Software Architecture Document

История изменений

Дата	Версия	Описание	Автор
	1.0		Karinski, Andrey

1. Введение

Введение должно содержать обзор всего документа и включать цель, область применения, определения и сокращения, ссылки на другие архитектурно-важные документы.

Этот документ содержит полный архитектурный обзор системы и использует ряд архитектурных представлений в формате «4+1» (Филиппа Крачтена) для отображения различных аспектов системы. Сам документ служит для отражения важных архитектурных решений, принятых при проектировании системы.

Модель 4+1 была разработана Филиппом Кратченом (одним из создателей нотации UML и методологии RUP) для описания архитектуры информационных систем, основанном на использовании нескольких взаимодополняющих представлений. Представления используются для описания системы с точки зрения разных заинтересованных лиц, таких как конечные пользователи, разработчики или руководители проекта. Четыре представления – это логическая структура, разработка, процессы и физическая структура системы. Дополнительно вводятся варианты использования, или сценарии, которые иллюстрируют описываемую архитектуру.

1.1. Цель

Весь документ в целом содержит полный архитектурный обзор системы, используя ряд различных архитектурных представлений для отображения различных важных аспектов системы.

Данный же раздел описывает определяет место документа во всей проектной документации и кратко описывает структуру документа. Так же важно указать проектные роли лиц, которые этот документ используют в работе.

Документ SAD содержит обзор архитектуры утилиты «Json2Uml». На текущий момент, утилита позиционируется как инструмент для двусторонней конвертации представлений информационной модели OSDU между представлениями в форматах Json Schema и XMI UML.

Содержимое документа важно в первую очередь проектной команде (разработчикам и аналитикам), но он также может быть интересен всем техническим специалистам компании NEDRA в учебных целях. Документ нужен для того, чтобы получить подробную информацию о способах реализации функциональных и нефункциональных требований.

Документ разрабатывается архитектором приложений, но может дорабатываться членами проектной команды, чьих навыков достаточно для проектирования и документирования архитектуры системы.

Первая версия документа создавалась после аналитической работы, но до перехода к программированию, следовательно, содержит лишь архитектурные концепции, которых следует придерживаться, но никак не подробную инструкцию по реализации. В дальнейшем документ будет дорабатываться и актуализироваться.

UML-диаграммы в документе были созданы в инструменте «Sparx Enterprise Architect».

1.2. Область действия

Краткое описание того, на что влияет данный документ: другие документы, роли, окружения.

Документ зависит от «Видения» (описания концепции системы), перечня Use Case Spec (Вариантов Использования), представляющих описание функциональных требований, и Supplementary Spec (Дополнительной Спецификации), содержащей нефункциональные требования и атрибуты качества системы.

Документ оказывает влияние на Use Case Realization Spec (Спецификацию реализации Вариантов использования), Deployment Plan (План развертывания) и Measurement Plan (описывает метрики системы и критерии их определения). Наконец, поскольку проектируемая утилита делается не по заказу ГПН, то не требуется создание документов документов "Упрощенный технический паспорт (УТП)" и "Презентация к АК и ТС".

1.3. Определения, сокращения и аббревиатуры

Здесь даны определения всех терминов и сокращений, необходимых для правильной интерпретации содержимого документа. Может содержать ссылку на раздел Глоссария, посвященный техническим терминам.

Общий для всего проекта перечень определений и сокращений представлен в документе "Glossary".

- OSDU (Open Subsurface Data Universe) - открытая спецификация платформы со стандартизированным набором интерфейсов (API) для у правления, хранения, обработки данных для нефтегазовой отрасли.
- API (Application Program Interface) - интерфейс прикладного программирования, который представляет собой набор готовых классов, процедур, функций, структур и констант, которые предоставляются сервисом для использования во внешних программных продуктах.

1.4. Ссылки

Здесь должен быть представлен полный список всех документов, на которые есть ссылки в других разделах данного документа. Не забыть указать источники.

Входящие документы:

- Glossary - перечень определений и акронимов.
- Stakeholder Requests - первичные требования заинтересованных лиц.
- Use Case Spec - спецификация вариантов использования.
- Supplementary Spec - дополнительная спецификация.

Исходящие документы:

- Use Case Realization Spec - спецификация реализации архитектурно-значимых вариантов использования.
- Deployment Plan - план по развертыванию утилиты.
- Measurement Plan - план и методика измерения различных показателей проекта и самой утилиты.

Ссылки на внешние источники и стандарты:

- osduforum - информационная страница стандарта OSDU.
- OSDU Data Definitions - репозиторий информационной модели OSDU.
- XMI - спецификация XMI 2.5.
- Json Schema - спецификация.

1.5. Обзор

Краткое описание содержания документа и принципа его организации.

Документ описывает архитектуру как серию представлений: Варианты Использования, Логическое, Процессное, Реализации, Развертывания. Каждый архитектурно важный момент может снабжаться дополнительными диаграммами, поясняющими концепцию.

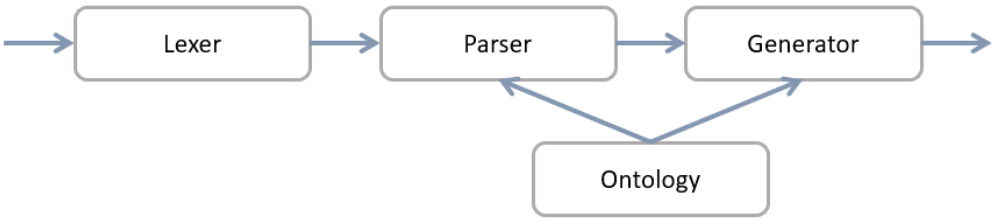
2. Архитектурное представление

Обобщенное и верхнеуровневое архитектурное представление системы, и то, как она представлена.
Описание метафоры архитектуры или основополагающего паттерна.

Кроме того, раздел должен содержать список необходимых архитектурных представлений (UseCase. Logical, Process, Deployment, Implementation) и для каждого представления объяснять, элементы модели они описывают.

Собственно, с архитектурной точки зрения конвертер из одного сложного формата в другой удобно воспринимать как Транслятор - преобразователь данных из одной модели в другую. Поскольку и входная, и выходная модели довольно сложны, разумно выполнить преобразование с использованием промежуточной Канонической Модели: сначала данные источника транслируются в промежуточный формат - каноническую модель, затем эта модель транслируется в целевой формат.

Типичная архитектура потоков данных транслятора похожа на то, что представлена на схеме ниже:



Такая архитектура предполагает, что сначала входная модель превращается в поток лексем (компонент Lexer), потом эти лексемы превращаются в грамматику в виде абстрактного синтаксического дерева (компонент Parser), затем синтаксическое дерево (AST) передается генератору, преобразующего его в выходную модель. Таким образом, обрабатываемые данные проходят ряд последовательных шагов преобразований, что напоминает конвейер. Хорошая метафора для такой последовательности действий - конвейер, а значит, логично применить шаблон проектирования `Pipes and Filters`, а точнее, более общий его случай - `Dataflow Architecture`.

Далее будут представлены детали того, как архитектурная метафора превращается в дизайн системы.

3. Архитектурные цели и ограничения

Описание требований и целей системы, которые оказывают существенное влияние на архитектуру (например, объем данных, информационные потоки и нагрузка, безопасность, конфиденциальность, надежность, т.п.)
Так же может содержать описание особых ограничений, таких как:

- *Стратегии проектирования и реализации*
- *Инструменты разработки*
- *Структура команды*
- *Доступные ресурсы*
- *Унаследованный код*

Исходные требования к утилите содержатся в документах Vision, Use Case Spec и Supplementary Spec. Краткое изложение наиболее важных и архитектурно значимых требований ниже:

- **Консольная утилита.**
  Приложение должно запускаться из командной троки и взаимодействовать с пользователем стандартным образом.

- **Сценарии работы - конвертация из Json Schema в XMI и наоборот**.
  Приложение должно запускать процесс конвертации, который должен выполняться достаточно быстро, потому что для консольного приложения ожидается достаточно быстрый отклик. Аргументы для процесса конвертации должны быть получены из командной строки.

- **Показ версии приложения**.
  Так как информационная модель OSDU постоянно меняется, то вполне возможна ситуация, когда утилита не сможет корректно обработать входящие данные. Для того, чтобы использовать утилиту в такой ситуации, необходимо точно понимать, какая ее версия используется в данный момент.

