Министерство Образования Российской Федерации
Нижегородский государственный университет им. Н. И. Лобачевского

Факультет Вычислительной Математики и Кибернетики
Кафедра Математического Обеспечения ЭВМ

Дипломная работа

Новая библиотека классов Colorer на базе XML технологий

Научный руководитель:
проф. кафедры МО ЭВМ
д. т. н. **Кетков Ю. Л.**

Рецензент:
Зав. кафедрой ННГАСУ
д. т. н. **Ротков С. И.**

Исполнитель:
студент группы 855
Русских И. В.

Допустить к защите в ГАК
/Зав. кафедрой МО ЭВМ, профессор Стронгин Р.Г./

_____

Нижний Новгород
2003

Содержание

1. Введение, постановка задачи

Подсветка и выделение основных синтаксических элементов в исходных текстах редактируемых программ облегчает их понимание, уменьшает вероятность ошибок и опечаток в тексте. Возможность расцветки синтаксиса языков программирования существует в большинстве интегрированных сред и специализированных текстовых редакторов. Основной недостаток расцветки синтаксиса в этих системах – поддержка фиксированного количества языков, отсутствующие или слабые возможности настройки цветового выделения и глубокая связь, зависимость от самой системы редактирования.

Существуют более мощные и универсальные системы редактирования текстов, которые не привязаны к конкретному языку, а позволяют настроиться на произвольный язык программирования. В большинстве из них для задания правил синтаксической раскраски текстов используются специальные настройки различной степени сложности и гибкости, но интегрированные и написанные специально для своего редактора.

Целью этой работы является создание и дальнейшее развитие специализированной библиотеки синтаксического анализа **«Colorer»**, обеспечивающей расцветку текстов на различных языках программирования. Библиотека должна обеспечивать максимальную гибкость в конфигурации, настраиваться на работу со всевозможными существующими языками программирования и не зависеть от архитектуры какой-то конкретной системы редактирования.

Исходя из анализа различных средств редактирования и интегрированных сред разработки, к создаваемому коду библиотеки предъявляется много требований. Основное из них – максимально возможная **переносимость кода** библиотеки и возможность интеграции его в существующие приложения. Это позволит использовать библиотеку с приложениями, написанными и работающими под различными аппаратными и программными платформами, что является неоспоримым преимуществом.

Основная цель – реализация функционирования библиотеки на платформах Windows и Unix/Linux с процессорами архитектуры x86. Для ее достижения основным средством написания был выбран язык C++ стандарта ANSI. Как и язык C, он широко распространен, его компиляторы существуют практически для всех программных и аппаратных платформ. В то же время, объектно-ориентированные возможности языка позволяют реализовать гибкую внутреннюю архитектуру системы, свободную от ограничений и недостатков структурных языков программирования. Еще одна причина заключается в том, что библиотека должна предоставлять приложению, использующему ее, удобный и расширяемый интерфейс для взаимодействия. Классы языка C++ с их возможностями наследования, полиморфизма, виртуальных функций, наилучшим образом подходят для решения поставленной задачи.

В дальнейшем, при анализе требований современного программного обеспечения, выяснилось, что помимо основного интерфейса, основанного на классах C++, требуется создание и тестирование более гибких интерфейсов, которые будет возможно использовать из различных языков программирования и которые не будут зависеть от особенностей конкретного языка. В качестве альтернативной платформы для функционирования библиотеки был выбран язык **Java**. Основной целью при этом была реализация интерфейса между базовой частью на языке C++ и высокоуровневыми классами языка Java. При этом, возможно, некоторые из компонентов библиотеки окажутся переносимыми и их можно будет полностью реализовать в среде Java.

Доступность интерфейса для языка Java предоставит огромные возможности при внедрении библиотеки в реальные системы, так как виртуальная машина Java представляет удобную объектную модель всех классов системы, и является полностью

объектно-ориентированной средой. Неоспоримым преимуществом ее является полная независимость от аппаратной и программной исполняемой платформы. Байт-код Java, единожды скомпилированный, выполняется везде. Это позволяет писать приложения, не задумываясь о конкретной среде их функционирования. На языке Java реализованы различные комплексные среды разработки, которые очень популярны у программистов. Ориентирование на них позволит расширить область применения библиотеки, реализовать поддержку новых возможностей и языков.

Хотя использование языка Java с компонентами кода на C++ и привязывает реализацию к конкретной платформе, в случае с библиотекой Colorer это не играет никакой роли. Все основные модули библиотеки полностью независимы от платформы, и компилируются под любой операционной системой. Таким образом, включение в Java-интерфейс версий библиотеки, скомпилированных под различными платформами, позволит добиться полной независимости от архитектуры системы. Этому будет способствовать универсальный интерфейс JNI, применяемый Java-машинами для связывания Java и native-кода (родного исполняемого кода платформы). Использование этого интерфейса позволяет не задумываться о внутренних особенностях виртуальной машины, его поддерживают все существующие реализации языка Java на всех аппаратных платформах.

Из необходимости отображения архитектуры библиотеки на классы языка Java вытекают некоторые особенности использования классов C++, которые придется учитывать в процессе разработки. Помимо Java практически все современные языки программирования предоставляют в распоряжение программиста объектно-ориентированные методы построения программ, представляющие компоненты в виде интерфейсов, абстрактных классов и способов их взаимодействия. В связи с этим важной особенностью структуры API библиотеки Colorer будет являться возможность прямого ее отображения на стандартные понятия объектно-ориентированных языков и различных спецификаций. При рассмотрении возможностей других языков и технологий (C#, COM, IDL) прослеживаются общие подходы к объектно-ориентированным принципам построения программного обеспечения и интерфейсов прикладного программирования. Многие возможности языка C++ не укладываются в эти рамки. Они позволяют реализовать более гибкие формы классов, объектов (как, например, множественное наследование). Иные, наоборот, не учитывают необходимости в некоторых формах абстракции, таких как формальное определение интерфейсов.

Для преодоления этих неудобств и обеспечения прямого переноса объектов и определений из C++ классов в классы и описания других языков на используемые в библиотеке возможности языка C++ наложены некоторые ограничения. В частности, интерфейсные объекты описываются классами C++ с абстрактными виртуальными объявлениями методов, не используется множественное наследование, за исключением случаев прямого наследования параллельно с реализацией интерфейсных классов. Помимо этого, во всех компонентах библиотеки широко используется обработка исключений, динамические преобразования классов. Большинство базовых понятий (символьные данные, файловый ввод/вывод, основные примитивы) представлено в виде абстрактных объектов, что упрощает перенос библиотеки на другие языки и отделяет ее от особенностей программно-аппаратной платформы.

Реализация строгих отношений между объектами системы и полноценная документация их поведения являются важнейшими ступенями в создании любого программного продукта. В случае с библиотекой Colorer для этого применяются системы встроенного документирования. Для исходных кодов на языке Java существует стандартное средство само-документирования `javadoc`, а исходные коды основной части библиотеки на языке C++ обрабатываются системой `doxygen`, возможности которой так же довольно широки.

Для анализа и расцветки текстов на произвольных языках программирования библиотека должна обеспечивать приемлемую форму описания их синтаксиса и конструкций, подлежащих подцветке. В библиотеке Colorer для этого используется специально разработанный язык **HRC** (Highlighting Resource Codes). Он позволяет с помощью различных средств описания структуры текста сопоставлять его частям цветовые и синтаксические регионы, которые интерпретируются библиотекой и отображаются визуально пользователю в виде подцвеченного содержимого файла. Язык HRC библиотеки Colorer должен обладать возможностями, позволяющими анализировать и расцвечивать структуру большинства существующих языков программирования. Основным инструментом в этом процессе являются регулярные выражения, анализатор которых реализован в виде модуля библиотеки.

Язык HRC развивался довольно долго, и в реализованной версии библиотеки он представляет собой мощный и гибкий инструмент анализа. Его собственный синтаксис основывается на стандарте языков разметки **XML** и определяет набор тэгов, атрибутов и их содержимого, которые описывают структуры и данные для синтаксического анализа. Использование XML в качестве основы хранения информации дает неоспоримые преимущества. Открытость формата позволяет использовать различные средства обработки и анализа содержимого XML-документов, трансформировать его в иные формы и представления. Работа с HRC на основе XML дает дополнительные преимущества в возможности сторонней проверки синтаксиса модулей с использованием описаний DTD или XML схем.

Одной из главных поставленных задач этой работы является тесное слияние библиотеки Colorer с XML-технологиями и оптимальное их использование. Основная идея состоит в том, что на основе описаний схем произвольных XML языков возможно **автоматически генерировать** соответствующие им описания HRC, которые будут использоваться в библиотеке для расцветки синтаксиса этих исходных XML языков. Такая архитектура позволит не задумываться о проблеме ручного создания и написания HRC-определений для каждого конкретного языка, а это очень важно, так как сегодня все больше и больше языков программирования, форматов разметки и структур данных используют XML как основу своего синтаксиса. Например, такие распространенные стандарты, как XHTML, XSLT, XSL-FO, MathML могут быть представлены в синтаксисе HRC автоматически. При этом библиотека сможет осуществлять проверку структуры этих документов «на лету» – некорректная вложенность элементов, неправильные значения атрибутов, нестандартные символы – все эти ошибки пользователь сможет контролировать еще на стадии набора текста.

Многие системы редактирования, работающие с XML-данными, часто используют визуальное представление документов. При этом организуется графический интерфейс, который определенным образом абстрагирует пользователя от реального содержимого документа. Такие модели представления довольно часто встречаются в синтаксически сложных языках, которые из-за своей сложности и громоздкости не совсем удобны для прямого текстового редактирования, и в то же время допускают удобную и естественную визуальную интерпретацию. Это, например, язык XML Schema, язык описания математических формул MathML, язык визуальной разметки текста XSL-FO.

Но, несмотря на это, библиотека Colorer занимает свою нишу в среде такого программного обеспечения. Довольно часто ручное редактирование и создание даже таких синтаксически сложных документов является единственным возможным вариантом. Так, язык трансформаций XSLT, не смотря ни на что, не может быть полноценно заменен какими-либо визуальными средствами – ручное редактирование кода здесь остается основным и единственным вариантом. Возможности, которые при этом сможет предоставить пользователю библиотека Colorer, намного облегчают задачу программиста.

Удобное визуальное разделение синтаксических конструкций, выделение цветом важных частей в тексте, мгновенное отображение большинства синтаксических и логических ошибок с одной стороны сочетают мощь и гибкость ручного редактирования кода, а с другой – действуют идентично визуальным средствам – позволяют сконцентрировать внимание автора на главном и уменьшают влияние сторонних помех (опечаток, ошибок).

Для реализации всех этих возможностей библиотека должна объединять в себе использование нескольких XML-стандартов, связывание которых позволяет достичь желаемых результатов. В первую очередь, это спецификация XML Schema и язык трансформаций XSLT. На основе трансформаций деревьев XML документов из исходного описания схемы документа в формате XML Schema создается конечное представление HRC-кода, который уже может использоваться библиотекой. Создание такого преобразования должно учитывать многие аспекты структуры XML схем и уметь переводить их в нужные синтаксические объекты языка HRC.

Конечной целью работы будет получение законченной библиотеки классов, использующей передовые информационные технологии, готовой к встраиванию в реальные функционирующие приложения. Созданная архитектура должна обеспечивать достаточную модульность, гибкость и расширяемость всех ее составляющих.

2. Технологии XML

До 1998 года обмен данными и документами был ограничен тем, что форматы документов либо находились в собственности своих владельцев, либо были неточно определены. Появление HTML – языка разметки для отображения интерактивных данных в Web-браузере – предоставило стандартный формат для обмена данными, сфокусированный на интерактивном визуальном содержании. Однако HTML является строго определённым языком и не может поддерживать все корпоративные типы данных. Эти недостатки послужили толчком к созданию **XML** (Extensible Markup Language - Расширенный Язык Разметки). Стандарт XML позволяет определять свои собственные языки разметки с акцентом на специфичных целях, таких как электронная коммерция, управление данными и публикациями.

По этим причинам XML быстро становится стратегическим инструментом определения данных в многочисленных областях использования приложений. Свойства разметки XML подходят для представления данных, концепций и контекстов открытым, независящим от платформы, разработчика и языка способом. Он использует теги – идентификаторы, которые сигнализируют о начале и конце блока связанных логически данных – для создания иерархии связанных компонентов данных, называемых элементами. В свою очередь эта иерархия предоставляет контекст и инкапсуляцию. В результате возникает возможность повторного использования этих данных вне приложения и вне источников данных, в которых данные впервые использовались.

Технология XML была успешно использована при создании зависящих от целевого назначения решений для обмена данными, публикаций и разработки программного обеспечения. Кроме того, XML стал стимулом для групп компаний внутри отдельных отраслей для совместной работы при определении специфичных для отрасли языков разметки (иногда называемых диалектами, словарями – *vocabulary*). Эти инициативы создали фундамент для совместного использования информации и обмена ею внутри целых областей вместо использования закрытых индивидуальных решений и форматов хранения данных.

2.1. Спецификация XML

2.1.1. Базовый синтаксис

Язык XML является синтаксисом для разработки специализированных языков разметки, которые добавляют идентификаторы (тэги) к отдельным символам, словам или фразам внутри документа. Эти идентификаторы позволяют впоследствии распознавать части документа и совершать над ними какие-либо действия в процессе будущей обработки. Результатом разметки документа или данных является формирование независящего от платформы, языка и разработчика иерархического контейнера, отделяющего содержание документа от среды, которая может его обрабатывать.

XML – в высшей степени платформо-независимый способ структуризации информации. Документ XML – это дерево элементов. Элемент может иметь набор атрибутов, в форме пар <ключ-значение>, и может содержать другие элементы, текст или их комбинацию. Элемент может ссылаться на другие элементы посредством специальных атрибутов, позволяя, таким образом, представлять произвольные структуры графов.

Эта базовая структура XML-документа позволяет формировать, хранить и обрабатывать произвольные данные в зависимости от их содержимого и назначения. Так, например, следующий XML-документ содержит информацию о произвольном адресе:

```
<address>
  <name form="full">Alice Smith</name>
  <street>123 Maple Street</street>
  <city>Mill Valley</city>
  <state>CA</state>
  <zip>90952</zip>
</address>
```

В общем случае элементы и их содержимое могут быть произвольными, они могут содержать произвольные атрибуты. Основное ограничение, которое накладывается на синтаксис XML – необходимость правильной вложенности элементов, отсутствие незакрытых тэгов, неправильно сформированных пар имя/значение у атрибутов элементов. Хотя элементы, их атрибуты и обычные текстовые данные и являются основными синтаксическими элементами в разметке XML, существуют дополнительные конструкции, служащие для упрощения работы с разметкой документа, поддержкой дополнительных свойств и атрибутов. К ним относятся комментарии, инструкции обработки (processing instruction), сущности (entities) и определения типов данных (DTD).

Структура XML документа не должна следовать каким-либо правилам, кроме тех, которые заложены в спецификации XML. Однако, для обмена документами значимым образом, требуется, чтобы их структура описывалась и ограничивалась так, что различные составляющие части интерпретировали ее корректно и единообразно. Этого можно достичь путем использования схемы (определения типов данных документа). Схема состоит из набора правил, ограничивающих структуру и содержимое компонентов документа, т.е. их элементов, атрибутов и текста. Схема также описывает (по меньшей мере неформально и часто неявно) подразумеваемое концептуальное значение компонентов документа. Другими словами, схема – это спецификация синтаксиса и семантики (потенциально бесконечного) набора XML документов. Документ действителен по отношению к схеме, если и только если он удовлетворяет ограничениям, описанным в схеме.

Первая реализация принципов соответствия XML документов схеме описывается в самом стандарте XML и является его неотъемлемой частью. Определения DTD (Data Type Definitions) используются для описания структуры документа и содержимого его элементов и атрибутов. Для описания ограничений используются два основных оператора: `ELEMENT` и `ATTLIST`. Первый описывает содержимое XML элемента, второй – список его атрибутов и их возможные значения.

Для определения возможного содержимого элемента используется запись, определяющая последовательность либо чередование элементов в теле родителя:

```
<!ELEMENT address (name, street, city, state?, zip?)>
```

При этом возможно задание самых простых конструкций в виде последовательности, альтернативы, либо их суперпозиции. В качестве элементов определения выступают либо имена допустимых тэгов, либо специальный идентификатор, означающий символьные данные (`#PCDATA`).

У каждого элемента определяется список возможных атрибутов с помощью конструкции, подобной следующей:

```
<!ATTLIST name
  form            NMTOKEN #REQUIRED
>
```

Для каждого атрибута указывается его тип дополнительные параметры, которые определяют, является ли атрибут необязательным, устанавливается ли ему возможное значение по умолчанию либо фиксированное значение.

Определения DTD выполняют роль не только схемы документа. Из-за тесной связи с синтаксисом XML, они частично дополняют и определяют его содержимое. Это

коренное отличие DTD от других возможных языков схем, которые только описывают структуру уже сформированного документа.

Внешние и внутренние сущности позволяют использовать ссылки на них в содержимом элементов и атрибутов. Эти ссылки разворачиваются до содержимого сущности, и, тем самым, компактно представляют различные конструкции документа. Содержимое определяемой сущности и его использование в контексте документа не должно нарушать общей структуры документа. Если значение сущности – последовательность тэгов, то они обязаны удовлетворять ограничениям на корректную вложенность в пределах этого определения. Такое поведение необходимо для того, чтобы непроверяющие (non-validating) XML-процессоры могли обрабатывать содержимое документа с неразрешенными внешними сущностями. Следующий пример вводит определение внутренней сущности и ссылается на нее из тела документа:

```
<?xml version="1.0"?>
<!DOCTYPE address [
  <!ENTITY nameref "Alice Smith"!>
]>
<address>
  <name form="full"> &nameref; </name>
  <street>123 Maple Street</street>
  <city>Mill Valley</city>
  <state>CA</state>
  <zip>90952</zip>
</address>
```

Еще одна особенность описаний DTD – изменение информационной модели документа при проверке его на корректность. Это связано с возможностью описывать в определении атрибутов их значения по умолчанию. Так как это значение XML процессор может узнать только из анализа определений DTD, информационная модель может быть различной при использовании XML-процессоров с различными уровнями соответствия спецификации.

Одно из важных свойств стандарта XML – строгое определение порядка функционирования программ, обрабатывающих XML документы. Спецификация явно указывает все возможные ошибки, которые могут возникать при синтаксическом разборе, и определяет требуемое от программных средств поведение при их обнаружении. Ошибки и нестандартные ситуации при обработке разделяются на различные степени важности и могут приводить к различным результатам – начиная от простого информационного сообщения и кончая полным прекращением синтаксического анализа с выдачей сообщения об ошибке. Такая модель позволяет унифицировать обработку XML файлов и гарантировать, что различные применяемые программные средства будут приводить к одинаковым результатам. Это очень важно, так как в случае неоднозначной интерпретации документа его содержимое может кардинально менять свой смысл, а это недопустимо.

Полная поддержка спецификации XML во всех ее подробностях – это очень сложная задача. Сюда входит не только анализ структуры самого документа, но и проверка его на соответствие DTD определениям, разбор внешних сущностей, предоставление высокоуровневого прикладного интерфейса для доступа к разобранному документу. Многим приложениям может оказаться достаточным лишь некоторых основных действий по синтаксическому разбору, другим может быть не нужна проверка структуры документа на корректность, и т.д. Именно поэтому вводится два уровня соответствия XML-процессоров спецификации – правильно сформированный документ (**well formed**) и корректный (**valid**).

Первый уровень соответствия намного проще в реализации, так как не требует от XML-процессора поддержки таких возможностей, как чтение внешних определений,

анализ структуры на соответствие DTD, установка атрибутов по умолчанию, и т.д. Правильно сформированный документ лишь подразумевает, что структура его элементов и атрибутов корректна и доступна для однозначной интерпретации. Данные, хранящиеся в таком документе, могут быть обработаны однозначно и без каких либо потерь. При обработке XML-документа процессором, реализующим Validation уровень соответствия, проводится полный анализ его содержимого, включая определения типов данных, проверку структуры на соответствие DTD, чтение внешних определений и некоторые другие операции.

В случае если процессор не поддерживает ни один из этих двух уровней, он вообще не соответствует спецификации. Тогда не существует гарантий, что приложение, обрабатывающее этот документ получит именно те данные, которые изначально были сохранены в документе.

2.1.2. Пространства имен XML

В связи с широким распространением языков и форматов данных, основанных на XML, возникла задача по упорядочиванию взаимодействия между ними, улучшению интеграции между различными словарями, основанными на этом стандарте. Основная проблема, возникшая при этом, состояла в том, что различные, основанные на XML, типы документов могли использовать произвольные имена элементов (тэгов) и атрибутов. Из-за этого во многих случая могла возникнуть путаница. В частности, это неприемлемо при интеграции различных словарей в один документ, при работе прикладной программы с различными типами документов, которые надо каким-то образом распознавать и обрабатывать.

Для преодоления этой проблемы была разработана дополнительная спецификация пространств имен XML (**XML Namespaces**). Она незначительно расширяет базовый XML синтаксис и предоставляет возможность привязки множества имен элементов и атрибутов к определенному пространству имен. Такое пространство имен несколько отличается от традиционного представления в виде множества, так как содержит разнородные по своему типу компоненты. Каждое пространство имен идентифицируется уникальной строкой, представляющей собой URI-адрес. В последствии через этот идентификатор можно ссылаться как на само пространство, так и на компоненты, в нем содержащиеся.

Для привязки компонентов документа к определенному пространству имен используются так называемые префиксы пространств имен. Префикс – это короткий временный псевдоним, который используется при объявлении элементов и атрибутов и сообщает о том, что соответствующий компонент документа привязывается к пространству имен, ассоциированному с этим префиксом. Префикс отделяется от имени элемента двоеточием.

Такое соглашение хотя и требует большей работы со стороны анализатора, поддерживающего пространства имен, но предоставляет большую гибкость в работе. Помимо этого использование такого синтаксиса полностью совместимо с базовым стандартом XML – это значит, что приложения, ничего не знающие о пространствах имен будут иметь возможность обработки документов, содержащих такую информацию.

Для объявления соответствия префикса пространству имен используется специальный атрибут `xmlns`:

```
<ad:address xmlns:ad='http://address.example.com'>
  <ad:name>Alice Smith</ad:name>
  <ad:street>123 Maple Street</ad:street>
  <ad:city>Mill Valley</ad:city>
  <ad:state>CA</ad:state>
  <ad:zip>90952</ad:zip>
</ad:address>
```

Сам префикс не имеет никакой смысловой нагрузки кроме того, что служит простым псевдонимом для идентификации принадлежности к конкретному пространству имен. Например, два различных префикса могут ссылаться на одно и то же пространство. И использовать можно любой из них или оба сразу – конечное содержание документа сохранится.

Помимо явного указания префикса существует возможность указать пространство имен по умолчанию. Таким образом, можно привязывать компоненты документа к пространству имен не изменяя их. Например, следующий код полностью идентичен предыдущему:

```
<address xmlns='http://address.example.com'
         xmlns:xx='http://address.example.com'>
  <name>Alice Smith</name>
  <street>123 Maple Street</street>
  <xx:city>Mill Valley</xx:city>
  <xx:state>CA</xx:state>
  <xx:zip>90952</xx:zip>
</address>
```

Конечно, в одном документе можно использовать сразу несколько объявлений пространств имен и принадлежащих им элементам. Это позволяет интегрировать в один документ информацию из различных источников, которая может в дальнейшем идентифицироваться и обрабатываться специализированными для каждого из объявленных пространств средствами.

В отличие от имен элементов, имена атрибутов по другому используют технику привязки к нужному пространству имен. Это связано с тем, что существует несколько возможных вариантов использования атрибутов. Большая часть атрибутов привязана к элементам, в которых они содержатся. Такие атрибуты описывают свойства содержащих их элементов и не имеют смысла в списке атрибутов других элементов. Поэтому у этих атрибутов не указывается пространство имен, и по умолчанию их подпространство имен идентифицируется парой [`пространство имен элемента; имя элемента`], следствием чего является невозможность иметь в элементе два локальных атрибута с одинаковыми именами.

Но, иногда бывает удобно иметь возможность работы с атрибутами, которые могут описывать свойства любого тэга. Такие атрибуты называются глобальными и используются с явным указанием префикса пространства имен. В следующем примере глобальный атрибут `html:class`, указывающий на принадлежность элемента к одному из HTML-классов, используется вместе с локальным атрибутом `class`, имеющим совсем иное смысловое значение:

```
<reservation>
  <name html:class="LargeSansSerif">Layman, A</name>
  <seat class="y" html:class="important">33b</seat>
  <departure>1997-05-24t07:55:00+1</departure>
</reservation>
```

Имя такого глобального атрибута уникально в рамках всего пространства имен. Обычные же атрибуты могут иметь повторяющиеся имена, но при этом принадлежать различным элементам (и нести различную смысловую нагрузку).

Пространства имен XML имеют структуру, отличающуюся от традиционного понимания термина **namespace**. Пространство имен в обычных языках (таких как C++, Java, C#) определяет множество элементов (имен), уникальных в границах этого множества. В случае с документом XML его пространство имен можно определить как совокупностью трех множеств (разделов пространства имен), в каждом из которых содержатся элементы с уникальными именами:

- Множество возможных имен тэгов (элементов) документа. Каждый элемент документа уникально идентифицируется через URI пространства имен и локальную часть своего имени.
- Множество возможных имен глобальных атрибутов. Каждый такой атрибут глобален для всего документа и идентифицируется парой [URI;локальное имя атрибута].
- Множество пар [имя элемента; имя локального атрибута]. Определяет все возможные локальные атрибуты (без указания префикса), принадлежащие каждому из допустимых элементов.

Таким разбиением любое пространство имен XML сводится к традиционным, и имя любого компонента в документе попадает в один из указанных разделов.

2.1.3. Информационное множество XML

Содержимое любого XML-документа – это однозначно определенная структура хранения данных. Для своего представления она использует два типа компонент: элементы с их содержимым и привязанные к ним атрибуты. Если атрибуты могут содержать только текстовые данные, то содержимое элементов – это чередование обычного текста и других вложенных элементов.

Синтаксис языка XML – это лишь форма представления и описания таких данных. Фиксированный и однозначный синтаксис – большое преимущество. Оно позволяет различным средствам, работающим с языком, давать одинаковый результат. Как пример, XML можно сравнить с его обобщением – языком разметки SGML (Standard Generalized Markup Language). В этом языке можно свободно конфигурировать синтаксические средства описания структурной разметки, использовать различные формы записи одной и той же структуры данных. Например, приводимый ранее пример XML документа в SGML мог записываться в сокращенной форме:

```
<address>
  <name fullform>Alice Smith</>
  <street>123 Maple Street</>
  <city/Mill Valley/
  <state/CA/
  <zip/90952/
</>
```

SGML допускает различные смягчения и сокращения синтаксиса, которые запрещены в XML. Несмотря на такие различия, можно сказать, что оба этих языка описывают информацию одинаковой структуры и между их элементами можно провести однозначное соответствие. Такое абстрактное представление информации называется **информационным множеством XML** (XML Infoset). Оно описывает абстрактную структуру документа без привязки к конкретным средствам ее выражения (элементам, атрибутам, сущностям подстановки).

