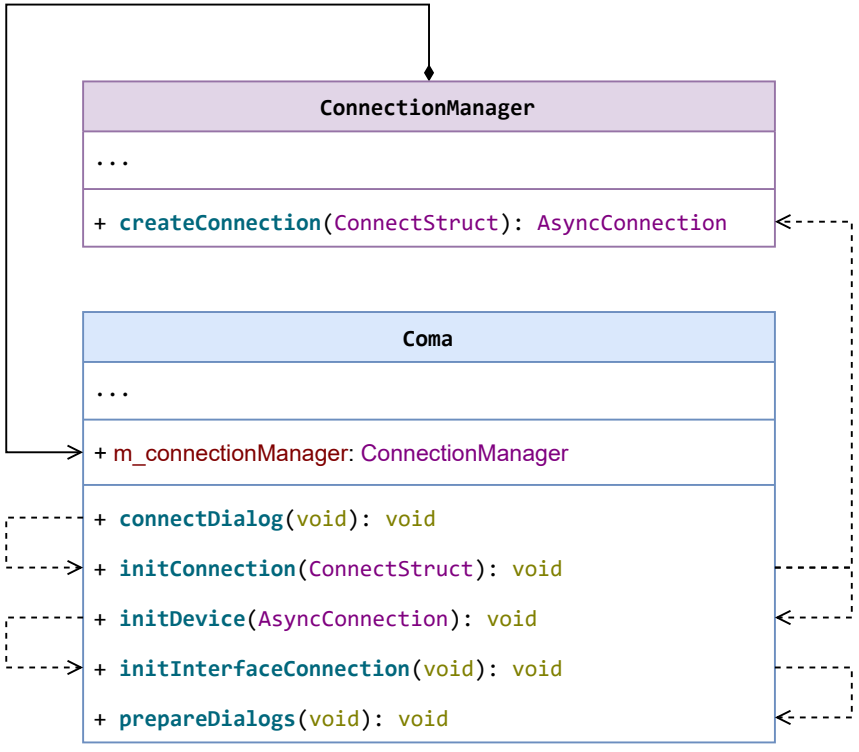
COMA v2.11.0 Core Architecture Guidelines

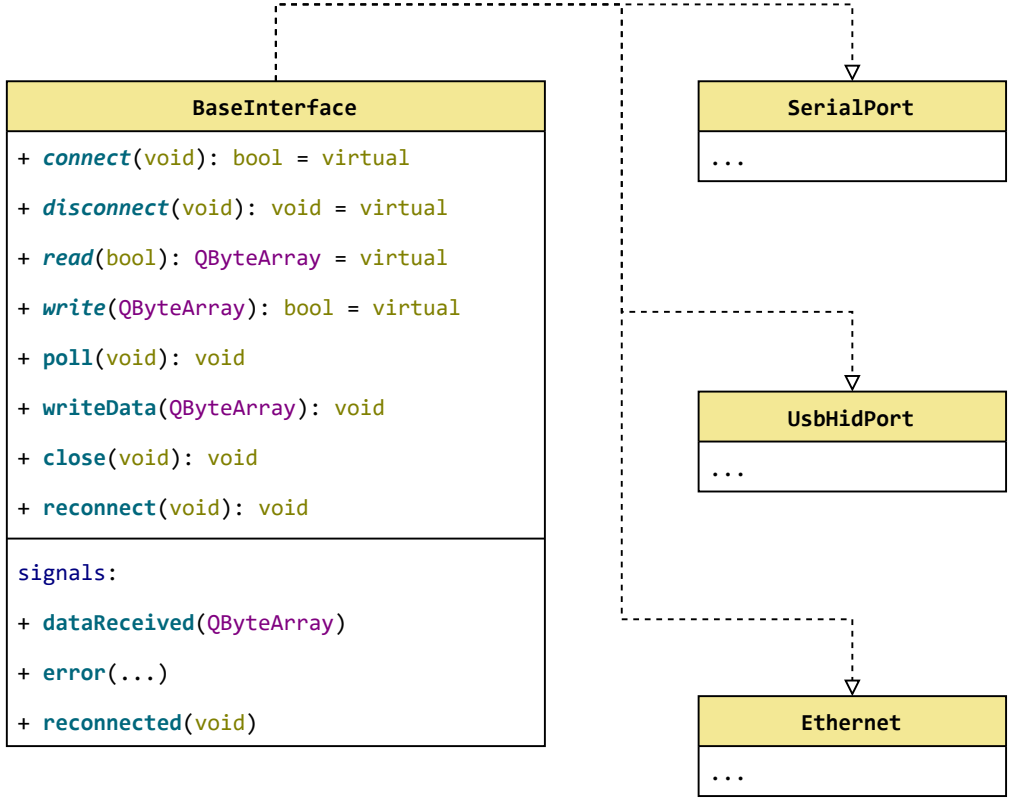
Соединение с устройством



Начнём с самого начала, а именно со связи с устройством. При нажатии на кнопку "Соединение" происходит следующее:

1. Сначала вызывается метод `Coma::connectDialog`, который создаёт диалог выбора соединения и его настроек.
2. Полученные настройки соединения передаются в метод `Coma::initConnection`, который передаёт их инстансу класса **`ConnectionManager`** (менеджеру соединений). Данный менеджер предназначен для создания по заданным настройкам активного соединения.
3. В случае успешного создания соединения вызывается метод `Coma::initDevice`, который создаёт инстанс класса **`CurrentDevice`**. Данный класс будет подробно рассмотрен далее.
4. Далее последовательно вызываются методы `Coma::initInterfaceConnection` и `Coma::prepareDialogs`: происходит парсинг и сохранение настроек для подключенного устройства, создаются диалоги, начинается основная работа приложения.