- **Подробная информация о выполнении конвертации**.
  Поскольку импортируемая модель может содержать ошибки или неоднозначности, а просто отказ от выполнения конвертации может быть неприемлемым, то пользователю необходимо предоставлять подробный отчет о выполненной конвертации, обраруженных ошибках и т.п.

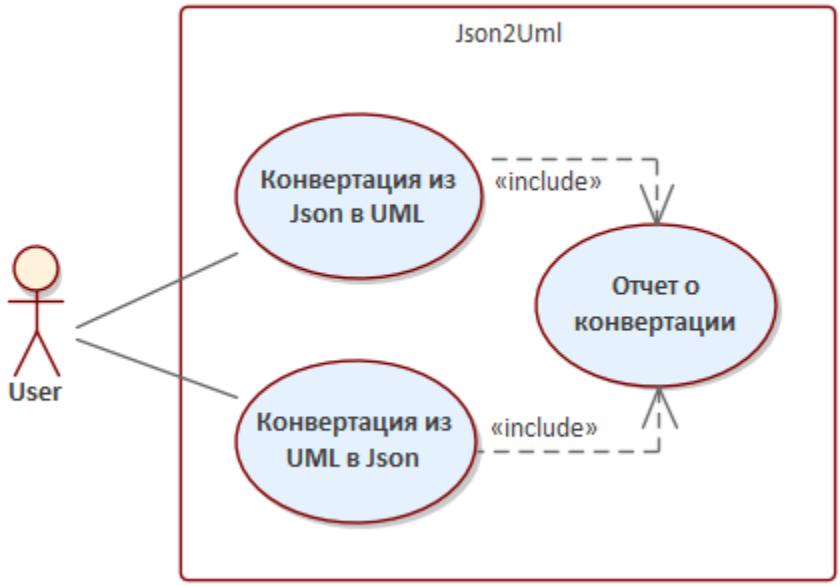
- **Высокая надежность работы**.
  Утилитой будут пользоваться сотрудники различных компаний, входящих в OSDU, поэтому явные ошибки и сбои могут нанести вред репутации компании NEDRA.

- **Удобство пользователя**.
  Консольная утилита должна стандартным образом показывать справочную информацию по поддерживаемым операциям, аргументам и т.п.

4. Представление Вариантов Использования (Use Case View)

Перечисление вариантов использования или пользовательских историй, если они представляют значительную важность для системы и обладают влиянием на архитектуру (то есть в процессе их реализации будет затронуто значительное число архитектурных артефактов), либо они подчеркивают или иллюстрируют важную часть архитектуры.

На диаграмме ниже показаны архитектурно-значимые варианты использования. Остальные требования не оказывают значимое влияние на архитектуру и дизайн, и поэтому не отображены.



5. Логическое представление (Logical View)

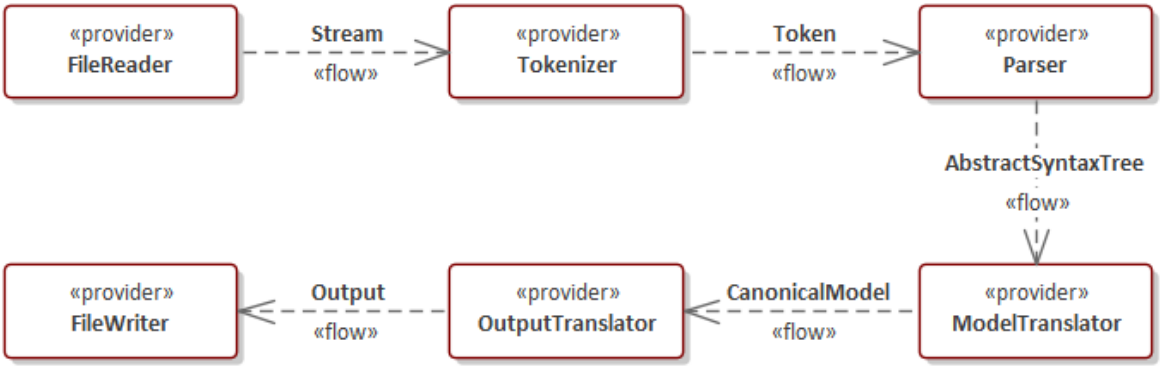
Описываются архитектурно значимые части модели системы, такие как ее разбиение на подсистемы, пакеты и модули. Для каждого важного пакета потребуется описать

- *значимые классы (разумеется, без подробностей, так как они будут проясняться только в процессе реализации)*
- *их обязанности*
- *несколько важных отношений, атрибутов и операций*

Так же стоит отметить, что описанные в разделе классы являются концептуальными (т.е. артефактами моделирования, а не реализации) и могут не соответствовать программным классам из модели реализации.

Рассмотрим основной вариант использования утилиты - конвертацию из одного формата в другой (сейчас принципиально не важно, в какой именно).

Как ясно из архитектурной метафоры (раздел 2), наиболее значимые концептуальные классы утилиты связаны между собой в конвейер, передающий данные по цепочке.

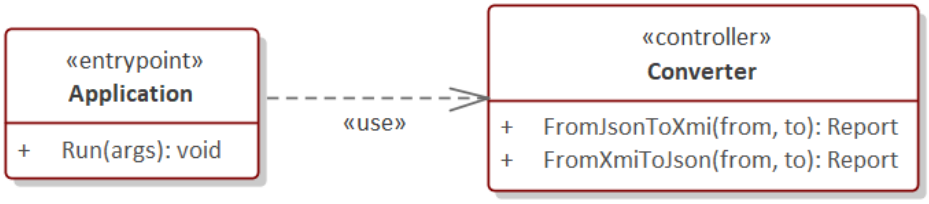


Концептуальные классы, их обязанности и передаваемые данные:

- **FileReader** - читает содержимое файла и преобразует в `Stream`.
- **Stream** - представляет данные файла как поток символов.
- **Tokenizer** - преобразует `Stream` в набор токенов `Token`.

- **Token** - представляет значимый элемент содержимого `Stream`.
- **Parser** - преобразует набор токенов `Token` в синтаксическое дерево `AbstractSyntaxTree`.
- **AbstractSyntaxTree** - представляет семантическое ядро абстрактной грамматики.
- **ModelTranslator** - дополняет каноническую модель изменениями из синтаксического дерева.
- **CanonicalModel** - представляет промежуточную модель, отображающую ООП-подобные сущности со связями, атрибутами, ограничениями и т.п.
- **OutputTranslator** - преобразует каноническую модель в целевой формат.
- **Output** - представляет целевой формат: это может быть XMI или Json Schema.
- **FileWriter** - записывает содержимое целевого формата в файл.

Вышеперечисленные классы затем объединяются в конвейер при помощи контроллера приложения (фасада), затем контроллер будет вызван через точку входа в консольное приложение.



Классы, организующие поток управления:

- **Application** - точка входа в утилиту, организует взаимодействие с пользователем.
- **Converter** - фасад для логики конвертации, организует и запускает конвейеры.

Контроллер (класс `Converter`) затем выберет, согласно полученным от `Application` аргументам, один из двух конвейеров, запустит его, а затем вернет отчет (`Report`) о выполнении. Затем отчет будет показан пользователю в командной строке, а утилита завершит работу стандартным для консольного приложения способом.