Понятие информационного множества очень важно, так как оно определяет, какие компоненты в документе несут значимый смысл, а какие могут использоваться лишь как вспомогательные механизмы разметки. Информационное множество определяет структуру возможного содержимого документа через введение информационных элементов (information item), каждый из которых имеет набор атрибутов. В понятие

атрибута информационного элемента включаются различные данные, в зависимости от его типа и структуры (Табл. 1).

Табл. 1. Элементы информационного множества XML.

Информационный элемент	***Значение и атрибуты***
Документ	Единственный корневой объект, содержащий в себе ссылки на все другие элементы. Его основными атрибутами являются упорядоченный список дочерних элементов, множество определенных нотаций и неразобранных сущностей.
Элемент	Свойства XML-элемента, и его содержимое, заключенное между начальным и конечным тэгами. Каждый элемент связан со списком его атрибутов, множеством отображенных префиксов пространств имен, локальным/развернутым именем и упорядоченным списком его дочерних элементов (символов, инструкций обработки, комментариев, других элементов).
Атрибут	Определение атрибута в элементе в виде пары имя-значение. Имя атрибута состоит из локальной части и идентификатора пространства имен, отображенного с помощью объявленного префикса. Значение атрибута нормализовано в соответствии с правилами схемы документа.
Инструкция обработки	Инструкция обработки, предназначенная для указания служебной информации. Пара имя(target) и значение (content).
Неразвернутая внешняя сущность	Ссылка на сущность, которая не была развернута в тело документа. Основные атрибуты – имя сущности, публичный и системный идентификаторы.
Символ	Отдельный символ в теле документа, объявленный явно, через символьную ссылку, либо через секцию `CDATA`.
Комментарий	Блок комментариев в любой части документа, за исключением содержимого секции DTD.
Определение типа документа	Содержимое секции DTD. Содержит в себе список всех инструкций обработки, определенных в этой секции.
Необработанная сущность	Необработанная сущность связана с определенной нотацией, которая указывает на тип этой сущности.
Нотация	Определение нотации в секции DTD. Включает в себя имя нотации, публичный и системный идентификаторы.
Пространство имен	Объявление отображения URI пространства имен на символьный префикс, используемый в имени атрибутов и элементов.

Любой правильно сформированный XML документ, удовлетворяющий спецификации пространств имен, может рассматриваться как дерево информационных объектов. Это очень важно с точки зрения интерфейсов прикладного программирования, предоставляющих доступ к содержимому XML документа. Такие интерфейсы, как DOM, XPath работают в терминах информационного множества XML и оперируют только его элементами.

Интерфейс DOM (Document Object Model), например, определяет набор объектов в виде IDL описаний, которые в совокупности предоставляют возможности обхода,

просмотра и изменения дерева документа. Так как для прикладных интерфейсов важна сама информация, то им не передаются данные о физической структуре документа – подстановках внутренних и внешних сущностей, значениях параметров по умолчанию. Все подобные понятия формируют структуру информационного множества XML документа на этапе его синтаксического разбора.

Один из базовых интерфейсов для работы с содержимым XML документа – это язык XPath. Он так же оперирует информационным множеством XML и позволяет обрабатывать запросы и делать выборки различных элементов из этого множества. Модель XPath позволяет ссылаться на части документа, используя различные способы обхода дерева. Все такие компоненты являются элементами модели информационного множества и, поэтому, обмен информацией на уровне интерфейсов осуществляется без привязки к физической структуре документа.

Модель информационного множества документа используют и языки описания схем (DTD, XML Schema, RELAX NG, др.). Все они ссылаются на объекты, определенные в этом стандарте и оперируют его элементами. Любые ограничения, создаваемые схемами, работают только с ними. Фактически, языки схем предназначены для наложения на структуру и содержимое информационного множества некоторых ограничений. Таким образом, описание схемы документов позволяет создавать структуру целевого языка описания данных в терминах информационного множества.

Так как спецификация информационного множества XML является лишь набором описаний, то другие стандарты, использующие эти определения должны корректно указывать степень соответствия используемых объектов. В частности, обязательно указывается, какие именно из определений используются, что происходит с другими информационными элементами (отбрасываются, не изменяются), явно указывается множество объектов, не определяемых информационным множеством. Взаимодействуя таким образом, иные стандарты призваны обеспечить однозначную интерпретацию содержимого документа, опираясь на определения информационного множества.

2.2. Язык XML Schema

Хотя спецификация XML сама описывает подмножество языка для написания определения типа документа (Document Type Definition), как схемы, DTD достаточно слабы. Они поддерживают определение простых ограничений на структуру и содержимое, но не предоставляют средств для описания типов данных или сложных структурных отношений – для многих приложений такой функциональности недостаточно. DTD позволяет убедиться в выполнении самых примитивных ограничений, накладываемых на структуру документа – при этом практически нет возможности работы с типами данных и сложными ограничениями.

Эти недостатки послужили причиной попыток определить более сложные языки схем, и существует рабочая группа W3C, разрабатывающая стандартный язык схем – *XML Schema*. Этот язык схем сам по себе является приложением XML, то есть определяет на основе синтаксиса XML структуру элементов и атрибутов, описывающих в итоге ограничения, накладываемые на содержимое целевого XML-документа.

Стандарт XML схем должен расширить возможности своего предшественника – DTD. Но, в то же время, XML схемы не являются полной его заменой. Причина этого в том, что синтаксис DTD предназначен для определения не только логической структуры документа, но и его физической разметки. DTD позволяет определять внешние и внутренние сущности, которые функционируют как компоненты синтаксического анализа, и не представлены в информационной модели документа.

Описания схем документов могут использоваться как совместно с DTD, так и отдельно от них. Стандарт не указывает конкретную форму связывания описаний схем

с конечными документами. Один из предлагаемых вариантов – через специальный атрибут, указывающий пространство имен и адрес соответствующей ему схемы. Но приложение может использовать схемы и иным образом: оно может проводить частичную проверку соответствия документа или же проверять лишь нужные ему части документа.

В связи с тем, что спецификация XML схем определяет довольно сложные структуры данных и вводит свою собственную типизацию данных, ее формальное описание в стандарте разделено на две части. Основная часть – определение структур данных – описывает средства, используемые для определения сложных типов данных. Особенностью спецификации является то, что это не просто словесное описание, а строгая формальная процедура, которая описывает алгоритмы проверки документов на соответствие и представляет содержимое схем в виде абстрактных объектов с присущими им свойствами и ограничениями. Это связано с тем, что сам по себе стандарт очень сложен и широк, такое описание фактически предписывает лишь реализовывать алгоритмы и структуры данных, уже описанные в спецификации. Вторая часть спецификации носит еще более формализованный характер и описывает все доступные простые типы данных. При этом описываются допустимые операции над ними, их свойства и атрибуты.

2.2.1. Обзор архитектуры

В архитектуре XML схем можно выделить некоторые базовые понятия, на которых основывается весь процесс построения модели документа. К ним относятся определения сложных и простых типов данных, элементов и атрибутов. Например, можно рассмотреть простейшую структуру документа, описывающего почтовый адрес:

```
<адрес>
  <область>Нижегородская</область>
  <город>Нижний Новгород</город>
  <улица тип='проспект'>Гагарина</улица>
  <дом>26</дом>
</адрес>
```

Содержимое корневого элемента «адрес» описывается сложным типом данных. В его описание входит допустимая структура элементов, порядок их следования в документе, возможные альтернативы. Для определения типа данных применяется набор квалификаторов, позволяющих задать альтернативу, последовательность, либо шаблон допустимых в этом типе элементов. В приведенном выше примере структуру корневого элемента можно описать следующим фрагментом схемы:

```
<complexType name='адрес'>
  <sequence>
    <element name='область' type='string'/>
    <element name='город' type='string'/>
    <element name='улица' type='ex:улица'/>
    <element name='дом' type='decimal'/>
  </sequence>
</complexType>
```

Такое описание предписывает всем дочерним элементам следовать только в указанном порядке. Каждому из них присваивается свой тип данных. Тип `string` – встроенный простой тип, определяющий произвольную последовательность символов. На простые типы данных могут накладываться различные ограничения. Существующий набор встроенных простых типов, от которых возможно порождать свои собственные, позволяет гибко выражать необходимые данные. К ним относятся различные числовые формы (целое, с плавающей запятой, знаковое, беззнаковое), строковые поля (`Name`, `NCName`, `QName`) и другие.

Хотя в приведенном примере элемент 'улица' имеет текстовое содержимое, он уже не является простым типом данных. Это связано с тем, что дополнительно с текстовыми данными он содержит атрибут. Для записи такого представления используется следующий тип:

```
<complexType name='улица'>
  <simpleContent>
    <extension base='string'>
      <attribute name='тип' type='string'/>
    </extension>
  </simpleContent>
</complexType>
```

Это описание создает сложный тип данных, наследуя его от простого типа string и расширяя посредством введения нового атрибута «тип». Так как атрибуты в модели XML могут содержать только текстовые данные, в схеме они моделируются простым типом данных (simple type).

Все типы данных, определяемых в языке схем, образуют общую структуру наследования типов, представляемую деревом наследования. Корнем дерева является абстрактный тип anyType, от которого уже наследуются все простые и сложные типы данных. Любой простой тип данных явно наследуется от одного из встроенных либо уже определенных типов. В случае со сложными типами структура наследования может быть намного разветвленней. Это объясняется тем, что само наследование описания типа может иметь две формы: расширение и ограничение. Расширение означает внесение в определение сложного типа данных таких изменений, при которых экземпляры объектов базового типа могут трактоваться как имеющие порожденный тип. Ограничение наоборот, сужает определение типа данных и уменьшает возможные варианты содержимого конечно элемента.

Описания элементов и атрибутов напрямую соотносятся со структурой целевого документа. Каждому допустимому элементу в документе присваивается сложный или простой тип данных, в соответствии с которым должно находиться содержимое этого элемента. Итоговое дерево документа представляется множеством его допустимых элементов с соответствующими им типами данных. Каждый элемент может содержать набор атрибутов. Эти атрибуты и их свойства так же задаются в XML схеме. Содержимое каждого элемента представляет собой последовательность других элементов либо текстовые данные, а конкретные множества вложенных элементов, порядок их следования и наличие текстовых данных описываются сложным типом данных. Процесс проверки произвольного документа на соответствие схеме заключается в обходе дерева и проверки содержимого каждого из элементов на соответствие описаниям данных в XML-схеме.

Стандарт XML схем отводит важное место спецификации пространств имен XML. Если его предшественник, язык DTD, работал с именами элементов и атрибутов на синтаксическом уровне и не предоставлял никаких средств для выражения данных с помощью разделения на пространства имен, то XML схемы полностью поддерживают пространства имен и реализуют все его возможности. Каждая схема определяет словарь (структуру) языка для набора элементов и атрибутов из явно указываемого целевого пространства имен (targetNamespace). При этом конечный XML документ должен так же придерживаться спецификации пространств имен и представлять любые свои элементы принадлежащими к некоторому пространству. В процессе проверки документа на соответствие схеме, учитывается уже не синтаксическая модель документа. Проверку на корректность проходят имена элементов с префиксами, которые разыменовываются и привязываются к своему пространству имен.

Такая модель позволяет создавать формальные словари описаний данных, используя XML схемы. А возможности взаимодействия между схемами позволяют в

одном документе совмещать информацию из различных областей. Этим достигается модульность построения схем, позволяющая разбивать схемы на логически законченные единицы, которые в дальнейшем могут использоваться как самостоятельно, так и в совокупности с другими схемами.

2.2.2. Расширенные возможности

Одним из важных требований реализации поддержки бизнес-логики была возможность установки ограничений и зависимостей между различными частями документа. Язык DTD ввел примитивную поддержку таких зависимостей – тип уникального идентификатора и ссылки на него. Часто этого было недостаточно, так как многие отношения между объектами не укладываются в рамки простой связки уникального идентификатора и ссылок на него. XML схемы позволяют создавать более сложные ограничения, позволяющие гибко контролировать уникальность содержимого элементов или их атрибутов. При этом есть возможность задавать область действия уникальности и используемые для этого элементы через XPath выражения.

XML схемы выделяются рядом уникальных особенностей, не имеющих альтернатив в синтаксисе DTD, и позволяющих гибко и оптимально использовать описания типов данных и структур в документе схемы. Например, такая возможность, как *группы подстановки*, позволяет компактно записывать сложные структуры данных, представление которых в DTD либо невозможно, либо потребовало бы излишних накладных расходов. Набор элементов, входящих в группу подстановки базового элемента группы, может использоваться в содержимом документа в любом месте, где возможно использование этого базового элемента. Определение, например, абстрактного базового элемента и множества элементов, формирующих его группу подстановки фактически вводит класс элементов, для использования которого не нужно перечислять все элементы этой группы – достаточно сослаться на базовый. При этом, последний может и вовсе быть абстрактным элементом и служит лишь для формирования этой группы.

Абстрактные типы данных – это еще одна возможность стандарта, которая тесно связана со всей структурой наследования. Объявление типа данных абстрактным запрещает его прямое использование в элементах, такие типы могут использоваться для наследования и определения порожденных типов, которые могут выражать уже определенную требуемую семантику. Структура наследования в этом плане сходна с объектно-ориентированными языками, при этом абстрактные типы данных играют роль интерфейсов (либо абстрактных классов).

Собственно отображение XML схем на структуры языков программирования и основывается на этих возможностях. Существуют средства, позволяющие генерировать по заданной схеме прототипы и определения на объектно-ориентированных языках программирования (Java, C#). Эти прототипы могут использоваться традиционным образом как обычные объекты – но вся их структура будет подчиняться структуре исходного документа схемы. Обратное преобразование подразумевает сериализацию содержимого этих объектов в документы XML, описываемые исходной схемой (иногда такую технологию называют marshaling/demarshaling).

2.2.3. Альтернативы XML Schema

Модели документов, которые могут быть выражены в синтаксисе XML схем, очень разнообразны и намного шире возможностей DTD. Но, все же, они не произвольны. Основной принцип построения схем на основе этого стандарта состоит в высоком уровне модульности компонентов, которые могут гибко взаимодействовать друг с другом и переопределять свое поведение. Основные понятия XML Schema – простые и сложные типы данных – составляют скелет всей архитектуры схем. Можно сказать, что

они определяют одну из возможных, и наиболее распространенную модель данных, которые представляются в XML синтаксисе.

Довольно часто содержимое XML документа должно удовлетворять требованиям, намного более сложным, чем те, что позволяют выразить схемы. Чаще всего такие ограничения носят прикладной характер, и их проверка, строго говоря, должна лежать на приложении, обрабатывающем документ. Например, для большинства языков, существуют дополнительные ограничения, которые зависят от структуры языка, его семантики, и могут быть проверены только конкретными алгоритмами. Но, иногда бывает удобно иметь инструменты, позволяющие накладывать сложные ограничения на содержимое и семантику документа и проверять эти ограничения.

XML схемы в таких случаях имеют ограниченную выразительную мощность. Это можно аргументировать тем, что могут существовать произвольные правила, ограничивающие корректность документа, и описание всего их множества сводится к написанию конкретного алгоритма на языке программирования, проверяющего нужные соотношения. А это фактически противоречит всей идеологии схем.

Все же, существуют альтернативные языки описания схем, и множество описываемых ими документов намного шире. Самый распространенный из языков этого класса – язык RELAX NG, поддерживаемый организацией OASIS. Этот язык схож с XML схемами, но основывается на более гибких регулярных шаблонах. Эти шаблоны позволяют описывать структуры тэгов произвольного характера, при этом существует возможность описания правил, описывающих взаимоисключающие неоднородные структуры деревьев. Недостатком RELAX NG схем является то, что их невозможно применять для строгой типизации сложных структур данных – изолированное понятие сложного типа данных как таковое отсутствует. Можно сказать, что этот язык оперирует больше синтаксической структурой документа, нежели его типизированной информационной моделью. Именно поэтому его использование оправдывается в задачах, не заботящихся о строгой типизированности содержимого документа, но требующих обязательной проверки комплексных условий.

Еще один представитель альтернативных языков схем – язык Schematron. Строго говоря, Schematron вообще не является языком описания схем. Это лишь оболочка, позволяющая удобным образом описывать конкретные алгоритмы, реализующие проверку нужных ограничений на содержимое документа. Это средство удобно использовать при необходимости автоматизации проверки документов и описания сложных ограничений содержимого в унифицированном формате и без лишних затрат.

2.3. Синтаксис XPath и XSLT

Для обработки содержимого XML документов существует огромное число программных средств. В простейшем случае приложение работает с документом через интерфейс, предоставляемый XML процессором. Оно может считывать, обрабатывать, изменять документ – все это зависит от уровня доступного сервиса и возможностей XML процессора. Такая непосредственная работа с данными носит низкоуровневый характер и пригодна в областях, где существует тесная интеграция XML документа и обрабатывающего его приложения. Часто возникает необходимость в представлении содержимого документа на более абстрактном уровне, при этом для стандартных задач обработки данных желательно иметь общедоступные инструменты, которые не будут зависеть от специфики целевой области. Именно такими возможностями обладают языки XPath и XSLT, разработанные специально для оперирования информационным множеством документа.

2.3.1. XPath выражения

XPath с помощью простого и ясного синтаксиса позволяет проводить сложные выборки данных из документа. Так как любой документ представляет собой дерево, узлы которого содержат различные атрибуты, работа с информацией в этом дереве сводится к его обходу и позиционированию на нужных элементах. Язык XPath реализует такую модель, используя понятие пути, указывающего нужные элементы структуры. При этом путь может содержать различного рода ограничения и проверки, позволяя производить позиционирование с учетом сложных условий. Язык XPath оперирует элементами информационного множества XML, он вводит свои типы данных, над которыми можно совершать различные операции. Работать с этим языком можно только в контексте более общей среды, которая предоставляет информационную модель. Одно из основных применений языка XPath – работа в контексте языка XSLT.

Язык трансформаций XML Stylesheet Language является полноценным языком программирования, ориентированным на обработку древовидных структур XML-данных. В совокупности эти два стандарта решают важную задачу – они позволяют описывать преобразования XML данных в иную произвольную форму (чаще всего в другие XML деревья). Иными словами, с их помощью становится возможным передача информации между различными системами и архитектурами.

Хотя основной областью применения XPath сегодня является программирование на языке XSLT, сам этот стандарт не указывает на какую-либо конкретную целевую среду, в которой он функционирует. Каждое XPath-выражение представляет собой путь, описывающий порядок выборки нужных элементов дерева. Путь состоит из нескольких шагов, разделенных символом косой черты `'//'`. Результатом работы каждого шага являются элементы из информационного множества XML, получаемые по определенным правилам. XPath обладает очень гибким и удобным синтаксисом. Это выражается в том, что элементарные запросы записываются очень компактно, а для создания более сложных конструкций может применяться расширенный синтаксис.

Основным понятием в XPath является выражение. Именно оно задает путь в дереве документа и указывает на нужные элементы. Модель XPath оперирует своим собственным набором типов данных. Выражение, полученное в результате обработки XPath-запроса, принадлежит одному из возможных базовых типов: множество узлов (node-set), строка, число, булево значение. Существуют правила преобразования этих типов данных, их обработки.

В связи с тем, что выражение XPath выполняется в контексте внешней среды, именно она должна определять текущее окружение, доступное для вычисления выражения. Контекст выражения состоит из нескольких понятий, определяющих положение XPath выражения в модели документа (Табл. 2).

Табл. 2. Контекст XPath выражения.

Элемент контекста	*Структура и описание*
Узел контекста	Текущий элемент в дереве, относительно которого ведется вычисление всего выражения.
Размер контекста	Общее число всех элементов в контексте.
Положение в контексте	Позиция текущего элемента в контексте.
Привязка переменных	Множество переменных, представленных в виде пар имя-значение. Значение переменной соответствует одному из возможных типов данных.
Библиотеки функций	Множество доступных функций, оперирующих переменными и выражениями.
Текущие привязки про-	Соответствия между идентификаторами пространств

странств имен	имен и префиксами, присвоенными им в текущем контексте.

Понятие узла контекста используется для указания положения выражения в дереве документа. Относительно него задаются параметры положения и размера контекста. Каждый элемент XPath-выражения может менять контекст определенным образом. В каждом XPath выражении внешняя среда может определять набор переменных, которым присвоены значения определенного типа. На эти переменные можно ссылаться из любой части выражения через синтаксис `$varname`. Помимо переменных для вычисления значения выражений доступны для использования различные функции, оперирующие значениями переменных и выражений. Стандартная библиотека функций оперирует стандартными четырьмя типами данных XPath. В то же время, расширения этой библиотеки могут принимать в качестве параметров иные типы данных, зависящие от требований и целевого назначения спецификации.

Основной составляющей частью XPath-выражения является путь адресации (location path). Он используется для нахождения и выборки требуемого множества узлов из дерева XML документа. Каждый компонент пути состоит из идентификатора оси данных, по которой производится выборка, шаблона выбираемого имени элемента и необязательных предикатов, фильтрующих множество полученных узлов на основе необходимых условий. Каждый следующий шаг в пути адресации оперирует со множеством элементов, полученном на предыдущем шаге.

Например, выражение

```
/child::doc/child::chapter[position()=5]/attribute::title
```

выберет атрибут `title` у пятого элемента `chapter` в корневом элементе `doc`. Представленный синтаксис – развернутый. Он полностью указывает оси на каждом шаге. Его сокращенная форма выглядит как `/doc/chapter[5]/@title`. В этом случае наиболее часто используемая ось `child`, означающая множество дочерних элементов относительно текущего, опускается. А ось `attribute`, выбирающая множество атрибутов в текущем элементе сокращается до знака `@`. Существуют другие оси, которые позволяют двигаться по структуре дерева в произвольном направлении. К ним относятся движения вверх по дереву (`ancestor`, `parent`), движение по соседним элементам (`following-sibling`, `preceding-sibling`).

Предикаты на каждом шаге заключатся в квадратные скобки. После вычисления возвращаются только те элементы, для которых значение предиката принимает логическое значение `true`.

2.3.2. Трансформации XSLT

Преобразования XSLT (XML Stylesheet Language Transformations) используются для трансформации входного дерева XML элементов в произвольную форму. Как правило, на выходе получается другое дерево XML, которое может полностью отличаться от исходного. Изначально язык XSLT позиционировался как развитие существующей спецификации CSS, определяющей визуальное представление языков логической разметки (HTML как правило). Но в своем развитии XSLT получил столь мощные возможности, что превратился в независимую спецификацию языка, который имеет мало отношения к визуальному форматированию и системам публикации.

Язык XSLT впитал в себя очень много идей, реализованных в языке DSSSL – *Document Style Semantics and Specification Language*. Этот язык использовался для обработки SGML документов и визуального их форматирования. Он основывался на языке обработки списковых данных `Lisp`. Собственно многие идеи XSLT являются отображением функциональности этого языка на синтаксис XML документов.

Как следствие, XSLT обладает одной важной особенностью – как и Lisp, он является больше декларативным, нежели императивным языком программирования. Совмещая довольно простой и понятный синтаксис шаблонного описания преобразования с мощными возможностями XPath выражений, сегодня язык XML преобразований является одним из центральных стандартов с огромной практической пользой.

Для описания требуемых преобразований вся структура языка состоит из шаблонов(`template`), которые фактически являются разновидностью неявных функций. Каждый шаблон сопоставлен со своим XPath выражением. Это выражение определяет множество узлов в документе, попадающих под определение этого шаблона. Процесс трансформации заключается в том, что при обходе дерева документа для каждого анализируемого элемента находится шаблон, его обрабатывающий. В теле шаблона описывается произвольная структура XML элементов, в которую могут быть внедрены различные операторы XSLT. Эта структура элементов и заменяет исходный анализируемый элемент. Таким образом происходит обход всего дерева документа.

Язык определяет множество различных операторов, которые позволяют управлять процессом обхода дерева, вызывать другие шаблоны (именованные), применять шаблонную обработку к произвольному множеству элементов, выбираемых с помощью XPath выражений. Операторы позволяют проводить итерационную обработку множества узлов (`xsl:for-each`, `xsl:sort`), направлять поток выполнения в зависимости от значений условий (`xsl:if`, `xsl:choose`). В шаблоне можно указывать не только явную выходную структуру XML-документа, но и создавать элементы и атрибуты по запросу с различными условиями с помощью операторов `xsl:element`, `xsl:attribute`.

Очень мощной возможностью шаблонов является использование режимов обработки (`modes`). Одному набору узлов может быть установлено в соответствие несколько шаблонов обработки, но с различными именами режимов. В процессе трансформации можно использовать специальные атрибуты, которые позволяют указать используемый режим обработки. Тогда при дальнейшей обработке в работу будут включаться шаблоны только с указанным именем режима. Это позволяет одно и то же содержимое документа интерпретировать по-разному в зависимости от условий и требуемого результата.

Язык XSLT сейчас активно развивается, за счет возможности внедрения функций расширения, в него добавляются нужные разработчикам возможности и операции. В процессе активного развития находятся модели обработки содержимого документов XPath2 и XSLT2. Уже существуют экспериментальные реализации этих стандартов (Saxon), свободные от некоторых ограничений предыдущего стандарта. Основным направлением в развитии этих технологий является введение поддержки типов данных XML Schema, возможности более строгой типизации операций. В этом аспекте спецификация XPath тесно примыкает к разрабатываемой модели XQuery, позволяющей определять произвольные выборки данных из содержимого документа.

2.4. Перспективы XML технологий

Любой XML-документ можно рассматривать как хранилище структурированной информации, которое использует определенные соглашения для ее определения и описания. Конечно же, существует огромное число способов описывать информацию и XML – это лишь один из возможных. Объявление структуры данных на любом языке программирования, реляционная база данных – все это такие же способы хранения информации, для которых разработаны свои средства их обработки.

С этой точки зрения язык XML предоставляет способ, позволяющий описывать информацию очень гибко и компактно. При этом полученный документ будет одно-

значно интерпретироваться как программами-обработчиками, так и обычными людьми, имеющими возможность редактировать его вручную. При этом существуют различные способы описания необходимой структуры и содержания документа (схемы документов, определения DTD), а так же методы обработки и преобразования документов (язык XSLT, программные XML-интерфейсы).

Хотя изначально языки разметки использовались в области публикаций и представления документов в сети, язык XML позволил расширить круг решаемых его предшественниками задач. Прежде всего, стало возможным использовать его для хранения и обработки сложных структурированных данных.

В отличие от реляционных баз данных, которые так же решают задачу хранения и обработки данных, XML позволяет работать со сложными структурами данных в простой и доступной для понимания форме. Базы данных оптимальны для использования при обработке больших объемов однородных данных, но при работе с небольшими объемами данных сложной структуры их использование затруднительно. XML не является заменой реляционных СУБД. Каждая из этих технологий занимает свою нишу. И все же, в последнее время возникают попытки их интеграции.