Интерфейсы

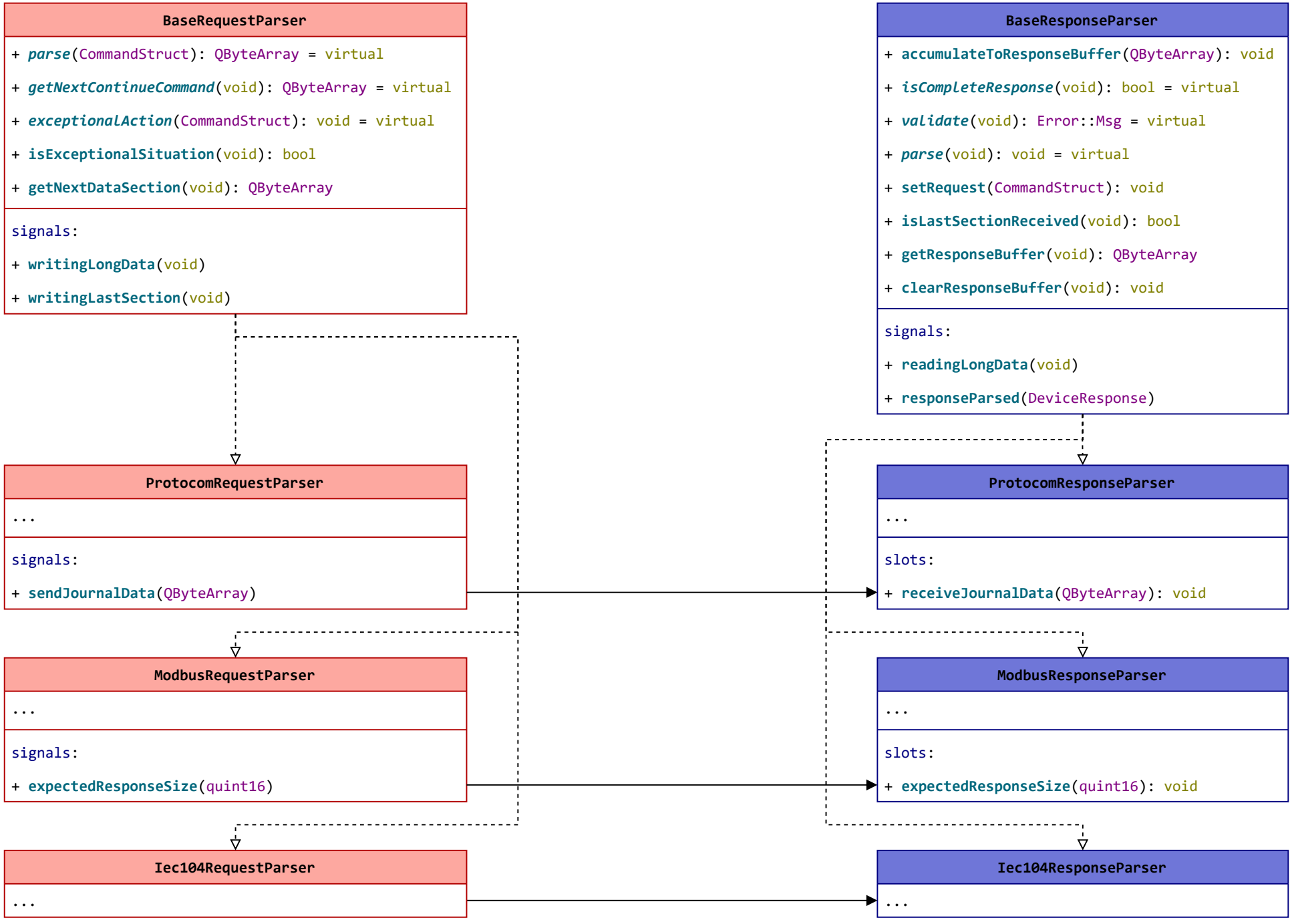


`BaseInterface` является базовым абстрактным классом для реализации интерфейсных классов. От него наследуются `SerialPort`, `Ethernet` и `UsbHidPort`. Поток исполнения находится постоянно в методе `BaseIntreface::`**`poll`**. Интерфейс имеет состояние, и в зависимости от его изменения может предпринимать различные действия. В обычном состоянии поток постоянно читает данные. Когда интерфейсу требуется прекратить работу, его состояние изменяется и поток исполнения завершает выполнение метода `BaseInterface::`**`poll`**.

Также интерфейс имеет возможность переподключиться к источнику данных, если произошёл разрыв соединения или произошло исключительное событие. Для этого у него имеется публичный слот `BaseInterface::`**`reconnect`**. Решение о попытке переподключиться принимает не сам интерфейс, а управляющая структура (или т.н. "верх", см. `ConnectionManager`).

`SerialPort`, `UsbHidPort` и `Ethernet` просты как классы - они лишь реализуют виртуальные методы `BaseInterface` для создания соединения, его разрыва, чтения и записи данных в интерфейс.

Парсеры запросов и ответов



BaseRequestParser

Данный класс представляет собой абстрактный парсер запросов к устройству. Под запросом в данном случае следует понимать также различные команды на запись и чтение данных, в том числе запись ВПО, конфигурационных файлов и запросы на чтение файлов.

Парсинг запроса в команду конкретного протокола осуществляется вызовом виртуального метода `BaserequestParser::`***`parse`***. Если для запроса требуются дополнительные действия, то метод `BaseRequestParser::`**`isExceptionalSituation`** вернёт `true`, для того, чтобы управляющая структура вызвала виртуальный метод `BaserequestParser::`***`exceptionalAction`*** для данного запроса.

В случае, если запрос не умещается в одной посылке, допустимой для протокола, она разбивается на несколько частей и вызывается сигнал `BaseRequestParser::`**`writingLongData`** для уведомления управляющей структуре (см. `DeviceQueryExecutor`). Для продолжения работы с текущей "длинной" командой используется метод `BaseRequestParser::`**`getNextDataSection`**.

BaseResponseParser

Данный класс представляет собой абстрактный класс парсера ответов от устройства. Для его использования вышестоящая управляющая структура должна выполнить следующую последовательность действий:

1. сложить полученные от устройства данные в буфер внутри парсера с помощью метода `BaseResponseParser::`**`accumulateToResponseBuffer`**;
2. вызвать виртуальный метод `BaseResponseParser::`***`isCompleteResponse`*** для проверки, является ли содержимое буфера парсера цельным ответом;
3. вызвать виртуальный метод `BaseResponseParser::`***`validate`*** для проверки корректности полученного ответа;
4. вызвать виртуальный метод `BaseResponseParser::`***`parse`***.

Полученные ответы от устройства парсер будет возвращать вызовом сигнала `BaseResponseParser::`**`responseParsed`**.

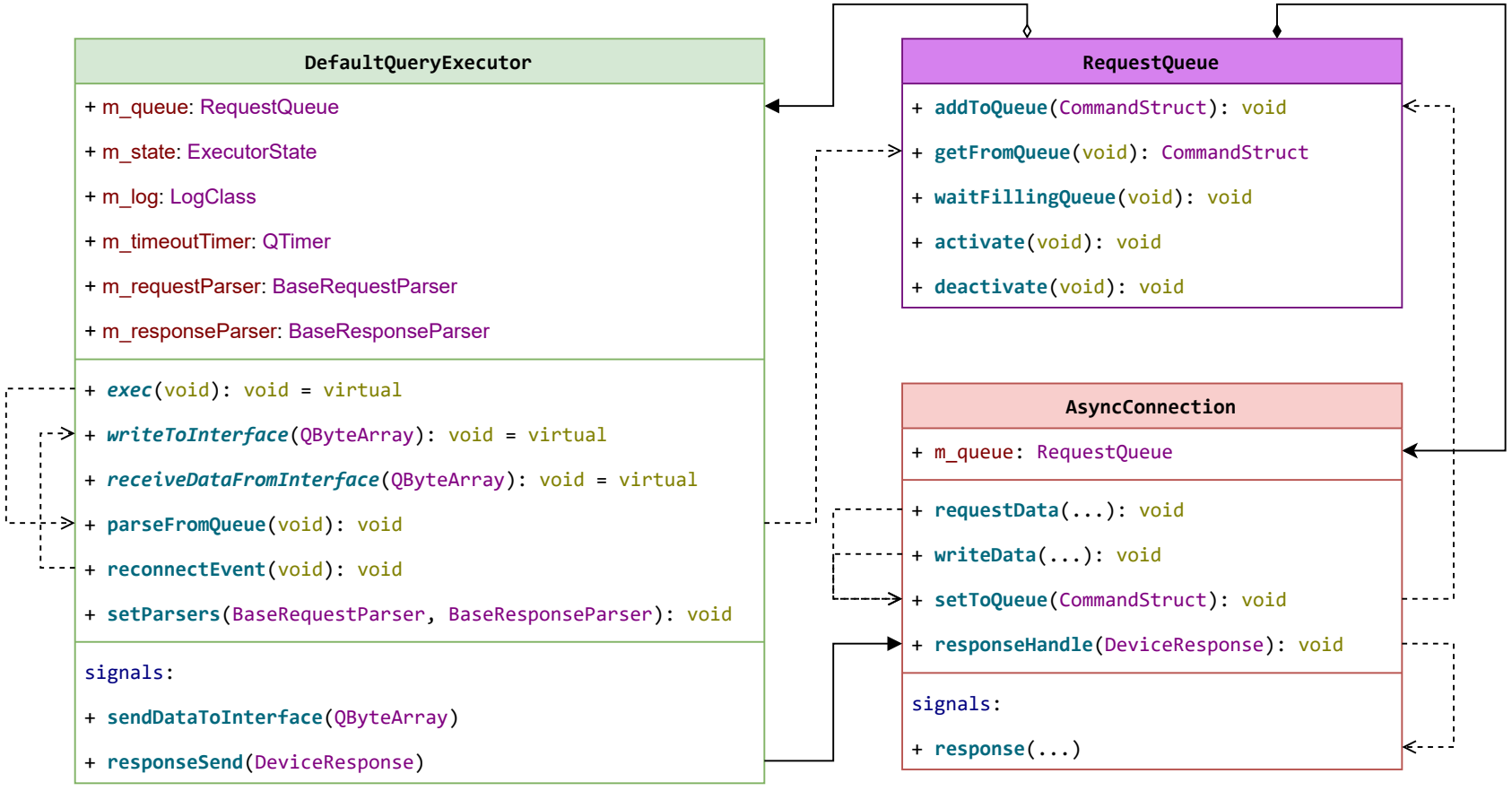
По аналогии с парсером запросов, парсер ответов может понять, когда ответ пришёл не полностью (например, чтения файла из устройства). В таком случае он имеет сигнал `BaseResponseParser::`**`readingLongData`**, информирующий об этом управляющую структуру. В таком случае, данная структура должна будет продолжить работать над данным запросом: она должна отправить устройству запрос на продолжение отправки данных. Данные запрос генерирует парсер запросов с помощью виртуального метода `BaseRequestParser::`**`getNextContinueCommand`**.

