# CActive и Все-Все-Все


## Бен Моррис

**Перевод с английского:**

**Александр Труфанов, Денис Григоренко**

**Рецензент: Владислав Шинкаренко**



## Публикуется Symbian Developer Network

**Версия: v1.9 – Ноябрь 2007**

# 1 Введение

Это второй документ из серии, посвященной идиомам Symbian C++, на которые так любят жаловаться разработчики. В этом месяце я рассматриваю активные объекты - конструкции, часто встречающиеся в Symbian OS. Если вы программировали в Symbian OS, то вы уже познакомились с ними, даже если не заметили этого. Скорее всего - заметили. Техника асинхронного программирования может сбить с толку разработчиков, воспитанных в более простом мире программ с последовательным, синхронным исполнением команд. А для тех, кто уже сталкивался с асинхронным программированием, объектно-ориентированный стиль активных объектов, возможно, покажется отличным от того, к чему они привыкли.

Активные объекты хорошо документированы в SDK, на многих сайтах, посвященных программированию в Symbian OS, и в книгах (ссылки на некоторые приводятся в конце этого документа). Но, тем не менее, популярность этой темы в Базе знаний и обсуждения на форумах подтверждают, что для новичков они кажутся сложными и не так доходчивы, как того заслуживают.

С помощью этого документа мы постараемся объяснить, почему активные объекты используются в Symbian OS, и почему они предпочтительнее многопоточности. Мы продемонстрируем вам некоторые типовые приемы, чтобы вы поняли, как использовать их на практике. И в завершение статьи мы поместили перечень того, что «можно делать» и «не следует делать» с активными объектами.

# 2 И снова основы: События, асинхронность, многозадачность и многопоточность

Активные объекты существуют для того, чтобы справиться с одной серьезной проблемой. Создание интерактивных приложений требует новых подходов в проектировании программ. В отличие от классического одноцелевого стиля программирования «программа-фильтр», поведение интерактивного приложения больше похоже на диалог с пользователем: последовательность выполнения команд в приложении обусловлена его откликами на действия пользователя. Этот «событийно-ориентированный» стиль программирования, стал популярен благодаря Mac OS и, чуть позже, Microsoft Windows. (Хотя, если поразмышлять - он используется с незапамятных времен. Любой конечный автомат является событийно-ориентированным, а конечные автоматы известны очень давно. Самый яркий пример - абстрактная машина Тьюринга).

Типичное событийно-ориентированное приложение тратит большую часть времени на ожидание пользовательских запросов, но не это является его характерной чертой; главное – то, как в нем организована управляющая логика. Она не начинается с `main()`, и не завершается после выполнения предопределённой последовательности операций. Событийно-ориентированное приложение принципиально (и, возможно, без преувеличений) просто большой оператор `switch`, обрабатывающий передающиеся ему системные события.

В результате такой стиль программирования по-сути является асинхронным. Для того чтобы программа была быстрой и хорошо управляемой, действия пользователя и косвенно вызванные ими события передаются полностью асинхронным методам-обработчикам, выполняющимся вне основной последовательности команд. Сам же метод распределения событий синхронный (или, во всяком случае, ведет себя таким образом) — он возвращает управление вызвавшему его методу сразу же после завершения, так как должен быть всегда готовым отреагировать на новые действия пользователя. Затем, когда асинхронный метод-

обработчик завершается, его результаты передаются в обработчик завершения, генерирующий завершающее событие. Все события, будь они результатом синхронных или асинхронных методов, и вне зависимости от того, вызваны они действиями пользователя или системой, оказываются в одной очереди событий, передаваемых в систему.

Асинхронное программирование существует уже долгое время, в частности, как решение проблемы потенциально высокой задержки отклика периферийных устройств. Периферийные устройства обычно являются интерфейсами для связи с «реальным» миром (включая нас с вами) и поэтому, их поведение достаточно непредсказуемо: принтеры могут отключаться, линии коммуникаций разрываться, телефонная трубка может плохо лежать, пользователь — отойти от компьютера. Асинхронное программирование позволяет событиям с более высокой допустимой задержкой обрабатываться вне основного управляющего потока запущенной программы. Благодаря этому, остальная часть системы может продолжать обрабатывать прочие события. Это применяется в любой системе, поддерживающей параллелизм (параллельное выполнение множества процессов, или эмуляция этого путём квантования процессорного времени).