Передовые разработчики баз данных (IBM DB2, Oracle) в последних версиях своих продуктов предоставляют возможности интеграции и обмена данными с XML документами. Наиболее интересная возможность из этого ряда – отображение документа на множество реляционных таблиц и установка между ними связей на основании схемы документа. Фактически, это позволяет хранить XML-данные в СУБД и обрабатывать их наравне с другими. Над сохраненными в специальном виде документами можно производить различные трансформации, которые вызываются прямо из SQL запроса. Еще большее сближение этих технологий может произойти после внедрения разрабатываемого сейчас языка XQuery – обобщения XPath, позволяющего создавать сложные SQL-подобные запросы, работающие на информационном множестве XML.

На базе XML создано огромное количество языков разметки, используемых в таких областях, как публикации, документирование. Главное преимущество использования XML в системах представления информации – возможность отделения содержимого от его визуальной формы. В сети Internet внутренняя структура сервера может представлять собой хранилище информации. Программное обеспечение web-сервера по запросу пользователя преобразует содержимое документов в нужную форму. При этом к одному документу могут применяться разные преобразования, позволяющие реализовать гибкую систему представления данных в сети. Использование языка XSLT позволяет отобразить один документ в таких форматах, как HTML и PDF, позволяет иметь несколько различных версий документа (например, для просмотра и для печати).

Реализация таких возможностей не требует, чтобы на машине пользователя было установленное программное обеспечение для обработки XML. Все преобразования могут выполняться сервером динамически – клиент получает лишь результат их работы. Хотя последние версии браузеров поддерживают большинство стандартов, такая структура более предпочтительна, так как уменьшает нагрузку на клиента и требует от него меньших вычислительных затрат.

В процессе развития XML было создано большое количество прикладных языков, основанных на нем. Многие из них считаются стандартами в области обмена информацией. В сфере публикаций – это язык представления разметки XSLFO, язык представления векторной графики SVG. Все большую популярность и поддержку приобретает язык описания математических конструкций – MathML. Для решения многих задач публикаций и систем документирования существуют уже разработанные и стандартизированные решения. Например, формат хранения документов DocBook позволяет гибко представлять техническую документацию, работать с документами и

преобразовывать их в различные целевые форматы (HTML, XSL-FO, PDF) с помощью XSLT преобразований. DocBook предоставляет в распоряжение гибкий и продуманный XML словарь, позволяющий описывать техническую документацию и решать иные сходные задачи. Изначально этот стандарт основывался на языке SGML, и уже после возникновения XML был переведен на новую технологию.

Развитие самой спецификации языка XML не стоит на месте. В связи с активным распространением и развитием стандарта Unicode возникла необходимость в некоторых модификациях описания XML-документов (это связано с тем, что в процессе развития набор символов Unicode непрерывно расширяется и дополняется). Поэтому уже сейчас создается модифицированная версия спецификации XML 1.1. Вместе с ней на стадии принятия находится спецификация XML Names 1.1, вносящая некоторые коррективы в исходную версию стандарта пространства имен.

3. Структура библиотеки классов Colorer

Для обеспечения доступа ко всем своим возможностям библиотека предоставляет прикладной интерфейс программирования (API), реализующий функции синтаксического разбора и анализа текста. Основная цель, которая достигалась в процессе реализации интерфейсов библиотеки – возможность гибкого использования библиотеки внешними приложениями вне зависимости от их архитектурных особенностей, аппаратной и программной платформы. Для достижения этого результата была выбрана объектно-ориентированная модель классов языка C++. Исходные модули в дальнейшем могут использоваться для отображения на структуры других языков.

Так как все объекты библиотеки Colorer представлены в виде классов и интерфейсов языка C++, их использование ограничивается приложениями, умеющими импортировать и работать с объектами этого языка. Чаще всего в этом случае само целевое приложение должно быть написано на языке C++. Для предоставления возможности работы с библиотекой из программ на произвольном языке, стандартным решением является реализация ее внешних интерфейсов в терминах структурного программирования. Это позволяет использовать Colorer как подключаемую библиотеку (DLL) с любыми приложениями на языках, поддерживающих динамическое связывание, а это практически все традиционные языки программирования (C, Pascal, и т.д.). Преимуществом библиотеки Colorer в этом случае является многослойное построение, позволяющее использовать ее возможности на разных уровнях при различных возникающих задачах. При этом прослойка для связывания с указанными языками может включать в себя только минимальные сервисы, либо расширяться до представления более глубоких уровней функциональности библиотеки.

3.1.Обзор

Библиотека синтаксического анализа Colorer представляет собой совокупность различных компонентов, реализующих интерфейсы для взаимодействия друг с другом и с целевым приложением-клиентом, в котором библиотека функционирует. Эти модули реализованы как классы языка C++, использующиеся для связи и обработки объектов системы.

Приложение-клиент предоставляет библиотеке информацию об обрабатываемом тексте обычно в виде последовательности строк, на которые разбит этот текст. Система анализирует поток поступаемых строк текста и генерирует на его основе выходную информацию о полученных синтаксических регионах (лексемах – элементарных структурных единицах текста), которая в дальнейшем может использоваться приложением для обработки (чаще всего для генерации цветовой информации).

С точки зрения обработки символьной информации, библиотека работает с несколькими ее источниками. Уже описанный разбираемый набор строк исходного текста поступает от прикладной программы. С другой стороны, вся информация о синтаксическом анализе задается в настройках на специализированном языке HRC. Он описывает структуры, необходимые для разбора синтаксиса каждого языка программирования в терминах регулярных выражений и схем (контекстов).

Вся выходная информация генерируется в виде потока событий, каждое из которых описывает изменение состояния в процессе синтаксического анализа. Обработчик этих событий может напрямую использовать их для отображения результатов анализа, либо сохранять в произвольной форме для дальнейшего использования. Обобщенная диаграмма потоков данных в библиотеке Colorer представлена на Рис. 1.

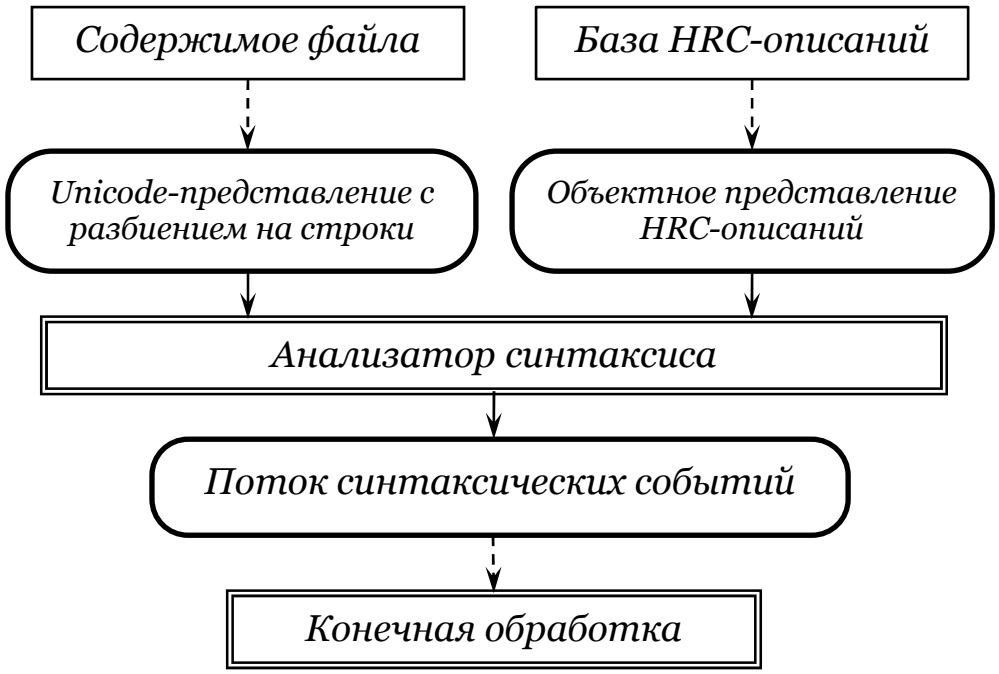

Рис. 1. Диаграмма потоков данных.

Модуль анализа синтаксиса является центральным компонентом, обрабатывающим поступающие на вход данные. Он опирается в своей работе на структуру HRC-описаний, преобразованную в объектную форму. Содержимое разбираемого файла абстрагируется от входного источника и представляется в виде интерфейса, предоставляющего доступ к любой его части.

3.2. Базовые компоненты

В процессе работы библиотека классов Colorer использует набор модулей, функций и классов, реализующих поддержку низкоуровневой функциональности и выполнение различных сервисных задач. Все эти компоненты не зависят от основных функций библиотеки и служат для удобства представления и абстракции данных. При необходимости они без изменений могут использоваться в иных проектах, требующих соответствующей функциональности.

Общая структура и связи между этими компонентами представлены на Рис. 2.

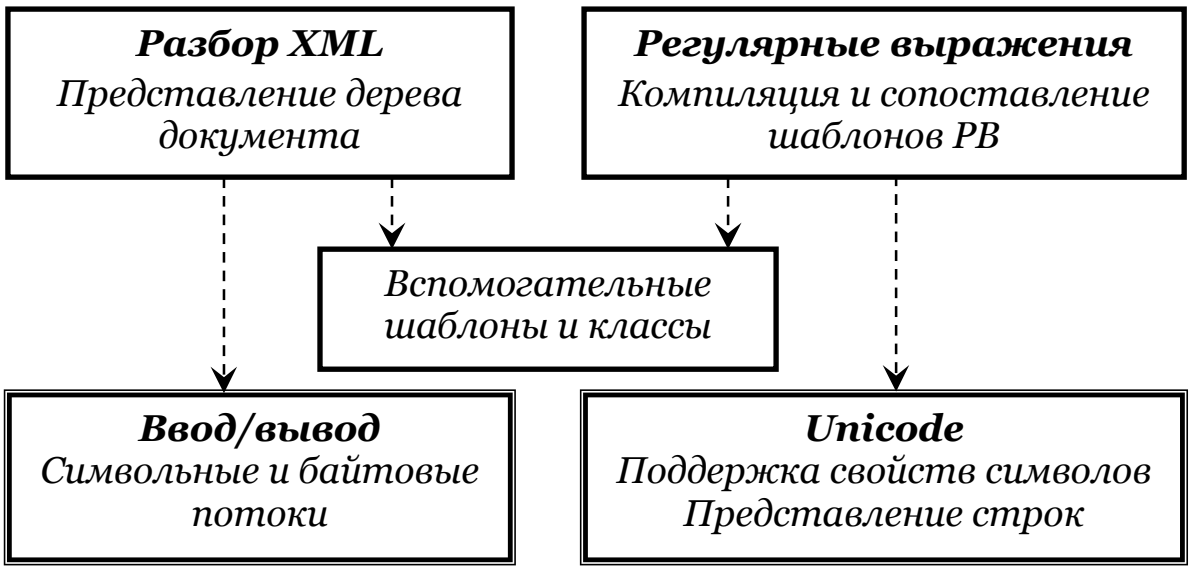

Рис. 2. Структура базовых модулей библиотеки.

3.2.1. Поддержка Unicode

Базовым слоем, реализующим поддержку стандарта Unicode в библиотеке, является набор Unicode-классов. В него входят два основных компонента: классы по работе с отдельными символами и реализующие представление строковых данных. Классы `Character` и `String` являются представлением символа и строки. Причем, если `Character` – это статический класс, возвращающий информацию об одиночных символах, то класс `String` является абстрактным базовым классом. Он определяет набор виртуальных методов, конкретная реализация которых может варьироваться.

Основные цели, преследуемые при создании набора этих классов – это универсальность, портируемость, простота поддержки и модификации. Универсальность выражается в возможности изменения внутренних алгоритмов работы при сохранении функциональности компонентов, пользующихся этими методами. То есть компоненты должны представлять определенный уровень абстракции (функционировать как «черный ящик») и исключать возможность зависимости кода от конкретной реализации.

Переносимость является естественным требованием для функционирования системы на различных программных и аппаратных платформах. Во избежание проблем с различной трактовкой типов данных компиляторами, необходимо однозначное соответствие и явное указание размера и знака используемых целочисленных типов.

Требование возможности модификации компонент связано с постоянным процессом пополнения базы данных Unicode и выпуском новых версий стандарта. Перекомпиляция библиотеки с новым вариантом базы данных не должна создавать проблем, требующих вмешательства в алгоритмы работы и их модификации.

Общая структура Unicode-классов представлена на Рис. 3.

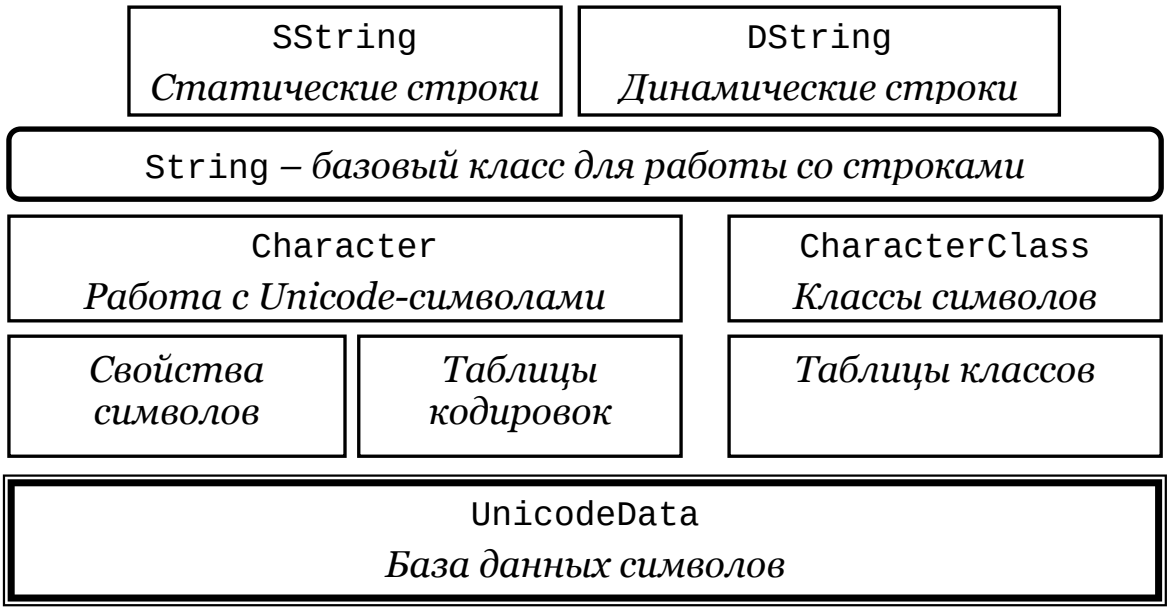

Рис. 3. Иерархия классов Unicode.

В наибольшей степени требования стандарта применимы к классу `Character`, реализующему базовые символьные операции. Unicode символ, представленный этим классом, является единицей хранения текстовой информации и обладает набором свойств и признаков, которые используются приложением. Хотя база данных символов Unicode описывает довольно большое количество этих свойств, реально при функционировании библиотеки необходимы не все из них. Основная используемая информация – это принадлежность к одному из тридцати определенных классов символов и соответствие между верхним, нижним и заглавным регистрами.

Для извлечения информации из базы данных Unicode, она преобразуется в сжатую форму, обеспечивающую компромисс между относительно небольшим размером хранимой базы данных и скоростью извлечения информации из нее. Основное решение, применяемое для достижения этой цели – двухуровневые таблицы доступа. Таблица первого уровня содержит индексы, необходимые для обращения к таблице второго уровня. Соответственно битовое представление символа разбивается на две части – верхняя служит для указания позиции в таблице индексов, а нижняя – для смещения относительно выбранного блока в таблице свойств. Реализация нескольких блоков таких таблиц, хранящих различные свойства символов, имеет размер 150-200 Kb при прямой выборке данных с двумя обращениями на чтение, что положительно сказывается на производительности. Двухуровневые таблицы свойств так же часто позволяют сократить дублирование информации, так как с точки зрения набора свойств различные блоки символов в базе данных могут иметь одинаковое сжатое представление.

Основным используемым в коде библиотеки модулем является набор классов по поддержке Unicode-строк. Эти классы реализуют хранение строк и различные операции над ними. Хотя строки можно представлять последовательностью `wchar` символов, и обрабатывать их аналогично `char*` строкам, в случае с библиотекой Colorer требуется более комплексный подход к реализации понятия строка символов. По большей части это связано со спецификой работы библиотеки и принимаемых для анализа данных. Приложение-клиент может работать с произвольным представлением строк. Оно может быть в любом из форматов хранения символов Unicode (UTF-8, 16, 32), или же в произвольной однобайтовой кодировке. В связи с тем, что библиотеке необходимо воспринимать и обрабатывать произвольную кодировку, нужно предусмотреть механизм, с помощью которого это можно гибко и без больших затрат реализовать.

Стандартная строка ASCIIZ (оканчивающаяся нулевым символом) хотя и является простейшей реализацией понятия строки (в совокупности с сервисными функциями), не учитывает указанных требований и имеет ряд недостатков. Так, нулевой символ, означающий конец строки, формально является символом Unicode, и таким образом, может присутствовать в строке. Длина строки не передается вместе с ней в явном виде, а должна вычисляться по необходимости. Для представления и обработки потока Unicode данных такое представление так же неудобно в связи с необходимостью обработки и выделения некоторых частей из этих данных, передача их как параметров. В итоге, для преодоления всех этих проблем была выбрана объектно-ориентированная реализация класса строки, в которой естественно сочетаются форма хранения данных и непосредственная обработка.

3.2.2. Вспомогательные шаблоны и классы

При оперировании сложными структурами данных удобно использовать базовые примитивы для представления таких понятий, как список, множество объектов. В библиотеке Colorer, где большая часть внутренних данных имеет списковую структуру, реализация примитивов для их представления существенно облегчает задачу и позволяет писать код в более компактном виде. Для функционирования большинства компонентов библиотеки Colorer требуется два вида контейнеров: хэш-таблица и вектор.

Хэш-таблица реализуется классом `Hashtable` и позволяет эффективно производить поиск в неупорядоченном множестве элементов по ключу. Класс описывается через шаблон языка C++, что позволяет объявлять различные его варианты для хранения данных в произвольной форме.

Вектор – это объект, эффективно реализующий динамический массив для хранения упорядоченной последовательности объектов и быстрого доступа к любому элементу по его индексу. Его реализацией в библиотеке Colorer является шаблонный класс `Vector`, позволяющий аналогично хэш-таблице задавать тип хранимых элементов.

Для контроля ошибок в библиотеке Colorer широко используется обработка исключительных ситуаций языка C++. Любой компонент, который при работе приводит к ошибочному состоянию, выбрасывает исключение. Любое исключение в библиотеке Colorer определяется как экземпляр класса, унаследованного от базового типа исключения – `Exception`. При обработке в коде, объект исключения передается в блок `catch` по ссылке, что позволяет избежать ненужного копирования объектов, ускорить процесс обработки и избавиться от возможных утечек памяти.

Использование собственных примитивов и отказ от их аналогов, предоставляемых стандартной библиотекой шаблонов (STL) языка C++, вызван несколькими причинами. Главная – это необходимость обеспечения максимальной переносимости библиотеки на различные платформы цель которой – добиться возможности компиляции кода на любой архитектуре без его модификации. Так как реализации STL на различных платформах могут отличаться, существует вероятность различного поведения кода, использующего STL.

С другой стороны, введение своих примитивов позволяет обеспечить возможность простого переноса кода на другие языки программирования (такие как Java, C#). Это очень важно, так как намного упрощается перевод алгоритмов, которые в случае с STL, возможно, потребовали бы более сложных изменений, касающихся самой структуры классов – а это крайне нежелательно. При этом существует возможность преобразования введенных классов и функционирования их как оболочек, обращающихся к другим реализациям алгоритмов (в частности к STL).

3.2.3. Ввод-вывод

Многие внутренние компоненты библиотеки взаимодействуют с внешней средой посредством файлового ввода/вывода. Так как во-первых, функционирование всей библиотеки необходимо вести относительно набора символов Unicode, а во вторых, в процессе реализации файловых операций нужно достичь максимальной независимости от аппаратной и программной платформы, то для реализации подсистемы ввода/вывода была создана объектная модель, абстрагирующая библиотеку от конкретных особенностей реализации.

Модель входных источников данных подразумевает, что запрашиваемый ресурс может представляться не только файлом и находиться в произвольном месте. Для указания полного пути используется строка в формате URL, в которой название схемы URL определяет тип хранилища, в котором располагается запрашиваемый файл. Различные модели хранилищ обрабатываются различными классами, которые являются реализациями абстрактного класса `InputSource`. Основные методы, объявленные в этом классе указаны в Табл. 3.

Табл. 3. Методы класса InputSource.

Метод	*Назначение*
`const byte *openStream()`	Открытие потока и возвращение указателя на байтовый массив с содержимым ресурса.
`void closeStream()`	Закрытие потока и освобождение памяти.
`int length()`	Длина открытого потока в байтах.
`String *getLocation()`	Строка, содержащая полный URL для этого ресурса.
`createRelative(String*)`	Создание ресурса с этим же типом схемы и с адресом, указанным относительно текущего.

Так как класс `InputSource` является абстрактным, то для создания экземпляров его реализаций может использоваться статический factory-метод `InputSource::newInstance(const String*)`. Он автоматически определяет схему переданного URL и вызывает конструктор соответствующей реализации. Его использование удобно в случаях, когда заранее не известен тип передаваемого URL. Библиотека Colorer реализует поддержку нескольких основных типов URL, использование которых в большинстве случаев покрывает все запросы пользователей (Рис. 4).

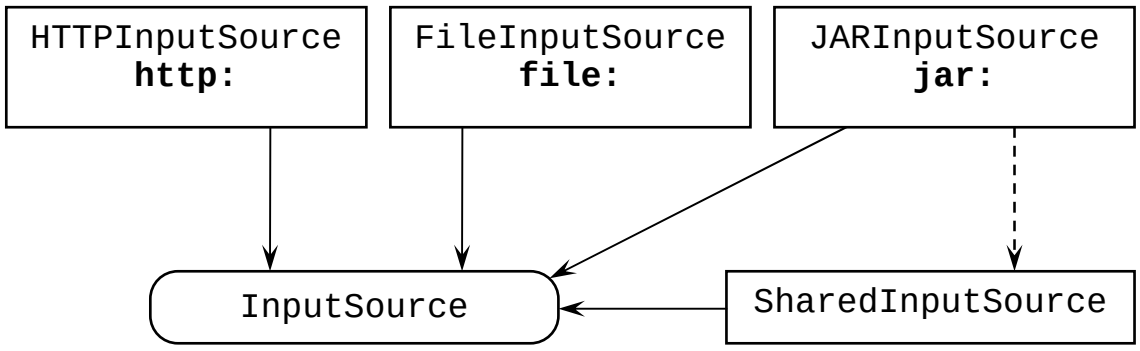

Рис. 4. Структура наследования InputSource.

Основной тип URL – локальная файловая система. По умолчанию при отсутствии схемы URL используется именно он. Класс `JARInputSource` используется для доступа к файлам в содержимом JAR(ZIP) архива. Для работы этого класса используется вспомогательный класс, являющийся оболочкой над базовым `InputSource` и

предназначенный для возможности его использования одновременно в нескольких экземплярах `JARInputSource`, работающих с одним базовым файлом.

3.2.4. Поддержка XML

Для разбора синтаксиса файлов на языке HRC, использующихся в библиотеке, применяется простой XML анализатор, строящий в памяти дерево элементов и атрибутов. Используемый XML процессор минимально удовлетворяет требованиям спецификации и реализует только те функции, которые необходимы библиотеке.

Намного больше внимания в разработке библиотеки уделялось соответствию самих HRC файлов стандарту XML. Это очень важно, так как при этом становится возможным использовать различные существующие средства визуального редактирования XML, использующие и анализирующие не только содержимое самих файлов, но и соответствующих им DTD и XSD схемам.

При разработке компонентов библиотеки, использующих XML преобразования для создания нужных HRC, работающих и с другими, уже существующими HRC кодами очень важным было достижение совместимости как структуры HRC так и описывающих ее схем с существующими стандартами. Встроенный XML-процессор корректно распознает стандартный синтаксис разметки XML в виде дерева элементов и атрибутов, умеет работать с CDATA секциями и предопределенными символьными и числовыми сущностями. Он полностью реализован на основе Unicode-классов библиотеки, чем достигается соответствие его стандарту Unicode: библиотека способна воспринимать HRC файлы в любой кодировке (однобайтовой или Unicode), все внутренние представления данных используют Unicode.

С другой стороны, в процессе реализации XML-процессора не ставилась целью корректная обработка синтаксических ошибок в тексте документа. Изначально процессор проектировался для разбора произвольной SGML подобной структуры, поэтому он довольно универсален и способен разбирать любой входной поток, обнаруживая в нем структурную разметку. Процессор не поддерживает создание DTD описаний, он игнорирует объявление внешних и внутренних сущностей, инструкций обработки и некоторых других возможностей XML.

Задача написания XML процессора, удовлетворяющего хотя бы первому уровню соответствия (well-formedness) довольно сложна, и при создании библиотеки Colorer не ставилось такой цели. В основном это связано с тем, что в дальнейшем можно безболезненно перейти на использование сторонних процессоров, более точно следующих спецификации – сегодня существует большой выбор реализаций, работающих на различных платформах и языках программирования.

3.2.5. Регулярные выражения

Регулярные выражения используются во многих компонентах библиотеки, но основное их применение – это синтаксический анализ на основе описаний HRC файлов. Одной из официальных спецификаций синтаксиса РВ является стандарт POSIX. Сам стандарт определяет базовые требования к UNIX-совместимым системам, набор доступных утилит и прикладных программ. Одним из его компонентов и является требование поддержки POSIX-регулярных выражений. Такие РВ используют минимальный синтаксис и определяют лишь некоторые из возможных операторов.

Большинство реальных систем, использующих и реализующих разбор регулярных выражений, расширяют этот стандарт и вводят свои определения и новые операторы. Это объясняется тем, что чаще всего использование РВ зависит от области применения и решаемых задач, и создать универсальный синтаксис невозможно – каждой реализации нужно использовать свои собственные возможности. Как пример, можно привести довольно популярные реализации РВ в языке Perl (и их аналог – биб-

лиотека PCRE). Они вводят большое количество операторов, специфичных для языка Perl, либо же просто расширяющих и упрощающих синтаксис. С другой стороны, спецификация синтаксиса РВ в языке XML Schema, хотя и использует синтаксис, более приближенный к стандартному, все равно вводит расширения, рекомендуемые уже стандартом Unicode. Аналогичный синтаксис применяется в модуле РВ, входящем в библиотеку Sun JDK 1.4.