Классы-наследники

Парсеры для протоколов Protocom и Modbus RTU реализуют публичные виртуальные методы, а также внутреннюю логику для работы с получаемыми и отправляемыми данными. Но есть некоторые отличия их в работе. Например, `ProtocomRequestParser` не умеет отправлять запросы на чтение файлов журналов, так как при подключении по USB имеется возможность читать файлы намного быстрее напрямую с виртуальной файловой системы подключённого устройства. В таком случае при выполнении метода `ProtocomRequestParser::`***`parse`*** выставляется флаг исключительной ситуации (*exceptional situation*), а при вызове метода `ProtocomRequestParser::`***`exceptionalAction`*** управляющей структурой нужный файл будет прочитан и передан через связку сигнал-слот `ProtocomResponseParser`, который перешлёт его наверх.

Также дополнительно расширяется и функциональность парсеров для протокола Modbus RTU. Только в данном случае дополнительный сигнал парсера запросов используется для того, чтобы передать парсеру ответов размер ожидаемого ответа от устройства.

Исполнитель запросов к устройству



Вместо предисловия

Пожалуй, начать эту часть стоит с того, что получает клиент, когда запрашивает у `ConnectionManager` соединения - инстанс класса `AsyncConnection`, с помощью которого основная часть программы может запрашивать данные у подключённого устройства и записывать в него данные (`AsyncConnection::`**`requestData`**, `AsyncConnection::`**`writeData`**, etc...). `AsyncConnection` предоставляет удобный пользовательский интерфейс для обмена данными с устройством, не зная подробностей о типе подключения и используемом протоколе связи.

Данные пользовательские запросы преобразуются в `CommandStruct` и добавляются в очередь запросов (инстанс класса `RequestQueue`) с помощью метода `AsyncConnection::`**`setToQueue`**. Очередь запросов - это просто очередь уникальных объектов типа `CommandStruct` с мьютексом для работы в нескольких потоках. Также эту очередь можно деактивировать (`RequestQueue::`**`deactivate`**) - в данном режиме новые запросы в неё не будут добавляться - и активировать (`RequestQueue::`**`activate`**). Ссылкой на эту очередь `AsyncConnection` делится с `DefaultQueryExecutor`, организовывая связь для обмена данными между потоками, в которых они работают.

DefaultQueryExecutor

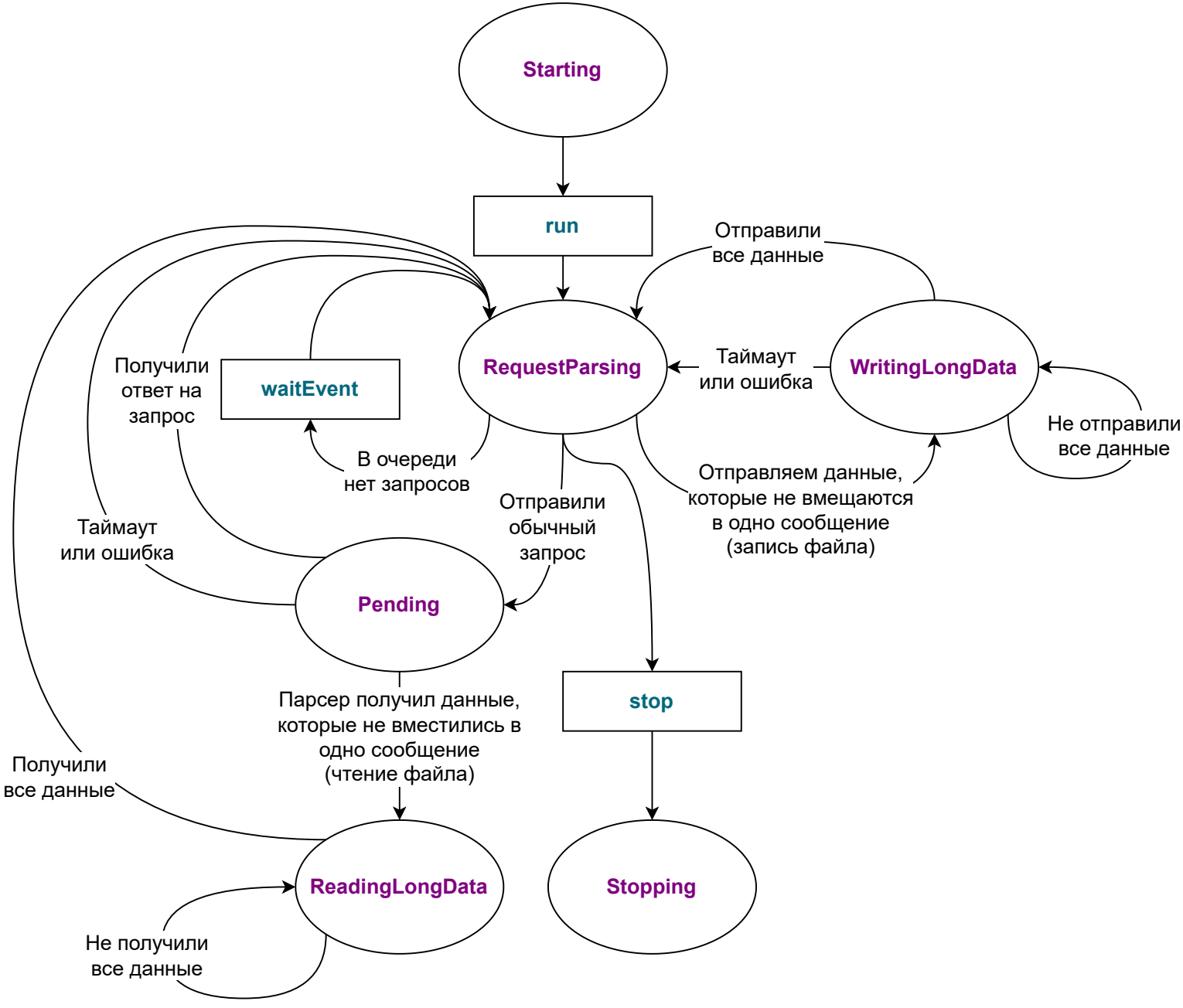
Название класса говорит само за себя - это исполнитель запросов к устройству. При создании соединения (**Connection**) создаётся и исполнитель (**Executor**) - в этот момент соединение передаёт исполнителю ссылку на очередь запросов. После этого оба объекта имеют общую (*shared*) очередь запросов. После этого исполнителю передаются созданные парсеры запросов и ответов с помощью метода `DefaultQueryExecutor::`**`setParsers`**. При инициализации контекста соединения (см. `ConnectionContext`) выполнение исполнителя переносится в отдельный поток - выполняется метод `DefaultQueryExecutor::`**`exec`**. По сути это аналог `BaseInterface::`**`poll`** - пока состояние исполнителя не определяется как `Stopping`, выполняются запросы к устройству. До начала работы исполнителя состояние определяется как `Starting`. После выполнение метода `DeviceQueryExecutor::`**`run`** состояние исполнителя меняется на `RequestParsing` - именно в этом состоянии выполняется парсинг клиентских запросов из `RequestQueue`.