Ранние системы GUI, такие как первые ОС Mac и Windows, реализовывали свои событийно-ориентированные модели программирования, используя невытесняющее планирование процессов, известное под термином кооперативная многозадачность. Это сильно отличалось от Unix-систем и мейнфреймов, использовавших вытесняющее планирование процессов. Разница в том, что в вытесняющей системе процесс, выполняющийся в данный момент, может быть в любое время остановлен планировщиком операционной системы. В невытесняющей системе планировщик выбирает процесс, который будет выполняться следующим, и всегда даёт ему возможность выполняться до конца. Реализация вытесняющего планирования приводит к дополнительным расходам ресурсов и, в целом, усложняет систему. Но невытесняющее планирование имеет серьезный недостаток: плохо спроектированные приложения могут препятствовать многозадачности, не уступая процессорное время другим процессам.

Эволюция, приведшая к многопоточным системам, может рассматриваться как попытка сохранить преимущества параллелизма, включая естественную асинхронность и событийно-ориентированный подход к программированию, и преимущества вытесняющих систем, без потери производительности обусловленной параллелелизмом процессов.

# 3 Сложность многопоточности

Главная проблема многопоточности в том, что она сложна для разработчиков приложений. В то время как многопоточное программирование распространилось повсеместно, на практике преобладание этого стиля произошло сравнительно недавно. Причин популярности многопоточного программирования три: Web сервера, Linux, и новые поколения многоядерных процессоров. Со временем, многопоточное программирование из загадочного искусства, доступного немногим посвященным (системным программистам в области коммуникаций) превратилось в стандартную технику программирования приложений.

Есть два источника сложности многопоточности:

- Синхронизация: контроль правильности порядка наступления событий. Это необходимо, т.к. порядок, в котором потоки завершат выполнение, заранее не известен.

- Разделяемые данные и проблема атомарности: обеспечение атомарности чтения и записи данных. Операции чтения и записи общих данных могут быть прерваны на полпути, при вытеснении потока другим потоком, выполняющим запись этих данных.

Многопоточная система должна предоставлять программисту системные примитивы для обеспечения взаимодействия потоков. Например, семафоры, блокировки, мьютексы и др. Программисты вынуждены сталкиваться с этой дополнительной сложностью, что в свою очередь может привести к появлению целого ряда новых ошибок: от забывания заблокировать критические секции (возникает опасность изменения потоком состояния программы, от которого зависит другой поток), до взаимных блокировок и состояний гонки.

Внезапно, разработчики столкнулись с необходимостью беспокоиться о вещах, которые раньше были само собой разумеющимися, таких как целостность и устойчивость контекста во время исполнения. Им пришлось заботиться об этом самим на уровне приложения. Хуже того, они вынуждены были учиться справляться с потенциально сложным взаимодействием потоков в рамках их программ, хотя раньше таких проблем просто не существовало, так как приложения были однопоточными.

Эта дополнительная сложность является основным аргументом против многопоточности. Споры об этом ведутся уже давно, и еще долго будут вестись. В своей знаменитой презентации на конференции Usenix Джон Остерaут, создатель языка программирования Tcl, заявил, что многопоточность - слишком трудна для большинства программистов и подытожил ее недостатки: сложна для отладки, нарушает абстракцию, и может ухудшить производительность, вместо ее увеличения. Даже его оппоненты согласны, что многопоточность достаточно сложна.

# 4 Новый подход в Symbian OS: Активные объекты

Symbian OS – это многопоточная операционная система. В частности, потоки в ней являются исполнимыми объектами, управляемыми, планируемыми, запускаемыми и переключаемыми посредством ядра ОС. Это полностью вытесняющая модель управления потоками. Symbian также является операционной системой реального времени (начиная с девятой версии), и время ожидания потоков в ней гарантированно ограничено (что спроектировано специально для обеспечения тайминга сигнального стека мобильного телефона). Но, в то время как другие операционные системы всё больше и больше адаптируют многопоточность для использования на прикладном уровне, активные объекты Symbian OS предлагают альтернативную и более простую модель для управления асинхронным поведением приложения в событийно-ориентированной системе. Активные объекты показали себя настолько эффективными на практике, что на большинстве уровней над ядром, включая системные службы, они стали стандартным шаблоном для реализации любого асинхронного поведения.

В результате, при программировании для Symbian OS практически отпала необходимость в семафорах, блокировках и прочих примитивах потока. Это значительно упростило разработку, и снизило вероятность возникновения потенциальных ошибок.