Ситуация с библиотекой Colorer усложняется тем, что с одной стороны нужно работать с реализацией РВ, максимально приближенной к синтаксису этих стандартов (для того, чтобы упростить процесс написания РВ), с другой же стороны библиотеке требуется расширенный набор операторов, которые будут отвечать за управление процессом синтаксического анализа и могут быть применимы только по отношению к терминам разбора текста.

В библиотеке Colorer реализованы РВ, так же расширяющие стандартный синтаксис и определяющие свои собственные операторы. С другой стороны, в их синтаксис заложена строгая поддержка стандарта Unicode, которая следует указаниям UTR#18 (Unicode Regular Expression Guidelines) и реализует мощные возможности по поддержке набора символов Unicode. Реализация полностью основывается на Unicode-модуле библиотеки Colorer, использует многие его функции и структуры данных. Это обеспечивает возможность разбиения множества символов по классам Unicode, присвоение им определенных свойств и атрибутов. Синтаксис РВ, в свою очередь, позволяет использовать все эти возможности при написании HRC-кодов.

Хотя регулярные выражения библиотеки могут использоваться самостоятельно, они определяют некоторые операторы, специфичные для библиотеки Colorer и используемые только ею. К этим операторам относятся возможности сдвига границ регулярных выражений, операции, работающие на взаимодействие нескольких экземпляров РВ: ссылки на группы из других объектов, связывание РВ в цепочку. Многие расширенные возможности, хотя и не специфичны для библиотеки Colorer, широко в ней используются. Это, например, расширенный синтаксис именованных скобок (групп), операторы просмотра вперед и назад.

Разбор регулярных выражений в библиотеке реализован в классе `CRegExp`. Этот класс при инициализации компилирует переданную строку РВ во внутреннее древовидное представление, в котором каждый узел является каким-либо оператором. После компиляции при вызове одного из методов `parse`, отвечающих за нахождение соответствия между шаблоном и переданной последовательностью символов, результат сопоставления содержится в структуре `SMatches`, которая описывает все совпавшие группы и их границы.

Наличие нескольких вариантов метода `parse` обусловлено тем, что в различных условиях анализатору РВ нужно передавать различные параметры. Например, можно передать только одну строку, характеризующую целевую последовательность символов, либо же передать несколько дополнительных параметров, определяющих особенности проведения анализа текста.

3.3. Структура библиотеки

Общая структура функционирования библиотеки определяется входными данными, поступающими на анализ, внутренней структурой синтаксического анализатора и представлением выходных данных, полученных от него. В соответствии с этим большинство классов библиотеки принадлежит одному из этих трех модулей.

Помимо этого, библиотека реализует набор классов и функций верхнего уровня, предназначенных для работы с полученными от анализатора данными и представлением их в более удобной форме. Эти интерфейсы высокого уровня абстрагируют

приложение от внутренних особенностей устройства библиотеки и позволяют использовать ее как «черный ящик», реализующий требуемую функциональность. Конечное приложение может использовать эти классы, либо, при необходимости, более тонко контролировать действия библиотеки.

3.3.1. Представление входных данных

В конечном итоге вся работа, которую проводит библиотека, ведется над содержимым текстового документа. Следует заметить, что библиотека ориентирована на разбор однородных текстовых данных, коими и являются языки программирования. Хотя, например, текстовый процессор так же работает с последовательностями символов, образующих в совокупности документ, такие данные не подходят к использованию в библиотеке Colorer, так как они не однородны, а структурированы (различным компонентам текста могут быть присвоены различные атрибуты). Помимо этого, в них часто даже визуально отсутствует явное деление на строки (они плавающие).

Особенность библиотеки Colorer состоит в том, что она использует представление текста в построчном виде – это наиболее естественное представление для всех языков программирования. Хотя само такое разбиение можно назвать мнимым и визуальным (в самой программе текст может храниться иным образом), библиотека использует его как основу для построения динамического синтаксического разбора, кэширования и возвращения результатов.

Существующие прикладные программы, ориентированные на редактирование однородных текстов (plain text), применяют самые различные методы для внутреннего представления текста в памяти. Это диктуется требуемыми от них возможностями редактирования, удаления, изменения и перегруппировки частей текста.

Вне зависимости от этого представления, библиотека требует реализацию универсальной формы передачи ей текстовых данных. Для этого используется абстрактный класс (интерфейс) `LineSource`. Приложение обязано реализовать три определенных в нем метода (Табл. 4).

Табл. 4. Интерфейс LineSource.

Метод	*Описание*
`String *`**`getLine`**`(int lno)`	Основной используемый библиотекой метод. По запросу должен возвращать указатель на строку с индексом `lno`. Строка должна быть представлена в форме Unicode-класса `String`.
`void` **`startJob`**`(int lno)`	Вспомогательный метод, используется для уведомления клиента о начале анализа и, как следствие, потока запросов на содержимое строк текста.
`void` **`endJob`**`(int lno)`	Вспомогательный метод, используется для уведомления клиента о завершении текущего этапа запросов.

Вспомогательные методы могут использоваться приложением-клиентом для инициализации и завершения процесса функционирования интерфейса (например, для освобождения используемых ресурсов).

Метод `getLine` возвращает указатель на объект `String`, который является абстрактным представлением последовательности Unicode-символов. При этом объект содержит в себе информацию о длине передаваемой строки, которую библиотека использует при анализе. Передача данных через этот объект позволяет абстрагировать их

обработку библиотекой от реальной формы представления (кодировки), которая может произвольной.

На практике, требуемое библиотекой разбиение по строкам довольно часто применяется для реального представления данных файла в памяти. Это объясняется тем, что такое представление позволяет очень удобно реализовать различные операции изменения текста – вставка, удаление. Но, существуют и варианты реализации, использующие так называемые gap-хранилища. Они основываются на предположении, что все модификации содержимого файла локализованы большей частью в одном месте, и, поэтому, становится возможным представлять содержимое всего файла в виде линейного массива с «брешью», которая используется при добавлении содержимого. Но даже такая модель все равно обязана визуально отображаться в построчное представление, а это означает, что и реализации интерфейса `LineSource` для такого отображения не составит труда.

3.3.2. Представление выходных данных

Результатом работы синтаксического анализатора является поток событий, описывающих разобранную структуру текста. Для того чтобы позволить приложению и высокоуровневым компонентам библиотеки произвольно конфигурировать конечное представление этой информации, весь поток представляется в виде последовательностей вызовов интерфейса `RegionHandler`, реализация которого и будет решать, в какой конечной форме представлять данные:

```
class RegionHandler{
public:
  virtual void startParsing(int lno){};
  virtual void endParsing(int lno){};
  virtual void clearLine(int lno, String *line){};
  virtual void addRegion(int lno, String *line,
                                    int sx, int ex, const Region *region) =
            0;
  virtual void enterScheme(int lno, String *line, int sx, int ex,
                                    const Region *region, const Scheme
            *scheme) = 0;
  virtual void leaveScheme(int lno, String *line, int sx, int ex,
                                    const Region *region, const Scheme
            *scheme) = 0;
protected:
  RegionHandler(){};
  virtual ~RegionHandler(){};
}
```

Каждый из методов абстрактного класса `RegionHandler` отвечает за определенное синтаксическое событие (Табл. 5).

Табл. 5. Интерфейс RegionHandler.

Метод	*Описание*
`startParsing`	Вызывается один раз при запуске процесса анализа текста. Передает строку в тексте, с которой началась данная стадия анализа.
`endParsing`	Вызывается один раз при окончании текущего этапа анализа. Номер передаваемой строки указывает на конечную проанализированную строку.
`clearLine`	Вызывается один раз для каждой строки в процессе анализа.
`addRegion`	Основной метод, передающий поток разобранных синтаксических регионов. Каждый регион идентифицируется номером строки, его начальной и конечной позицией в строке и указателем на объект `Region`, описывающий тип этого региона.

enterScheme	Событие изменения контекста разбора (смену текущей схемы). Событие описывается начальной позицией, регионом контекста и самой схемой, на которую происходит переключение.
leaveScheme	Событие возврата из вложенного контекста. Параметры идентичны событию входа в контекст.

Все методы, за исключением информационных, получают указатель на текущую разбираемую строку, которым могут пользоваться по усмотрению (либо игнорировать). Основная нагрузка при разборе ложится на три последних метода, которые и определяют дерево регионов. При этом узлы этого дерева – события входа во вложенные схемы, а листья – обычные синтаксические регионы.

Библиотека накладывает некоторые ограничения на функционирование этого интерфейса. В частности, гарантируется корректность вложенных вызовов `enterScheme/leaveScheme`, все методы обязательно вызываются последовательно по отношению к номерам строк – поток не может «вернуться» назад по строкам. С другой стороны, анализатор не гарантирует отсутствие пересечения и монотонную упорядоченность вызовов `addRegion` в пределах одной строки – при необходимости обработчик должен сам принимать меры, обеспечивающие это.

Хотя приложение может предоставить библиотеке свой собственный обработчик событий, часто необходимости в этом нет. Библиотека содержит встроенный универсальный обработчик `LineRegionsSupport`. Этот класс использует набор вспомогательных объектов и позволяет сохранять структуру возвращаемых регионов в динамический массив (представленный вектором). Для работы с сохраненными регионами используется связанный список объектов `LineRegion`. Каждый такой объект содержит ссылку на соответствующий ему тип синтаксического региона, его позицию и размер. Помимо этого объект хранит указатель на другой объект – потомок `RegionDefine`. Задача этого объекта – определять расширенный свойства региона, которые будут использоваться при визуальном или ином отображении структуры на графический интерфейс. Сама библиотека реализует два варианта хранения такой информации, которых в большинстве случаев хватает с избытком. Основное представление – отображение в графическую (цветовую) информацию классом `StyledRegion`. При этом каждому региону ставится в соответствие цвет фона, текста и атрибуты шрифта (жирный, наклонный, и т.д.). Другой вариант – дополнение региона текстовой информацией, используемой для текстового форматирования конечного потока (например, в какой-либо язык разметки). Для этой цели используются объекты класса `TextRegion`.

Дополнение синтаксических регионов HRC расширенной информацией реализуется отдельными служебными классами, внешними по отношению к базе HRC кодов. Структура этих классов показана на Рис. 5. Внешние свойства хранятся в дополнительном формате HRD, который ставит в соответствие именам регионов различные расширенные свойства. Разделение визуального форматирования и логической синтаксической разметки является одним из базовых свойств библиотеки Colorer и всей ее базы HRC-кодов. Такая идеология отделяет визуальное представление от синтаксической структуры и позволяет использовать библиотеку в любых средах. Единая база HRC может отображается в различные цветовые схемы раскраски, которые можно менять и конфигурировать не затрагивая основные HRC коды.

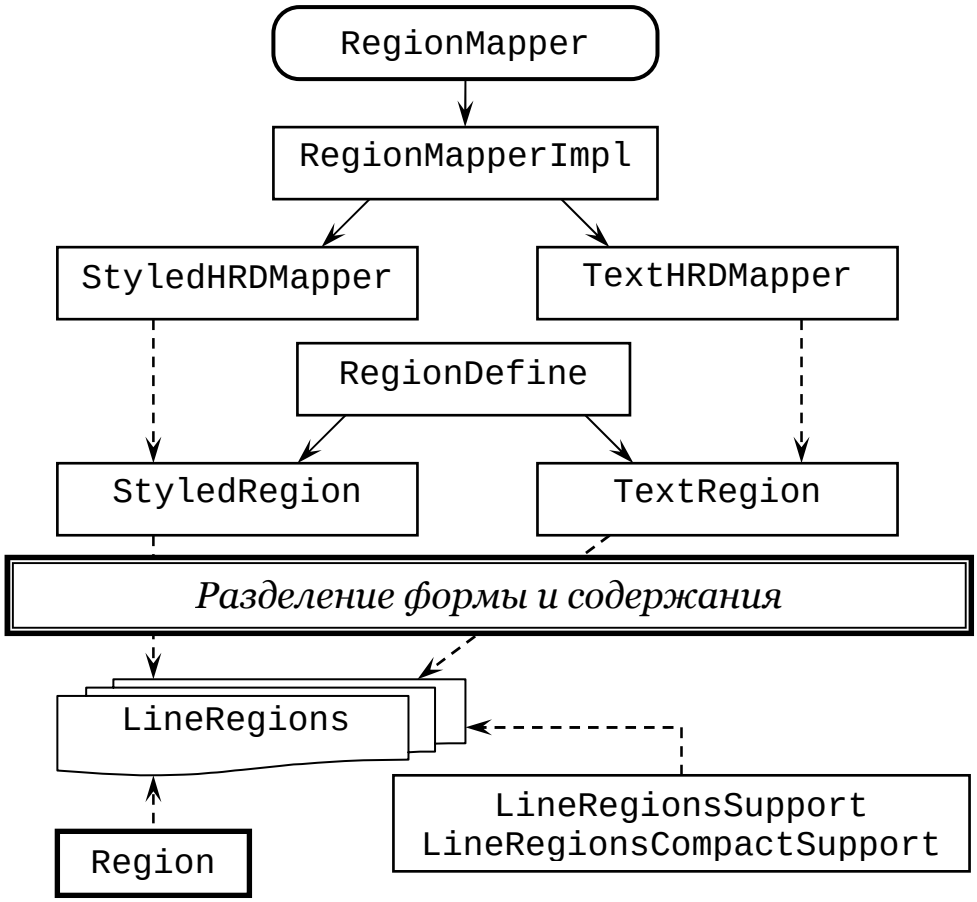

Рис. 5. Поддержка внешних свойств регионов HRC.

Помимо основного обработчика событий возможно использование его модифицированной версии (`LineRegionsCompactSupport`). Этот класс содержит дополнительные алгоритмы, обеспечивающие непересекающееся распределение регионов по строке. Такое представление часто необходимо в системах, где конечное отображение цветов в пользовательском интерфейсе редактора не имеет «цветового буфера» (который, например, есть в консольных системах).

3.3.3. Синтаксический анализатор

В процессе работы синтаксический анализатор использует входные данные, которые можно разделить на две категории: непосредственно содержимое разбираемого документа, представленное в форме программного интерфейса `LineSource`, и данные, описывающие структуру текста в терминах компонентов синтаксического анализа. Эти структуры хранятся в формате HRC, при инициализации они считываются и сохраняются в объекты библиотеки. Формат HRC основывается на синтаксисе XML, и для его разбора используется встроенный XML-процессор. В процессе разбора анализатор использует базу HRC для выделения в тексте синтаксических элементов и общей структуры и генерирует соответствующие события, которые затем передаются установленному обработчику.

В структуре синтаксического анализатора можно выделить два основных компонента. Первый отвечает за предварительный разбор и представление в памяти данных HRC для различных языков. Второй компонент – непосредственно синтаксический анализатор, использующий эти данные и обрабатывающих входной текст, получаемый от системы редактирования. Оба компонента представлены своими основными классами и набором вспомогательных.

Анализом HRC-кодов занимается класс `HRCParser`. В задачи этого класса входит начальная инициализация структур данных, представляющих HRC. Каждый компонент HRC моделируется соответствующим классом (Рис. 6).

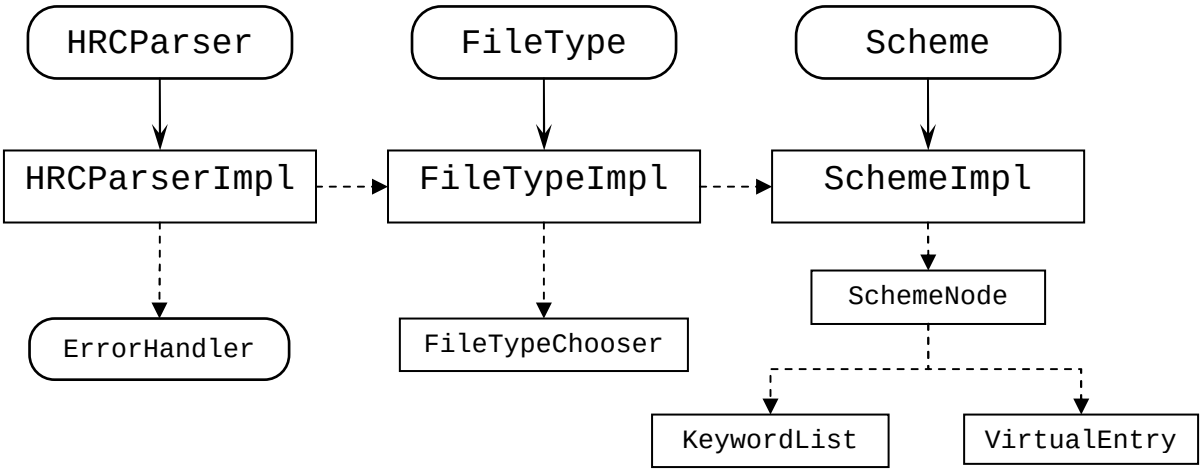

Рис. 6. Компоненты работы с HRC-кодами.

Загружаемая структура классов не просто повторяет модель исходных XML документов, в задачи загрузчика входит преобразование этой модели и подготовка ее к использованию анализатором. При этом необходимо провести максимально возможную подготовку исходных данных, так как последующий анализ наиболее критичен к скорости функционирования. При загрузке HRC-кодов происходит разрешение всех перекрестных ссылок на схемы, регионы и сущности HRC. В случае обнаружения конфликтов или ошибок об этом сообщается приложению через специальный обработчик `ErrorHandler`. Все регулярные выражения компилируются и подготавливаются к выполнению, списки ключевых слов сортируются и для них создаются вспомогательные структуры, использующиеся при анализе.

Анализ текста реализуется классом `TextParser` (Табл. 6). Этот класс использует структуры HRC, и для выбранного языка программирования осуществляет синтаксический разбор текста.

Табл. 6. Методы класса TextParser.

Метод	*Описание*
`setFileType`	Выбор типа файла (языка программирования), который будет использоваться для работы этого анализатора.
`setLineSource`	Установка входного источника текстовых данных в виде интерфейса `LineSource`.
`setRegionHandler`	Установка приемника разобранной синтаксической структуры текста.
`parse`	Запуск анализа с использованием кэширования. Анализ проводится для отрезка текста, указанного в параметрах вызова.
`breakParse`	Принудительный останов синтаксического анализа. Применяется при многопоточной обработке текста для отмены анализа из другого потока.
`clearCache`	Принудительный сброс кэша для всего текста.

Особенностью работы библиотеки Colorer является то, что она должна учитывать возможность изменения текста и поддерживать синтаксические структуры в соответствии с действиями пользователя. Для реализации этой возможности библиотека осуществляет частичный разбор текста по запросу приложения. При любом изменении приложение, для того, чтобы получить обновленную информацию о структуре, вызывает анализатор, который проводит разбор требуемой части текста. При этом анализ начинается с места изменения и продолжается для всей визуальной (видимой на экране) части текста.

Для того чтобы поддерживать такую модель разбора, анализатор в процессе работы ведет кэширование внутренних структур данных. Это кэширование полностью прозрачно и не влияет на работу приложения. Используя эту информацию, анализатор способен восстановить процесс разбора с любого места уже разобранного текста. Разбиение структуры входного текста на строки и объясняется необходимостью кэширования данных. Именно индексы строк в тексте используются для задания требуемых границ разбора и для сохранения данных во внутреннем кэше.

Классы и интерфейсы, используемые на этапе анализа, показаны на Рис. 7.

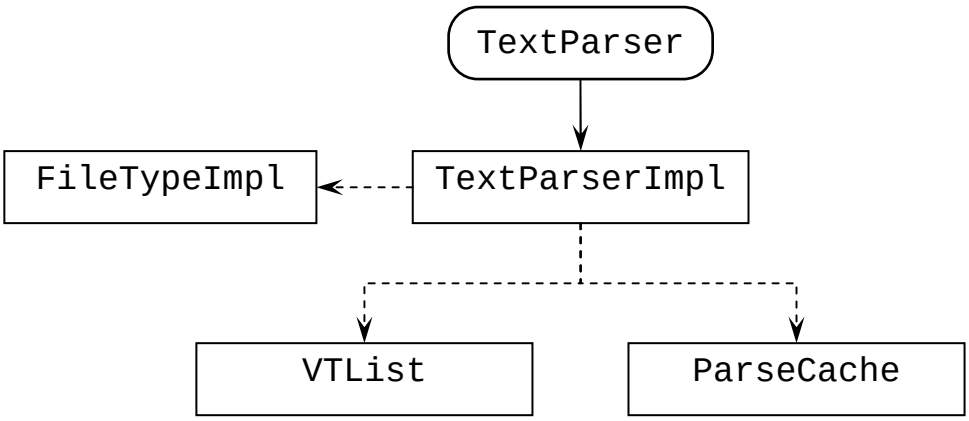

Рис. 7. Структура модуля анализа текста.

Структуры объектов `VTList` используются для реализации динамических последовательностей списков виртуальных схем. Эти объекты содержат указатели на списки подстановок соответствующих схем, реализуют поддержку этих списков, операции с ними и сохранение/восстановление текущего своего состояния с использованием кэша.

В процессе анализа последовательно для всего текста производится сопоставление с ним компонент схем, начиная с корневой схемы выбранного языка программирования. При нахождении совпадения генерируется соответствующее ему событие, которое передается внешнему обработчику. Этот процесс продолжается для всего текста с учетом рекурсивных входов во вложенные контексты и обходы наследуемых схем. Одновременно с анализом строится или модифицируется структура кэша, которая используется на следующих этапах разбора.

Все основные классы модуля синтаксического анализа отображаются на соответствующие им абстрактные интерфейсы. Это сделано для достижения большей модульности и заменяемости компонент. Клиентская часть приложения общается с синтаксическим анализатором через интерфейсы, реальный же объект создается централизовано специальным factory-методом в классах высокоуровневой прослойки.

3.3.4. Высокоуровневые интерфейсы

Непосредственное использование классов библиотеки конечными приложениями довольно нетривиально, так как помимо управления основными объектами анализа необходимо вести учет различных вспомогательных объектов и структур и работать с

ними. Часто это довольно сложная задача. Помимо того, большинство требований программ по поддержке библиотеки Colorer сходны и не сильно различаются.

Для того чтобы избежать необходимости в дублировании кода по работе с самой библиотекой и по обработке данных, полученных от нее, был создан набор классов и модулей, реализующий «прослойку» верхнего уровня и представляющий процесс взаимодействия с библиотекой в упрощенной форме. Эта реализация, во-первых, включает в себя гибкое управление всеми ресурсами библиотеки через класс `ParserFactory`. Этот класс поддерживает простую структуру файла-каталога, в котором перечислены все доступные файловые ресурсы библиотеки и их расположение. К ресурсам относится информация о положении основных HRC-файлов, описание и обозначение существующих в поставке цветовых схем (HRD-файлов). Кроме того, этот класс может создавать соответствующие этим ресурсам объекты библиотеки, автоматически устанавливать между ними нужные связи и передавать их клиентской части приложение. Таким образом, приложение избавляется от необходимости вручную управлять размещением ресурсов библиотеки. Файл каталога (`catalog.xml`) использует простой формат хранения данных, путь к этому файлу может быть указан приложением явно, либо же библиотека предпримет ряд попыток найти его в стандартных местах.

Хотя класс `ParserFactory` избавляет от основной рутинной работы по поддержке ресурсов библиотеки, он не избавляет приложение от прямого общения с синтаксическим анализатором и его данными. Для того чтобы упростить эту часть библиотеки, был создан класс `BaseEditor`, который предназначен для связи непосредственно с кодом целевого приложения. Этот класс берет на себя всю работу по вызову синтаксического анализатора и по его настройке, а сам предоставляет приложению простой интерфейс, который уже не привязан к внутренним структурам библиотеки. Набор основных методов класса приведен в Табл. 7.

Табл. 7. Методы класса BaseEditor.

Метод	*Описание*
`setRegionMapper`	Выбор требуемой цветовой схемы, используемой для отображения синтаксических регионов в цвета.
`setFileType`	Выбор используемого языка программирования.
`getPairMatch`	Поиск начальной парной конструкции в указанной позиции строки.
`searchLocalPair`	Поиск конечной парной синтаксической конструкции по указанной начальной в пределах видимой области.
`searchGlobalPair`	Поиск конечной парной синтаксической конструкции по указанной начальной в пределах всего текста.
`getLineRegions`	Возвращает (и при необходимости вычисляет) список синтаксических регионов для запрошенной строки.
`validate`	Принудительно обновляет синтаксическую структуру текста по указанный номер строки.
`addRegionHandler`	Добавляет внешний обработчик синтаксической структуры в процесс анализа.
`removeRegionHandler`	Удаляет ранее установленный обработчик.
`modifyEvent`	Оповещает объект о модификации текста начиная с определенной позиции.
`visibleTextEvent`	Событие изменения визуально отображаемой части текста.
`lineCountEvent`	Событие изменения общего числа строк в тексте.

После требуемой настройки объекта, которая соответствующим образом отображается в конфигурирование его внутренних свойств, прикладной код сообщает обо всех изменения в визуальном представлении текста через набор событий. В ответ на это, вызов метода `getLineRegions` возвращает список синтаксических (цветовых) регионов, привязанных к указанной строке. Приложение, используя его, реализует расцветку синтаксиса по своим правилам и в зависимости от своей архитектуры.

Помимо этого класс полноценно реализует возможность работы с парными синтаксическими конструкциями (поиск, переход по ним). Для этого используется специальный объект `PairMatch`, который сохраняет информацию о найденной паре элементов, их свойствах и атрибутах. Используя его, приложение может реализовать мощные возможности визуальной подсветки этих пар в тексте (например, при наведении курсора на них), перехода и выделения.

Доступные служебные методы установки произвольного обработчика регионов позволяют целевому приложению извлекать дополнительную информацию из анализа текста. Один из таких обработчиков (класс `Outliner`), уже реализованный в библиотеке Colorer, отвечает за обработку и построение структуры текста исходя из передаваемых синтаксических регионов. На основе этого обработчика, например, реализуются возможности отображения пользователю списка всех функций, классов и объектов в редактируемом тексте, выдача всех обнаруженных синтаксических ошибок с возможностью перехода по ним.

Эти возможности реализуются на основе анализа типов регионов, передаваемых обработчику. Описания HRC могут использовать некоторые из базовых объявленных типов, и в произвольном языке вводить возможности создания таких структур (outline views). Сама эта возможность очень удобна, так как позволяет пользователю в процессе редактирования текста быстро переходить на нужные его части и объявления. Например, сейчас просмотр структуры реализован не только в обычных языках программирования (C/C++, Java, Pascal), но и в некоторых других (HTML, XML). Добавление в язык поддержки отображения структуры текста не требует модификации исходных кодов библиотеки, а реализуется на уровне редактирования их HRC-описаний. Общие связи между описанными классами представлены на Рис. 8.