Выполнение запросов к устройству происходит следующим образом: выполняется метод `DeviceQueryExecitor::`**`parseFromQueue`** - он вытаскивает из очереди запрос устройству, если таковой имеется, в противном случае исполнитель выполняет метод `DefaultQueryExecutor::`**`waitEvent`**, чтобы дождаться момента, когда для него появится работа. `BaseInterface` и `AsyncConnection` вызывают метод wakeUp для его пробуждения, когда получен новый запрос или ответ от устройства. Когда получен запрос, он передаётся парсеру запросов, который вернёт запрос в описанном протоколе (`BaseRequestParser::`**`parse`**) или выполнится исключительное действие (`BaseRequestParser::`**`exceptionalAction`**). В первом случае состояние исполнителя изменится на `Pending` или `WritingLongData` - в зависимости от запроса и сигнала `BaseRequestParser::`**`writingLongData`**. Во втором случае исполнитель приступит к следующему запросу из очереди, поскольку состояние исполнителя не изменилось и всё так же обозначено как `RequestParsing`.

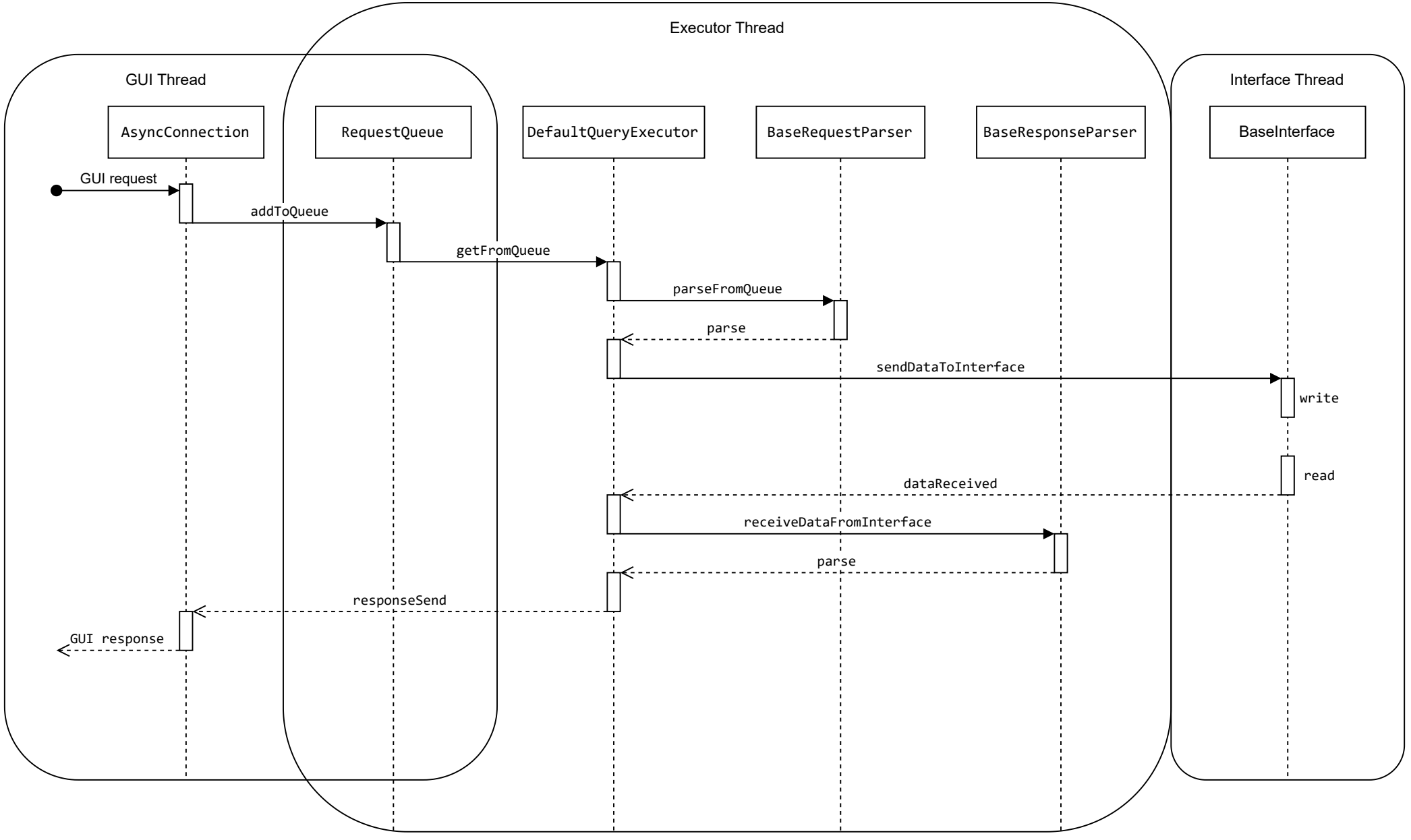
В состоянии `Pending` исполнитель будет ожидать ответа от устройства, который попадёт в слот `DefaultQueryExecutor::`**`receiveDataFromInterface`** или произойдёт таймаут и отправленный запрос будет отменён. Полученные от интерфейса данные будут переданы парсеру ответов и выполнена последовательность действий, описанная ранее для `BaseResponseParser`. В зависимости от выполненных действий состояние исполнителя изменится на `ReadingLongData` или `RequestParsing`.

В состояниях `WritingLongData` и `ReadingLongData` исполнитель продолжит выполнять запрос с помощью методов `BaseRequestParser::`**`getNextDataSection`** и `BaseRequestParser::`**`getNextContinueCommand`** соответственно. При этом очередь запросов будет временно переведена в неактивный режим, чтобы избежать её переполнения.

