_channel_t * a_tcp_srvsock =
      new so_4::rt::comm::a_raw_srv_channel_t(
        a_main_t::tcp_agent_name(),
        so_4::socket::channels::
          create_server_channel( ip_address ) );

    so_4::rt::agent_t * agents[] =
      {
        a_main, a_tcp_srvsock
      };

    return new so_4::rt::dyn_agent_coop_t( "srvsock1",
        agents, sizeof( agents ) / sizeof( agents[ 0 ] ) );
  }

int
main( int argc, char ** argv )
  {
    if( 2 == argc )
      {
        // Создаем таймер, диспетчер. Затем запускаем
        // run-time SObjectizer-a.
        so_4::ret_code_t rc = so_4::api::start(
          // Диспетчер будет уничтожен при выходе из start().
          so_4::disp::one_thread::create_disp(
            // Таймер будет уничтожен диспетчером.
            so_4::timer_thread::simple::create_timer_thread(),
            so_4::auto_destroy_timer ),
          so_4::auto_destroy_disp,
          create_coop( argv[ 1 ] ) );
        if( rc )
          {
            // Ошибка старта.
            std::cerr << "start: " << rc << std::endl;
          }

        return rc;
      }
    else
      std::cerr << "sample_raw_channel_tcp_srv <ip-address>" << std::endl;
```

```
    return 0;
  }
```

A.12 Исходный код примера parent_insend

A.12.1 Файл main.cpp

```
/*
  Демонстрация insend-событий и взаимоотношений кооперaций.

  Пример работает в интерактивном режиме. Оператор указывает
  момент регистрации и дерегистрации кооперации маршрутизатора.
  Агент-маршрутизатор в своем событии evt_start создает
  дочернюю кооперацию с агентом серверного сокета.

  Далее агент-маршрутизатор отслеживает сообщения о подключениях
  новых клиентов. Для каждого нового клиента создается агент-обработчик.
  Все приходящие от клиента данные пересылаются маршрутизатором
  агентам-обработчикам.

  Агент серверного сокета и агенты-обработчики являются активными
  агентами. Агент-маршрутизатор является пассивным агентом, но
  он использует insend-события.
*/

#include <iostream>
#include <map>

#include <cpp_util_2/h/lexcast.hpp>

#include <threads_1/h/threads.hpp>

#include <so_4/api/h/api.hpp>
#include <so_4/rt/h/rt.hpp>

#include <so_4/rt/comm/h/a_raw_srv_channel.hpp>
#include <so_4/socket/channels/h/channels.hpp>

#include <so_4/timer_thread/simple/h/pub.hpp>
#include <so_4/disp/active_obj/h/pub.hpp>

// Имя агента-маршрутизатора и его кооперации.
const std::string router_agent_name( "a_router" );
// Имя агента-серверного сокета.
const std::string server_agent_name( "a_server" );

// Класс агента, который обслуживает конкретный канал.
```

```
class a_listener_t
  : public so_4::rt::agent_t
  {
    typedef so_4::rt::agent_t base_type_t;
  public :
    a_listener_t( const std::string & agent_name )
      : base_type_t( agent_name )
      {
        so_4::disp::active_obj::make_active( *this );

        std::cout << so_query_name() << " created" << std::endl;
      }
    virtual ~a_listener_t()
      {
        std::cout << so_query_name() << " destroyed" << std::endl;
      }

    // Владеет сообщением msg_data, тип которого уже определен.
    typedef so_4::rt::comm::msg_raw_package msg_data;

    virtual const char *
    so_query_type() const;

    void
    so_on_subscription()
      {
        so_subscribe( "evt_data", "msg_data" );
      }

    void
    evt_data( const msg_data & cmd )
      {
        std::cout << "agent: " << so_query_name()
          << ", channel: " << cmd.m_channel
          << ", received: " << cmd.m_package.size()
          << " byte(s)" << std::endl;

        cmd.unblock_channel();
      }
  };

SOL4_CLASS_START( a_listener_t )