Целевой рынок устройств для Symbian OS, по сути, является потребительским, и надежность — главный фактор на этом рынке. Поэтому, отсутствие ошибок при программировании – прямой путь к успеху. Есть еще одна важная причина создания активных объектов — это экономия процессорного времени и энергии. Symbian OS рассчитана на устройства, работающие от батарей, и обладающих процессорами с достаточно скромной производительностью. Операционные системы персональных компьютеров, даже после миграции на мобильные устройства, хуже приспособлены к работе с подобным аппаратным обеспечением.

Переключение между активными объектами, работающими в одном потоке (это наиболее распространенная ситуация), позволяет избежать затрат на переключение контекста потоков. Поэтому, приложение, реализующее асинхронность с помощью активных объектов обладает большей производительностью, чем приложение, использующее несколько потоков.

С точки зрения проектирования, Symbian OS изначально создавалась как объектно-ориентированная система, практически полностью написанная на C++. Поэтому, объектно-ориентированная реализация асинхронного программирования была естественным решением.

Активные объекты - прекрасный пример того, как объектно-ориентированное проектирование используется для инкапсулирования и скрытия сложности примененных алгоритмов, а также для абстракции концепций ядра операционной системы. Активный объект инкапсулирует механизмы для отправки запроса асинхронной службе и обработки результата его выполнения. При этом, несколько таких объектов могут находиться в одном и том же потоке, не блокируя его на период ожидания исполнения запроса и (с точки зрения разработчика) без использования многопоточности. Активные объекты ограждают разработчиков приложений от трудностей работы с многопоточностью, пряча эту сложность во внутренних механизмах ОС.

Активные объекты позволяют разработчиками не задумываться о тонкостях обработки асинхронных событий. Их использование тоже не так просто, но в большинстве приложений система берет на себя управление активными объектами. Главный поток каждого приложения автоматически получает свой планировщик активных объектов и специальный поток для них. Все активные объекты, запущенные из главного потока, выполняются в специальном потоке активных объектов без вытеснения, в порядке приоритета. Если приложение имеет всего один поток, тогда все его активные объекты работают совместно. В контексте системы, их выполнение может приостанавливаться, чтобы дать процессорное время активным объектам других приложений или системным потокам. Активные объекты предоставляют мощный и простой шаблон для проектирования приложений в соответствии с событийно-ориентированной интерактивной моделью. Это позволяет сохранить и простоту однопоточности, и реактивность событийно-ориентированной модели, взяв за основу удобный объектно-ориентированный подход.

Основной задачей, решаемой с помощью концепции активных объектов, является упрощение асинхронного событийно-ориентированного программирования. Устранение этих сложностей исключило причину многих потенциальных ошибок программирования, тем самым, улучшив устойчивость и надёжность программ.

Если вам все же придется работать с многопоточностью напрямую, то все необходимые для этого механизмы имеются в API. В дополнение к собственному API, Symbian OS 9-й версии обладает улучшенной совместимостью с POSIX (в виде библиотек P.I.P.S., или POSIX для Symbian), тем самым упрощая перенос на Symbian с других платформ уже имеющийся код, использующий многопоточность. Более того, P.I.P.S. делает доступным создание приложений для разработчиков, не знающих Symbian C++. В результате, большая часть многопоточного кода может быть портирована на эту платформу. Но это не является наиболее эффективным подходом, и многопоточность лучше использовать лишь в крайнем случае, в силу причин, которые мы ранее обсуждали.

# 5 Основы активных объектов

Базовые активные объекты определены в файле e32base.h, который вы можете найти в SDK. Существует два главных базовых класса: это CActive, описывающий абстрактный базовый класс, от которого наследуются все активные объекты; и CActiveScheduler – класс, определяющий и реализующий планировщик активных объектов. (Обычно приложения обращаются к методам CActiveScheduler, вместо того, чтобы создавать наследуемый от него объект).

## *5.1 CActive*

CActive порожден от CBase - стандартного базового класса Symbian OS для создаваемых в куче объектов. Каждый активный объект наследуется от CActive.

- CActive объявляет чистые виртуальные методы RunL() и DoCancel(), которые обязаны реализовывать порожденные классы.
- CActive объявляет стандартный метод RunError(), который вызывается системой, в случае, если в RunL() происходит сброс. Для того чтобы обрабатывать ошибки самостоятельно, разработчик должен перекрыть метод RunError() в своем классе. Возвращаемое RunError() значение передается планировщику активных объектов.
- Все активные объекты имеют приоритет, используемый планировщиком для управления ими. В CActive определены значения, которые может принимать приоритет: EPriorityIdle (-100) – используется в объектах, предназначенных для выполнения в фоновом режиме; EPriorityLow (-20) и EPriorityStandard (0) – подходят для большинства активных объектов; EPriorityUserInput (10) – предназначен для объектов, принимающих вводимые пользователем данные; EPriorityHigh (20) - должно использоваться только для высокоприоритетных активных объектов, например, таймеров, для того, чтобы они получили преимущество, когда планировщик активных объектов будет выбирать следующий объект для исполнения.
- CActive содержит член TRequestStatus iStatus, в который система помещает код завершения, когда асинхронный запрос активного объекта исполняется.
- CActive реализует конкретные методы SetActive(), Cancel(), и SetPriority().