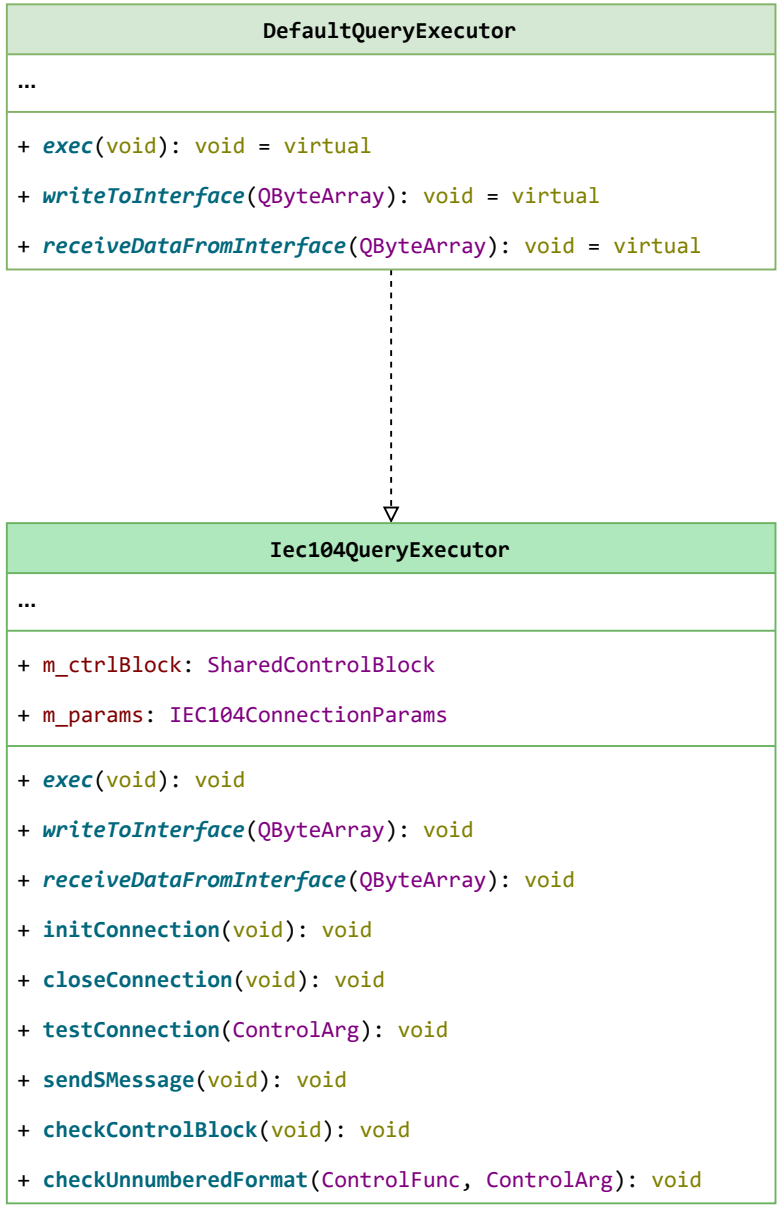
Диаграмма состояний исполнителя запросов



Пайплайн одного запроса



Почему DefaultQueryExecutor? Бывает не Default?



Изначально `DefaultQueryExecutor` должен был стать многофункциональным классом, используемым всеми протоколами связи. Но как всегда бывает в таких случаях, есть нюанс. Связь с устройствами осуществляется с помощью 3 протоколов:

- **Protocom** - используется для связи с устройствами по USB с помощью HID API;
- **Modbus** - используется для связи с устройствами по интерфейсу RS-485;
- **ГОСТ Р МЭК 60870-5-104** - используется для связи с устройствами по интерфейсу Ethernet.

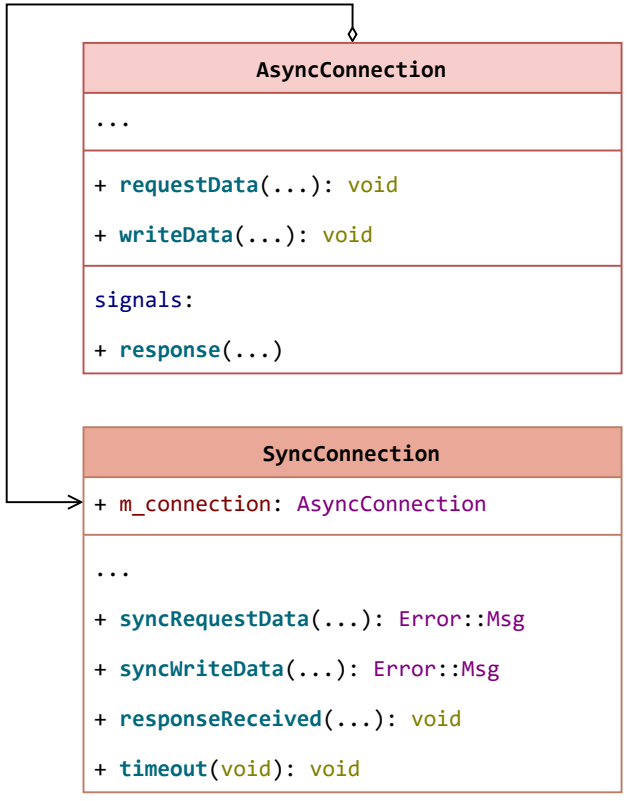
Остановимся на последнем. ГОСТ Р МЭК 60870-5-104 в отличие от Modbus и Protocom является асинхронным протоколом, что позволяет присылать устройству данные спорадически, т.е. без запроса клиента. Изначально `DefaultQueryExecutor` проектировался для работы с синхронными протоколами. Это означает, что при работе с конкретными парсерами исполнитель запросов полагается на то, что полученный ответ всегда будет соответствовать отправленному запросу. ГОСТ Р МЭК 60870-5-104 работает иначе: перед ответом на отправленный запрос исполнитель может получить несколько фреймов различного содержания, которые он должен передать парсеру запросов для их корректной обработки.

В связи с вышеперечисленным было решено сделать некоторые методы `DefaultQueryExecutor` виртуальными и, используя его в качестве базового, был написан класс `Iec104QueryExecutor`. Помимо этого данный класс берёт на себя ответственность по некоторым операциям, нужным только для реализации связи по протоколу ГОСТ Р МЭК 60870-5-104:

- проверяет контрольный блок со статистикой и отправляет устройству S-посылки с информацией о принятых и отправленных I-посылках в соответствии с протоколом;
- в зависимости от полученной U-посылки выполняет внутреннюю логику, например, посылает ответ на тестовый фрейм, начинает или завершает свою работу.

Указанная ранее диаграмма состояний исполнителя запросов характерна и для `Iec104QueryExecutor`. Указанный ранее пайплайн одного сообщения характерен отчасти, т.к. ответ от устройства может быть получен не сразу, т.е. асинхронно.

А что насчёт AsyncConnection? Бывает SyncConnection?