  SOL4_MSG_START( msg_data, a_listener_t::msg_data )
    SOL4_MSG_CHECKER( a_listener_t::msg_data::check )
  SOL4_MSG_FINISH()

  SOL4_EVENT_STC( evt_data, a_listener_t::msg_data )
```

```
  SOL4_STATE_START( st_normal )
    SOL4_STATE_EVENT( evt_data )
  SOL4_STATE_FINISH()

SOL4_CLASS_FINISH()

// Класс агента-маршрутизатора.
class a_router_t
  : public so_4::rt::agent_t
  {
    typedef so_4::rt::agent_t base_type_t;
  private :
    // Кооперация с агентом серверного сокета является
    // атрибутом агента-маршрутизатора. При этом кооперация
    // регистрируется как статическая, а не динамическая. Т.е. она не
    // будет уничтожена через delete при дерегистрации.
    so_4::rt::comm::a_raw_srv_channel_t m_srv_channel;
    so_4::rt::agent_coop_t  m_srv_channel_coop;

    // Тип карты подключенных клиентов.
    typedef std::map< so_4::rt::comm_channel_t, std::string >
      client_map_t;

    // Карта подключенных клиентов.
    // Доступ к ней не синхронизируется, т.к. маршрутизатор работает
    // только на контексте нити агента серверного сокета.
    client_map_t  m_clients;

    // Счетчик для создания уникальных имен дочерних агентов.
    int m_child_counter;

    // Вспомогательный метод для упрощения hook-а подписки
    // на сообщения агента серверного сокета.
    void
    setup_subscr_hook(
      const std::string & event,
      const std::string & msg )
      {
        so_4::rt::def_subscr_hook( m_srv_channel_coop,
          *this, event, m_srv_channel, msg, 0, &std::cerr,
          so_4::rt::evt_subscr_t::insend_dispatching );
      }

  public :
    a_router_t(
      const std::string & ip )
      : base_type_t( router_agent_name )
```

```
  , m_srv_channel( server_agent_name,
      so_4::socket::channels::create_server_channel( ip ) )
  , m_srv_channel_coop( m_srv_channel )
  , m_child_counter( 0 )
  {
    // Серверный агент должен быть активным агентом.
    so_4::disp::active_obj::make_active( m_srv_channel );

    std::cout << so_query_name() << " created" << std::endl;
  }
virtual ~a_router_t()
  {
    std::cout << so_query_name() << " destroyed" << std::endl;
  }

virtual const char *
so_query_type() const;

virtual void
so_on_subscription()
  {
    so_subscribe( "evt_start",
      so_4::rt::sobjectizer_agent_name(), "msg_start" );
  }

void
evt_start()
  {
    // Регистрируем кооперацию с серверным сокетом.
    // И подписываемся на сообщения агента серверного сокета.
    setup_subscr_hook( "evt_srv_channel_success", "msg_success" );
    setup_subscr_hook( "evt_srv_channel_fail", "msg_fail" );
    setup_subscr_hook( "evt_client_connected",
      "msg_client_connected" );
    setup_subscr_hook( "evt_client_disconnected",
      "msg_client_disconnected" );
    setup_subscr_hook( "evt_channel_data",
      "msg_raw_package" );

    // Регистрируем кооперацию.
    // Мы должны быть родителями для кооперации.
    m_srv_channel_coop.set_parent_coop_name(
      so_query_coop()->query_name() );
    so_4::api::register_coop( m_srv_channel_coop );
  }

void
evt_srv_channel_success()
```

```
  {
    std::cout << "server channel created" << std::endl;
  }

void
evt_srv_channel_fail(
  const so_4::rt::comm::a_srv_channel_base_t::msg_fail & cmd )
  {
    std::cout << "server channel not created: "
      << cmd.m_ret_code << std::endl;
  }

void
evt_client_connected(
  const so_4::rt::comm::msg_client_connected & cmd )
  {
    std::cout << "client connected: "
      << cmd.m_channel << std::endl;