5.1. Обзор

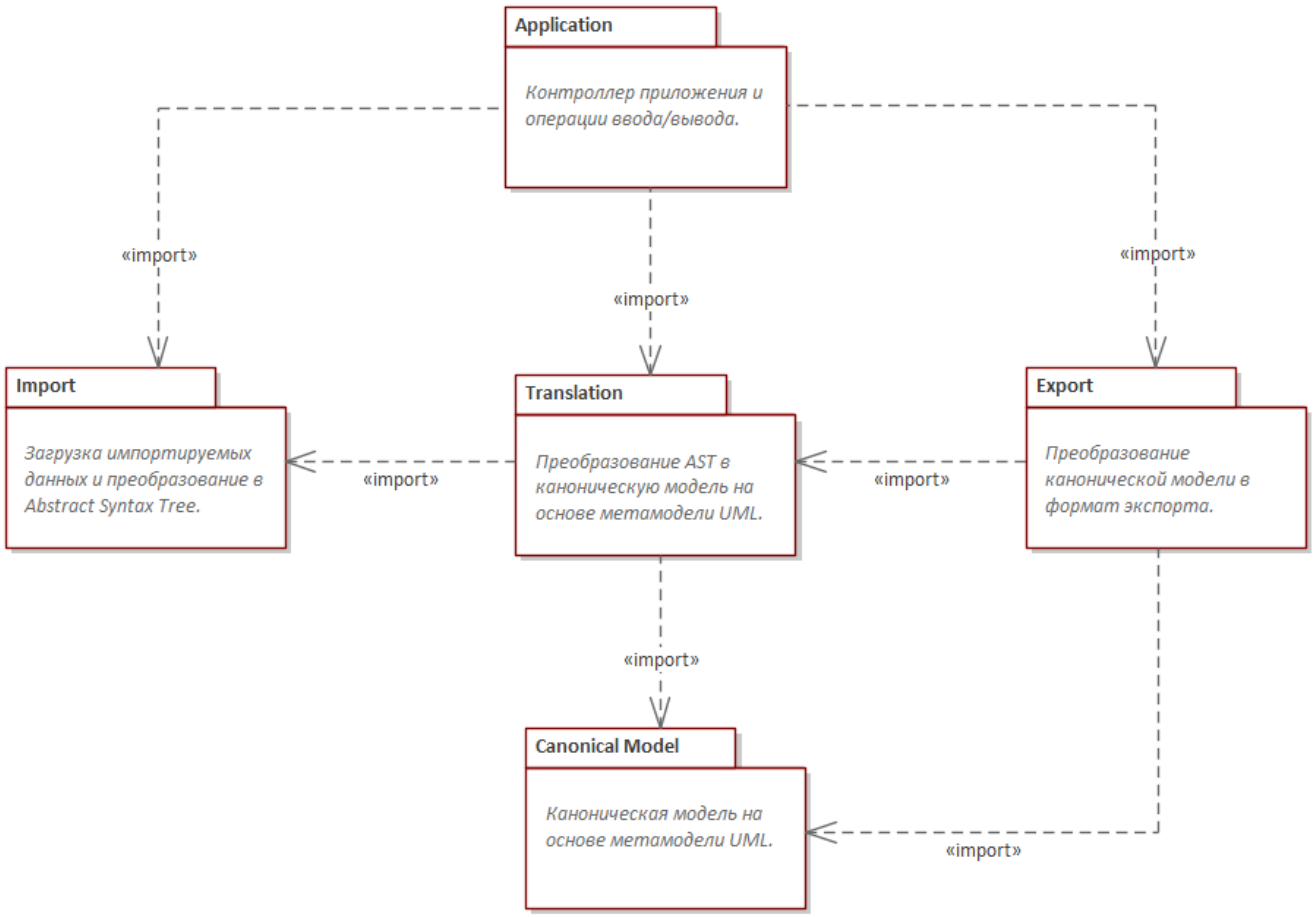
Описание общих принципов декомпозиции модели с точки зрения слоев и иерархии пакетов.

Все классы утилиты располагаются в нескольких пакетах строго по назначению/ответственности. Далее будут рассмотрены наиболее важные из них.

5.2. Архитектурно значимые пакеты

Для каждого значимого пакета включить подраздел с его описанием и диаграммой со всеми значимыми содержащимися в нем классами и пакетами. Для каждого значимого класса указать его имя, краткое описание, и при необходимости некоторые из его основных обязанностей, операций и атрибутов.

Классы утилиты можно сгруппировать согласно их обязанностям в несколько взаимозависимых пакетов. Следует помнить, что иклические зависимости между пакетами недопустимы. Ниже представлена диаграмма архитектурно-значимых пакетов и их краткое описание.



- **Application** - точка входа, контролер приложения, настройки и операции ввода-вывода.
- **Import** - загрузка импортируемых данных и преобразование в синтаксическое дерево.
- **Translation** - преобразование синтаксического дерева в каноническую модель.
- **Export** - преобразование канонической модели в выходной формат данных и экспорт.
- **Canonical Model** - каноническая (промежуточная) модель и механизмы конструирования.

5.3. Реализация Вариантов использования

Раздел иллюстрируют, как система работает внутри, посредством описания реализаций нескольких выбранных (и важных) сценариев; объясняет, как для этого используются различные элементы модели.

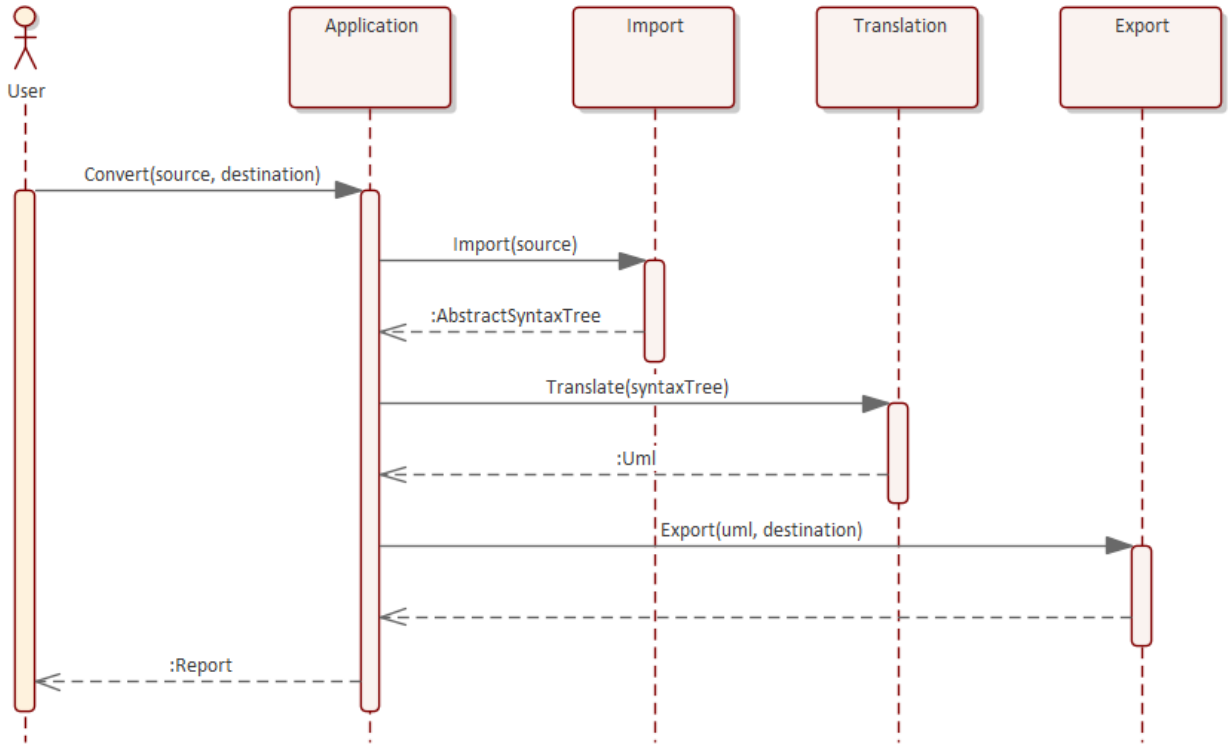
В разделе 5.0 показано взаимодействие классов, из которых и состоит 2 главных вариантов использования - конвертация из Json Schema в XMI и конвертация из XMI в Json Schema. Дополнительно описывать поведение концептуальных классов в рамках вариантов использовния не имеет смысла.

6. Процессное представление (Process View)

Описание декомпозиции системы на бизнес-процессы. Удобно организовать раздел по группам взаимодействующих процессов. Хорошо так же проследить зависимость от соответствующих use cases.

На диаграмме ниже показано типичное взаимодействие между основными компонентами утилиты (в данном случае вместо компонетов показаны пакеты), демонстрирующее, как реализуются основные варианты использования.