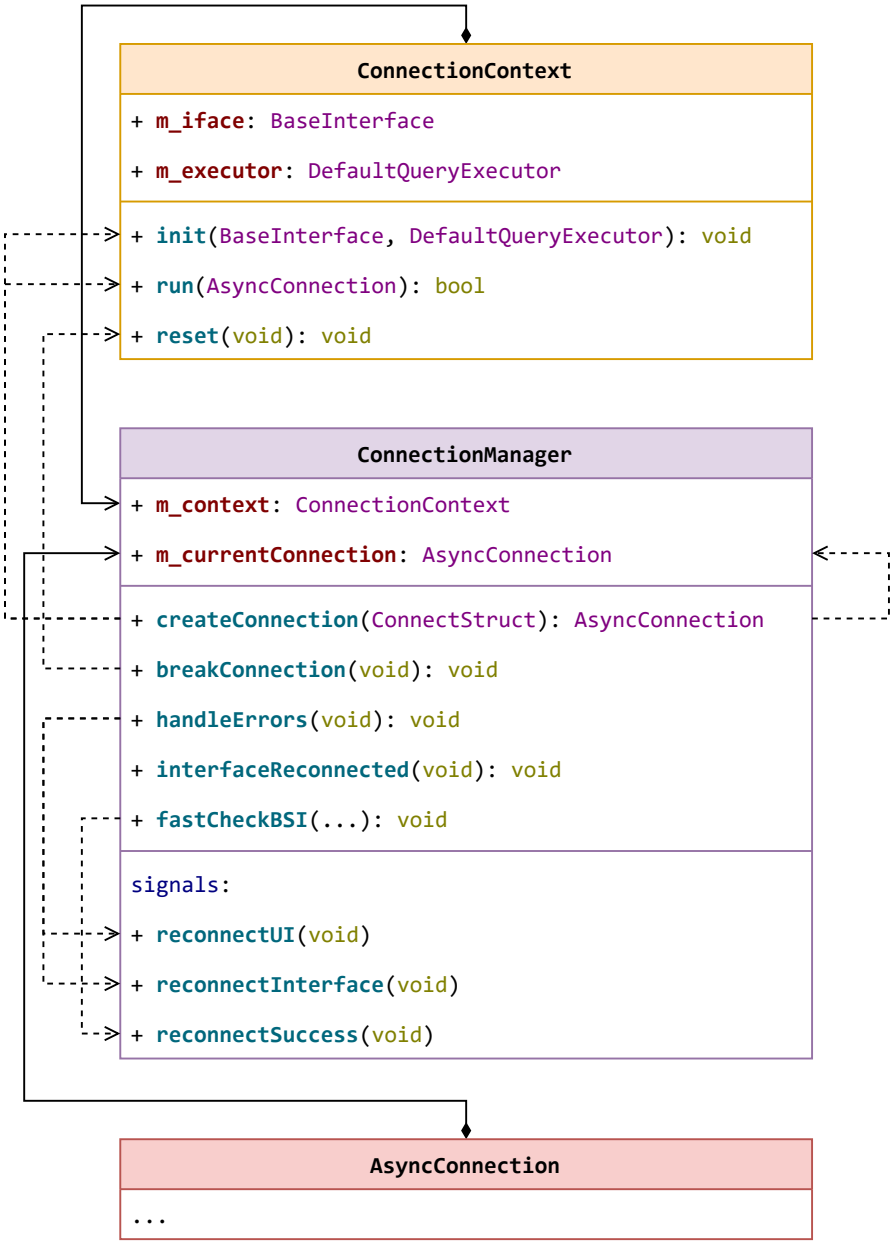


Да, помимо `AsyncConnection` в распоряжении клиента также есть класс-адаптер `SyncConnection`, который представляет собой обёртку для синхронной работы с подключённым устройством. Данный класс получает при создании инстанс `AsyncConnection` и использует его для обмена данными. Для этого в `SyncConnection` предусмотрено несколько слотов: `SyncConnection::`**`responseReceived`** и `SyncConnection::`**`timeout`**.

Методы с префиксом *sync* будут использовать инстанс `AsyncConnection` и event loop для синхронного ожидания ответа на отправленный запрос.

Синхронная работа с устройством существенно упрощает разработку и может быть использована в различных сценариях, когда приложение не может приступить к следующему шагу алгоритма, не получив результат отправленного устройству запроса. В текущий момент синхронная связь с устройством чаще всего используется при регулировке.

ConnectionManager

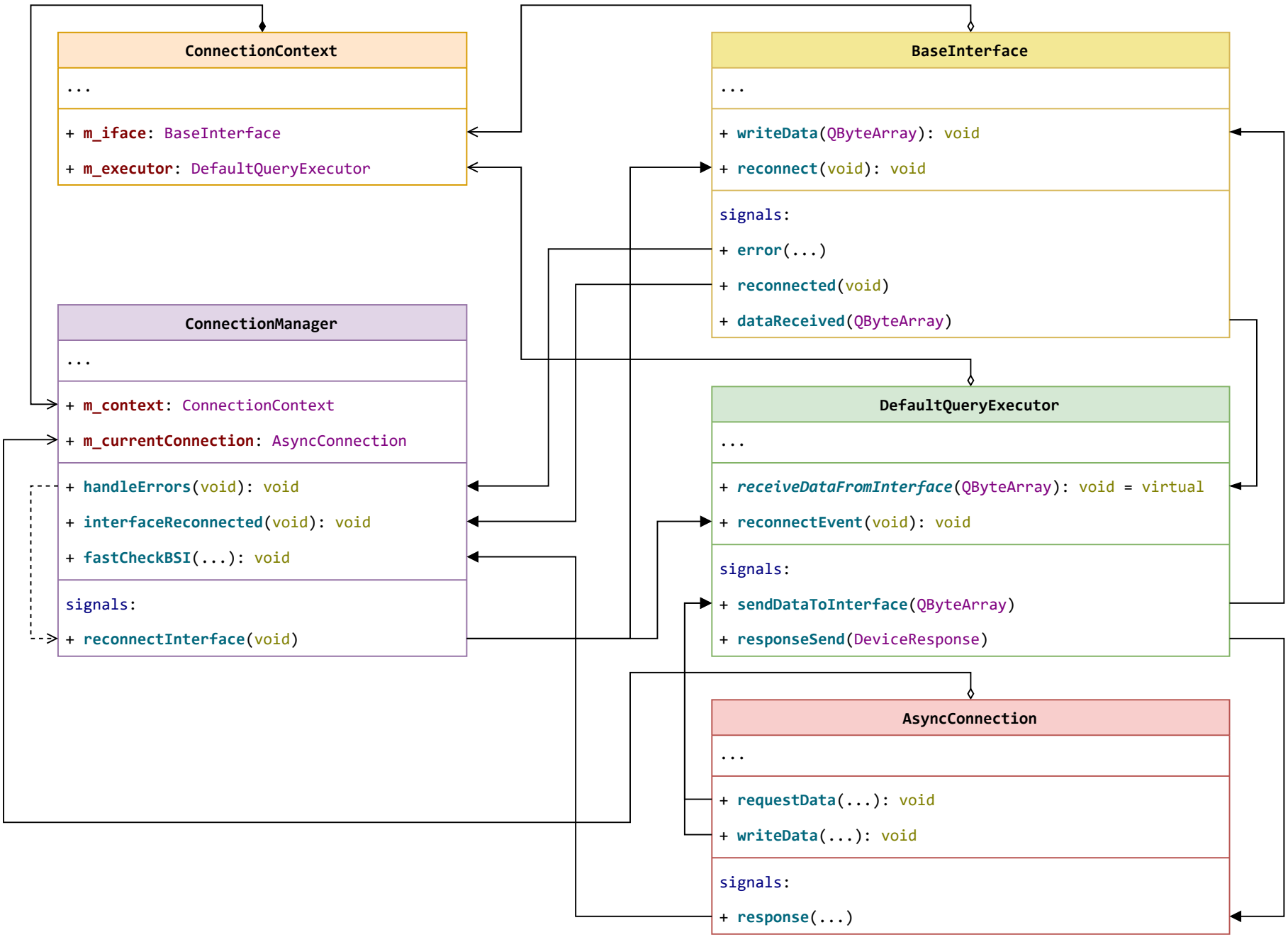


Как говорилось ранее, класс `Coma` для создания соединения с устройством вызывает метод `ConnectionManager::`**`createConnection`** с указанными настройками подключения, который возвращает инстанс класса `AsyncConnection`, если подключение прошло успешно.