## *5.2 CActiveScheduler*

CActiveScheduler порожден от CBase - стандартного базового класса Symbian OS для создаваемых в куче объектов. В обычном GUI приложении планировщик активных объектов создается системой автоматически. Если вам придется создать планировщик явно, то вы можете воспользоваться экземпляром класса CActiveScheduler без наследования от него.

- CActiveScheduler реализует ряд конкретных методов. В частности, метод Install(), используемый для установки планировщика, если система еще не сделала это автоматически. Методы Start() и Stop() используются для запуска и остановки планировщика. Метод Add() используется для регистрации и активации объекта планировщиком.
- CActiveScheduler предоставляет реализацию метода Error(), вызываемого системой в случае необработанного сброса в методе RunL() активного объекта.

- `CActiveScheduler` хранит ассоциации между состоянием запроса в активном объекте `TRequestStatus iStatus` и методом `RunL()` этого объекта.
- `CActiveScheduler` реализует механизмы планирования, выбирающие и запускающие следующий готовый к работе активный объект текущего потока с наибольшим приоритетом.

## 5.3 Таймеры и другие, готовые к использованию активные объекты

Кроме двух базовых классов, в системе также определено несколько полезных, уже готовых к использованию активных объектов. Например:

- конкретный класс `CIdle` – готовый низкоприоритетный активный объект, который может непосредственно использоваться для выполнения задач в фоновом режиме. Для этого при создании ему передается метод обратной связи (callback), в котором разработчик реализует необходимые действия.
- абстрактный класс `CAsyncOneShot`. Наследовав от него ваш класс, вы получите готовый к использованию низкоприоритетный активный объект, вызывающий метод обратной связи лишь один раз.
- различные таймеры, порожденные от `CActive` для дальнейшего наследования или прямого использования: `CTimer`, `CDeltaTimer`, `CHeartBeat`, `CPeriodic`.

При программировании для Symbian OS, класс `CTimer`, наряду с порожденными от него таймерами, решает практически все связанные с отсчетом времени и периодичностью задачи.

## 5.4 Активные объекты и потоки

Активные объекты используются в Symbian OS для того, чтобы, обеспечив асинхронное поведение, избежать сложностей многопоточного программирования. Однако ничто не мешает программисту смешивать активные объекты и многопоточность, в тех случаях, когда это действительно необходимо.

Важно помнить, что активные объекты и планировщики активных объектов являются локальными по отношению к потоку, в котором они были созданы и установлены. Каждый поток, в котором создаются активные объекты, должен иметь установленный планировщик активных объектов. Все созданные в потоке активные объекты управляются планировщиком активных объектов, установленным в этом потоке. Попытка вызвать активный объект, созданный и зарегистрированный в одном потоке, из другого потока, приведет к ошибке.

В частности, для GUI приложений система автоматически создает один планировщик активных объектов и устанавливает его в главном потоке. Если приложение запускает дополнительные потоки, то вы сами должны позаботиться о создании и установке планировщика активных объектов в каждом из них.

## 5.5 Активные объекты и приоритетные прерывания

Потоки в Symbian OS являются вытесняемыми. Так как планировщики активных объектов принадлежат потокам, то выполнение активных объектов одного потока может быть приостановлено для выполнения активных объектов другого потока, и наоборот.

Однако сам по себе планировщик активных объектов не реализует механизмов вытеснения на основе приоритетов. Поэтому, в рамках одного потока плохо спроектированный активный объект может заблокировать работу всех остальных активных объектов, что необходимо иметь в виду при разработке.

# 6 Активные объекты на практике

Этот документ не является всеобъемлющим руководством по использованию активных объектов (существует множество других материалов, ссылки перечислены в конце документа). Цель данного раздела - продемонстрировать достаточное количество примеров, чтобы дать вам представление о том, как используются активные объекты на практике, не углубляясь в детали.