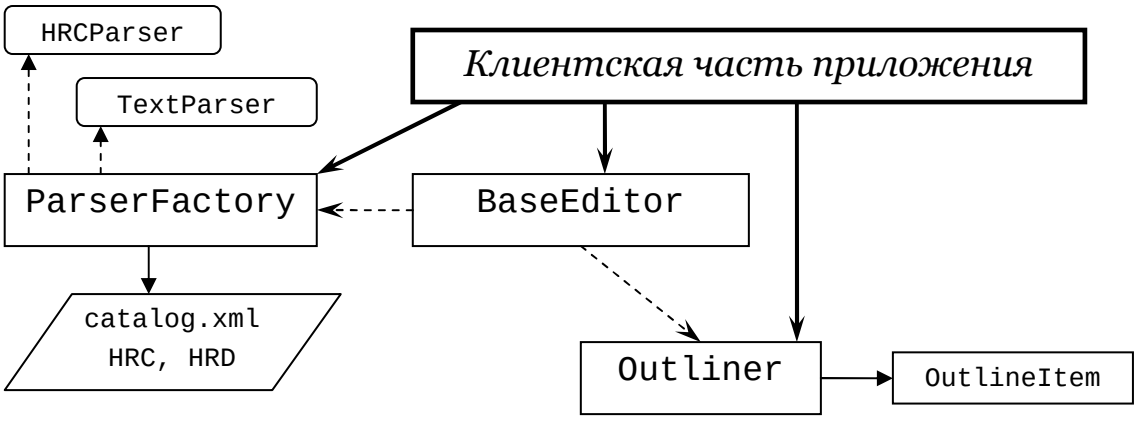

Рис. 8. Модули верхнего уровня библиотеки.

Таким образом, для использования библиотеки достаточно работы с парой классов – `ParserFactory` и `BaseEditor`. Возможностей, реализуемых ими, хватает для большинства систем редактирования. Использование каталога `ParserFactory` фактически стало стандартным, так как удобно и централизовано позволяет управлять ресурсами библиотеки. На основе класса `BaseEditor` реализовано несколько вспомогательных функций, связанных с преобразованием проанализированных данных в

текстовую форму HTML-файлов, описывающих расцвеченные входные исходные тексты. Эта возможность может использоваться для публикации исходных кодов, ознакомления с ними в удобной и визуальной форме.

3.4. Интерфейс Java

При внедрении библиотеки Colorer в реальные системы редактирования был создан интерфейс для виртуальной машины Java с использованием JNI (Java Native Interface). В процессе разработки это позволило в наиболее общей форме определить требования к API библиотеки, формализовать многие понятия и принципы взаимодействия ее компонентов. В результате созданная структура классов и интерфейсов языка Java практически полностью повторяет соответствующую структуру языка C++. Отображение многих объектов заключается лишь в минимальных изменениях синтаксиса их описаний – все структуры и параметры методов полностью совпадают.

Для связывания интерфейсов Java с реально функционирующей C++ частью библиотеки используется стандартный интерфейс JNI, позволяющий осуществлять взаимодействие Java-классов с native-кодом (кодом, скомпилированным под текущую платформу). Интерфейс JNI основывается на довольно простом и открытом способе управления Java-объектами, представляющим собой множество экспортируемых функций, принимающих различные параметры. В совокупности они обеспечивают способы взаимодействия с виртуальной машиной Java, независимые ни от платформы, ни от внутренней архитектуры самой виртуальной машины.

JNI шлюз библиотеки Colorer интересен тем, что в его функции входит поддержка передачи данных в обе стороны. От Java-кода необходимо принимать текстовую информацию для синтаксического разбора, со стороны C++ модулей – передавать разобранные данные для дальнейшей обработки. При этом такой шлюз должен обеспечивать взаимодействие между двумя объектными моделями, работая как простой набор функций языка C. Если для объектов C++ это нормально, то для поддержки особенностей языка Java необходима дополнительная обработка. В частности, JNI шлюз поддерживает модель сборщика мусора (garbage collector), обеспечивая автоматическое освобождение ресурсов библиотеки по мере удаления соответствующих Java-объектов. Из соображений производительности уменьшается число возможных переключений контекстов выполнения Java/Native за счет создания Java-копий некоторых примитивов библиотеки.

Общая структура Java-классов абстрагирует целевое приложение от информации о функционирующей native-прослойке. Этим достигается возможность изменять внутренние связи с C++ библиотекой и конфигурировать функциональность интерфейса для достижения оптимальной производительности. Так, некоторые из компонентов высокого уровня были реализованы не через JNI, а переписаны целиком на Java. Следует заметить, что созданный интерфейс не утратил основного свойства языка Java – независимости от используемой аппаратной платформы. Native-часть библиотеки без изменений компилируется и работает под любой процессорной архитектурой.

4. Язык HRC

Формат HRC (Highlighting Resource Codes) используется в библиотеке Colorer для хранения правил разбора исходных текстов, синтаксической и цветовой информации. Библиотека использует ее для анализа предоставляемого приложением-клиентом файла и выделения в нем синтаксических регионов для дальнейшей раскраски и других действий. Формат HRC определяет набор правил, которые используются для описания синтаксических и лексических элементов языка: схем (контекстов), регулярных выражений и ключевых слов (лексем). Он определяет так же отношения между схемами по наследованию, вызову и виртуализации.

Язык HRC использует XML для выражения своих конструкций и определяет свой собственный словарь XML. Помимо соответствия спецификации XML, для синтаксиса HRC определяются пространство имен и XML схема, его описывающая. Использование схемы позволяет автоматизировать проверку корректности всех HRC-файлов, входящих в состав библиотеки Colorer. Использование XML как основы синтаксиса языка дает очень много преимуществ. Конечные HRC файлы становится возможным обрабатывать различными стандартными средствами, начиная от XML-редакторов и кончая возможностью создания преобразований, переводящих содержимое HRC в другие форматы и наоборот.

Текущая версия библиотеки содержит HRC-описания более чем для 100 различных текстовых форматов и языков. При анализе существующих языков программирования, скриптов и технологий становится ясно, что большинство из них образуют группы, которые тесно взаимосвязаны друг с другом и часто не могут существовать обособленно. Изолированная реализация синтаксиса каждого нужного языка привела бы к большой избыточности данных, отсутствию гибкости, неудобству работы. HRC, помимо описания синтаксиса языков, имеет возможности создания связей между различными языковыми элементами, позволяющими интегрировать и пересекать различные языки. Это приводит к уменьшению дублирования описаний синтаксиса и увеличению гибкости всей языковой базы, предоставляемой библиотекой Colorer.

4.1. Структура языка

Для описания языков Colorer использует язык разметки XML. Использование XML позволяет стандартизировать язык HRC и облегчить процесс редактирования описаний. Хотя сейчас библиотека использует свой внутренний анализатор для разбора HRC, ничто не мешает в будущем подключить внешние модули анализа, к примеру, для переноса библиотеки на другую платформу. Вместе с HRC кодами поставляются DTD и XSD схемы, а так же простое XSLT преобразование, переводящее HRC в HTML и позволяющее пользователю визуально оценить и разобрать содержимое HRC файла.

В соответствии со спецификацией XML все HRC файлы имеют следующий стандартный заголовок:

```
<?xml version="1.0"?>
<!DOCTYPE hrc PUBLIC "-//Cail Lomecb//DTD Colorer HRC take5//EN"
  "http://colorer.sf.net/2003/hrc.dtd">
<hrc version="take5" xmlns="http://colorer.sf.net/2003/hrc"
     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
     xsi:schemaLocation="http://colorer.sf.net/2003/hrc
             http://colorer.sf.net/2003/hrc.xsd">
```

В этот заголовок входит определение типа документа (`DOCTYPE`), указание пространства имен документа, используемого по умолчанию, и сопоставление пространства имен XML схеме, его проверяющей (атрибут `schemaLocation`).

Корневым элементом любого HRC файла является тэг `<hrc version="take5">`. Его параметр `version` используется библиотекой и при-

кладной программой для проверки соответствия версии HRC-базы и текущего анализатора.

4.1.1. Прототипы

Загрузка базы начинается с анализа корневых файлов, которые определяют прототипы всех доступных языков программирования. Основным корневым файлом является файл `/hrc/colorer.hrc`, который ссылается на другие файлы HRC-базы, содержащие описания синтаксиса. Каждый прототип определяет свойства одного языка программирования (Табл. 8).

Табл. 8. Компонент prototype языка HRC.

Атрибут	*Описание*
`name`	Имя определяемого языка программирования. Используется как уникальный идентификатор для работы с этим языком и его компонентами
`group`	Группа, к которой принадлежит данный язык. Используется для удобства при работе со списком языков (для группировки их по областям)
`description`	Символическое описание языка, которое используется для отображения в пользовательском интерфейсе.
`targetNamespace`	Целевое пространство имен. Применим к автогенерируемым языкам программирования, основанным на XML. Используется для создания возможных связей между ними.
Элемент	***Описание***
`location`	Элемент используется для указания URL файла, содержащего определение синтаксиса этого языка программирования. Этот файл должен определять тип (`type`) с именем, соответствующим имени этого прототипа.
`filename`	Регулярное выражение, используемое для выбора тех имен файлов, которые соответствуют этому языку программирования.
`firstline`	Регулярное выражение, которое так же используется для выбора языка, соответствующего загружаемому файлу, но выбор при этом происходит исходя из анализа первой строчки в файле.
`parameters`	Список произвольных параметров, которые могут использоваться прикладной программой, работающей с библиотекой, для хранения различных дополнительных свойств языка.

Определение прототипа включает в себя данные, использующиеся в алгоритме определения языка, соответствующего загружаемому на редактирование исходному файлу. Целевая система редактирования может содержать свой алгоритм, определяющий какой тип HRC нужно использовать для расцветки редактируемого файла. Но, библиотека Colorer предоставляет свой собственный вариант определения типа загружаемого языка. Этот алгоритм очень гибок и может использоваться как самостоятельно так и вместе с другими, внешними условиями.

При определении типа файла очень часто бывает недостаточно просто информации о его расширении. У многих языков существуют пересекающиеся имена расширений, сами же языки различаются по содержимому. Поэтому в библиотеке введено два оператора, выявляющих тип языка – `filename` и `firstline`. Оба содержат регулярные выражения, которые в первом случае сопоставляются имени файла, а во втором – первой строке файла. Каждый из этих операторов при объявлении может указываться несколько раз с различным содержимым РВ. При этом каждому экземпляру присваивается воображаемый вес (который можно задать явно через атрибут `weight` или оставить значение по умолчанию). При загрузке конкретного файла из всех прототипов выбирается первый, у которого суммарный вес максимален. Этот алгоритм хорошо зарекомендовал себя, так как позволяет различать даже очень смежные языки программирования (например, HTML и XHTML, ASP с различными скриптовыми языками).

Выделение прототипов из определений самих языков обусловлено тем, что вся база HRC, которая находится в распоряжении библиотеки, определяет очень много языков программирования и единовременная загрузка их всех – довольно долгий процесс. Поэтому при первой инициализации библиотеки строится лишь список прототипов всех языков, каждый из которых имеет ссылку на файл с определением самого типа языка (параметр `location`). Как только в процессе работы пользователь запросит на редактирование файл определенного языка программирования, библиотека, используя прототип языка, загружает соответствующий ему тип, в котором содержатся все требуемые определения синтаксиса. Такая модель позволяет не тратить ресурсы и время на загрузку языков, которые пользователем никогда не используются.

4.1.2. Описания языков программирования

Определение типа языка включает в себя всю информацию, требуемую для синтаксического разбора текста и его расцветки. Язык HRC определяет три компонента уровня типа языка, с помощью которых реализуется синтаксический разбор: регионы, сущности и схемы.

Регион в HRC – это базовое понятие, описывающее разобранную неделимую синтаксическую единицу. Множество синтаксических регионов является результатом работы анализатора. Каждый регион в зависимости от своего определения может в дальнейшем играть различные роли при анализе. Он может соответствовать определенному цвету, которым будет выделена соответствующая лексема в тексте, либо же играть роль метки, с помощью которой приложение может разбираться в структуре текста и предоставлять различные расширенные возможности.

Каждому региону присваивается уникальное имя, используя которое на этот регион будут ссылаться другие компоненты HRC. Помимо этого регион может содержать расширенное описание, которое будет отображаться пользователю. Все объявленные регионы образуют древовидную структуру, так как каждый регион может указывать своего родителя. Эта возможность позволяет ввести типизацию регионов и разделить их по назначению. Так, множество регионов, унаследованных от `def:Outlined` обрабатываются библиотекой как скрытые (то есть не будут соответствовать какому-либо цвету), но при этом будут учитываться при построении отдельного дерева структуры текста. Например, для языков C/C++ определяется следующий набор структурных регионов:

```
<region name="ClassOutline" parent="def:Outlined"/>
<region name="StructOutline" parent="def:Outlined"/>
<region name="UnionOutline" parent="StructOutline"/>
<region name="EnumOutline" parent="StructOutline"/>
<region name="FuncOutline" parent="def:Outlined"/>
<region name="DefineOutline" parent="def:Outlined"/>
```

Каждый из этих регионов определяет компонент структуры языка, которые могут отображаться приложением в отдельном окне структуры языка (outline view).

Следующий компонент верхнего уровня в HRC – это *сущности*. Сущности HRC сходны по функциональности с сущностями XML, но имеют узкую направленность на работу в компонентах регулярных выражений. Каждая сущность определяет часть регулярного выражения, которая потом может использоваться как сокращение при описании одного или нескольких регулярных выражений. Отличие их от XML сущностей состоит в том, что сущности HRC, как и любой другой компонент, могут использоваться из разных типов языков, являясь структурным объектом языка, а не синтаксическим. Второе отличие и назначение – возможность сокращения и оптимизации использования памяти при разборе HRC, которая может быть реализована путем компиляции сущности в компонент PB и создания простых ссылок на нее из точек вызова внутри PB.

Последний, и основной компонент языка – это *схема* (scheme). Схема – это однородная область, содержащая определенный набор синтаксических элементов. Понятие схемы сходно с понятием состояния, контекста (в некоторых других системах синтаксического анализа этот термин так и называется: *контекст*). Схемой, например, будет язык C – эта схема содержит элементарные синтаксические элементы – лексемы (ключевые слова, числа, строки). В то же время, схемой является и содержимое комментария вида `/* */` – это пустая схема, так как она не содержит никаких лексем. Из этого примера видно, что существуют схемы, у которых нет явных границ – схемы верхнего уровня, и схемы, которым присуще понятие границ – начала и конца. Одно из отличий системы Colorer от других систем анализа состоит в том, что в ней лексемы, описывающие границы схемы, не являются атрибутами этой схемы. То, в какие границы заключить схему, решает вызывающая ее схема высшего уровня. Это свойство позволяет абстрагировать схему от ее языка – и, тем самым, использовать возможности межъязыковых взаимодействий.

Большинство схем верхнего уровня соответствуют каким-либо реальным языкам. Для определения того, какая из схем в данном типе языка – самая верхняя, используется имя типа. Та схема, локальное имя которой совпадает с именем ее типа и является корневой для этого языка.

4.1.3. Пространства имен HRC

Каждый тип (язык программирования) фактически определяет свое собственное пространство имен, в котором описываются все другие компоненты языка HRC. Внутри этого типа ссылки между элементами языка разрешаются однозначно, для ссылок между различными типами нужно использовать полное имя элемента – с префиксом типа, которому он принадлежит.

Пространство имен HRC разбивается на подпространства, каждое из которых соответствует имени объявленного типа. В свою очередь, каждое из этих подпространств не является однородным, а содержит три линейных пространства имен, в которых определяются соответствующие компоненты HRC верхнего уровня: регионы, сущности и схемы. В соответствии с таким разбиением локальные имена у различных компонентов одного типа могут пересекаться. Это не вызывает проблем, так как все ссылки на эти компоненты разрешаются с учетом контекста – при этом известен тип запрашиваемого компонента с таким именем. Аналогично могут совпадать и локальные имена у компонентов одного типа, но принадлежащих разным языкам. При этом для разрешения конфликтов используются правила именования глобальных имен (с префиксом типа).

Глобальные имена компонентов (с префиксом имени типа) используются для взаимодействия между различными типами. Внутри определения типа можно исполь-

зовать как локальные так и глобальные имена компонентов. Локальные имена ссылок на компоненты других типов допустимы, если в текущем типе объявляется импорт имен из других типов. Операция импорта так же является компонентом верхнего уровня в определении типов языков, но не несет никакой действенной нагрузки, а позволяет упростить и сократить описание HRC.

4.2. Схемы и контексты

Если все описанные выше структуры языка HRC работают в качестве прикладных и вспомогательных средств, то содержимое тэга `scheme` (описание схемы) имеет непосредственное отношение к работе синтаксического анализатора. В этом блоке задаются все синтаксические элементы схемы языка и связи этой схемы с другими языками.

Любая схема идентифицируется в базе HRC по своему уникальному имени (паре `имя типа:имя схемы`). Схема может содержать четыре управляющие синтаксисом конструкции в любом количестве и порядке. Порядок описания элементов схемы важен и влияет на процесс синтаксического разбора. Каждый из этих элементов прямым или косвенным образом определяет синтаксические и лексические элементы, начиная простыми и заканчивая сложными многосхемными конструкциями (Табл. 9).

Табл. 9. Компонент scheme языка HRC.

Элемент	*Описание*
`keywords`	Определение списка ключевых слов с однородными свойствами. Любому из перечисленных в этом блоке слов при нахождении в тексте присваивается определенный синтаксический регион.
`regexp`	Регулярное выражение – это базовый компонент, использующийся в анализе текста. С помощью РВ возможно распознать требуемую последовательность символов и присвоить ей целиком, либо ее частям различные синтаксические регионы.
`block`	Компонент, служащий для переключения контекста. Содержит два РВ, между которыми в тексте текущая схема изменяется на указанную в этом элементе.
`inherit`	Оператор, служащий для связывания различных схем. В простейшем случае позволяет включить в содержимое текущей схемы содержимое указанной (наследуемой) схемы.

4.2.1. Списки ключевых слов

Блок `keywords` используется для задания списка ключевых слов. Набор ключевых слов (лексем), указанный внутри него с помощью тэгов `word` или `symb` задает список слов (или произвольных последовательностей символов), подлежащих выделению регионом, указанным в параметре `region`. Указанный список слов сортируется на этапе загрузки HRC, и обрабатывается через кэш, что позволяет ускорить синтаксический анализ и нахождение в тексте нужных последовательностей. Параметр `ignorecase` указывает анализатору, что данный набор ключевых слов в тексте не различается по регистру символов (в схеме Pascal, к примеру, этот флаг должен быть установлен). Дополнительный параметр `worddiv` служит для изменения установленного по умолчанию набора символов разделителей слов. В некоторых языках класс символов слов отличается от принятого по умолчанию (буквы + цифры + знак подчер-

кивания). Например, в языке *Форт* идентификатор может состоять практически из любых символов кроме пробела и скобок. Таким образом, для этого языка параметр должен быть установлен в `worddiv="[\s\[\]\{\}\(\)]"`. Класс символов задается в форме, принятой в регулярных выражениях.

Хотя функциональности блока `keywords` можно добиться, используя синтаксис регулярных выражений, этот вариант предпочтительней в случае большого количества ключевых слов. Предварительная обработка и оптимизированный поиск по последовательности позволяет добиться высокой скорости даже на очень больших наборах лексем.

4.2.2. Регулярные выражения

Основной и наиболее часто используемый элемент HRC – `regexp`. Он определяет через регулярное выражение произвольный синтаксический элемент и задает соответствующие ему синтаксические (цветовые) регионы.

Через регулярные выражения задаются все основные лексемы в схемах языков: числа, строки, комментарии, языковые конструкции – но в пределах одной строки. Именно это ограничение и создает специфику работы синтаксического цветового анализатора. Colorer работает как модуль в системе редактирования текстов, а любая такая система в конечном счете оперирует не целым текстом, а его блоками – чаще всего строками. То есть, нет возможности расширить область действия регулярного выражения более чем на строку. Другая причина – необходимость ограничения возможных движений по тексту, просмотров вперед и откатов. Это связано с особенностью работы синтаксического анализатора в режиме реального времени, при котором нужно обеспечивать специальное кэширование результатов анализа.

Результатом проверки на совпадение любого регулярного выражения является ответ о его совпадении/несовпадении. Помимо этого регулярное выражение возвращает границы совпадения в тексте и информацию по областям совпадения (их границам и количеству). Эта информация и используется для сопоставления тексту синтаксических регионов. Например, выражение

```
<regexp match="/\b 0[xX][\da-fA-F]+ (?{NumberSuffix}[UuLl]{1,2}|(i64)|(i128))?
            \b/x" region="NumberHex"/>
```

задает шестнадцатеричное число, записанное в правилах C/C++. Элемент `region` выделяет все совпавшее регулярное выражение как область (цвет) `NumberHex`, а именованный регион `NumberSuffix` выделяет возможный суффикс числа. Первый регион в регулярном выражении – это первая группа, равная всему РВ. Всего возможно использование шестнадцати регионов для одного регулярного выражения: `region0` – `regionf`. Каждый из этих нумерованных регионов будет соответствовать номеру соответствующего блока в РВ (номеру группирующих скобок). Возможно применение расширенного синтаксиса именованных скобок, в которых скобкам не присваивается номер, а заданное имя используется как имя синтаксического региона: `(?{NumberSuffix} ... )`.

Еще один возможный параметр тэга `regexp` – это флаг `lowpriority`, принимающий значения `yes/no`. Он используется при взаимодействиях между схемами и влияет на приоритет разбора схем.

4.2.3. Переключение контекста

Одни только регулярные выражения не позволяют реализовать все существующие языковые конструкции. Первая причина состоит в том, что большинство из них просто выходит за рамки регулярных языков. Вторая причина – невозможность работы регулярных выражений на нескольких строках. Обычный многострочный комментарий в *C++* или *Pascal* стиле уже не может быть представлен регулярным выражением (в

силу ограничений действия РВ на одну строку). С другой стороны любая конструкция контекстно-свободного языка (например, неограниченно вложенные скобки) даже в пределах одной строки не может быть разобрана (регулярные выражения охватывают только класс регулярных языков). Для преодоления этих ограничений, помимо синтаксических структур (регулярных выражений) используются лексические структуры, которые могут связывать различные схемы.

Тэг `block` использует регулярные выражения для задания границ вложенной схемы. При обнаружении признака начала блока с найденной позиции начинает работать не текущая схема, а указанная в параметре `scheme` этого блока. По достижении признака закрытия блока происходит возврат из схемы и продолжение разбора текущей схемы. На языке контекстов происходит переключение контекста и его восстановление. Такой процесс функционирования сходен с работой конечных автоматов с магазинной памятью. При этом сами конечные автоматы соответствуют обычным регулярным конструкциям (РВ и ключевым словам), а работа с магазинной памятью ведется через описанный компонент `block` (сами схемы при этом оказываются элементами магазинной памяти).

С точки зрения цветового выделения, начало и конец блока (атрибуты `start` и `end`) выделяются аналогично простым регулярным выражениям, и, дополнительно, вводится регион для выделения всего блока. Исходя из этого механизма, обработка таких структур, как многострочные комментарии происходит путем переключения на пустую схему. Аналогично обрабатываются иные рекурсивные конструкции:

```
<scheme name="cNestedComment-internal">
  <inherit scheme="Comment"/>
  <block scheme="cNestedComment-internal" region="Comment">
    <start region="PairStart">/\/\*/</start>
    <end region="PairEnd">/\*\//</end>
  </block>
</scheme>

<scheme name="cNestedComment">
  <block scheme="cNestedComment-internal" region="Comment">
    <start region="PairStart">/\/\*/</start>
    <end region="PairEnd">/\*\//</end>
  </block>
</scheme>
```

В этом примере пара схем реализует рекурсивно вложенный С-комментарий. Из него так же видно, что схема свободно может вызывать саму себя. Параметр `scheme` может ссылаться даже на еще не определенную схему – из-за специфики разбора это не вызывает никаких проблем. Параметр `region` указывает на общий цветовой регион вызываемой схемы, а параметры `region0i` и `region1j` задают регионы для регулярных выражений `start` и `end`.

Работа тэга `block` тесно связана с работой регулярных выражений его определяющих. Вызываемая схема так же содержит регулярные выражения и вызовы других схем. Поэтому тэг `end` взаимодействует, конкурирует с ними, и из-за этого могут возникать конфликты. Самый простой конфликт – откат возврата схемы. Он происходит, когда на строке найдено завершающее блок выражение, но до того, как анализатор дойдет то этого места, другое регулярное выражение или блок из работающей схемы захватит эту часть строки и запретит возврат из схемы. В конкретных примерах эта ситуация может быть благоприятной (то есть повторять структуру языка) или же нежелательной. Например, в языке *C* при вызове языка *Assembler* может быть конфликт при закрытии блока:

```
__asm{
  mov ax, 4C00h
  int 21h
  // close bracket in comment }
};
return true;
```

В этом примере анализатор находит закрывающую скобку `asm` блока, но последующий разбор регулярного выражения (комментария) захватывает эту скобку и откатывает возврат схемы. Из структуры языка должно следовать именно такое поведение. Аналогичная ситуация, но с языком ASP (Active Server Pages). Этот язык визуально представляет собой HTML текст со вставками кода, которые выполняются на стороне web-сервера:

```
<table>
..
<td width="<%=tdSize() REM visual basic comment %>">
..
</table>
```

В этом случае ASP код, выделенный в блок `<% ... %>` содержит комментарий. Но этот комментарий не осуществляет отката: он должен выделяться до закрывающего ASP тэга `%>`.

В связи с тем, что нельзя отдать предпочтение какой-то одной ситуации, необходимо ввести явное указание необходимого поведения. Контролируется такое поведение регулярных выражений параметром `priority`. Этот флаг может присутствовать как у линейного регулярного выражения, так и у блока и принимает два возможных значения: `normal` и `low`. Последний указывает анализатору, что данное регулярное выражение имеет пониженный приоритет (по умолчанию приоритет обычный) и не может вызывать откат возврата схемы. Блоки и регулярные выражения по умолчанию имеют высокий приоритет, который может быть понижен, а списки ключевых слов – низкий. Следует заметить, что приоритет блока (`block`) вводится аналогично приоритету регулярного выражения (`regexp`): понижение приоритета действует на начальное регулярное выражение блока, так как на этапе поиска совпадения для вложения схемы (`start`) механизм анализа работает аналогично линейному регулярному выражению.

Параметр понижения приоритета действует на отдельные компоненты в схеме, которые его явно указывают. Использование этого параметра явно подразумевает, что содержимое схемы (или совокупности схем) реализует заданные приоритеты. Но, часто возникают ситуации, когда вызываемая схема ничего не знает о требуемом от нее поведении. При этом вызывающей схеме необходимо каким-то образом управлять приоритетом вызываемой. Для реализации этой возможности служит атрибут content-priority, который применим только к компоненту `block`. Будучи установленный в значение `low`, он принудительно понижает приоритет всех компонентов вызываемой схемы. Хотя того же эффекта можно добиться путем выставления `priority='low'` у всех компонентов схемы, этот атрибут необходим при создании схем, взаимодействующих с различными языками, так как изменять код в схемах «чужого» языка не всегда приемлемо и возможно.