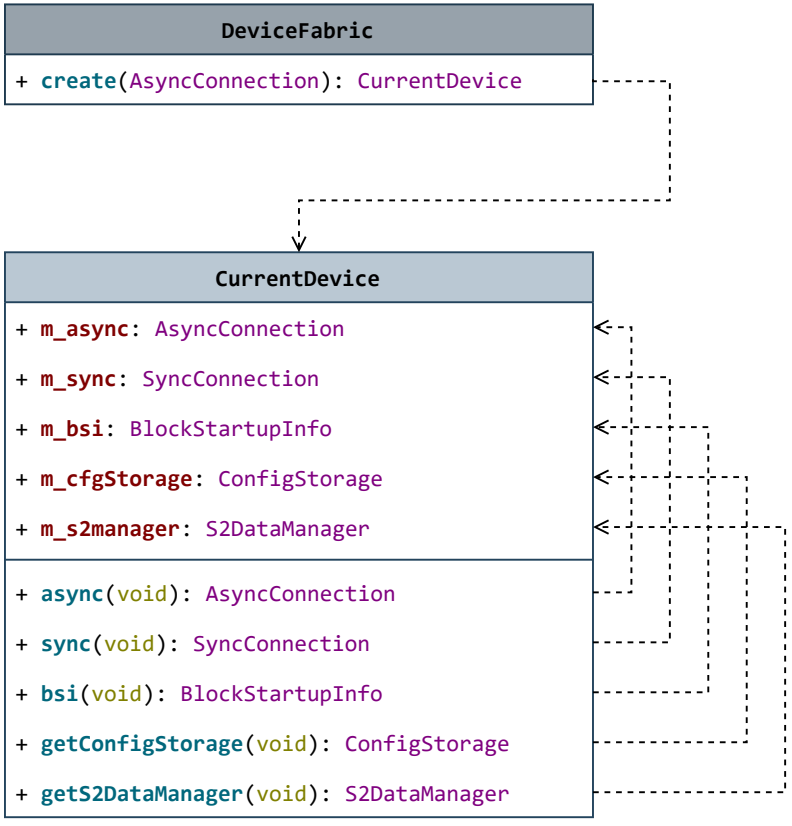
Внутри `ConnectionManager` создаёт соответствующие инстансы `BaseInterface` и `DefaultQueryExecutor` в зависимости от полученных настроек. После этого настраивается контекст соединения - эту работу выполняет класс `ConnectionContext`. В метод `ConnectionContext::`**`init`** передаются экземпляры исполнителя запросов и интерфейса, в котором соединяются их слоты и сигналы, создаются потоки в рамках которых эти инстансы будут выполняться и в контекст которых переносится их выполнение. Метод `ConnectionContext::`**`run`** принимает инстанс класса `AsyncConnection` и связывает его с классами исполнителя запросов и интерфейса через сигналы и слоты. Метод `ConnectionContext::`**`reset`** используется для остановки работы интерфейса и исполнителя запросов, уничтожения контекстов их выполнения и закрытия текущего соединения.

Помимо создания соединения с устройством и его контекст `ConnectionManager` берёт на себя управление ошибками текущего соединения и менеджментом переподключения. Слот `ConnectionManager::`**`handleErrors`** вызывается, если `BaseInterface` не смог прочитать/записать данные в интерфейс или `DefaultQueryExecutor` получил таймаут на отправленный запрос. Если число полученных ошибок превышает определённый допустимый порог, вызываются сигналы `ConnectionManager::`**`reconnectInterface`** (вводит `BaseInterface` в состояние переподключения и уведомляет об этом `DefaultQueryExecutor`) и `ConnectionManager::`**`reconnectUI`** (уведомляет UI о том, что предпринимается попытка переподключиться к устройству). Как только `BaseInterface` получилось подключиться к интерфейсу со старыми параметрами, то он уведомляет об этом менеджер и вызывается слот `ConnectionManager::`**`interfaceReconnected`**. В нём временном активируется очередь запросов и посылается запрос BSI. Если соединение корректно обработало этот запрос и вернуло ответ, то вызывается сигнал `ConnectionManager::`**`reconnectSuccess`** для уведомления UI и исполнителя запросов о том, что соединение успешно восстановлено.

Соединяем пазл воедино



CurrentDevice

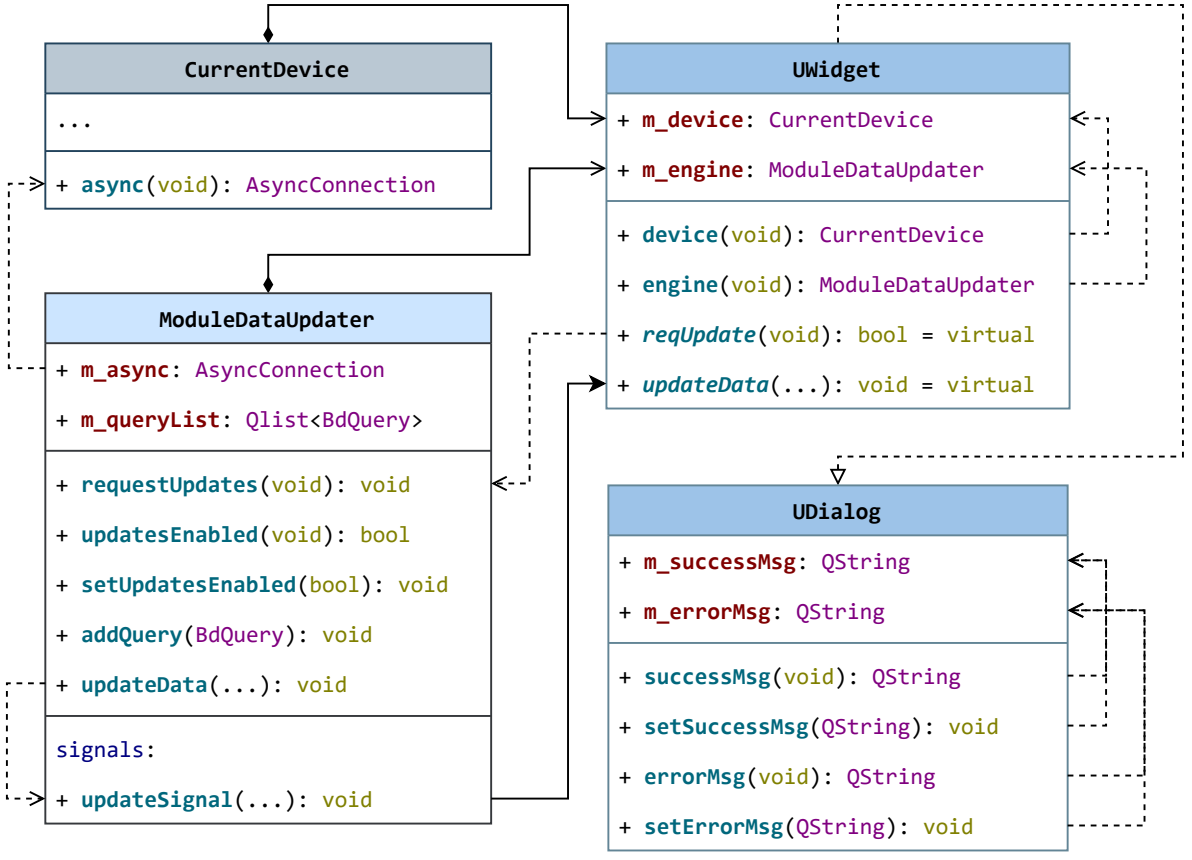


Временно вернёмся к самой первой страницу данного руководства - пришло время рассказать о классе **`CurrentDevice`**.

После установления связи с подключенным устройством полученный от менеджера соединений инстанс `AsyncConnection` передаётся в метод `DeviceFabric::`**`create`** класса-фабрики **`DeviceFabric`**. Она передаёт асинхронное соединение в конструктор `CurrentDevice`. Класс текущего устройства на основе асинхронного соединения (`AsyncConnection`) создаёт инстанс синхронного соединения (`SyncConnection`). Также этот класс хранит блок BSI (см. АВМР.424457.001 Д2 Приложение 1 "Состав блока стартовой информации BSI") и содержит специальные контейнеры, хранящие XML-настройки для текущего устройства и конфигурационные настройки в форматах S2 и S2B (см. АВМР.424457.001 Д5 Параграфы 1 и 7).

Все перечисленные выше настройки необходимы для создания диалогов пользовательского интерфейса и определения перечня возможных команд для обмена данными с подключенным устройством. Например, диалогу "Конфигурация" надо знать о всех возможных конфигурационных параметрах, доступных для текущего устройства, а также о формате ввода-вывода этих конфигурационных параметров на пользовательский интерфейс с помощью различных виджетов. Также прочитанная конфигурация устройства в формате S2 может использоваться созданными диалогами, например, для связи с пользователем в случае ошибок.