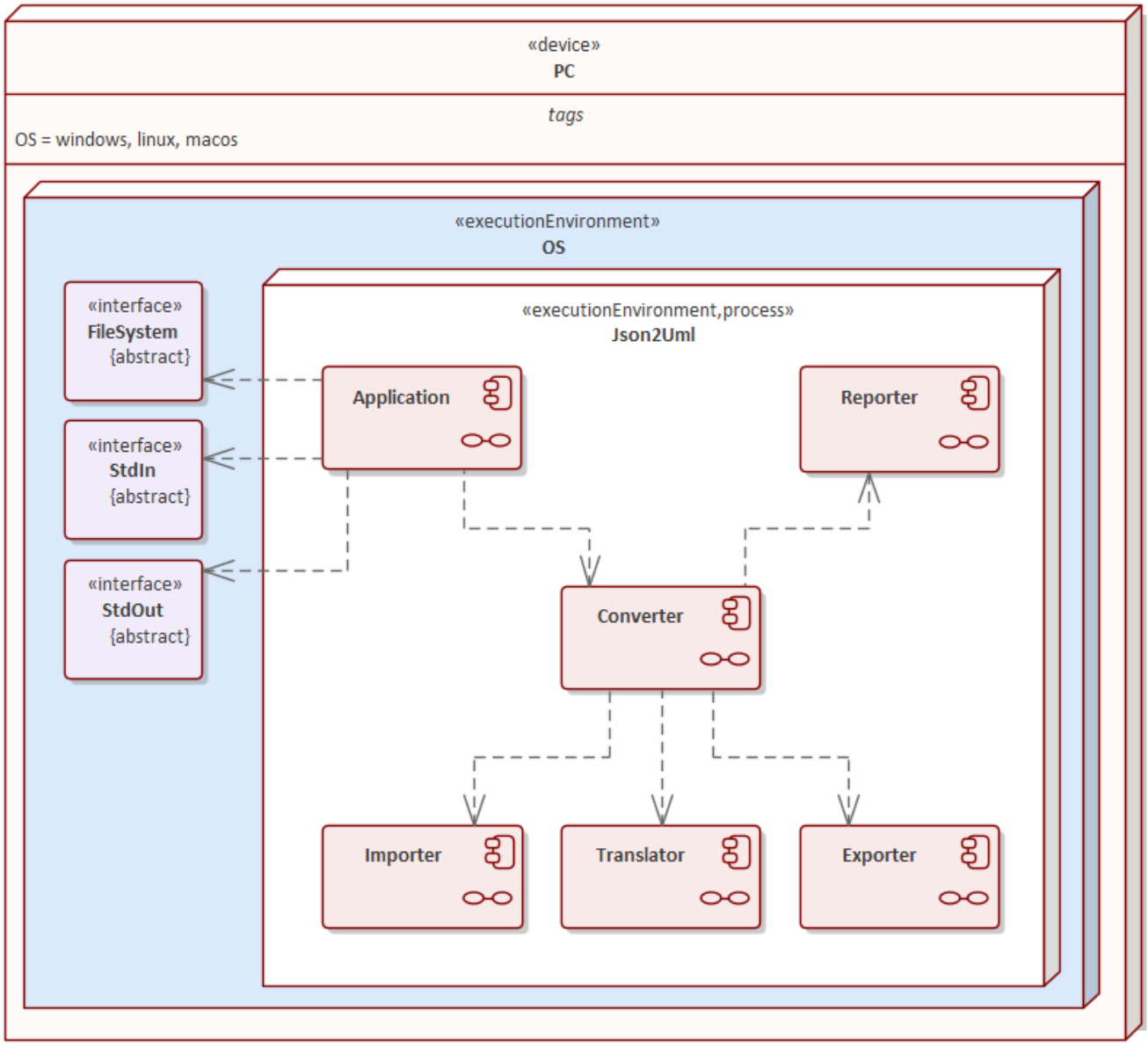
Утилита проектируется так, что все ее операции строго однопоточные. Это решение не может претендовать на хорошую утилизацию ресурсов, но зато заметно упрощается логика и исключаются многочисленные и трудноуловимые ошибки параллелизма. Следовательно, разделу Process View не требуется детализация.



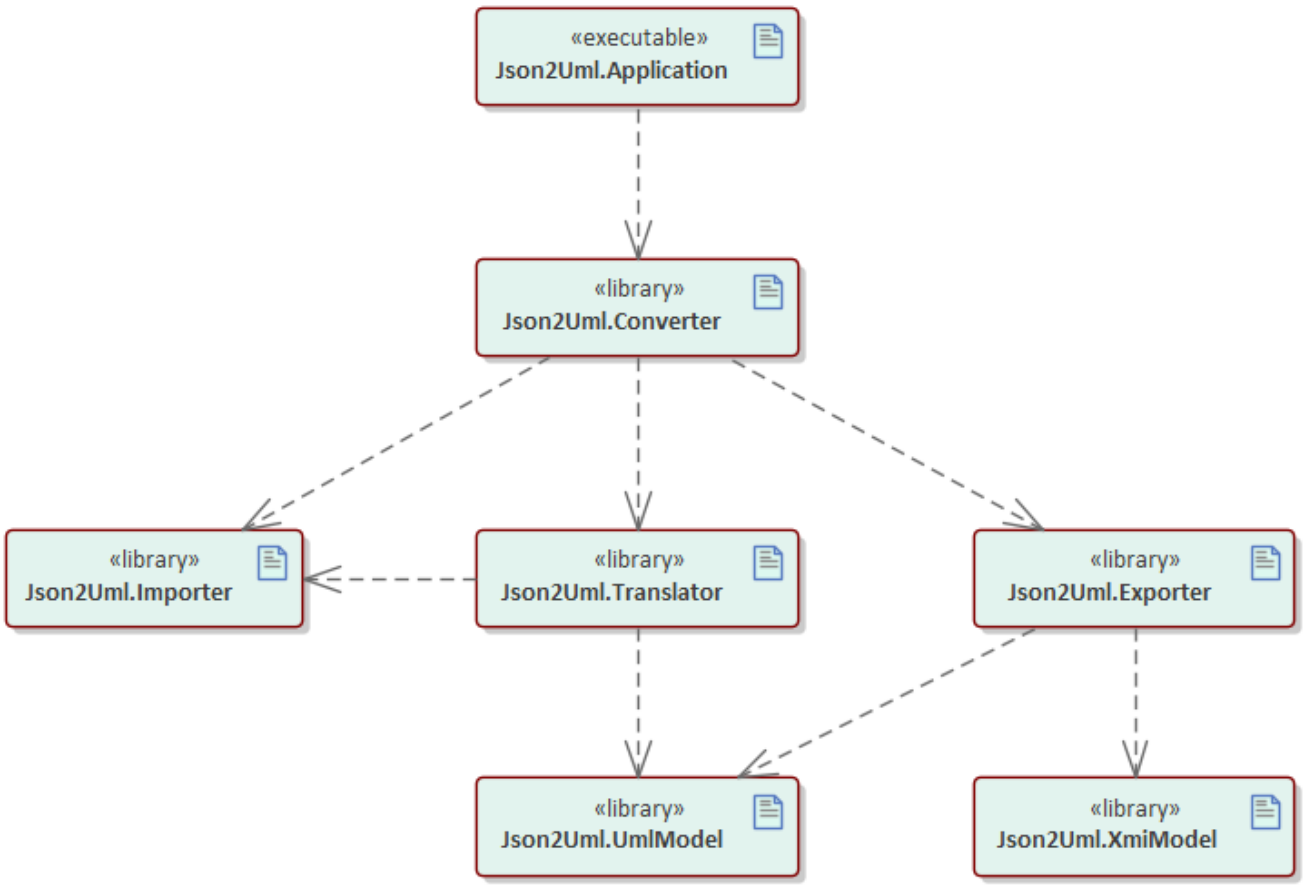
7. Представление развертывания (Deployment View)

Описание физических (сетевых, аппаратных) конфигураций, в которых будет развернута и запущена создаваемая система. Для каждой конфигурации должны быть указаны физические узлы (сервера, процессоры), выполняющие код, и их взаимосвязи (сетевые соединения, очереди, т.п.), отображение процессов на физических узлах.

Поектируемая консольная утилита, очевидно, запускается локально в командной строке. Следовательно, схема развертывания совсем проста: утилита представляет собой единственный процесс `Json2Uml`, запускающийся внутри операционной системы *windows*, *linux* или *macos*. Внутри памяти процесса располагаются основные компоненты утилиты. Компонент `Application` взаимодействует с API операционной системы: `FileSystem` API и `StdIn/StdOut`.



Физически компоненты располагаются в *NuGet*-пакетах, загружаемых во время сборки приложения. На диаграмме ниже показано, как *NuGet*-пакеты (а значит и компоненты внутри них) зависят друг от друга, что важно для правильной загрузки библиотек, правильной политики обновления и т.п.