    // Создаем для этого канала новую кооперацию.
    std::string agent_name( "a_listener_" +
      cpp_util_2::slexcast( ++m_child_counter ) );
    so_4::rt::dyn_agent_coop_t * coop(
      new so_4::rt::dyn_agent_coop_t(
        new a_listener_t( agent_name ) ) );
    // Указываем, что мы являемся родителями этой кооперации.
    coop->set_parent_coop_name(
      so_query_coop()->query_name() );

    // Регистрируем новую кооперацию.
    so_4::rt::dyn_agent_coop_helper_t h( coop );
    if( !h.result() )
      {
        m_clients[ cmd.m_channel ] = agent_name;
      }
  }

void
evt_client_disconnected(
  const so_4::rt::comm::msg_client_disconnected & cmd )
  {
    std::cout << "client disconnected: "
      << cmd.m_channel << std::endl;

    client_map_t::iterator it( m_clients.find( cmd.m_channel ) );
    if( it != m_clients.end() )
      {
        // Такой клиент нам известен. Нужно уничтожить прикладного
```

```
            // агента для этого клиента.
            so_4::api::deregister_coop( it->second );
            m_clients.erase( it );
          }
      }

    void
    evt_channel_data(
      const so_4::rt::comm::msg_raw_package & cmd )
      {
        // Определяем, какому агенту нужно переслать эти данные.
        client_map_t::iterator it( m_clients.find( cmd.m_channel ) );
        if( it != m_clients.end() )
          {
            so_4::api::send_msg_safely( it->second, "msg_data",
              new a_listener_t::msg_data( cmd ) );
          }
      }
};

SOL4_CLASS_START( a_router_t )

  SOL4_EVENT( evt_start )
  SOL4_EVENT( evt_srv_channel_success )
  SOL4_EVENT_STC( evt_srv_channel_fail,
    so_4::rt::comm::a_srv_channel_base_t::msg_fail )
  SOL4_EVENT_STC( evt_client_connected,
    so_4::rt::comm::msg_client_connected )
  SOL4_EVENT_STC( evt_client_disconnected,
    so_4::rt::comm::msg_client_disconnected )
  SOL4_EVENT_STC( evt_channel_data,
    so_4::rt::comm::msg_raw_package )

  SOL4_STATE_START( st_normal )
    SOL4_STATE_EVENT( evt_start )
    SOL4_STATE_EVENT( evt_srv_channel_success )
    SOL4_STATE_EVENT( evt_srv_channel_fail )
    SOL4_STATE_EVENT( evt_client_connected )
    SOL4_STATE_EVENT( evt_client_disconnected )
    SOL4_STATE_EVENT( evt_channel_data )
  SOL4_STATE_FINISH()

SOL4_CLASS_FINISH()

//
// Нить, на которой будет происходить запуск SObjectizer-а
//
class sobj_thread_t
```

```
  : public threads_1::thread_t
  {
  public :
    sobj_thread_t();
    virtual ~sobj_thread_t();

  protected :
    virtual void
    body();
  };

sobj_thread_t::sobj_thread_t()
  {}

sobj_thread_t::~sobj_thread_t()
  {}

void
sobj_thread_t::body()
  {
    so_4::ret_code_t rc = so_4::api::start(
      // Диспетчер будет уничтожен при выходе из start().
      so_4::disp::active_obj::create_disp(
        // Таймер будет уничтожен диспетчером.
        so_4::timer_thread::simple::create_timer_thread(),
        so_4::auto_destroy_timer ),
      so_4::auto_destroy_disp, 0 );
    if( rc )
      {
        std::cerr << "start: " << rc << std::endl;
      }
  }

void
create_coop( const std::string & ip )
  {
    so_4::rt::dyn_agent_coop_helper_t helper(
      new so_4::rt::dyn_agent_coop_t(
        new a_router_t( ip ) ) );
  }

void
destroy_coop()
  {
    so_4::api::deregister_coop( router_agent_name );
  }

int
```

```
main( int argc, char ** argv )
  {
    if( 2 == argc )
      {
        sobj_thread_t thread;
        thread.start();