4.2.4. Границы схем и приоритеты

Помимо зависимостей между приоритетами вызываемой/вызывающей схем, на взаимодействие вложенных схем влияет понятие границ регулярных выражений. Имен границы РВ используются при определении зависимостей компонент и сравнении их приоритетов. Помимо обычных областей группировки (задающихся скобками) существует специальная нулевая группа. Этой группе при совпадении регулярного выражения ставится в соответствии вся строка совпавшего РВ: начальной позиции группы отвечает

начало совпавшего шаблона, конечной – последняя позиция. «Координаты» именно этой группы используются при установке цветовых регионов `region0` (тэг `regexp`) и `region00, region10` (тэг `block`). Нулевая группа играет важную роль: при работе анализатор посимвольно продвигается по тексту, и по нахождению любой лексемы выделяет ее и идет дальше с позиции окончания нулевой группы. Этим обуславливается последовательность разбора различных лексем.

Хотя описанный процесс имеет место по умолчанию, в языке HRC есть средства для изменения этого поведения. Специальные операторы регулярных выражений `\m` и `\M` позволяют в регулярном выражении явно указать его границы. В простейшем случае это позволяет создавать частично или полностью *прозрачные* регулярные выражения. Такое регулярное выражение обрабатывается и выделяется цветом как обычное, но потом может «отдать» часть совпавшей строки другим лексемам. Этим достигается наслоение лексем и, как следствие, возможность описания сложных цветовых комбинаций.

Схожим образом границы используются и в межсхемном взаимодействии. При инициализации вызова схемы начальное и конечное регулярные выражения вызывающего схему блока формально принадлежат вызываемой схеме. На самом же деле разбор вызванной схемы ведется с позиции окончания нулевой группы атрибута `start` и заканчивается позицией начала нулевой группы атрибута `end`. Таким образом, анализ схемы происходит внутри границ начала и конца схемы. Использование в совокупности с включением схем некоторых расширенных возможностей регулярных выражений позволяет добиться создания сложных конструкций, иногда даже выходящих за класс контекстно-свободных языков, и, все же, часто встречающихся в языках программирования. Одной из таких возможностей является «сцепление схем», которое реализуется через использование оператора «~» (оператор привязки к началу схемы) и через манипуляции с границами блоков (их совмещением). По этой схеме реализована обработка «строенных» конструкций в Perl, анализ произвольных XML файлов и другие схемы:

```
<!-- набор произвольных XML тэгов -->
<division
   id="West"
   lang="en">
       <name title="West Division"/>
       <revenue>6</revenue>
       <growth>-1.5</growth>
       <bonus>2</bonus>
</division>

# 'Строенная' Perl-конструкция замены регулярного выражения
# результатом работы исполняемого кода
s {
     E<
     ( [A-Za-z]+ )
     >
} {
     do {
       exists $HTML_Escapes{$1}
       ? do { $HTML_Escapes{$1} }
          : do {
            warn "Unknown escape: $& in $_";
            "E<$1>";
          }
     }
}egx;
```

Во многих схемах применяется мощный оператор `\yN`, который позволяет сделать ссылку из конечного регулярного выражения блока `end` в начальное `start` (`N` – номер запрашиваемой совпавшей группы):

```
<block start="/(COMMENT) (.)/i" end="/\y2/" scheme="Comment"
                    region="asmComment"
                    region01="asmDefinition" region02="asmDefinition"/>
```

В этом примере выделяется блок комментария на языке ассемблера, который состоит из ключевого слова `COMMENT` и любого символа после него. Заканчивается комментарий этим же произвольно выбранным символом, что и реализуется оператором `\y2`, ссылающимся на вторую группу (скобку) параметра `start`.

Рассмотренные элементы языка HRC позволяют реализовать поддержку языковых конструкций в широких пределах. Добавление к ним дополнительных элементов будет либо избыточным, и они будут выражаться через существующие, либо же новые элементы будут использоваться для слишком специфических или бессмысленных целей. Через совокупность схем и регулярных выражений выражаются практически все синтаксические конструкции, встречающиеся в реальных языках. Проблема упрощается еще и тем, что синтаксический анализатор, имеющий своей задачей раскраску текста, совсем не обязательно должен полностью дублировать синтаксис и грамматику целевого языка – в простейших случаях достаточно реализации разбора элементарных конструкций.

4.3. Межсхемные связи

Все же, из анализа потребностей некоторых языков вытекает необходимость в дополнительных конструкциях, работающих со схемами. Большинство языков на уровне синтаксиса имеют возможность связи с другими языками. Таковы, к примеру, языки Си и Паскаль, использующие коды Ассемблера. Еще отчетливее это проявляется в языках, ориентированных на Internet. Такие языки и разметки как HTML, CSS, XML, JavaScript, Visual Basic тесно связаны друг с другом и составляют единую группу. HTML, например, может включать в себя помимо вышеперечисленных еще и такие языки как Perl (Perl Script), Java (Java Server Pages), ASP (Active Server Pages), PHP. И этим список не ограничивается. Аналогичные примеры глубоких взаимосвязей языков можно найти и в других областях программирования.

В простейшем случае можно воспользоваться вызовом схемы нужного языка – переключаться на схему CSS по достижении в HTML коде пары тэгов `<style> ... </style>`. Но в реальных примерах оказывается, что простого переключения схемы недостаточно. Практически всегда язык, включающий в себя другой язык как некоторое подмножество, переопределяет некоторые его синтаксические правила в соответствии со своими правилами. Например, при вставке asm-блока в C/C++ код, используются комментарии в Си стиле, а не в стиле Ассемблера. Это лишь простейший пример – на практике часто от исходного языка требуется только какая-то часть (к примеру, только список ключевых слов).

Еще одна проблема, возникающая при создании схем смешанных языков, это проблема дублирования информации о языке. Она является следствием первой проблемы и состоит в том, что многие языки бывают очень схожи и различаются только некоторыми признаками. Таковы языки C и C++. Базовая синтаксическая структура этих языков одинакова, и отличие состоит только в наборах ключевых слов (часть из которых опять же общая) и некоторых расширениях. При использовании только описанных возможностей языка HRC единственный способ реализации таких схем – это полное дублирование общей информации. Если для небольших объемов данных это приемлемо, то для крупных языковых структур это вызовет резкое увеличение объема базы описаний и, как следствие, уменьшение скорости работы библиотеки в целом.

4.3.1. Наследование

Для решения всех поставленных выше проблем в HRC введена дополнительная операция *наследования* схем. Ее реализует четвертый элемент в схеме – `inherit`. В простейшем виде он записывается как `<inherit scheme='name'/>`. Эта запись указывает анализатору, что в текущую схему необходимо добавить все определения

(ключевые слова, регулярные выражения, блоки, другие элементы `inherit`) схемы с указанным именем. При работе библиотеки в этом случае не происходит дублирования данных: анализатор сам переходит со схемы на схему без реального копирования данных. Этим достигаются как повышение быстродействия за счет экономии памяти, так и преимущества при разработке схем языков. Следует заметить, что тэг `inherit` *наследует* параметры от другой схемы; из этого следуют естественные ограничения на наследуемую схему – она должна быть уже ранее определена. В противном случае произошло бы зацикливание анализатора при обработке схемы. Несмотря на внешнее сходство, тэги `inherit` и `block` выполняют разные действия. Использование тэга `inherit` эквивалентно прямому копированию всех определений наследуемой схемы в текущую.

Использование этого тэга позволяет избавиться ото всех описанных выше проблем. С его помощью реализуются элементарные формы полиморфизма схем: схема может доопределяться операторами с начала или с конца, а так же простейшая утилизация кода: наследование небольших схем, определенных в базовом типе `def` (`default.hrc`) можно рассматривать как использование часто встречающихся макросов. Использование элемента `inherit` никак не расширяет класс поддерживаемых библиотекой языков, но этот тэг вводит в язык HRC возможности объектно-ориентированного программирования, что намного упрощает работу разработчика HRC-кода.

4.3.2. Подстановки схем

Особенно ярко объектно-ориентированные свойства языка становятся видны при использовании дополнительных параметров элемента `inherit` – набора тэгов `virtual`. Каждый тэг из этого набора задает пару схем, а в совокупности они задают список виртуализации текущего оператора `inherit`. При наследовании схемы список виртуализации используется для подстановки, замены определенных схем на некоторые другие, возможно специально определенные. При работе анализатор создает и обрабатывает списки виртуализации динамически, поэтому необходимость в изменениях схем и в каких-либо их дубликатах отпадает.

Система виртуализации схем – это мощный механизм, который позволяет устанавливать между схемами нетривиальные связи, максимально использовать возможности языка HRC и, в частности, оператора `inherit`. У системы виртуализации большой круг применения. На виртуальных схемах построены группы Internet языков (ASP, PHP) – они позволяют описать один из вариантов реализации языка, а потом с помощью наследования и виртуализации перенести этот код на работу с другими целевыми языками. Именно так реализована работа различных версий ASP-скриптов (Perl, JScript и VBScript версии). Еще более интересная область применения – это написание схем с «интерфейсами». Большей частью это касается группы схем по поддержке XML стандарта. Схема общего синтаксиса языка XML описывает синтаксис любого файла в этом формате и дополнительно предоставляет некоторые схемы для виртуализации с целью конкретизации синтаксиса языка: например, уточнения его ключевыми словами до стандартов XSL, XML Schema. Аналогично без изменений в базовом описании XML, разработчик может элементарно реализовать поддержку своих форматов, основывающихся на XML и использующих его компоненты.

Если обычное наследование схемы можно рассматривать, как наследование с возможностью изменять поведение только схемы верхнего уровня, то виртуализация позволяет намного шире управлять поведением наследуемых схем с возможностью изменения свойств схем на любых уровнях вложенности по наследованию и включению.

При самом процессе виртуализации подменяются не только схемы, наследуемые на других уровнях: виртуализации подвергаются и вызываемые схемы (тэг `block`). В связи с произвольным уровнем вложенности тэгов `inherit` и неограниченным количеством реально действующих списков виртуализации, общая картина подмен схем может быть очень сложной. В любом случае выполняются следующие условия: виртуализации подлежит набор схем, уже прошедший через механизмы виртуализации низших уровней (выполняется суперпозиция подстановок); при корректно прошедшей подстановке схемы, все списки виртуализации, находящиеся на более низком уровне, чем текущая примененная подстановка, на время работы виртуальной схемы отключаются (игнорируются сторонние подстановки).

Возможности виртуализации и наследования языка HRC позволяют гибко описывать и связывать различные языки программирования, достигая тем самым сцепленности языковой базы и утилизации существующего кода. Они вводят в HRC понятия, аналогичные понятиям полиморфизма, наследования и виртуальных функций в ООП, а это позволяет добиться от базы HRC большой гибкости, устойчивости к изменениям, общей архитектурной и идеологической стройности и отсутствия избыточности. Эти возможности положительно отличают язык описаний HRC и саму библиотеку Colorer от других аналогов.

5. Автоматизированное создание схем HRC

Библиотека Colorer использует свое представление данных о синтаксической и лексической структуре текста, которое реализует разбор некоторого класса языков и ориентировано на особенности синтаксического анализа в системах редактирования реального времени. В библиотеке уже существуют HRC-описания различных языков, и в связи с широким использованием языка XML естественным образом возникла задача расцветки синтаксиса исходных текстов, использующих этот синтаксис. Для этого необходимо описать на языке HRC грамматику языка XML, как она определена в спецификации W3C.

Так как стандарт XML представляет грамматику языка полностью в EBNF, то существует возможность создать HRC-синтаксис, разбирающий структуру XML-документа с учетом вложенности элементов и наличия у них атрибутов. Стандартными средствами языка HRC описываются простые синтаксические конструкции – инструкции обработки, комментарии, ссылки на сущности. Полученная HRC-грамматика соответствует произвольному правильно-сформированному документу. Она выполняет очень важную роль – позволяет выявить практически все синтаксические ошибки в тексте документа.

Следующий шаг – реализация возможности проверки не только правильно-сформированных документов, но и корректности документов на соответствие схеме (DTD или XML Schema). Этот процесс намного сложнее, так как описывается не на уровне синтаксиса (грамматик и продукций), а на уровне семантики содержимого. Данные в документе интерпретируются как абстрактное информационное множество, схема позволяет накладывать на это множество различного рода ограничения: фиксированную структуру дерева элементов, строгую типизацию содержимого атрибутов.

Хотя XML-процессор оперирует информационным множеством, полученным из уже завершенного синтаксического анализа, для специализированного разбора на языке HRC (с целью подцветки синтаксиса) эти две стадии можно объединить, реализовав возможность одновременной проверки как синтаксиса, так и семантики XML-документа.

Конечно, синтаксический разбор с учетом структуры схемы документа не может служить полноценной заменой процессу проверки корректности (validation). Он может учитывать лишь некоторые конструкции, из которых наиболее важна проверка корректной вложенности элементов в документе и проверка содержимого символьных типов данных. Реализация этих компонентов схем позволит создать оптимальную структуру HRC-синтаксиса, учитывающую семантику целевого документа и функционирующую с приемлемой скоростью.

5.1. Реализация разбора синтаксиса XML в HRC

Так как информационная модель любого XML-документа базируется на его синтаксическом описании, то реализация разбора синтаксиса без учета ограничений (для произвольного правильно-сформированного документа) будет служить основой для дальнейшего наложения определений схемы документа.

Сам язык XML полностью описывается в синтаксисе грамматик EBNF, и HRC позволяет довольно точно описать его грамматику. Следует заметить, что грамматика HRC накладывает довольно много ограничений на синтаксические конструкции, которые она способна описать. Это связано со спецификой работы библиотеки Colorer в реальном масштабе времени при редактировании документа. Несмотря на это, удается создать правила HRC, которые выделяют все основные лексемы языка XML, контролируют и выявляют наличие синтаксических и логических ошибок.

Большая часть продукций в грамматике XML переходит в соответствующие схемы HRC с дополнительными условиями, ограничениями и вложениями. В связи с ограничением на построчный разбор текста, многие компоненты грамматики записываются в другом, более обобщенном виде. Например, следующий набор продукций определяет структуру заголовка XML файла:

```
[23]   XMLDecl   ::= '<?xml' VersionInfo EncodingDecl? SDDecl? S? '?>'
[24]   VersionInfo   ::=   S 'version' Eq ("'" VersionNum "'" | '"' VersionNum
           '"')/* */
[25]   Eq            ::=   S? '=' S?
[26]   VersionNum    ::=   ([a-zA-Z0-9_.:] | '-')+
[80]   EncodingDecl  ::=   S 'encoding' Eq ('"' EncName '"' | "'" EncName "'" )
[81]   EncName       ::=   [A-Za-z] ([A-Za-z0-9._] | '-')*
[32]   SDDecl        ::=   S 'standalone' Eq (("'" ('yes' | 'no') "'") | ('"'
           ('yes' | 'no') '"'))
```

Так как в грамматике XML нетерминал S (символ пробела) включает в себя и перевод строки, то продукции при переносе в HRC обобщаются и фактически представляются в следующем виде (для `[23]`):

```
XMLDecl      ::=   '<?xml' (VersionInfo | EncodingDecl | SDDecl)? '?>'
```

Соответствующее представление в HRC разбивается на пару схем: определяющую границы конструкции `XMLDecl` и ее содержимое:

```
<scheme name="XMLDecl">
  <block start="/^(&lt;\?)(?{XMLDecl.name}xml)\M(\s|$)/"
         end="/(\?>)/" scheme="XMLDecl.content"
         region="XMLDecl"
         region00="PairStart" region01="XMLDecl.start"
         region10="PairEnd" region11="XMLDecl.end"/>
</scheme>

<scheme name="XMLDecl.content">
  <regexp match="/(\s|^)(?{VersionInfo}version)
                  \s* (?{XMLDecl.eq}\=) \s*
                  (?{VersionNum}(?{q}[&#x22;&#x27;])
                  [a-zA-Z0-9_.:\-]+
                  \p{q})
                 /x"/>
  <regexp match="/(\s|^)(?{EncodingDecl}encoding)
                  \s* (?{XMLDecl.eq}\=) \s*
                  (?{EncName}(?{q}[&#x22;&#x27;])
                  [a-zA-Z][a-zA-Z0-9_.:\-]*
                  \p{q})
                 /x"/>
  <regexp match="/(\s|^)(?{SDDecl}standalone)
                  \s* (?{XMLDecl.eq}\=) \s*
                  (?{SDDecl.yn}(?{q}[&#x22;&#x27;])
                  (yes|no)
                  \p{q})
                 /x"/>
  <regexp match="/\S+/" region0="badChar" priority="low"/>
</scheme>
```

Такой принцип разбиения применяется для большинства продукций. В дальнейшем, подобное разбиение структуры языка на схемы, соответствующие его основным нетерминалам позволит ссылаться на них из генерируемых схем, определяющих семантику типов документов. Тем самым уменьшается дублирование кода и увеличивается степень интеграции различных модулей HRC.

Описания многих лексем языка XML используют определения множества доступных символов. Эти множества отображаются в соответствующие HRC-сущности, которые затем используются в конкретных регулярных выражениях в качестве макроподстановок. Так как библиотека Colorer полностью поддерживает стандарт Unicode, определяемые классы символов и составленные из них шаблоны идентификаторов полностью соответствуют XML спецификации. Это позволяет при редактировании

документа распознавать некорректно используемые символы (например, коды из зарезервированной области пространства Unicode).

Так как по умолчанию текст, не попавший ни под какое из определений HRC, пропускается в процессе анализа, то для индикации ошибочных состояний и конструкций применяется явное выражение, захватывающее оставшиеся неразобранные данные:

```
<scheme name="badChar">
  <regexp match="/\S/" region="badChar" priority="low"/>
</scheme>
```

В случаях, когда возможно отследить ошибочную конструкцию, это явно описывается через регулярные выражения. Выделение ошибочных выражений позволяет не только визуально их обозначить, но и производить в дальнейшем различные действия над такими областями. Например, выводить список всех найденных в тексте ошибок и предлагать их исправить.

В дальнейшем, при генерации схем конкретных XML-языков, необходимо будет обеспечить основу для реализации разбора синтаксиса тэгов в документе. Этот синтаксис должен учитывать все возможные формы записи начального и конечного тэгов (полную и сокращенную). При этом начальный тэг может иметь произвольное число параметров. В EBNF это выражается следующими продукциями:

```
[39]   element      ::=   EmptyElemTag | STag content ETag
[40]   STag         ::=   '<' Name (S Attribute)* S? '>'
[42]   ETag         ::=   '</' Name S? '>'
[44]   EmptyElemTag ::=   '<' Name (S Attribute)* S? '/>'
```

Для записи такой структуры в языке HRC нужно учитывать, что нетерминал S включает в себя перевод строки. Так как HRC поддерживает разбор лексем в пределах одной строки текста, то для содержащих перевод строки конструкций нужно находить неделимые лексемы и описывать схемы на их основе. При этом конструкции «EmptyElemTag» и «STag content ETag» нужно объединить и разбирать за один проход, так как невозможно осуществить просмотр вперед на несколько строк текста. Это реализуется следующей связкой схем:

```
<scheme name="element">
  <block start="/\M &lt; (%Name; ([\s\/&gt;]|$) )/x"
         end="/ &gt; /x"
         scheme="elementContent"/>
</scheme>
<scheme name="elementContent">
  <block start="/~( (&lt;) (  ((%xml:NCName;) (:) )? (%xml:Name;) ) \M
         &gt;?   )/x"
         end="/( (&lt;\/) (\y3 (?{}\s|$|\>)?= )?= ( (%xml:NCName;) (:) )?
         (%xml:Name;) \b \M \s* (&gt;?)
          | (\/ \M &gt;) )/x"
         region01="PairStart" region02="element.start.lt"
         region05="element.nsprefix" region06="element.nscolon"
         region07="element.start.name"
         region11="PairEnd" region12="element.end.lt"
         region15="element.nsprefix" region16="element.nscolon"
         region17="element.end.name" region18="element.end.gt"
         region19="element.start.gt"
         scheme="xml:elementContent2"/>
  <inherit scheme="badChar"/>
</scheme>
<scheme name="elementContent2">
  <block start="/>/" end="/ \M ( &lt;/%Name; ) /x"
         region00="element.start.gt"
         scheme="content"/>
  <inherit scheme="Attribute"/>
</scheme>
```

Такая тройка схем реализует разбор, практически эквивалентный исходной грамматике. Она осуществляет разбор структуры тэгов языка XML, а рекурсивная вло-

женность разбора элементов достигается возможностью порождения схемы `element` схемой `content`.

В приведенной структуре определяется довольно много синтаксических регионов HRC. Это связано с тем, что потенциально для каждой лексемы (и компонентов лексем) можно выделить различные по значимости синтаксические конструкции, каждой из которых в дальнейшем можно сопоставить визуальное представление. Кроме того, одновременно с визуальной расцветкой синтаксиса, схемы языка XML реализуют возможности анализа и обработки структуры вложенностей тэгов, которые используются для обеспечения быстрой навигации по тексту документа и представлению компактной структуры текста (outline).

5.2. Структура модуля XSD2HRC

Очевидно, что условие действительности по отношению к схеме намного жестче, чем просто соответствие синтаксису спецификации XML (well formed) и, хотя анализ семантики текста выходит за рамки синтаксического анализа и контекстно-свободных грамматик, в случае с XML языками существует возможность создать правила HRC уже для конкретных типов документов XML.

Каждый конкретный тип документов XML вводит пространство имен XML, в котором определяются значимые элементы и атрибуты языка. Помимо этого большинство XML-языков имеют структуру зависимостей между элементами и различные ограничения на представление данных. Именно эта информация, как наиболее значимая, интересна для обработки в HRC, так как основная задача библиотеки – обеспечение удобства редактирования текста и максимальный контроль за ошибками и опечатками еще на стадии ввода текста программы. Если HRC-определения большинства языков программирования в библиотеке Colorer введены вручную, то в случае с XML-языками возникает необходимость автоматизации создания их синтаксиса. Ведь языков, основанных на XML, может быть неограниченное множество, и каждый из них определяет свою структуру и свои зависимости между элементами.

Исходными данными в этой задаче будут описания типов документов XML в формате XML Schema. Это и есть схема документа, определяющая его структуру, семантику и содержимое. Конечная информация, получаемая в результате преобразований, является описанием структуры этого языка в терминах HRC. В дальнейшем она считывается и обрабатывается библиотекой как описания для любых других языков программирования.

Традиционно простейшим способом описания синтаксиса XML документов служит язык DTD. Несмотря на свою простоту, он очень слабо типизирован и позволяет выражать лишь элементарные формы зависимостей. Помимо многих других недостатков он обладает еще одним, очень значимым: он не основан на XML и, как следствие, не позволяет использовать доступные средства анализа и обработки XML-данных. Большинство его недостатков устраняет стандарт XML Schema, который явно ориентируется на типизацию данных и лишен большинства недостатков DTD (поддерживает пространства имен XML, сложные отношения между компонентами, механизмы наследования/переопределения типов данных). Кроме того, создаваемые XML-схемы документов сами представлены в формате XML, что делает их универсальными, открытыми и доступными для дальнейшего анализа.

Таким образом, имея структуру языка и пространства имен, описанные в XML Schema, библиотека Colorer использует эти данные для автоматического преобразования их в свой формат представления синтаксиса (HRC) и дальнейшего синтаксического анализа (Рис. 9).

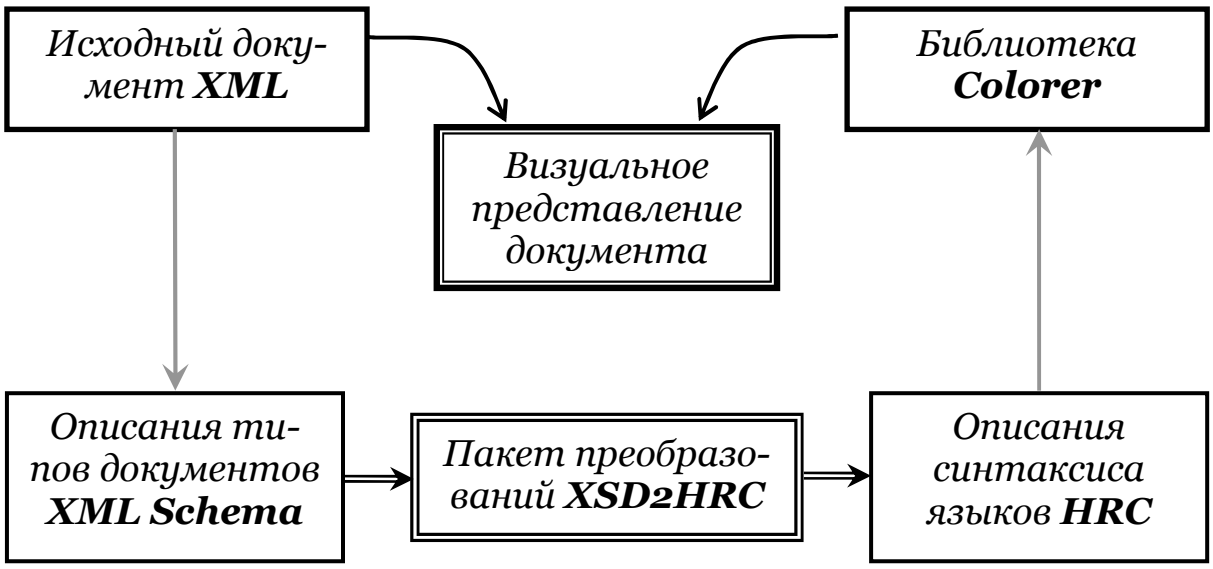

Рис. 9. Взаимосвязь компонентов библиотеки при преобразовании XSD2HRC.

Для описания необходимых преобразований и получения конечного кода HRC в библиотеке Colorer используется язык XSLT (версии 1.0). Реализованный модуль `XSD2HRC` представляет собой пакет XSLT преобразований, принимающий на вход XML схему (XSD файлы) и создающий на выходе соответствующий HRC файл.

5.2.1. Возможности настройки пакета XSD2HRC

Для поддержки гибкого и мощного процесса генерации HRC файлов, модуль XSLT преобразований библиотеки реализован в виде расширяемого компонента, который может принимать на вход не только требуемую XML схему, но и набор параметров, настраивающих процесс его работы под конкретный генерируемый тип XML документа (Табл. 10). Часть параметров управляет процессом генерации HRC схем, часть влияет на принципы обработки входных схем XSD, позволяя незначительно изменять структуру документа.

Табл. 10. Входные параметры пакета XSD2HRC.