8. Представление реализации (Implementation View)

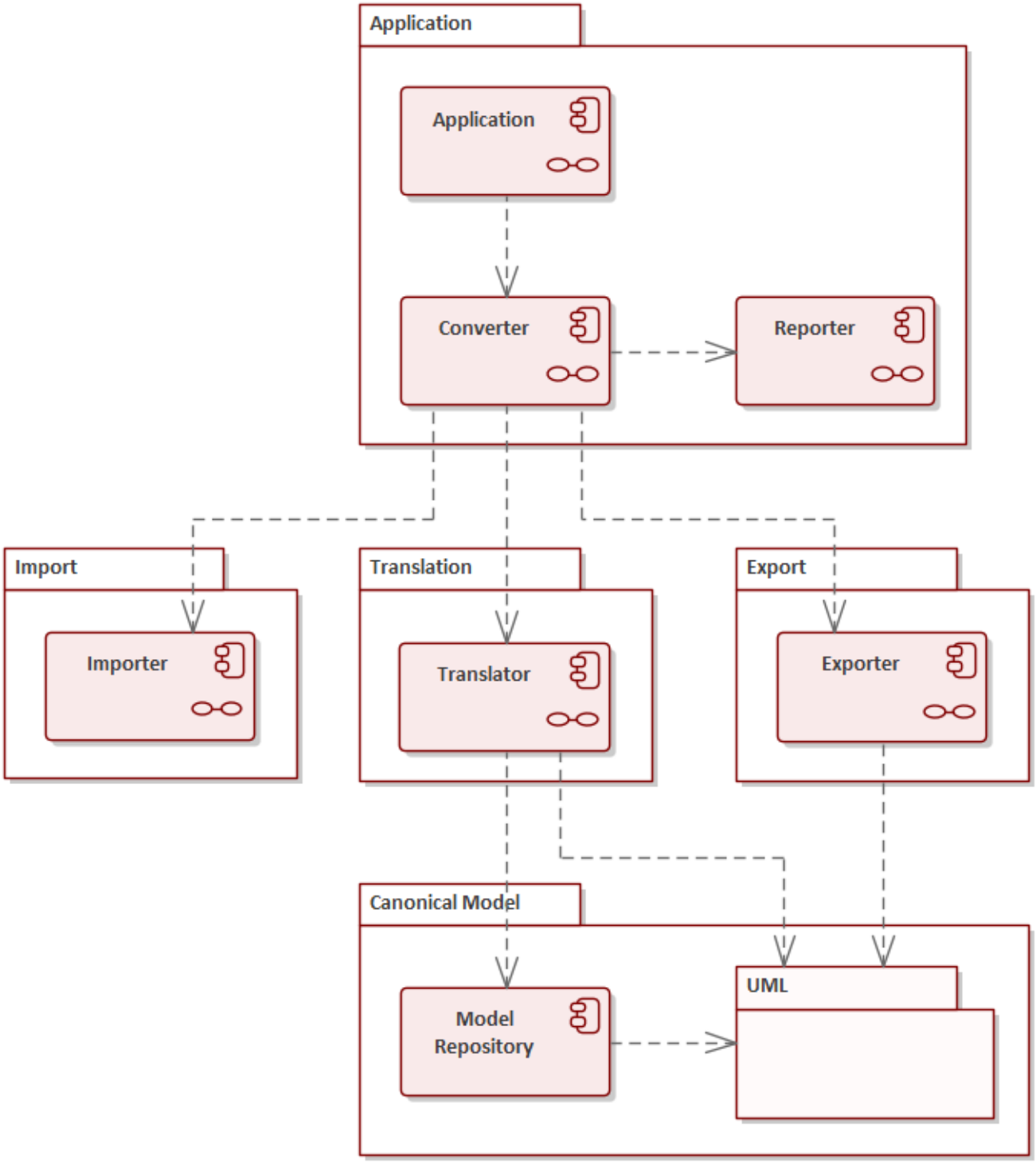
Описание общей структуры модели реализации, декомпозиция на уровни и подсистемы, любые архитектурно значимые элементы реализации.

8.1. Обзор

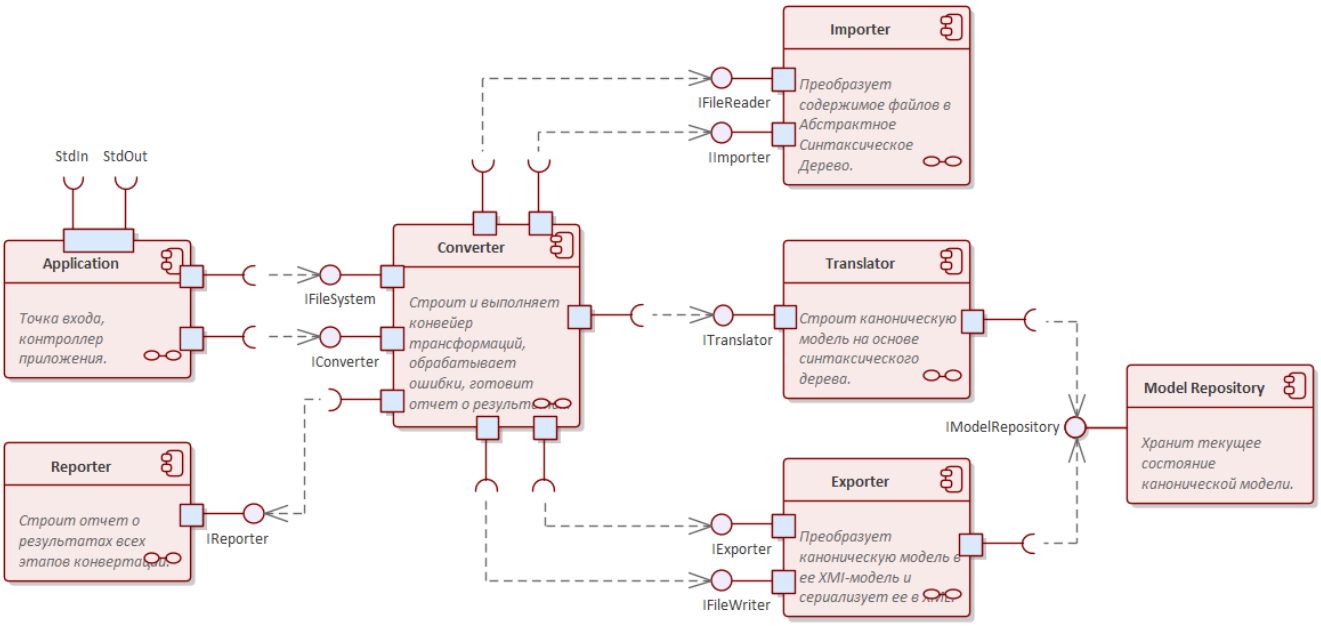
Описание основных уровней и слоев системы с точки зрения реализации, правила разделения на слои, границы между слоями. Имеет смысл описывать в виде диаграмм компонентов.

Не предусмотрено разделение артефактов системы на слои в классическом понимании. Компоненты системы, из которых строится реализация утилиты, распределны по основным пакетам, описанным в разделе 5.2.

Компоненты, отвечающие за запуск утилиты, взаимодействие с пользователем, настройки и т.п., расположены в пакете `Application`. Компоненты `Import, Translation, Export` расположены в соответствующих индивидуальных пакетах. В пакете `Canonical Model` хранится вложенный пакет UML с классами, представляющими элементы канонической модели, и компонент `Model Repository` - простое *in-memory* - хранилище для текущего состояния канонической модели.



Концептуальные классы, описанные в разделе 5, могут быть (не обязательно) представлены в разделе "Имплементация" в виде компонентов, имеющих четко описанный API в виде публикуемых (*provided*) и требуемых (*required*) интерфейсов. Так, можно выделить ряд компонентов, формирующих представление реализации утилиты.