Активный объект инкапсулирует методы создания запроса и обработки результатов его выполнения. Кроме того, он может обработать прерывание запроса. Благодаря этой инкапсуляции, активные объекты предоставляют полностью самодостаточный механизм для обработки асинхронных событий.

1. Создание запроса:

   ```
   iMyRequestor -> Cancel();
   iMyRequestor -> MakeAsynchRequest();
   ```

   Здесь `iMyRequestor` - это активный объект типа `CMyRequestor`, цель которого выполнять асинхронные вызовы системных служб. В этом примере запрос к службе инкапсулирован в методе `MakeAsynchRequest()`, который определяет и реализует класс `CMyRequestor`. В качестве службы может выступать любая из множества системных служб Symbian OS, предоставляющих асинхронное API для клиентов - в данном примере выполняется запрос к файловому серверу `RFs::Notify()`.

   Перед созданием системного запроса вызывается метод `Cancel()`, определенный в базовых классах, для того чтобы прервать имеющиеся невыполненные запросы. См. далее для более подробной информации.

2. Обработка результатов запроса:

   Обработка результатов выполнения запроса производится в `CMyRequestor::RunL()`. В базовых классах `RunL()` определен как чистый виртуальный метод. Именно в этом методе вы должны реагировать на результаты выполнения вашего асинхронного запроса. Система вызовет `RunL()` как только служба завершит выполнение вашего запроса и будет готова принять новый.

   При реализации `RunL()` необходимо помнить о двух важных правилах:

   - `RunL()` должен быть кратким и быстрым; метод должен оперативно завершить свои действия и вернуть управление, поскольку другие активные объекты, управляемые тем же планировщиком, не смогут продолжить свое выполнение до тех пор, пока текущий `RunL()` не завершится.
   - Если в результате выполнения `RunL()` может возникнуть сброс (leave), то ваш активный объект должен перекрыть метод `RunError()`, определенный в базовых классах. Вызов метода `RunL()` осуществляется в рамках ловушки (TRAP), которая при возникновении сброса вызоет метод `RunError()`. Система содержит реализацию метода `RunError()` по-умолчанию, генерирующую панику (panic). Конечный пользователь не должен сталкиваться с таким поведением вашей программы, поэтому вы должны переопределить этот метод, выполнить необходимые действия и вернуть `KErrNone`.

Обработать результаты выполнения предыдущего асинхронного запроса и инициализировать новый - это распространенная схема, использующийся при реализации RunL(). Иными словами в RunL() выполняется шаг №1 для создания повторяющихся, периодических асинхронных служб.

3. Отмена запроса:

   Вы можете отменить выполнение запроса с помощью вызова CActive::Cancel(). Если есть невыполненный запрос, который ожидается активным объектом, то вызов CActive::Cancel() приведет к вызову метода DoCancel() этого активного объекта (данный метод объявлен в предке как чистый виртуальный). Ваша собственная реализация DoCancel() должна отменить запрос с помощью методов асинхронной службы, с которой взаимодействует ваш класс.

Итоговое объявление класса CMyRequestor может выглядеть следующим образом:

```
class CMyRequestor : public CActive
     {
public:
     // Стандартный деструктор C++, двухфазный конструктор Symbian OS
     ~CMyRequestor();
     static CMyRequestor* NewL();
     // Асинхронный запрос
     void MakeAsynchRequest();
protected:
     CMyRequestor(); // C++ конструктор по умолчанию
     void ConstructL(); // конструктор второй фазы, может привести к сбросу
protected:
     // Методы CActive
     virtual void RunL();
     virtual void DoCancel();
     virtual TInt RunError(TInt aError);
private:
     // Далее объявление специфичных элементов, необходимых для вашего класса
     RFs iFs;   // Идентификатор сессии файлового сервера, используется для
                // создания асинхронных запроса
     HBufC* iPath;   // Аргумент, который передается в запросе
     };
```

При реализации активного объекта необходимо перекрыть методы RunL() и DoCancel(), которые объявлены в предке как чистые виртуальные, а также метод RunError(), если в результате выполнения RunL() может возникнуть сброс.

Типичная реализация RunL() может выглядеть следующим образом:

```
void CMyRequestor::RunL()
      {
      // Если возникла ошибка - обработать ее в RunError()
      User::LeaveIfError(iStatus.Int());

      // Немедленно выполнить новый асинхронный запрос,
      // если это периодическая или непрерывная служба
```

```
iFs.NotifyChange(ENotifyAll, iStatus, *iPath);
SetActive(); // Отметить данный объект как активный

// Обработать результаты
...
}
```

Для создания объекта необходимо использовать механизм двухфазного конструирования - стандартный подход Symbian OS, который предполагает наличие публичной функции-фабрики `NewL()`, а также закрытые или защищенные C++ конструктор и метод `ConstructL()`.