Параметр преобразования	*Значение*
`hrctype`	Уникальный идентификатор типа языка, получаемого на выходе преобразования. Этот идентификатор используется для интеграции сгенерированного синтаксиса в общую базу HRC-описаний
`include-prototype`	Логический параметр, указывающий на необходимость генерации прототипа описания языка. В дальнейшем прототип должен быть вручную сопоставлен с нужным типом файлов.
`catalog-path`	Путь к глобальному файлу с описаниями прототипов всех доступных языков (`colorer.hrc`). Используется для связывания сгенерированных схем различных XML языков на основе уникального идентификатора пространства имен каждого типа документа.
`custom-defines`	Путь к необязательному конфигурационному файлу, определяющему параметры генерируемого языка более подробно.

allow-common-attr	Контролирует процесс генерации схемы. Позволяет в конечном документе использовать базовые глобальные атрибуты, определенные спецификацией XML (`xml:space, xml:base, xmlns:*`).
allow-any-attr	Позволяет использовать в конечном документе атрибуты из произвольных пространств имен, которые не контролируются на правильность содержимого.
allow-unknown-elements	Позволяет в любой позиции дерева документа использовать элементы из произвольного пространства имен, на которые не распространяется процесс проверки.
allow-unknown-root-elements	Позволяет начать процесс проверки содержимого на соответствие XML-схеме не с корневого элемента дерева, а с произвольной позиции в структуре документа.
force-single-root	Принудительно вводит ограничения на синтаксис, которые позволяют использовать в документе только один корневой элемент. Это условие не обязательно для внешних сущностей, которые могут не удовлетворять ему.
top-level-element	Явное указание, какой из доступных глобальных элементов нужно использовать в качестве корневого в процессе проверки документа. По умолчанию любой глобальный элемент может выступать в роли корневого.

Хотя многие из параметров, управляющие структурой генерируемых схем (`allow-unknown-elements, allow-any-attr, allow-common-attr`), имеют эквивалентное выражение в XML Schema, часто бывает удобнее использовать их, чем модифицировать содержимое исходных XSD файлов.

Помимо простых параметров (передаваемых в командной строке) существует возможность использования специального конфигурационного файла, который определяет тонкие настройки генерации описания документа и позволяет произвольным образом интегрировать языки XML Schema и HRC в полуавтоматическом режиме. С его помощью становится возможным описание языков, которые невозможно полноценно определить в терминах XML схем. XSLT, например, допускает встраивание своих конструкций в атрибуты элементов генерируемого им документа, его элементы могут быть дочерними для элементов из произвольного пространства имен, существует сокращенная форма активизации шаблонов XSLT. Все подобные возможности не могут быть в полной мере описаны в схемах, но использование конфигураций в модуле преобразований позволяет минимальными коррективами добиться нужных результатов синтаксического разбора.

Файл конфигурации (`custom.*.xml`) включает так же настройки, которые невозможно передать в виде простых параметров из-за их комплексной структуры. Этот файл имеет простую структуру, сопоставляющую набор параметров одному или нескольким генерируемым типам документов. Сопоставление производится на основе используемого URI пространства имен. Основные возможности, предоставляемые конфигурацией, перечислены в Табл. 11.

Табл. 11. Конфигурация пакета XSD2HRC.

Параметр конфигурации	*Описание*
Префиксы пространства имен	Список обычно используемых в документе префиксов пространств имен. Как вариант, возможно использование произвольных префиксов, либо их отсутствие вообще.
Корневые элементы/группы	Более мощная форма параметра `top-level-element` позволяет задать список возможных корневых элементов или их групп.
Структурные элементы	Множество элементов для которых генерируется дополнительный код поддержки выносной структуры документа (outline). Для каждого структурного элемента возможно задание способа его представления в структуре и краткого описания.
Переопределение HRC типов	Основная возможность, используемая в конфигурационных файлах пакета `XSD2HRC`. Параметр позволяет встроить в генерируемый HRC-файл измененные определения, что позволяет вручную подстраивать генерацию схем.

Переопределение HRC типов является очень мощной возможностью, так как позволяет вмешиваться в процесс преобразования и модифицировать поведение некоторых компонентов. При этом модификация может относиться как к компонентам сложных типов данных, так и к содержимому простых типов (реализуя их специализированную обработку).

5.2.2. Принципы разбора структуры XML Schema

Возможности настройки преобразования `XSD2HRC` используются в частности для автоматизации процесса генерации первичного документа «схемы для схем». Полноценная реализация отображения формальной части спецификации схем – схемы, описывающей саму XML схему, является самым важным шагом на начальном этапе создания модуля преобразований. Такое описание возможно, так как документ XML схемы ничем не отличается от любого другого XML документа. «Схема для схем» (Schema for Schemas) является в некотором роде абстрактным документом, так как связывает воедино некоторые априорные понятия о типах и объектах (magic types) с другими, наследуемыми и выводимыми из них. Любая другая схема ссылается на определенные в схеме для схем базовые типы данных (build-in datatypes).

При генерации конечных HRC-схем расцветки необходимо, чтобы эти ссылки каким-либо образом разрешались. Это делается путем импортирования определений в генерируемый HRC файл из уже сгенерированного HRC-описания для XML схем. Импортирование (или связывание) объектов HRC – это встроенная возможность этого языка, реализованная в библиотеке Colorer. Использование такой архитектуры решает сразу две проблемы. Первая – необходимость ссылаться на базовые типы XML схем без дублирования кода. Вторая – вытекающее из этого обобщение реализации возможностей импорта и включения XML схем (операции `xs:import` и `xs:include`).

Априорность определения типов данных в XML Schema выражается в том, что все простые типы данных не могут быть обработаны и определены средствами самой спецификации XML Schema. Их поддержка должна быть изначально встроена в XML-процессор, занимающийся соотнесением документов со схемами их описывающими.

Так, например, схема для схем определяет все базовые типы унаследованными от абстрактного `anySimpleType`, хотя в спецификации этот тип представлен в виде объединения всех базовых типов данных.

В пакете `XSD2HRC` аналогом априорной информации является ручное описание синтаксиса каждого простого типа данных в файле конфигурации преобразования. В дальнейшем, при первичной генерации «схемы для схем», эта информация автоматически внедряется в полученный HRC файл, реализуя тем самым поддержку проверки синтаксиса примитивных типов данных.

Проверка содержимого простых типов данных – достаточно важный момент, так как именно они хранят конечную символьную информацию в документе. Реализация поддержки разбора и проверки таких типов сводится к уже описанной первичной генерации HRC описаний «схемы для схем». Любые простые типы данных, определяемые в других схемах, наследуются от примитивных. К условиям наследования применяются заданные ограничения и обработка большинства простых типов данных происходит автоматически за счет ссылок на уже определенные вручную типы из XML Schema. Конечно, при необходимости более точного описания синтаксиса простого типа можно воспользоваться конфигурационным файлом и реализовать разбор необходимого синтаксиса вручную.

Отображение сложных типов данных основывается на базовой структуре HRC-описаний для грамматики языка XML. Эта грамматика, работающая для произвольного правильно сформированного документа, расширяется и уточняется каждым из сложных типов для того, чтобы обеспечить корректное вложение структур дочерних элементов и возможных списков атрибутов (которые также конкретизируются). В итоге порожденная HRC-грамматика способна распознавать корректные дочерние атрибуты и элементы в каждом сложном типе данных. Помимо этого реализуются возможности визуальной индикации ошибочных состояний.

Язык XML схем очень гибок, и предоставляет в распоряжение программиста большой набор примитивов, описывающих структуру конечного документа. Простые и сложные типы данных – это лишь основа, на которой строится вся архитектура XML Schema. Модуль `XSD2HRC` библиотеки поддерживает большинство конструкций, что позволяет работать практически с произвольными описаниями документов на языке XML схем (Табл. 12).

Табл. 12. Отображение компонентов схем в HRC-описания.

Компоненты	*Описание*	*Реализация*
`xs:include` **`xs:redefine`**	Включение/изменение определений из других схем	Прямое отображение на структуру HRC с использованием операторов `hrc:inherit/hrc:virtual`
`xs:import`	Импорт описаний из схем других документов	Установление идентичной связи между генерируемым описанием и HRC-описанием подключаемой схемы
`xs:extension` **`xs:restriction`**	Основная операция наследования типов через расширение или ограничение базового	Идентичные расширения/сужения в структуре схем HRC-описания.
`facets`	Ограничения на простые типы данных	Поддержка `xs:enumeration` и `xs:pattern` через регулярные выражения HRC

xs:list xs:union	Определение простых типов через список/объединение	Расширение контекстов соответствующих простых типов с использованием `hrc:inherit`
xs:sequence xs:choice xs:all	Квалификаторы структуры вложенных элементов в сложных типах данных	Обобщение до `xs:choice` и генерация операторов переключения контекста для каждого дочернего элемента
xs:group xs:attributeGroup	Группы атрибутов и модели данных	Группировка в схемы HRC и ссылки через оператор `hrc:inherit`

Реализация некоторых возможностей схем XML не укладывается в рамки синтаксиса HRC. Это связано с тем, что многие ограничения схем описываются в алгоритмической форме и их невозможно выразить даже в более общей, чем HRC, контекстно-свободной грамматике. К таким ограничениям относятся ссылочные связи между элементами (операторы схемы `xs:key`, `xs:keyref`), ограничения на уникальность (`xs:unique`). Игнорируются некоторые другие компоненты языка схем: поддержка шаблонов имен, строгого порядка и ограничений на количество повторений элементов (атрибуты `minOccurs`, `maxOccurs`). Положительным является то, что реализация таких возможностей в расцветке синтаксиса не нужна. В случае такого специфического разбора, который реализует библиотека Colorer, достаточно поддержки самых основных понятий и определений XML схем, описывающих структуру документа в целом.

Процесс перевода схем в формат HRC подразумевает отображение всех компонентов XML схем в идентичные по функциональности HRC схемы. В процессе работы преобразования происходит обход дерева анализируемой XML схемы в различных режимах применения шаблонов XSLT (modes). Каждый из режимов отвечает за свои компоненты, которые трансформируются в нужные коды HRC. Краткое описание каждого режима приведено в Табл. 13.

Табл. 13. Режимы обработки содержимого XML Schema.

Режим обработки	*Назначение*
по умолчанию	Создание общих для всех языков заголовков, определений. Сюда входят базовые схемы, используемые в каждом языке, переопределенные синтаксические регионы (для возможного изменения представления документа). Из этого режима инициируются режимы `root` и `typedefs`, рекурсивно разбирающие содержимое XSD файла и создающие соответствующие HRC-определения.
root	Генерация определений глобальных элементов, атрибутов и их групп. Этот режим отвечает также за связывание схем, состоящих из нескольких файлов (путем рекурсивного перебора всех подключаемых в схему файлов).
typedefs	Режим, генерирующий описания всех найденных сложных и простых типов данных. К ним относятся не только именованные типы, но и вложенные безымянные, каждому из которых присваивается уникальный внутренний идентификатор для дальнейшей работы.
include-content	Реализация содержимого сложных и простых типов данных на уровне элементов. Этот режим обрабатывает

	любое возможное содержимое, описывающее тип данных (расширение, ограничение) и поддерживает все его возможные формы реализации. Учитываются шаблоны имен элементов (wildcard) и ссылки на группы модели (model groups)
include-attr	Реализация содержимого сложных и простых типов данных на уровне атрибутов. Обрабатывает списки атрибутов, допустимых для каждого типа данных, учитывая при этом возможные шаблоны имен (wildcard) и ссылки на группы атрибутов (`attributeGroup`).
root-elements	Вспомогательный режим, формирующий список глобальных элементов, доступных для использования в качестве корня дерева документа. Учитывает возможность использования схем из нескольких определений.

Между режимами существуют связи, реализующие в совокупности полную рекурсивную обработку дерева XML схемы.

Помимо основной структуры шаблонов XSLT пакет `XSD2HRC` определяет несколько вспомогательных именованных шаблонов (выполняющих роль функциональных элементов), с помощью которых реализуются различные алгоритмы, необходимые для построения HRC кодов.

К сервисным процедурам можно отнести именованный шаблон `replace-string`, служащий для замены в переданном параметре вхождений определенных последовательностей символов на другие подпоследовательности. Этот шаблон используется при переносе регулярных выражений XML схем в соответствующие регулярные выражения HRC. Хотя в общих чертах синтаксис обоих реализаций совпадает, существуют незначительные отличия в применении операторов и метасимволов. Для их преобразования в синтаксис библиотеки Colorer достаточно механической замены в соответствующие регулярные выражения HRC.

Именованные шаблоны `qname2hrcname`, `element-call` и `attribute-call` применяются для отображения полных имен (qualified names) типов из пространства имен XML Schema в соответствующие эквиваленты HRC. Если в схеме используются ссылки на типы данных, элементы или атрибуты из другого пространства имен (через предварительно объявленное выражение `xs:import`), то они разрешаются с использованием соответствующего HRC-типа, ассоциированного с этим пространством имен через атрибут `hrc/prototype/@targetNamespace`. При этом преобразованные HRC-определения типов данных должны находиться в уже сгенерированном файле, с которым связывается текущая схема. Если адресуемые элементы и атрибуты находятся в том же пространстве имен, что и генерируемая схема (при проверке этого учитывается возможность использования произвольных префиксов), происходит ссылка на HRC-схемы, соответствующие их типам данных.

5.3. Примеры преобразований

Основной сгенерированный HRC-тип для самого языка XML схем используется всеми остальными языками и, поэтому, наиболее важен для рассмотрения. В приведенном ниже файле конфигурации пакета `XSD2HRC` описываются параметры генерации «схемы для схем» и подстановки для некоторых схем базовых типов. Эти подстановки

позволяют необходимым образом обработать содержимое простых типов данных и проверить его на корректность. Файл конфигурации выглядит следующим образом:

```
<custom xmlns="uri:colorer:custom">
  <!-- XML Schema magic simple types definition -->
  <custom-type targetNamespace="http://www.w3.org/2001/XMLSchema">
    <prefix>s</prefix>
    <prefix>xs</prefix>
    <prefix>xsd</prefix>
    <prefix>wxs</prefix>
    <empty-prefix/>
    <top-level>
      <element>schema</element>
    </top-level>

    <outline>
      <element name='schema' extract='withAttribute'/>
      <element name='element' extract='withAttribute'/>
      <element name='group' extract='withAttribute'/>
      <element name='attribute' extract='withAttribute'/>
    </outline>

    <type xmlns="http://colorer.sf.net/2003/hrc">

      <scheme name="anyType-content"/>
      <scheme name="anySimpleType-content"/>
      <scheme name="anySimpleType-AttributeContent">
        <inherit scheme="AttributeContent">
          <virtual scheme="xml:AttValue.content.stream"
                           subst-scheme="anySimpleType-content"/>
        </inherit>
      </scheme>

      <scheme name="string-content">
        <regexp match="/[%xml:Char;]/" priority="low"/>
      </scheme>
      <scheme name="boolean-content">
        <regexp match="/\b(true|1)\b/" region="BooleanConstant"/>
        <regexp match="/\b(false|0)\b/" region="BooleanConstant"/>
      </scheme>
      <scheme name="decimal-content">
        <regexp match="/ [\-+]? (\d+\.?\d* | \.\d+\b) /ix" region="Number"/>
      </scheme>
      <scheme name="float-content">
        <inherit scheme="FloatNumber"/>
        <regexp match="/ INF|-INF|NaN /x" region="Number"/>
      </scheme>
      <scheme name="duration-content">
        <regexp match="/[\-+]? P \d+Y\d+M\d+D (T \d+H \d+M (\d+(\.\d*)?)S)? /x"
                        region="Date"/>
      </scheme>
      <scheme name="dateTime-content">
        <regexp match="/[\-+]? \d{4,} - \d\d - \d\d T \d\d:\d\d:(\d\d(\.\d+)?)
                                (Z|[+-]\d\d:\d\d)?/x" region="Date"/>
      </scheme>
... ...
```

В заголовке файла определяется целевое пространство имен `targetNamespace` (оно установлено в значение URI пространства имен языка XML Schema: `http://www.w3.org/2001/XMLSchema`). Далее определяются возможные префиксы этого пространства (при необходимости их набор может быть расширен даже без перегенерации схемы языка), единственный корневой элемент языка (`xs:schema`), элементы и их свойства, используемые для структурирования текста документа (блок `outline`). Основная часть конфигурационного файла – переопределение некоторых конечных схем языка HRC. При генерации указанные схемы будут использоваться

вместо автоматически созданных. Это позволяет внести нужные коррективы в процесс генерации без необходимости дальнейшей модификации уже сгенерированного кода.

Еще один интересный пример – преобразование схемы языка XSLT в синтаксис HRC. Основная проблема здесь заключается в том, что некоторые из конструкций этого языка невозможно выразить в синтаксисе XML Schema, но можно представить в кодах HRC через настройку процесса преобразования.

Одна из особенностей синтаксиса XSLT – возможность сокращенной записи шаблонов, в которой опускается заголовочная часть, определяющая свойства преобразования и его параметры. Корневым элементом XSLT преобразования может быть любой элемент (из произвольного пространства имен). Сами же инструкции XSLT (элементы из его пространства имен) могут встречаться уже в содержимом других элементов на любых уровнях вложенности. Для реализации этой особенности в пакете преобразований XSD2HRC предусмотрен параметр `allow-unknown-root-elements`, при использовании которого к списку корневых элементов языка добавляется следующая конструкция:

```
<xsl:if test="$allow-unknown-root-elements = 'yes'">
  <inherit scheme="xml:element">
    <virtual scheme="xml:element" subst-scheme="{$hrctype}-root"/>
  </inherit>
</xsl:if>
```

Эта конструкция позволяет произвольным элементам (их синтаксис определен в схеме `xml:element`) функционировать в качестве корневых. Для того, чтобы разрешить присутствие в дереве, начинающемся с этого элемента, компонентов языка XSLT, применяется виртуализация рекурсивно вложенной схемы `xml:element` на корневую схему текущего типа `{$hrctype}-root` (в данном случае это будет `xslt-root`).

С описанной особенностью XSLT связана другая возможность его синтаксиса – внедрение операторов XSLT в любую часть результирующего дерева. Такие вставки операторов в данные для обработки невозможно отследить с использованием XML Schema, в то же время именно на них основывается вся мощь преобразований, и отсутствие поддержки этой возможности в синтаксисе HRC сделает бессмысленным применение сгенерированного типа для расцветки таких документов. Но использование файла конфигурации и замена определений некоторых схем позволяет просто и эффективно ввести поддержку такого специализированного представления кода:

```
<scheme name="result-element-group-content">
  <inherit scheme="instructions-group"/>
  <inherit scheme="xml:content"/>
</scheme>
<scheme name="result-element-group">
  <inherit scheme="result-element-group-old">
     <virtual scheme="xml:content"
       subst-scheme="result-element-group-content"/>
    <virtual scheme="xml:AttValue.content.stream"
       subst-scheme="avt-content-error"/>
  </inherit>
</scheme>
```

Этот фрагмент HRC заменяет две автоматически генерируемые схемы, внося в логику обработки документов XSLT возможность обнаружения операторов в результирующем дереве. При замене схем их аналогами из конфигурационного файла, оригинальные схемы не удаляются. Они сохраняются в тексте HRC под другим именем (с суффиксом `–old`). Это позволяет использовать старые схемы и переопределять их поведение произвольным образом, что и делается в приведенном выше примере для схемы `result-element-group`. Эта схема соответствует содержимому сложного типа данных в XSD, определяющему обработку конечных структур элементов (Result Tree

Fragment), получаемых в преобразовании XSLT. В конфигурационном файле она просто дополняется схемой `instructions-group`, содержащей набор операторов языка XLST и, тем самым, реализует необходимую функциональность модуля.

5.4. Использование пакета XSD2HRC

Созданный модуль `XSD2HRC` играет очень важную роль – он реализует полноценную поддержку разбора синтаксиса XML и языков на нем основанных. Для его работы требуются лишь XML и XSLT процессоры, поэтому возможна его явная интеграция с основной частью библиотеки Colorer либо с пользовательским интерфейсом целевой системы редактирования. Тогда распознавание типа языка, загрузка его схемы и генерация нужного синтаксиса для разбора может происходить в автоматическом или полуавтоматическом режиме. Это избавит от необходимости ручного запуска генерации синтаксиса, позволит интегрировать поддержку XML документов в целевую систему редактирования.

Помимо самого модуля преобразований XSD2HRC в библиотеку Colorer входит программа `dtd2xsd.pl` на языке Perl. Она реализует возможность преобразования описаний DTD в формат XML Schema. Такое преобразование очень полезно, так как сегодня язык DTD еще очень широко распространен, и не для всех типов документов существуют описания в стандарте XML Schema. Эта программа в пакетном режиме преобразует файлы DTD в модули XSD. Программа использует простейшие алгоритмы перевода данных, но позволяет учитывать некоторые общепринятые стандарты написания DTD и на их основе генерировать схемы, более пригодные для дальнейшего использования. Конечно, существуют и другие средства преобразования DTD→XSD и библиотеке Colorer не важно, каким образом этот перевод происходит.

Автоматическая генерация схем, реализованная в библиотеке Colorer, позволяет работать практически с произвольными языками и может использоваться как основа для введения поддержки других, смешанных языков, таких как JSP/Taglib, ASP.NET. Эти языки смешивают представление исходных данных со специальной разметкой, содержащей код на традиционном языке (Java, C#, JavaScript). Обработку таких «гибридов» так же можно реализовать в автоматическом режиме и получить приемлемые результаты. Все это позволит оптимально использовать возможности по обработке и редактированию XML-разметки как в самой библиотеке Colorer, так и в приложениях, основанных на ней.

6. Заключение

Результатом данной работы является создание полнофункциональной версии библиотеки Colorer – первой системы синтаксического анализа и раскраски текстов, которая работает как отдельная встраиваемая библиотека классов, не зависит от целевой аппаратной платформы и среды применения. Помимо основной возможности цветового выделения, библиотека предоставляет дополнительные функции, базирующиеся на синтаксическом анализе. К ним относится поиск и отображение парных языковых конструкций, фигурных скобок, границ строк, комментариев, XML и HTML тэгов. На основе данных синтаксического анализа, проводимого в реальном времени при редактировании текста, библиотека строит древовидные структуры элементов, классов и методов, предоставляя возможности быстрой навигации по тексту, поиска значимых компонентов в тексте, ошибок и опечаток. Эти возможности широко используются и востребованы в большинстве систем редактирования.

Библиотека проектировалась как независимый от аппаратной и программной архитектуры компонент, который можно использовать на Windows и Unix платформах. Код библиотеки написан на языке C++ стандарта ANSI и компилируется любым C++ совместимым компилятором, для функционирования требуется только стандартный набор системных библиотек, доступный на всех платформах. Одним из значимых результатов данной работы является созданный и отлаженный интерфейс для взаимодействия библиотеки с виртуальной машиной Java.

Изначально заложенные принципы проектирования подтвердились на практике: на сегодня существуют функционирующие и отлаженные версии библиотеки Colorer для Windows и Linux. Кроме того, работа библиотеки тестировалась на компьютерах Apple Macintosh (MacOS X) – безо всяких модификаций заработал не только основной код, но и прослойка JNI/Java. Проводилось так же тестирование кода на аппаратных платформах Sun Sparc и Alpha.

Библиотека обладает огромными преимуществами перед аналогичными системами. Первое и самое главное – она не привязана к конкретному приложению, а реализована как независимый компонент. Библиотека Colorer не является самостоятельным приложением и предназначена для функционирования в целевых системах редактирования (IDE, редакторах общего назначения), которые пользуются ее сервисами и реализуют заложенные в ней возможности. Библиотека реализует широкий спектр возможностей, связанных с анализом текста, сама расцветка является лишь одним из приложений этого анализа. Библиотека предоставляет уникальные возможности, призванные упростить и облегчить процесс написания исходных кодов программ. Для некоторых существующих систем, поддерживающих возможности расширения, библиотека работает как дополнительный модуль (plugin).

В процессе внедрения библиотеки, для анализа и тестирования основного интерфейса на языке C++, был написан подключаемый модуль к файловому менеджеру FAR Manager. Он вводит во встроенный редактор этой программы возможность синтаксического выделения. Помимо этого на базе этого редактора были опробованы все расширенные возможности библиотеки – отображение общей структуры документа (списки процедур и функций), удобная работа с парными синтаксическими конструкциями, работа с синтаксическими ошибками. Версия для оболочки FAR Manager является наиболее известной реализацией функций библиотеки. На сегодня число только зарегистрированных пользователей plugin-модуля Colorer превысило полторы тысячи человек.

Оболочка FAR Manager не ориентируется исключительно на редактирование файлов, поэтому было очень важным опробовать библиотеку Colorer в работе со специализированными средами разработки, где редактор текстов является центральным

компонентом. Для этого был создан подключаемый модуль на языке Java, который может использоваться в среде разработке Eclipse. Эта среда изначально ориентирована на разработку приложений и редактирование исходных кодов. Ее преимуществами является уникальная расширяемая архитектура, которая за счет своей открытости позволяет реализовать практически любые возможности. Подключаемый модуль библиотеки (EclipseColorer) тесно интегрируется в среду разработки и предоставляет программисту возможность использования в среде Eclipse редактора, специально сконфигурированного и использующего все функции библиотеки Colorer.

Создание этого модуля и интерфейса для языка Java позволило формализовать и выявить необходимые свойства и требования целевых приложений, использующих библиотеку Colorer. Это позволит в дальнейшем без модификаций перенести разработанный API на другие объектные языки. Ближайший аналог Java, язык C#, предоставляет идентичную функциональность в выражении объектно-ориентированных структур и является одним из возможных претендентов на реализацию в нем API Colorer. Конечно, список не ограничивается только этими языками, и в планах развития библиотеки Colorer стоит создание интерфейсов для иных систем и языков. В основном, требования к переносу библиотеки на другие языки состоят в том, чтобы отобразить существующую объектную модель в структуры целевого языка. Причин, по которым встает вопрос о переносе библиотеки, довольно много. Большинство из современных языков имеют уже встроенную поддержку таких стандартов, как Unicode, XML. Так как C++ версия библиотеки поддерживает их, процесс переноса и установления связей между различными языками намного упрощается. Большую роль в этом играет строгая формализация этих стандартов, которая позволяет использовать их вне зависимости от аппаратной и программной архитектуры систем.

Кроме основного целевого назначения библиотеки – редактирования текстов – стало возможным использование ее в виде самостоятельной утилиты, реализующей гибкую функциональность (`ConsoleTools, colorer.exe`). Эта программа использует простой интерфейс командной строки и позволяет генерировать выходные файлы, расцвеченные с использованием HTML-тэгов, что позволяет использовать библиотеку как средство улучшения читабельности текстов, выкладываемых в общий доступ. Кроме того, на основе этой утилиты был создан PHP-сервис, работающий на веб-сервере Apache под управлением операционной системы FreeBSD. Сервис предоставляет возможность, не устанавливая библиотеку на компьютер клиента, получать расцвеченные HTML-представления исходных текстов, передавая данные по HTTP протоколу.

В разработке библиотеки большое внимание уделялось не только архитектуре основного кода, но и качеству HRC описаний, используемых для представления синтаксиса языков. Для каждого типа файла (языка программирования) существует соответствующее ему HRC-описание, определяющее синтаксис этого языка. Посредством редактирования и создания HRC файлов, возможности библиотеки по поддержке языков могут расширяться.