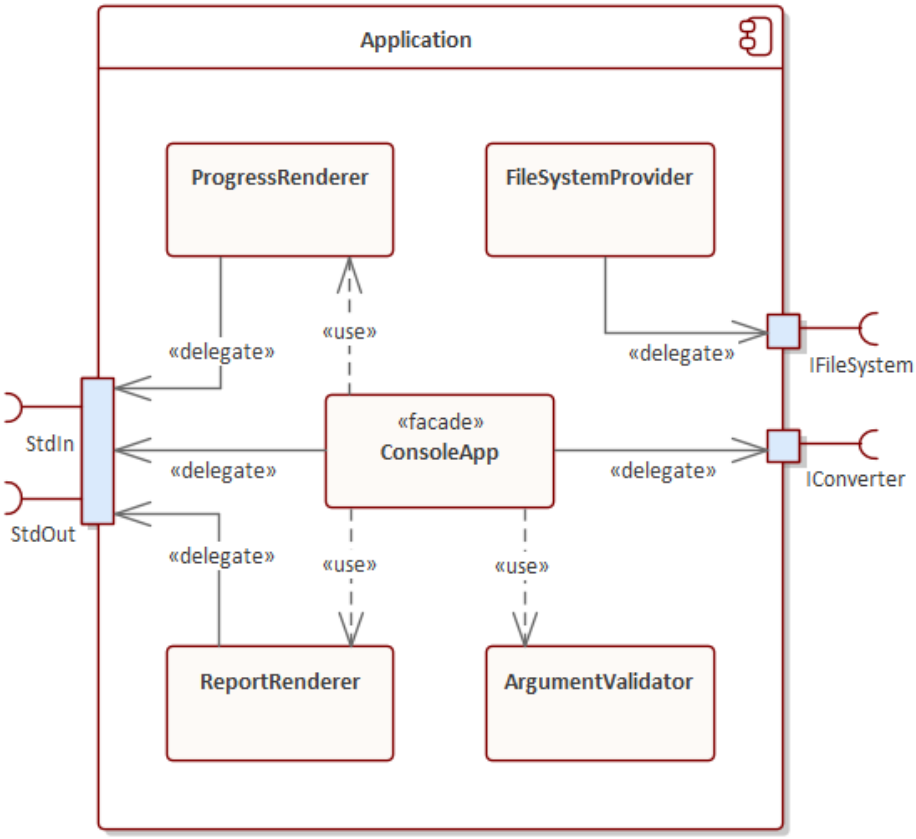


8.2. Слои

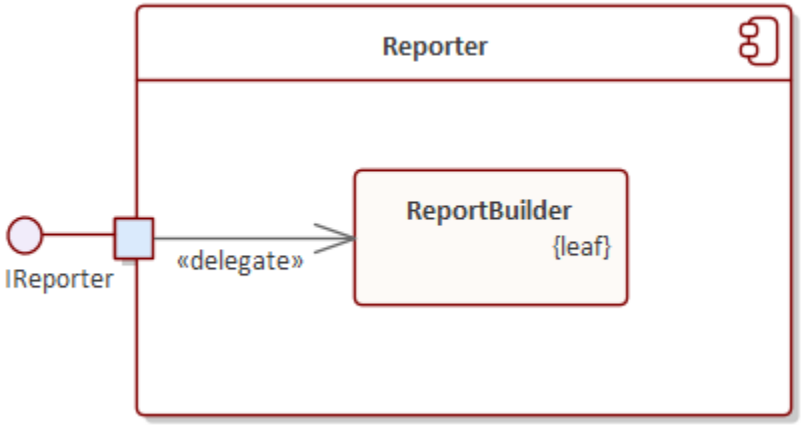
Для каждого уровня включить подраздел с его описанием, перечнем подсистем, диаграмму компонентов.

Перечень архитектурно-значимых компонентов:

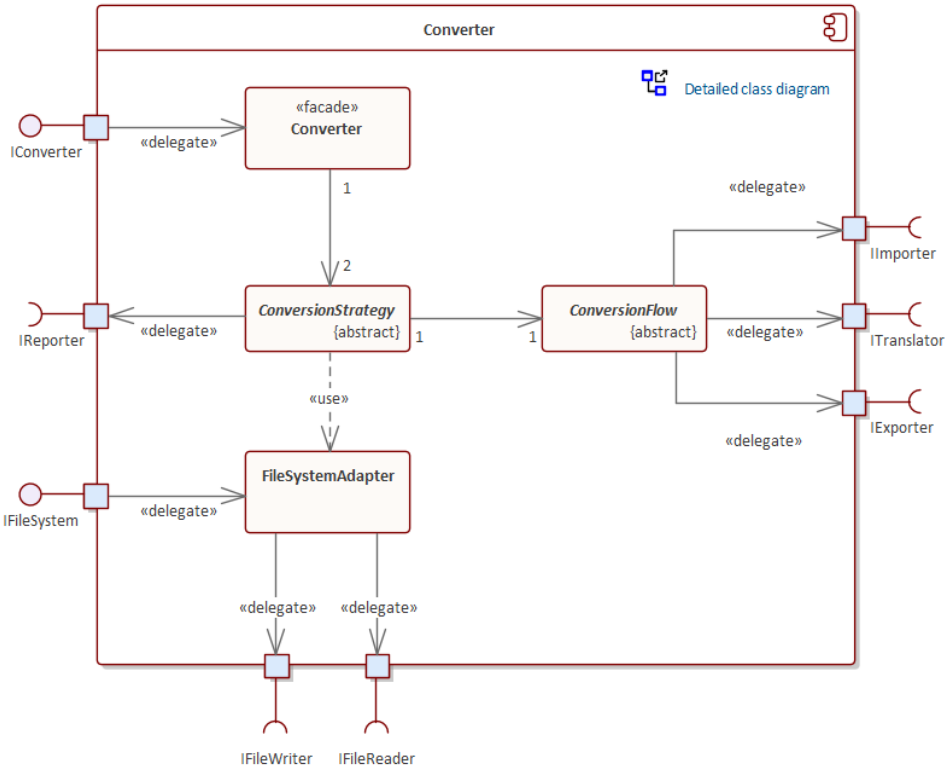
- **Application** - точка входа, контроллер приложения.
  Фасадный класс - `ConsoleApp`, формирующий консольное приложение. Он взаимодействует с окружением через интерфейсы `StdIn` (для получения аргументов командной строки), `StdOut` (для вывода текстового ответа и логирования), и `IConverter` (API для взаимодействия с собственно основной бизнес-логикой утилиты). После выполнения конвертации через `IConverter` и получения ответа в виде экземпляра `Report` класс `ConsoleApp` (для формирования легко читаемого пользователем отчета) обращается к `ReportRenderer`, а он в свою очередь выводит результат на консоль через `StdOut`.
  Отдельно существует класс `FileSystemProvider` - он адаптирует интерфейс `IFileSystem` (из компонента `Converter`) к файловому API операционной системы. Это нужно для того, чтобы только компонент `Application` зависил от окружения и инфраструктуры, а все остальные компоненты были полностью изолированы и общались только через интерфейсы.



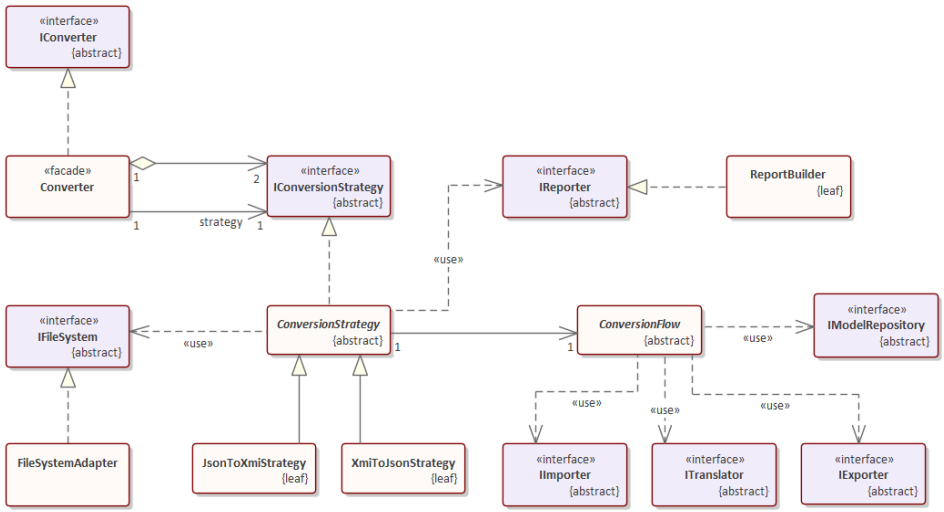
- **Reporter** - строит итоговый отчет о выполнении процесса конвертации.
  Класс `ReporterBuilder` (через интерфейс `IReporter`) предоставляет компоненту `Converter` возможность вернуть отчет вызывающему компоненту `Application` для последующего рендеринга в `StdOut`. Такой отчет агрегирует все ошибки и предупреждения, вызванные конвертаций каждого элемента моделей.