Конструктор C++ должен выполнять следующее:

- устанавливать приоритет активного объекта, используя синтаксис списка инициализации
- регистрировать активный объект в планировщике, используя статический метод `Add()` класса `CActiveScheduler`.

Пример:

```
// C++ конструктор активного объекта
CMyRequestor::CMyRequestor()
     : CActive(CActive::EPriorityStandard)
     {
     CActiveScheduler::Add(this);
     }
```

И, конечно же, активный объект должен реализовывать запрос к поставщику асинхронного сервиса (async. service provider) - это и есть главная причина создания класса. В данном примере сервис, к которому выполняется запрос - это файловый сервер, который должен выполнить оповещение при изменении файла или каталога заданного в аргументе `*iPath`.

```
// Метод создает асинхронные запросы
CMyRequestor::MakeAsynchRequest()
     {
     // Только один запрос может ожидать выполнения в любой момент времени
     // Необходимо вызвать Cancel() перед вызовом этого метода

     // Проверка на существование запроса, ожидающего выполнения
     if (IsActive())
          {
          // Это программная ошибка, поэтому паника допустима
          _LIT(KMyRequestorPanic, "CMyRequestor");
          User::Panic(KMyRequestorPanic, KErrInUse);
          }

     // Иначе выполнить запрос
     iFs.NotifyChange(ENotifyAll, iStatus, *iPath);
     SetActive(); // Отметить данный объект как активный
     }
```

Отметим, что в приведенном коде:

- `CActive::IsActive()` – стандартный метод, проверяющий, существует ли невыполненный запрос, который ожидается данным объектом. Если результат положительный (такой запрос существует) - генерируется паника, что является приемлемым, так как это программная ошибка. Вы всегда должны выполнять такую проверку перед обращением к асинхронной службе и вызовом `SetActive()` в запросе. Возможна реализация данной проверки с использованием выражения `ASSERT`, в этом случае код может выглядеть следующим образом:

  ```
  _LIT(KMyRequestorPanic, "CMyRequestor");
  __ASSERT_ALWAYS(!IsActive(), User::Panic(KMyRequestorPanic, KErrInUse));
  ```

- `iStatus` - это защищенный объект класса `TRequestStatus`, который объявлен в классе `CActive`. Потомки `CActive` не должны инициализировать данный элемент класса, а только использовать. `iStatus` всегда передается в качестве параметра при обращении к асинхронной службе. Присутствие такого параметра в объявлении метода является признаком того, что метод асинхронный.

- `CActive::SetActive()` - это метод, отмечающий данный активный объект в списке зарегистрированных объектов планировщика как активный (ожидающий выполнения запроса). Вы *должны* вызвать этот метод сразу же после активации асинхронного запроса.

Данный пример не является завершенным, так как существует ограничение на объем демонстрационного кода. Третий том издания *Symbian OS C++ for Mobile Phones* (Harrison and Shackman, 2007) содержит несколько полных примеров готовых активных объектов, которые вы можете загрузить со страницы Symbian Press на сайте Symbian Developer Network. Глава 6 этого издания содержит полное и детальное описание активных объектов.

# 7 Контрольный список

В следующем перечне мы подытожим наиболее важные сведения об использовании активных объектов.

- Приложение получает планировщик активных объектов автоматически. Он предоставляется системой, уже установлен и управляет активными объектами главного потока. Но, если вы создаете новые потоки в приложении, то вы должны сами создать и установить по одному экземпляру планировщика активных объектов в каждом новом потоке, работающем с активными объектами. Вы также должны создать и установить планировщик активных объектов если вы создаете серверное или консольное приложение (например, тестовую программу) использующие активные объекты.

  Пример создания и установки экземпляра планировщика в потоке показан ниже. Более подробную информацию вы сможете найти в материалах, ссылки на которые указаны в конце статьи.

  ```
  CActiveScheduler* scheduler = new(ELeave) CActiveScheduler;
  CleanupStack::PushL(scheduler);
  CActiveScheduler::Install(scheduler);
  ```

  После установки, для запуска и остановки планировщика используются методы `CActiveScheduler::Start()` и `CActiveScheduler::Stop()`.

- Создавая активный объект, добавляйте его в список планировщика сразу же. Лучше всего делать это в конструкторе по умолчанию. Для добавления воспользуйтесь методом `CActiveScheduler::Add()`. Помните, что `Add()` - статический метод, поэтому вам не нужно указывать ссылку на экземпляр установленного планировщика.