ModuleDataUpdater, UWidget и UDialog



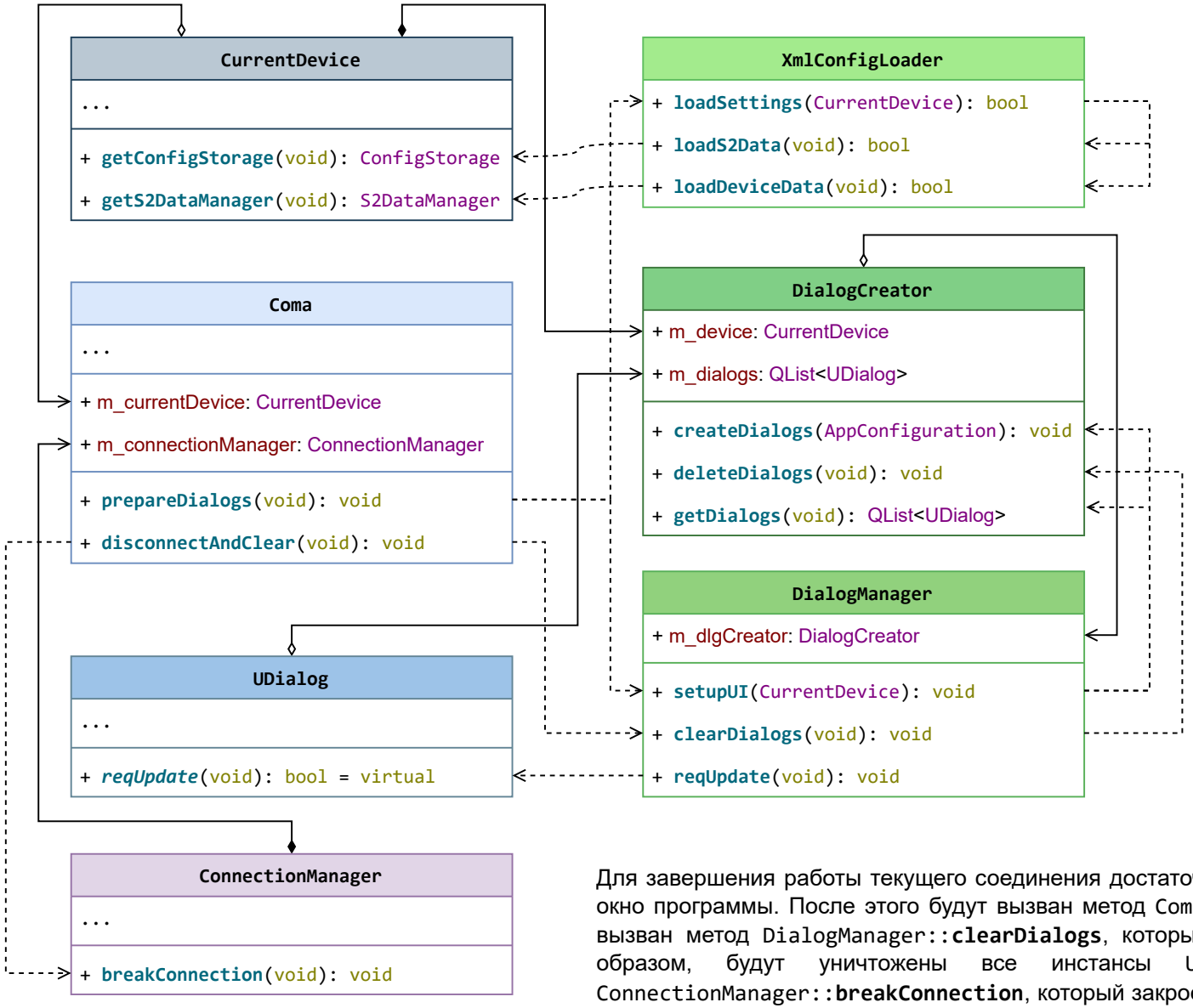
Ранее говорилось, что в методе `Coma::`**`prepareDialogs`** создаются диалоги, предоставляющие UI для обмена данными с подключенным устройством, но не обговаривалось, как это происходит.

Дело в том, что все диалоги являются потоками класса **`UDialog`**, который в свою очередь является потомком **`UWidget`**. Последний имеет два поля для хранения инстанса текущего устройства и инстанса класса **`ModuleDataUpdater`**. Данный класс предоставляет удобный интерфейс движка (*engine*) обмена данными для асинхронного соединения. Он хранит список запросов, который отправляет устройству по асинхронному соединению для текущего виджета с помощью вызова метода `ModuleDataUpdater::`**`requestUpdates`**. Ответ от устройства принимает слот `ModuleDataUpdater::`**`updateData`** и с помощью сигнала передаёт в слот `UWidget::`**`updateData`** текущего виджета.

Слот `UWidget::`**`updateData`** является виртуальным, поэтому все дочерние по отношению к `UWidget` классы имеют возможность его переопределить, предоставив собственные обработчики получаемых от устройства данных. Метод `UWidget::`**`reqUpdate`** также является виртуальным, так что его тоже можно переопределить. Базовая реализация в классе `UWidget` просто вызывает метод `ModuleDataUpdater::`**`requestUpdates`**.

Главным отличием `UDialog` от `UWidget` является то, что он имеет дополнительную реализацию `UDialog::`**`updateData`** для получения и обработки ответов типа **`DataGeneralResponse`** от устройства. Данный обработчик проверяет полученный ответ на наличие ошибки, и при её отсутствии создаёт окно пользовательского интерфейса с сообщением `m_successMsg`. В случае ошибки появится окно с сообщением `m_errorMsg`. Для управления этими сообщениями предназначены методы `UDialog::`**`successMsg`**, `UDialog::`**`setSuccessMsg`**, `UDialog::`**`errorMsg`** и `UDialog::`**`setErrorMsg`**.

Ещё немного классов



Но прежде чем создавать экземпляры `UDialog`, надо произвести ещё несколько действий. Например, загрузить XML-настройки для подключенного устройства - этим занимается класс **`XmlConfigLoader`**. Его задача - найти XML файлы для текущего устройства и с помощью парсеров заполнить `ConfigStorage` и `S2DataManager` внутри `CurrentDevice`.

Созданием инстансов `UDialog` занимается класс **`DialogCreator`**, который создаётся внутри метода `DialogCreator::`**`createDialogs`**. Его вызывает класс **`DialogManager`** в методе `DialogManager::`**`setupUI`** после создания экземпляра `DialogCreator` и передачи ему инстанса `CurrentDevice`. После этого менеджер диалогов отрисовывает полученные диалоги и запускает таймер, по которому вызывается метод `UDialog::`**`reqUpdate`** для текущего активного диалога.

Таким образом, складывается цельная архитектура: после создания соединения и получения настроек для конкретного устройства приложение на основе этих настроек создаёт диалоги обмена данных с устройством.

Для завершения работы текущего соединения достаточно нажать кнопку "Разрыв соединения" или закрыть окно программы. После этого будут вызван метод `Coma::`**`disconnectAndClear`**. Сначала внутри него будет вызван метод `DialogManager::`**`clearDialogs`**, который вызовет `DialogCreator::`**`deleteDialogs`** - таким образом, будут уничтожены все инстансы `UDialog`. После этого будет вызван метод `ConnectionManager::`**`breakConnection`**, который закроет текущее активное соединение и прекратит обмен с устройством.

Архитектура всего приложения, зависимости внутренних библиотек друг от друга