        // Засыпаем, чтобы дать стартовать SObjectizer.
        // Это самый простой способ синхронизации с sobj_thread_t.
        threads_1::sleep_thread( 1000 );

        bool is_continue = true;
        while( is_continue )
          {
            std::string choice;

            std::cout << "Choose action:\n"
              "\t0 - quit\n"
              "\t1 - create router coop\n"
              "\t2 - destroy router coop\n> "
              << std::flush;

            std::cin >> choice;

            if( choice == "0" )
              {
                // Завершаем работу.
                so_4::api::send_msg(
                  so_4::rt::sobjectizer_agent_name(),
                  "msg_normal_shutdown", 0,
                  so_4::rt::sobjectizer_agent_name() );
                is_continue = false;
              }
            else if( choice == "1" )
              // Создаем кооперацию клиента.
              create_coop( argv[ 1 ] );
            else if( choice == "2" )
              // Уничтожаем кооперацию клиента.
              destroy_coop();
          }

        // Ожидаем завершения SObjectizer.
        thread.wait();

        return 0;
      }
    else
      {
```

```
        std::cerr << "sample_parent_insend <server_sock_addr>"
          << std::endl;

        return -1;
      }
  }
```

A.13 Исходный код примера dyn_coop_controlled

A.13.1 Файл main.cpp

```
/*
  Демонстрация работы с объектами, чье время жизни контролируется
  динамической кооперацией.
*/

#include <iostream>

#include <so_4/api/h/api.hpp>
#include <so_4/rt/h/rt.hpp>

#include <so_4/timer_thread/simple/h/pub.hpp>
#include <so_4/disp/one_thread/h/pub.hpp>

// Класс объекта, чье время жизни контролируется динамической кооперацией.
// Он специально не производен от so_4::rt::dyn_coop_controlled_obj_t,
// чтобы продемострировать работу so_4::rt::dyn_coop_controlled().
class my_obj_t
  : private cpp_util_2::nocopy_t
  {
  private :
    std::string m_value;

  public :
    my_obj_t(
      const std::string & value )
      : m_value( value )
      {}
    ~my_obj_t()
      {
        // Печать в деструкторе покажет, в какое время
        // будет уничтожен объект.
        std::cout << "~my_obj_t: " << m_value <<std::endl;
      }

    const std::string &
    get() const
      {
```

```
        return m_value;
      }
  };

// Класс тестового агента.
// Данный агент, совместно с другими объектами, использует объекты
// типа my_obj_t, но не контролирует время их жизни.
class a_my_t
  : public so_4::rt::agent_t
  {
    typedef so_4::rt::agent_t base_type_t;
  private :
    // Объекты, которые контролируются динамической кооперацией,
    // но совместно используются агентами кооперации.
    const my_obj_t & m_a;
    const my_obj_t & m_b;

  public :
    a_my_t(
      const std::string & self_name,
      const my_obj_t & a,
      const my_obj_t & b )
      : base_type_t( self_name )
      , m_a( a )
      , m_b( b )
      {}
    virtual ~a_my_t()
      {
        // Печать в деструкторе покажет, в какое время будет
        // уничтожен агент.
        std::cout << "~a_my_t: " << so_query_name() << std::endl;
      }

    virtual const char *
    so_query_type() const;

    virtual void
    so_on_subscription()
      {
        so_subscribe( "evt_start",
          so_4::rt::sobjectizer_agent_name(), "msg_start" );
      }

    // Реакция на начало работы агента -- печать содержимого.
    void
    evt_start()
      {
        std::cout << so_query_name() << ":\n\t"
```

```
            << m_a.get() << "\n\t"
            << m_b.get() << std::endl;
      }
  };

SOL4_CLASS_START( a_my_t )

  SOL4_EVENT( evt_start )