- Класс активного объекта всегда имеет член `iStatus`, который должен быть передан по ссылке в качестве параметра асинхронного метода. Вам не нужно объявлять еще один член типа `TRequestStatus` - это часто встречающаяся ошибка, просто используйте тот, который определен в базовом классе `CActive`.

- Поставщик асинхронного сервиса автоматически присваивает `iStatus` значение `KRequestPending`. Это значение используется системой для идентификации активного объекта, ожидающего завершения асинхронного запроса. Вам не нужно делать этого самостоятельно.

- Перед выполнением асинхронного запроса проверяйте, завершился ли предыдущий запрос с помощью `CActive::IsActive()`. Эта проверка очень важна. Если вы выполните новый запрос, не дождавшись завершения предыдущего, то это, скорее всего, приведет к потерянному сигналу и системной панике.

- Всегда выполняйте `CActive::SetActive()` сразу же после вызова асинхронного запроса.

- Обрабатывайте завершение асинхронного запроса в методе `RunL()` вашего класса.

- Обрабатывайте ошибки в методе `RunError()`. Возвращайте `KErrNone` для ошибок, которые вы можете обработать, или их код, если не можете. Если вы не обрабатываете ошибки в активном объекте, то управление перейдет к методу `Error()` планировщика активных объектов, и его действия могут вас не удовлетворить (например, если он вызовет системную панику).

- Прерывайте выполнение активного объекта с помощью `CActive::Cancel()`. Это всегда следует делать перед его удалением. Если вы удалите активный объект, ожидавший завершения асинхронного вызова, то рано или поздно планировщик попытается вызвать уже не существующий метод `RunL()`. Поэтому, вы должны поместить `CActive::Cancel()` в деструкторе. Всегда вызывайте именно этот системный метод, а не метод `DoCancel()` вашего активного объекта.

- Параметр `TRequestStatus aStatus` в сигнатуре метода всегда указывает на то, что он выполняется асинхронно. Во многих API существуют пары эквивалентных синхронных и асинхронных методов. (обычно, синхронная версия просто вызывает асинхронную и ждет её завершения).

- Если вы смешиваете активные объекты и потоки, помните, что каждый активный объект закреплен за планировщиком, а тот в свою очередь является локальным по отношению к потоку. Попытка обратиться к активному объекту одного потока из другого потока приведет к ошибке. Все работа по вызову и обработке асинхронного запроса должна выполняться в одном потоке.

- Не создавайте активные объекты в стеке (как обычные переменные), т.к. в этом случае завершение выполнения асинхронного запроса, скорее всего, произойдет за областью видимости такого объекта, а значит, он будет уже удален.

- Важно выбрать подходящие приоритеты для ваших активных объектов, т. к. это влияет на порядок их выполнения. Для большинства случаев подойдет

EPriorityStandard (0), но для обработки пользовательского ввода, или для главного объекта сервера (если вы создаете сервер), используйте EPriorityUserInput (10). Для долго выполняющихся задач подойдет EPriorityIdle.

- Метод RunL() должен завершаться в разумные сроки, т.к. все активные объекты того же планировщика будут помещаться в очередь, и ожидать его завершения. Задача, требующая длительной обработки, должна быть разбита на подзадачи. К примеру, сохранение чего-либо можно выполнять поблочно, а не за одну операцию.
- Самая часто встречающаяся проблема при использовании активных объектов - паника, вызванная потерянным сигналом. Наиболее частые причины этого следующие:
    - не вызван CActiveScheduler::Add(), поэтому, ваш активный объект не зарегистрирован
    - не вызван SetActive() после отправки запроса поставщику сервиса. В результате - ваш запрос не зарегистрирован
    - запуск очередного запроса до завершения предыдущего.

# 8 Активные объекты в повседневной жизни

Активные объекты широко применяются в Symbian OS. К примеру, подсистема пользовательского интерфейса содержит два активных объекта: один - для обработки пользовательского ввода, второй - для обработки перерисовки содержимого окон. Они скрываются за многими стандартными методами приложений: HandleCommandL() (в классе CAppUi), Draw(), OfferKeyEventL(), HandlePointerEvent() и прочими в Control Environment (CONE). Большое количество активных объектов используется для выполнения задач в фоновом режиме.

Еще один пример - активные объекты используются в Сервере управления окнами (Window Server) для обработки повторных нажатий при удерживании клавиши телефона, и в коде обработки событий. Фактически, события Сервера управления окнами и есть активные объекты класса CEvent, наследованного от CActive.