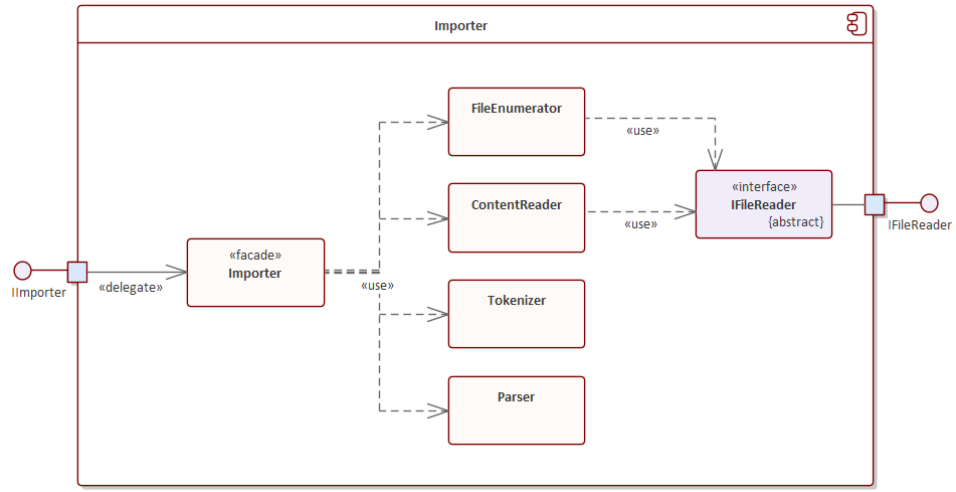
- **Converter** - компонент для самой логики конвертации.
  Фасадный класс `Converter` выбирает одну из двух стратегий `ConversionStrategy` и делегирует ей задачу конвертации. Конкретный класс-наследник `ConversionStrategy` содержит настроенный конвейер обработки данных (собственно конвертации) в классе `ConversionFlow`, который управляет работой компонентов `Importer`, `Translator` и `Exporter` через их API.
  Класс `FileSystemAdapter` отделяет интерфейсы `IFileWriter` и `IFileReader`, нужные компонентам `Importer` и `Exporter`, от файловой системы и делегирует вызовы компоненту `Application`, который и обращается к файловой системе напрямую.



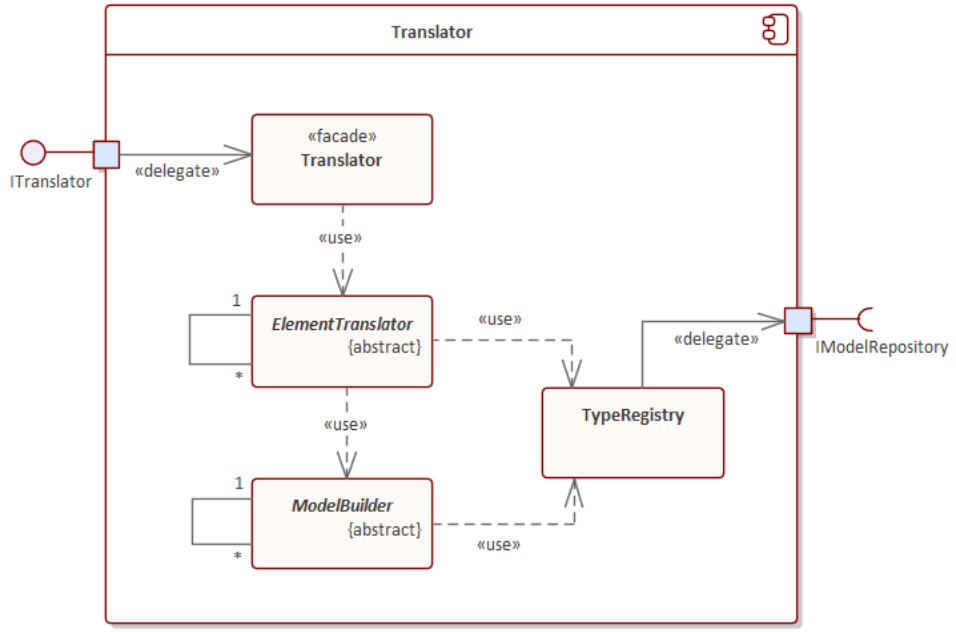
Ниже на диаграмме классов более детально показано, как предполагается организовать конвейер конвертации:



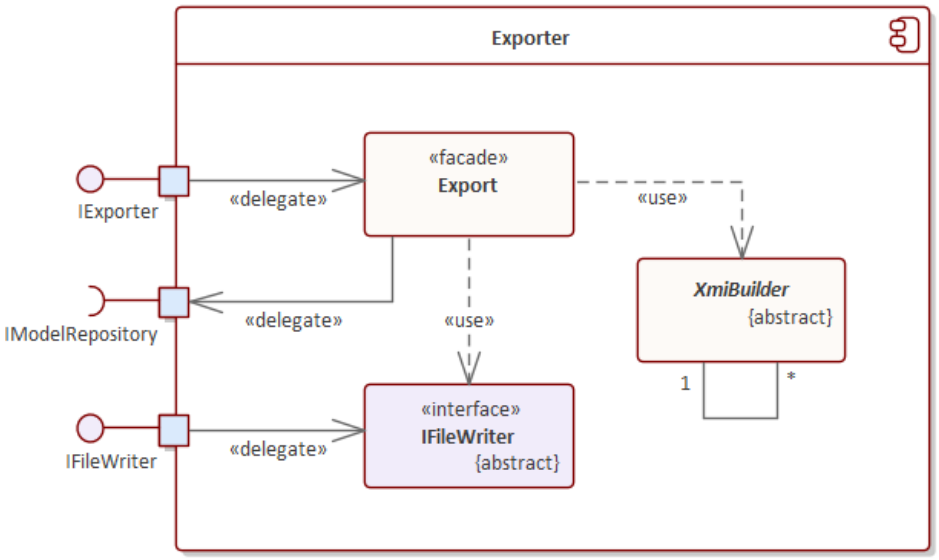
- **Importer** - преобразует содержимое импортируемых файлов в абстрактное синтаксическое дерево.
  Фасадный класс `Importer` выстраивает в цепочку вызовы классов `FileEnumerator`, `ContentReader`, `Tokenizer` и `Parser` таким образом, чтобы на вход следующего класса поступал результат предыдущего и в итоге класс `Importer` возвращал абстрактное синтаксическое дерево `AbstractSyntaxTree`.
  Класс `FileEnumerator` возвращает коллекцию файлов, которые нужно затем загрузить, класс `ContentReader` - считывает содержимое отдельного файла в тектовый буфер/поток, класс `Tokenizer` превращает содержимое буфера в набор токенов (очищенных от мусора семантически-значимых элементов), класс Parser анализирует токены и строит на их основе синтаксическое дерево. Таким образом, на выходе из Importer получается коллекция из `AbstractSyntaxTree`, представляющая коллекцию импортированных файлов.



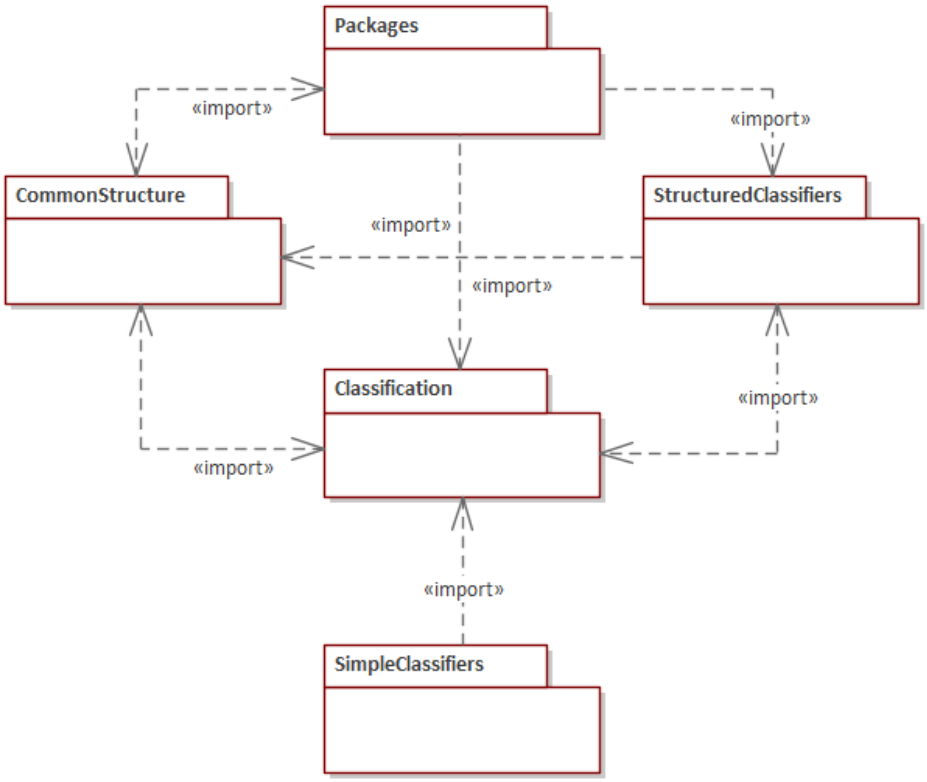
- **Translator** - строит каноническую модель на основе синтаксических деревьев.
  Фасадный класс `Translator` получает на вход коллекцию синтаксических деревьев (`AbstractSyntaxTree`), каждое из этих деревьев (представляющее сущность информационной модели) превращает в сущность промежуточной канонической модели (`CanonicalModel`) и дополняет ею текущую каноническую модель и подстраивает необходимые отношения с другими сущностями. То есть существует одно текущее состояние канонической модели, содержащее множетсво сущностей и связей, и каждое новое абстрактное синтаксическое дерево инкрементально дополняет каноническую модель новой сущностью.
  Класс `Translator` делегирует преобразование синтаксического дерева сущности агрегату, построенному из наследников `ElementTranslator`, каждый из которых предназначен для трансляции отдельных семантически-значимых частей дерева - самого типа, его атрибутов, отношений (ассоциаций, наследования), комментариев, ограничений и т.п. Каждый из подклассов `ElementTranslator` в свою очередь использует наследников `ModelBuilder`, предоставляющий *fluent api* для удобного конструирования элементов канонической модели.
  Класс `TypeRegistry` используется классами `ElementTranslator` и `ModelBuilder` для нахождения уже ранее созданных элементов, нужных для построения отношений, или для регистрации только что созданного элемента. Класс `TypeRegistry` использует интерфейс `IModelRepository`, принадлежащий пакету `Canonical Model`, для сохранения в оперативной памяти текущего состояния модели.