  SOL4_STATE_START( st_normal )
    SOL4_STATE_EVENT( evt_start )
  SOL4_STATE_FINISH()

SOL4_CLASS_FINISH()

// Класс агента, который завершит работу теста.
class a_shutdowner_t
  : public so_4::rt::agent_t
  {
    typedef so_4::rt::agent_t base_type_t;
  public :
    a_shutdowner_t()
      : base_type_t( "a_shutdowner" )
      {}
    virtual ~a_shutdowner_t()
      {}

    virtual const char *
    so_query_type() const;

    virtual void
    so_on_subscription()
      {
        so_subscribe( "evt_start",
          so_4::rt::sobjectizer_agent_name(), "msg_start" );
      }

    void
    evt_start()
      {
        so_4::api::send_msg( so_4::rt::sobjectizer_agent_name(),
          "msg_normal_shutdown", 0 );
      }
  };

SOL4_CLASS_START( a_shutdowner_t )

  SOL4_EVENT( evt_start )
```

```
  SOL4_STATE_START( st_normal )
    SOL4_STATE_EVENT( evt_start )
  SOL4_STATE_FINISH()

SOL4_CLASS_FINISH()

int
main()
  {
    // Создаем кооперацию из 3 агентов и 3 вспомогательных объектов.
    my_obj_t * a = new my_obj_t( "a" );
    my_obj_t * b = new my_obj_t( "b" );
    my_obj_t * c = new my_obj_t( "c" );

    a_my_t * a_1 = new a_my_t( "a_1", *a, *b );
    a_my_t * a_2 = new a_my_t( "a_2", *b, *c );
    a_shutdowner_t * a_shutdowner = new a_shutdowner_t();

    so_4::rt::agent_t * agents[] = { a_1, a_2, a_shutdowner };
    so_4::rt::dyn_agent_coop_t * coop = new so_4::rt::dyn_agent_coop_t(
      "sample", agents, sizeof( agents ) / sizeof( agents[ 0 ] ) );
    // Заставляем динамическую кооперацию контролировать
    // вспомогательные объекты.
    so_4::rt::dyn_coop_controlled( *coop, a );
    so_4::rt::dyn_coop_controlled( *coop, b );
    so_4::rt::dyn_coop_controlled( *coop, c );

    so_4::ret_code_t rc = so_4::api::start(
      // Диспетчер будет уничтожен при выходе из start().
      so_4::disp::one_thread::create_disp(
        // Таймер будет уничтожен диспетчером.
        so_4::timer_thread::simple::create_timer_thread(),
        so_4::auto_destroy_timer ),
      so_4::auto_destroy_disp,
      coop );
    if( rc )
      {
        std::cerr << "start: " << rc << std::endl;
      }

    return int( rc );
  }
```

A.14 Исходный код примера destroyable_traits

A.14.1 Файл main.cpp

```
/*
  Демонстрация работы со свойствами агентов, которые уничтожаются
  в деструкторе агентов.
*/

#include <iostream>

#include <so_4/api/h/api.hpp>
#include <so_4/rt/h/rt.hpp>

#include <so_4/timer_thread/simple/h/pub.hpp>
#include <so_4/disp/one_thread/h/pub.hpp>

// Свойство агента, созданное для данного примера.
class my_traits_t
  : public so_4::rt::agent_traits_t
  {
  private :
    // Имя агента, которому принадлежит свойство.
    std::string m_agent;

  public :
    my_traits_t( const std::string & agent )
      : m_agent( agent )
      {}
    ~my_traits_t()
      {
        // Печать в деструкторе покажет, в какое время будет
        // уничтожен объект-свойство.
        std::cout << "~my_traits_t: " << m_agent << std::endl;
      }

    // Реализация основных методов свойства.
    virtual void
    init( so_4::rt::agent_t & agent )
      {
        std::cout << "my_traits_t::init(): " << m_agent << std::endl;
      }

    virtual void
    deinit( so_4::rt::agent_t & agent )
      {
        std::cout << "my_traits_t::deinit(): " << m_agent << std::endl;
      }
```