Самым ярким примером того, насколько глубоко проникло использование активных объектов в Symbian OS, является реализация клиент-серверной архитектуры. Базовые классы для проектирования серверов порождены от активных объектов, и имеют (не публичные) методы RunL() и DoCancel().

Активные объекты в Symbian OS предоставляют элегантное решение проблем асинхронного программирования, управляемое на уровне приложения.

# 9 Заключение

Хочется надеяться, что я привел достаточно доводов необходимости использования активных объектов. К сожалению, это не делает их использование простым, но истинная причина этой сложности скрыта не в конструкции активных объектов, а в самой концепции асинхронного поведения приложения. Решения на базе активных объектов более надежны, и вероятность допустить ошибку при работе с ними гораздо меньше, чем при использовании альтернативных подходов. Именно поэтому, они широко используются в Symbian OS.

Некоторые могут перефразировать это немного жестче. Одно время в Symbian имелся внутренний документ, начинающийся со слов:

> *Вы не уверены в правильности использования активных объектов? Их преимущества для вас неочевидны? Тогда не читайте этот документ а займитесь чем-то другим.*

Активные объекты в большом почете у Symbian, и они проделали большой путь. В 1991 году они были впервые реализованы в 16-ти битной системе Psion SIBO, предшественнице Symbian OS. Когда началась работа над Symbian, активные объекты были уже испытаны, а их преимущества доказаны.

Принципы работы активных объектов были описаны в книге Pattern Languages of Program Design 2 (1996), а позднее, в известной статье Мартина Кэрролла из Bell Labs (1998).

# 10 Дополнительная литература и web ресурсы

## *10.1 Литература*

Любая книга Symbian Press, посвященная разработке на C++, содержит информацию об основах активных объектов. Я рекомендую начать со следующих трех изданий:

**Richard Harrison и Mark Shackman (2007)**

*Symbian OS C++ for Mobile Phones, Volume 3, Wiley/Symbian Press*

- Последнее издание ставшей классикой книги. Глава 6 посвящена активным объектам.

**Jo Stichbury и Mark Jacobs (2006)**

*The Accredited Symbian Developer Primer, Wiley/Symbian Press*

- Глава 9 содержит принципы событийно-ориентированного программирования, и объясняет использование активных объектов.

**Steve Babin (2007)**

*Developing Software for Symbian OS, Second Edition, Wiley/Symbian Press*

- Глава 8 подробно описывает активные объекты.

Некоторые сведения об истории создания Symbian OS, включая активные объекты, и об перспективах архитектуры вы можете почерпнуть из моей книги:

**Ben Morris (2007)**

The Symbian OS Architecture Sourcebook, Wiley/Symbian Press

- Описывает особенности проектирования и некоторый исторический контекст.

Прочие материалы:

**John Vlissides, Jim Coplien, and Norm Kerth (1996)**

*Pattern Languages of Program Design 2, Addison-Wesley Longman*

- Книга от идеологов технологии паттерного проектирования, ставшая классикой; содержит описание концепции активных объектов.

Статья Мартина Кэррола в Software Practice and Experience том 28, издание 1, стр. 1-21 (1998)

## *10.2 Web ресурсы*

Критика многопоточного программирования Джона Остераута (1996) *Why threads are a bad idea (for most purposes)*. Дискуссия на конференции Usenix. http://home.pacbell.net/ouster/threads.pdf.

Герб Суттер в статье *The Free Lunch is Over* приводит аргументы в поддержку параллелизма, не отрицая сложности многопоточности. http://www.gotw.ca/publications/concurrency-ddj.htm.

В сообществе Forum Nokia вы можете найти множество интересных материалов в Forum Nokia Wiki http://wiki.forum.nokia.com/index.php/Active_object; там также публикуется Symbian OS basics workbook.

Возможно, вам покажутся интересными две статьи из базы знаний Symbian Developer Network, дающие представление о типичных ошибках при использовании активных объектов (обе посвящены паникам View Server): FAQ-0900 и FAQ-0920.

И конечно же, Symbian Developer Network: developer.symbian.com.

# 11 Об Авторе

Бен Моррис является известным писателем и разработчиком программного обеспечения, специализирующимся на Symbian OS. Он - автор книги The Symbian OS Architecture Sourcebook: Design and Evolution of a Mobile Phone OS, изданной Symbian Press. Вы можете связаться с ним при помощи его web-сайта: www.benmorris.eu.