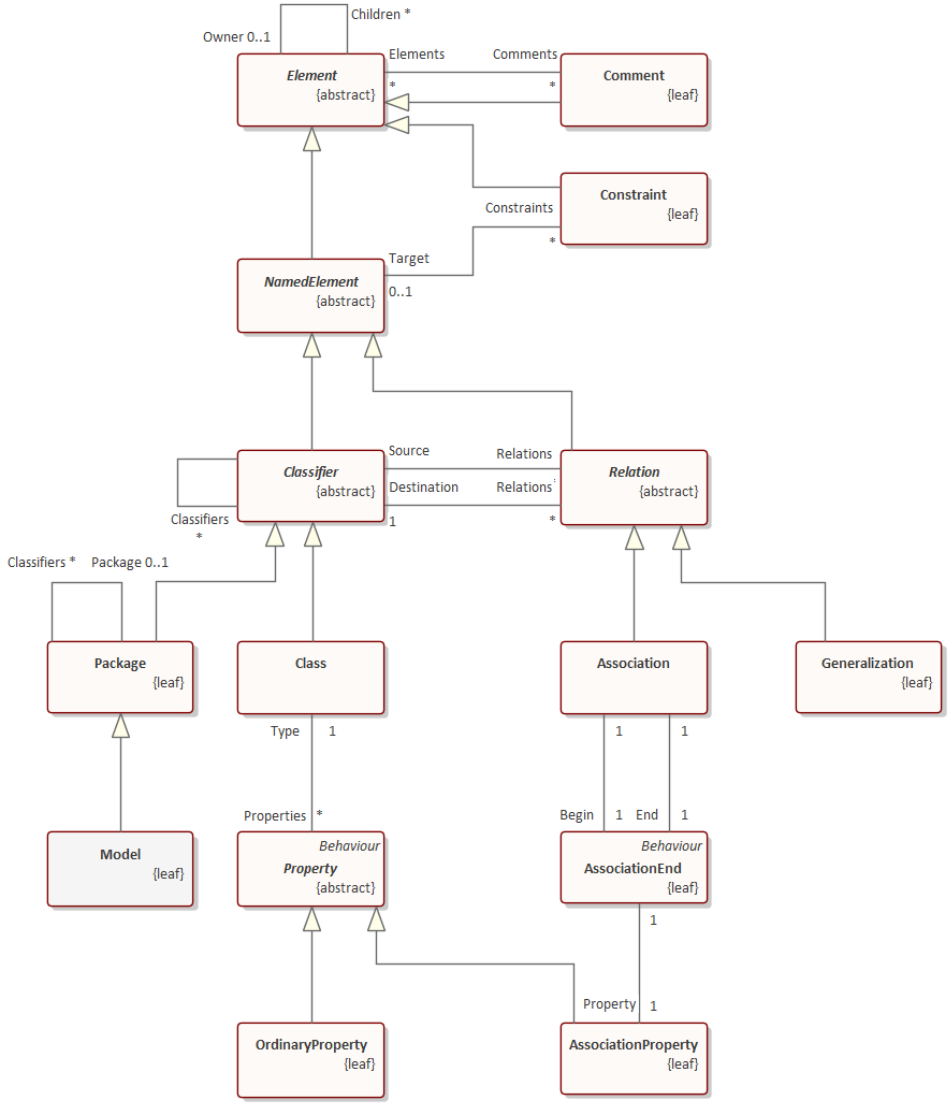
- **Exporter** - преобразует каноническую модель в XMI-представление и сохраняет в XML-файл.
  Фасадный класс Exporter делегирует обязанности преобразования агрегату из наследников абстрактного класс `XmiBuilder`, каждый из подклассов которого предназначен для конвертации в элемент модели XMI определенного элемента канонической модели, а в результате получается полная XMI-модель. Затем класс `Exporter` сериализует XMI-модель в XML-представление (подклассы `XmiBuilder` сами должны сериализоваться) и сохраняет в файл при помощи интерфейса `IFileWriter`.



- **Model Repository** - хранит текущее состояние канонической модели.
  Реализация тривиальна и в описании не нуждается.
  Компонент `Model Repository` использует классы канонической модели. Каноническая модель представляет собой переработку и упрощение метамодели UML. Ниже показаны пакеты, содержащие элементы метамодели:



Классы метамодели позволяют построить модель типов, их атрибутов, ассоциаций, наследования, ограничений и т.п. Ниже приведена верхнеуровневая диаграмма классов метамодели:



9. Представление данных (опционально)

Описание схемы хранения данных. Раздел не является обязательным и наполняется в случае, если хранение данных нетривиально и значимо с точки зрения общей архитектуры.

10. Размер и Производительность

Описание основных характеристик системы в плане объема данных и требований к производительности, особенно влияющих на архитектуру.

Предполагается, что утилита будет работать с файлами, содержащими модель, относительно небольшого размера и вполне умещающиеся в оперативной памяти типичного персонального компьютера. Следовательно, утилите не требуется промежуточное хранилище данных (БД).

Исходя из предыдущего параграфа, специальные требования к производительности также не предъявляются. Консольная утилита должна выполнять операции достаточно быстро, но при этом должна быть спроектирована в однопоточном стиле.

11. Качество

Описание того, как архитектура системы отражает влияние нефункциональных требований и атрибутов качества, таких, как безопасность, масштабируемость, надежность, повторное использование и т.п.

11.1. Отражение нефункциональных требований

Здесь нужно привести ссылки на нефункциональные требования, перечисленные в Supplementary Spec, и для каждого из требований описать, как оно отражается в архитектуре.

11.2. Дополнительные требования имплементации

11.2.1. Обработка ошибок

Поскольку основа организации логики конвертации - конвейер, то классический ООП-подход - обработка исключений - не подходит. Хорошим решением выглядит применение паттерна *Optional*, то есть возврат из ключевых методов конвейера специального объекта, содержащего либо результат метода, либо ошибку. Такой подход позволит построить конвейер в функциональном стиле.

Неустранимые же ошибки должны быть обработаны классическим образом: принудительно закрыты открытые файлы, освобождены другие ресурсы, исключение записано в лог и `StdOut` и приложение закрыто с соответствующим кодом возврата.

11.2.2. Управление зависимостями

Для организации связей между отдельными компонентами должен применяться принцип инверсии управления: сопоставление контрактов и их имплементаций должно быть перенесено в отдельный модуль `CompositionRoot`, расположенный в пакете `Application`. Взаимодействовать с окружением и ОС должен только компонент `Application`, остальные же компоненты должны делегировать ему эти обязанности через соответствующие интерфейсы.