```
  };

// Класс тестового агента.
// Создает в своем конструкторе объект-свойство.
class a_my_t
  : public so_4::rt::agent_t
  {
    typedef so_4::rt::agent_t base_type_t;
  public :
    a_my_t(
      const std::string & self_name )
      : base_type_t( self_name )
      {
        // После создание свойство живет своей жизнью, никто
        // про него не знает и не может получить указатель на
        // свойство. Но свойство будет уничтожено вместе с агентом.
        so_add_destroyable_traits( new my_traits_t( self_name ) );
      }
    virtual ~a_my_t()
      {
        // Печать в деструкторе покажет, в какое время будет
        // уничтожен агент.
        std::cout << "~a_my_t: " << so_query_name() << std::endl;
      }

    virtual const char *
    so_query_type() const;

    virtual void
    so_on_subscription()
      {
        // Эта печать покажет, когда был вызван метод.
        std::cout << "a_my_t::so_on_subscription(): "
          << so_query_name() << std::endl;

        so_subscribe( "evt_start",
          so_4::rt::sobjectizer_agent_name(), "msg_start" );
      }

    virtual void
    so_on_deregistration()
      {
        // Эта печать покажет, когда был вызван метод.
        std::cout << "a_my_t::so_on_deregistration(): "
          << so_query_name() << std::endl;
      }

    // Реакция на начало работы агента -- печать содержимого.
```

```
    void
    evt_start()
      {
        std::cout << so_query_name() << std::endl;
      }
  };

SOL4_CLASS_START( a_my_t )

  SOL4_EVENT( evt_start )

  SOL4_STATE_START( st_normal )
    SOL4_STATE_EVENT( evt_start )
  SOL4_STATE_FINISH()

SOL4_CLASS_FINISH()

// Класс агента, который завершит работу теста.
class a_shutdowner_t
  : public so_4::rt::agent_t
  {
    typedef so_4::rt::agent_t base_type_t;
  public :
    a_shutdowner_t()
      : base_type_t( "a_shutdowner" )
      {}
    virtual ~a_shutdowner_t()
      {}

    virtual const char *
    so_query_type() const;

    virtual void
    so_on_subscription()
      {
        so_subscribe( "evt_start",
          so_4::rt::sobjectizer_agent_name(), "msg_start" );
      }

    void
    evt_start()
      {
        so_4::api::send_msg( so_4::rt::sobjectizer_agent_name(),
          "msg_normal_shutdown", 0 );
      }
  };

SOL4_CLASS_START( a_shutdowner_t )
```

```
  SOL4_EVENT( evt_start )

  SOL4_STATE_START( st_normal )
    SOL4_STATE_EVENT( evt_start )
  SOL4_STATE_FINISH()

SOL4_CLASS_FINISH()

int
main()
  {
    // Создаем кооперацию из 3 агентов.
    a_my_t * a_1 = new a_my_t( "a_1" );
    a_my_t * a_2 = new a_my_t( "a_2" );
    a_shutdowner_t * a_shutdowner = new a_shutdowner_t();

    so_4::rt::agent_t * agents[] = { a_1, a_2, a_shutdowner };
    so_4::rt::dyn_agent_coop_t * coop = new so_4::rt::dyn_agent_coop_t(
      "sample", agents, sizeof( agents ) / sizeof( agents[ 0 ] ) );

    so_4::ret_code_t rc = so_4::api::start(
      // Диспетчер будет уничтожен при выходе из start().
      so_4::disp::one_thread::create_disp(
        // Таймер будет уничтожен диспетчером.
        so_4::timer_thread::simple::create_timer_thread(),
        so_4::auto_destroy_timer ),
      so_4::auto_destroy_disp,
      coop );
    if( rc )
      {
        std::cerr << "start: " << rc << std::endl;
      }
    else
      std::cout << "successful finish" << std::endl;

    return int( rc );
  }
```