В процессе развития библиотеки были созданы HRC типы для большинства современных языков программирования. Помимо описания структур расцветки они включают в себя логику обработки парных конструкций, встречающихся в каждом языке, и дополнительные синтаксические конструкции, используемые в расширенном представлении структуры текста. Поддержку любого языка можно делать с различным уровнем детализации и точности. Для простых схем языков часто было достаточно линейных структур, разбирающих основные встречающиеся лексемы в тексте. В то же время, более точное следование оригинальной грамматике языка позволяет отслеживать и находить различные нетривиальные ошибки в синтаксисе. Так, например, схемы языков C, C++, Java вводят разделение структуры текста на несколько контекстов, в

каждом из которых строго контролируется набор доступных операторов и лексем. Это позволяет сразу же обнаруживать многие логические ошибки.

Наиболее активно такой принцип применяется при разборе языков с XML синтаксисом. Созданный модуль преобразования `XSD2HRC` позволяет ввести поддержку языка XML Schema в библиотеку. HRC-описания, автоматически генерируемые им, полностью отслеживают контекст выражений и четко контролируют имена тэгов, доступных на каждом уровне вложенности. В процессе тестирования этого модуля были сгенерированы HRC-схемы для различных типов XML документов. Основные стандарты, представляющие интерес – это XSLT и XML Schema. Анализ схем этих языков позволил выявить основные требования к разбору XML-документов и выделить структуры XML схем, которые возможно транслировать в синтаксис HRC.

Пакет `XSD2HRC` можно применить и для схемы самого языка HRC. Полученный «HRC для HRC» корректно описывает структуру языка и, что очень важно, позволяет обнаружить практически все возможные ошибки, которые можно допустить как в синтаксисе XML представления HRC, так и в синтаксисе регулярных выражений этого языка.

В текущем состоянии для работы модуля необходимо предоставлять полную схему соответствующего типа документа. На практике же часто оказывается, что у некоторых языков нет явных схем, а они состоят из нескольких несвязанных компонентов. Конечно, всегда можно создать схему, описывающую нужный синтаксис (либо составить ее из существующих компонент), но часто это является довольно сложной задачей. В перспективе можно рассмотреть создание некоего параметризованного шаблона, который бы позволял быстро объединять нужные структуры схем и их определения. Уже на основе этого шаблона будет генерироваться вся структура HRC кодов. В этой же области лежит и решение другой проблемы – создание межъязыковых структур схем.

Очень часто реальный синтаксис языка позволяет внедрять в свою структуру элементы других типов документов. Например, стандарт XHTML в будущем будет тесно интегрироваться с разметкой XForms, уже сейчас некоторые из браузеров поддерживают визуализацию выражений MathML. Сами схемы этих языков могут и не указывать подобные взаимодействия. И, хотя всегда существует возможность изменить саму схему, для большинства из них, представляющих глобальные стандарты это не приемлемо, так как вносит элементы ручной работы в уже автоматизированный процесс.

К этой проблеме примыкает и проблема поддержки расширенных синтаксисов XML, которые, возможно не полностью могут быть описаны в XML Schema. Например, средство пакетного построения Ant использует свои конфигурационные XML файлы, которые могут иметь различное содержимое в зависимости от текущей настройки программы и установленных в ней расширений. Еще один аналогичный пример, плохо поддающийся структуризации в XML схемах – это конфигурации подключаемых модулей в среде Eclipse. Поддержка таких сложных синтаксисов разметки в полуавтоматическом режиме намного упростит использование пакета `XSD2HRC` и предоставит большую свободу в генерации их описаний.

Следует отметить, что технологии XML используются в библиотеке Colorer не только на уровне программирования. Многие задачи, возникавшие в процессе создания библиотеки, были решены с использованием языка XML. Так, например, сопроводительная документация по языку HRC описывается в формате DocBook. Она ссылается на документы XML Schema, описывающие синтаксис языка HRC. Помимо формальной части эти схемы содержат развернутые текстовые аннотации ко всем объявляемым компонентам. С использованием трансформаций XSLT эти пояснения вставляются в текст документации, наполняя ее формальным описанием объектов. Сами документы в

формате представления DocBook преобразуются в конечные HTML и PDF (XSL-FO) формы.

В перспективе у библиотеки Colorer существует много возможностей для дальнейшего роста и развития. Некоторые из модулей могут быть еще больше оптимизированы для достижения максимальной производительности, стоит рассмотреть возможность расширения набора высокоуровневых сервисов, обеспечивающих удобство работы с интерфейсами библиотеки и реализацию разнообразных вспомогательных алгоритмов. Основные требования, возникающие у пользователей в процессе работы с библиотекой, состоят в необходимости еще большего упрощения набора текстов и поддержки его со стороны приложения. В этой области следует рассмотреть возможность внедрения контекстно-зависимых шаблонов автодополнения, которые прикладная программа сможет использовать для повышения комфорта пользователей. Кроме этого, на основе анализа синтаксических регионов можно реализовать и иные операции, зависящие уже от требований и структуры обрабатываемого языка программирования.

Созданная библиотека классов Colorer, несомненно, занимает свою нишу в архитектуре интегрированных сред разработки и систем редактирования. Она обладает уникальными возможностями работы с языками программирования, основывается на гибких синтаксических правилах HRC, поддерживает синтаксис XML языков. Все эти возможности выделяют библиотеку из ряда аналогов и делают ее универсальным средством, обеспечивающим комфортное редактирование исходных текстов.

7. Литература

1. А. Ахо, Д. Ульман. Теория синтаксического анализа, перевода и компиляции. т.1,2 - М.: Мир, 1978.
2. Основы конструирования компиляторов. В.А.Серебряков, М.П.Галочкин, 2001.
3. Aho, Alfred V., Ravi Sethi, and Jeffrey D. Ullman. Compilers: Principles, Techniques, and Tools. Reading: Addison-Wesley, 1986, rpt. corr. 1988.
4. Extensible Markup Language (XML) 1.0 Second Edition.
Tim Bray, Jean Paoli, and C. M. Sperberg-McQueen, Eve Maler, editors.
W3C Recommendation, 2000. (http://www.w3.org/TR/2000/REC-xml-20001006)
5. The Unicode Consortium. The Unicode Standard, Version 3.0. Reading, Mass.: Addison-Wesley Developers Press, 2000.
(http://www.unicode.org/unicode/uni2book/u2.html)
6. XML Path Language (XPath).
James Clark, editor. W3C Recommendation, 1999. (http://www.w3.org/TR/xpath)
7. XSL Transformations (XSLT).
James Clark, editor. W3C Recommendation, 1999. (http://www.w3.org/TR/xslt)
8. XML Information Set. John Cowan, Richard Tobin, editors.
W3C Recommendation, 2001. (http://www.w3.org/TR/xml-infoset)
9. XML Schema Part 1: Structures.
Henry S. Thompson, David Beech, Murray Maloney, Noah Mendelsohn, editors.
W3C Recommendation, 2001. (http://www.w3.org/TR/xmlschema-1)
10. XML Schema Part 2: Datatypes.
Paul V. Biron, Ashok Malhotra, editors.
W3C Recommendation, 2001. (http://www.w3.org/TR/2001/xmlschema-2)
11. RELAX NG Specification.
James Clark, Makoto MURATA, editors.
OASIS, 2001. (http://www.oasis-open.org/committees/relax-ng/spec.html)
12. Unicode Regular Expression Guidelines.
Mark Davis. The Unicode Consortium, 2000.
(http://www.unicode.org/unicode/reports/tr18.html)
13. Java Language Specification, Second Edition.
James Gosling, Bill Joy, Guy Steele. Sun Microsystems, Inc., 2000.
(http://java.sun.com/docs/books/jls)
14. Java Native Interface Specification. Sun Microsystems, Inc. May 16, 1997.
15. Effective Java Programming Language Guide. Joshua Bloch. Addisson Wesley.
16. Технология XSLT. Валиков А.Н., BHV, 2002 г.
17. Объектно-ориентированный анализ и проектирование с примерами приложений на C++, Гради Буч, 1998 г.
18. Effective C++ / More Effective C++, Scott Douglas Meyers, 1998.
(http://www.awl.com/cseng/meyerscd)
19. C# и платформа .NET. Эндрю Троелсен, 2002 г.
20. DocBook: The Definitive Guide. Norman Walsh, Leonard Muellner. O'Reilly & Associates, Inc., 2002. (http://www.docbook.org)

8. Приложение А. Основные классы библиотеки

ErrorHandler.h

```
#ifndef _COLORER_ERRORHANDLER_H_
#define _COLORER_ERRORHANDLER_H_

#include<common/Common.h>

/**
  Informs application about different
  Parsing problems and warning.
  @ingroup colorer
*/
class ErrorHandler
{
public:
  /**
    Nonfatal parse error.
    Called when target class finds some non-fatal error.
    @param msg Error message
  */
  virtual void error(const String &msg) = 0;
  /**
    Fatal parse error.
    Called when target class finds fatal error, which could
    significantly change or break expected behaviour.
    @param msg Error message
  */
  virtual void fatalError(const String &msg) = 0;
  /**
    Simple warnings/info messages.
    Called when target class want to inform application
    about some warnings or informational messages.
    @param msg Warning message
  */
  virtual void warning(const String &msg) = 0;

  virtual ~ErrorHandler(){};
};

#endif
```

FileType.h

```
#ifndef _COLORER_FILETYPE_H_
#define _COLORER_FILETYPE_H_

#include<common/Common.h>
#include<colorer/Scheme.h>

class Scheme;

/** HRC FileType (or prototype) instance.
    @ingroup colorer
*/
class FileType{
public:

  /** Public name of file type (HRC 'name' attribute).
      @return File type Name
  */
  virtual const String *getName() = 0;

  /** Public group name of file type (HRC 'group' attribute).
      @return File type Group
  */
  virtual const String *getGroup() = 0;
```

```
  /** Public description of file type (HRC 'description' attribute).
      @return File type Description
  */
  virtual const String *getDescription() = 0;

  /** Returns the base scheme of this file type.
      Basically, this is the scheme with same public name, as it's type.
      If this FileType object is not yet loaded, it is loaded with this call.
      @return File type base scheme, to be used as root scheme of text
     parsing.
  */
  virtual Scheme *getBaseScheme() = 0;

  /** Returns parameter value of this file type.
      Parameters are stored in prototypes as
      <pre>
      \<parameters>
        \<param name="name" value="value"/>
      \</parameter>
      </pre>
      @note Parameters could be used to store client-application
            specific information about each type of file.
      @param name Parameter's name
  */
  virtual const String *getParameter(const String &name) = 0;
protected:
  FileType(){};
  virtual ~FileType(){};
};

#endif
```

HRCParser.h

```
#ifndef _COLORER_HRCPARSER_H_
#define _COLORER_HRCPARSER_H_

#include<common/io/InputSource.h>

#include<colorer/ErrorHandler.h>
#include<colorer/FileType.h>
#include<colorer/Region.h>


/**
  Informs application about internal HRC parsing problems.
  @ingroup colorer
*/
class HRCParserException : public Exception{
public:
  HRCParserException(){};
  HRCParserException(const String& msg){
    message->append(DString("HRCParserException: ")).append(msg);
  };
};


/** Abstract template of HRCParser class implementation.
    Defines basic operations of loading and accessing
    HRC information.
  @ingroup colorer
*/
class HRCParser
{
public:
  /** Error Handler, used to inform application about different error
     conditions
      @param eh ErrorHandler instance, or null to drop error handling.
  */
  virtual void setErrorHandler(ErrorHandler *eh) = 0;
```

```
  /** Loads HRC from specified InputSource stream.
      Referred HRC file can contain prototypes and
      real types definitions. If it contains just prototype definition,
      real type load must be performed before using with #loadType() method
      @param is InputSource stream of HRC file
  */
  virtual void loadSource(InputSource *is) = 0;

  /** Enumerates sequentially all prototypes
      @param index index of type.
      @return Requested type, or null, if #index is too big
  */
  virtual FileType *enumerateFileTypes(int index) = 0;

  /**
      @param name Requested type name.
      @return File type, or null, there are no type with specified name.
  */
  virtual FileType *getFileType(const String *name) = 0;

  /** Searches and returns the best type for specified file.
      This method uses fileName and firstLine parameters
      to perform selection of the best HRC type from database.
      @param fileName Name of file
      @param firstLine First line of this file, could be null
      @param typeNo Sequential number of type, if more than one type
                    satisfy these input parameters.
  */
  virtual FileType *chooseFileType(const String *fileName, const String
     *firstLine, int typeNo = 0) = 0;

  /** Total number of declared regions
  */
  virtual int getRegionCount() = 0;
  /** Returns region by internal id
  */
  virtual const Region *getRegion(int id) = 0;
  /** Returns region by name
      @note Also loads referred type, if it is not yet loaded.
  */
  virtual const Region *getRegion(const String *name) = 0;

  /** HRC base version.
      Usually this is the 'version' attribute of 'hrc' element
      of the first loaded HRC file.
  */
  virtual const String *getVersion() = 0;

  virtual ~HRCParser(){};
protected:
  HRCParser(){};
};
#endif
```

LineSource.h

```
#ifndef _COLORER_LINESOURCE_H_
#define _COLORER_LINESOURCE_H_

#include<common/Common.h>

/**
  Interface for editor line information requests.
  Basic data source interface, used in TextParser processing.
  @note Methods startJob and endJob are optional, and
  could be implemented or not depending on system archtecture.
  @ingroup colorer
*/
class LineSource
{
public:
```

```
  /** Called by parser, when it starts text parsing.
      @param lno Line number, which will be used as
      initial position of all subsequend parsing.
  */
  virtual void startJob(int lno){};

  /** Called by parser, when it has finished text parsing.
      Could be used to cleanup objects, allocated by last
      #getLine() call.
  */
  virtual void endJob(int lno){};

  /**
    Returns line of text with specified number.
    Returns String class pointer, which incapsulates information
    about line with number <code>lno</code>.
    @note Returned pointer must be valid until next getLine method call.
    @param lno Requested line number
    @return Unicode string, enwrapped into String class.
  */
  virtual String *getLine(int lno) = 0;
protected:
  LineSource(){};
  virtual ~LineSource(){};
};
#endif
```

ParserFactory.h

```
#ifndef _COLORER_PARSERFACTORY_H_
#define _COLORER_PARSERFACTORY_H_

#include<xml/xml.h>
#include<colorer/TextParser.h>
#include<colorer/HRCParser.h>
#include<colorer/handlers/DefaultErrorHandler.h>
#include<colorer/handlers/StyledHRDMapper.h>
#include<colorer/handlers/TextHRDMapper.h>
#include<colorer/ParserFactoryException.h>


/** Maintains catalog of HRC and HRD references.
    This class searches and loads <code>catalog.xml</code> file
    and creates HRCParser, StyledHRDMapper, TextHRDMapper and TextParser
     instances
    with information, loaded from specified sources.

    If no path were passed to it's constructor,
    it uses next search order of 'catalog.xml' file:

    - global:
      - \%COLORER5CATALOG%
      - \%HOME%/.colorer5catalog
      - \%HOMEPATH%/.colorer5catalog

    - win32 systems:
      - \%SYSTEMROOT%/.colorer5catalog (or \%WINDIR% in w9x)
      - image_start_dir, image_start_dir/../, image_start_dir/../../

    - unix systems:
      - ./catalog.xml
      - ../catalog.xml
      - /usr/shared/colorer/catalog.xml

    @ingroup colorer
*/
class ParserFactory{
public:

  /** ParserFactory Constructor.
      Searches for catalog.xml in the set of predefined locations
      @throw ParserFactoryException If can't find catalog at any of standard
     locations.
```

```
  */
  ParserFactory();

  /** ParserFactory Constructor with explicit catalog path.
      @param catalogPath Path to catalog.xml file. If null,
             standard search method is used.
      @throw ParserFactoryException If can't load specified catalog.
  */
  ParserFactory(const String *catalogPath);
  virtual ~ParserFactory();

  static const char *getVersion();

  /** Enumerates all declared hrd classes
  */
  const String *enumerateHRDClasses(int idx);

  /** Enumerates all declared hrd instances of specified class
  */
  const String *enumerateHRDInstances(const String &classID, int idx);

  /** Returns description of HRD instance, pointed by classID and nameID
     parameters.
  */
  const String *getHRDescription(const String &classID, const String &nameID);

  /** Creates and loads HRCParser instance from catalog.xml file.
      This method can detect directory entries, and sequentally load their
      contents into created HRCParser instance.
      In other cases it uses InputSource#newInstance() method to
      create input data stream.
      Only one HRCParser instance is created for each ParserFactory instance.
  */
  HRCParser  *getHRCParser();

  /** Creates TextParser instance
  */
  TextParser *createTextParser();

  /** Creates RegionMapper instance and loads specified hrd files into it.
      @param classID Class identifier of loaded hrd instance.
      @param nameID  Name identifier of loaded hrd instances.
      @throw ParserFactoryException If method can't find specified pair of
              class and name IDs in catalog.xml file
  */
  StyledHRDMapper *createStyledMapper(const String *classID, const String
     *nameID);
  /** Creates RegionMapper instance and loads specified hrd files into it.
      It uses 'text' class by default.
      @param nameID  Name identifier of loaded hrd instances.
      @throw ParserFactoryException If method can't find specified pair of
              class and name IDs in catalog.xml file
  */
  TextHRDMapper *createTextMapper(const String *nameID);

  /** Returns currently used global error handler.
      If no error handler were installed, returns null.
  */
  ErrorHandler *getErrorHandler(){
    return fileErrorHandler;
  };

private:
  void init();
  String *searchPath();

  String *catalogPath;
  InputSource *catalogFIS;
  ErrorHandler *fileErrorHandler;
  Vector<const String*> hrcLocations;
  Hashtable<Hashtable<Vector<const String*>*>*> hrdLocations;
```

```
  Hashtable<const String *>hrdDescriptions;
  HRCParser  *hrcParser;
  CXmlEl *catalog;

  ParserFactory(const ParserFactory&);
  void operator=(const ParserFactory&);
};
#endif
```

ParserFactoryException.h

```
#ifndef _COLORER_PARSERFACTORY_EXCEPTION_H_
#define _COLORER_PARSERFACTORY_EXCEPTION_H_

#include<common/Common.h>

/** Exception, thrown by ParserFactory class methods.
    Indicates some (mostly fatal) errors in loading
    of catalog file (catalog.xml), or in creating
    parsers objects.
    @ingroup colorer
*/
class ParserFactoryException : public Exception{
public:
  ParserFactoryException(){};
  ParserFactoryException(const String& msg){
    message->append(DString("ParserFactoryException: ")).append(msg);
  };
};

#endif
```

Region.h

```
#ifndef _COLORER_REGION_H_
#define _COLORER_REGION_H_

#include<common/Common.h>

/**
  HRC Region implementation.
  Contains information about HRC Region and it attributes:
  <ul>
    <li>name
    <li>description
    <li>parent
  </ul>
  @ingroup colorer
*/
class Region{
public:
  /** Full Qualified region name (<code>def:Text</code> for example) */
  virtual const String *getName() const{ return name; };
  /** Region description */
  virtual const String *getDescription() const{ return description; };
  /** Direct region ancestor (parent) */
  virtual const Region *getParent() const{ return parent; };
  /** Quick access region id (incrementable) */
  virtual int getID() const{ return id; };
  /** Checks if region has the specified parent in all of it's ancestors.
      This method is useful to check if region has specified parent,
      and use this information, as region type specification.
      For example, <code>def:Comment</code> has <code>def:Syntax</code>
     parent,
      so, some syntax checking can be made with it's content.
  */
  bool hasParent(const Region *region) const{
    const Region *elem = this;
    while(elem != null){
      if (region == elem) return true;
      elem = elem->getParent();
    };
    return false;
```

```
  };
  /**
    Basic constructor.
    Used only by HRCParser.
  */
  Region(const String *_name, const String* _description, const Region
     *_parent, int _id){
    name = new SString(_name);
    description = null;
    if (_description != null) description = new SString(_description);
    parent = _parent;
    id = _id;
  };
  virtual ~Region(){
    delete name;
    delete description;
  };
protected:
  /** Internal members */
  String *name, *description;
  const Region *parent;
  int id;
};

#endif
```

RegionHandler.h

```
#ifndef _COLORER_REGIONHANDLER_H_
#define _COLORER_REGIONHANDLER_H_

#include<colorer/Region.h>
#include<colorer/Scheme.h>

/** Handles parse information, passed from TextParser.
    TextParser class generates calls of this class methods
    sequentially while parsing the text from top to bottom.
    All enterScheme and leaveScheme calls are properly enclosed,
    addRegion calls can inform about regions, overlapped with each other.
    All handler methods are called sequentially. It means, that
    if one of methods is called with some line number, all other calls
    (before endParsing event comes) can inform about events in the same,
    or lower line's numbers. This makes sequential tokens processing.
    @ingroup colorer
*/
class RegionHandler{
public:
  /** Start of text parsing.
      Called only once, when TextParser starts
      parsing of the specified block of text.
      All other event messages comes between this call and
      endParsing call.
      @param lno Start line number
  */
  virtual void startParsing(int lno){};

  /** End of text parsing.
      Called only once, when TextParser stops
      parsing of the specified block of text.
      @param lno End line number
  */
  virtual void endParsing(int lno){};

  /** Clear line event.
      Called once for each parsed text line, when TextParser starts to parse
      specified line of text. This method is called before any of the region
      information passed, and used often to clear internal handler
      structure of this line before adding new one.
      @param lno Line number
  */
  virtual void clearLine(int lno, String *line){};
```

```
  /** Informs handler about lexical region in line.
      This is a basic method, wich transfer information from
      parser to application. Positions of different passed regions
      can be overlapped.
      @param lno Current line number
      @param sx Start X position of region in line
      @param ex End X position of region in line
      @param region Region information
  */
  virtual void addRegion(int lno, String *line, int sx, int ex, const Region
     *region) = 0;

  /** Informs handler about entering into specified scheme.
      Parameter <code>region</code> is used to specify
      scheme background region information.
      If text is parsed not from the first line, this method is called
      with fake parameters to compensate required scheme structure.
      @param lno Current line number
      @param sx Start X position of region in line
      @param ex End X position of region in line
      @param region Scheme Region information (background)
      @param scheme Additional Scheme information
  */
  virtual void enterScheme(int lno, String *line, int sx, int ex, const Region
     *region, const Scheme *scheme) = 0;

  /** Informs handler about leaveing specified scheme.
      Parameter <code>region</code> is used to specify
      scheme background region information.
      If text parse process ends, but current schemes stack is not balanced
      (this can happends because of bad balanced structure of source text,
      or partial text parse) this method is <b>not</b> called for unbalanced
      levels.
      @param lno Current line number
      @param sx Start X position of region in line
      @param ex End X position of region in line
      @param region Scheme Region information (background)
      @param scheme Additional Scheme information
  */
  virtual void leaveScheme(int lno, String *line, int sx, int ex, const Region
     *region, const Scheme *scheme) = 0;

protected:
  RegionHandler(){};
  virtual ~RegionHandler(){};
};

#endif
```

Scheme.h

```
#ifndef _COLORER_SCHEME_H_
#define _COLORER_SCHEME_H_

#include<colorer/FileType.h>

class FileType;

/** HRC Scheme instance information.
    Used in RegionHandler calls to pass curent region's scheme.
    @ingroup colorer
*/
class Scheme
{
public:
  /** Full qualified schema name.
  */
  virtual const String *getName() = 0;
  /** Returns reference to FileType, this scheme belongs to.
  */
  virtual FileType *getFileType() = 0;
protected:
```

```
  Scheme(){};
  virtual ~Scheme(){};
};

#endif
```

TextParser.h

```
#ifndef _COLORER_TEXTPARSER_H_
#define _COLORER_TEXTPARSER_H_

#include<colorer/FileType.h>
#include<colorer/LineSource.h>
#include<colorer/RegionHandler.h>

/** Basic lexical/syntax parser interface.
    This class provides interface to lexical text parsing abilities of
    Colorer library.
    It uses LineSource as source of input data, and RegionHandler
    as interface to transfer results of text parse process.

    Process of syntax parsing supports internal caching algorithims,
    which allows to store internal parser state and reparse text
    only partially (on change, on request).

    @ingroup colorer
*/
class TextParser
{
public:
  /** Sets root scheme (filetype) of parsed text
      @param type FileType, which contains reference to it's baseScheme.
             If parameter is null, there will be no any kind of parse
             over the text.
  */
  virtual void setFileType(FileType *type) = 0;
  /** LineSource, used as input of parsing text
  */
  virtual void setLineSource(LineSource *lh) = 0;
  /** RegionHandler, used as outputter of parsed information
  */
  virtual void setRegionHandler(RegionHandler *rh) = 0;

  /** Performs cachable text parse.
      Builds internal structure of contexts,
      allowing appriсation to continue parse from any already
      reached position of text. This guarantees the validness of
      result parse information.
      @param from Start parsing line
      @param num Number of lines to parse
  */
  virtual int parse(int from, int num) = 0;

  /** Performs break of parsing process from external thread.
      It is used to stop parse from external source. This needed
      in some editor systems implementation, when editor system
      detects background changes in highlighted text, for example.
  */
  virtual void breakParse() = 0;
  /** Clears cached text stucture information
  */
  virtual void clearCache() = 0;

  virtual ~TextParser(){};
protected:
  TextParser(){};
};


#endif
```

Editor Interfaces

BaseEditor.h

```
#ifndef _COLORER_BASEEDITOR_H_
#define _COLORER_BASEEDITOR_H_

#include<colorer/ParserFactory.h>
#include<colorer/handlers/FileErrorHandler.h>
#include<colorer/handlers/LineRegionsSupport.h>
#include<colorer/handlers/LineRegionsCompactSupport.h>

#include<colorer/editor/PairMatch.h>

/** Base Editor functionality.
    This class implements basic functionality,
    which could be useful in application's editing system.
    This includes automatic top-level caching of hilighting
    state, outline structure creation, pair constructions search.
    This class has event-oriented structure. Each editor event
    is passed into this object and gets internal processing.
    @ingroup colorer_editor
*/
class BaseEditor : public RegionHandler{
public:
  /** Initial constructor.
      Creates uninitialized base editor functionality support.
      @param pf ParserFactory, used as source of all created
             parsers (HRC, HRD, Text parser). Can't be null.
      @param lineSource Object, that provides parser with
             text data in line-separated form. Can't be null.
  */
  BaseEditor(ParserFactory *pf, LineSource *lineSource);
  ~BaseEditor();

  /** This method informs handler about internal form of
      requeried LineRegion lists, which returned after parse
      process. Compact regions are guaranteed not to overlap
      with each other (this is achieved with more internal processing
      and more extensive cpu use); non-compact regions are placed directly
      as they created by TextParser and can be overlapped.
      @note By default, if method is not called, regions are not compacted.
      @param compact Creates LineRegionsSupport (false) or
     LineRegionsCompactSupport (true)
             object to store lists of RegionDefine's
  */
  void setRegionCompact(bool compact);

  /** Installs specified RegionMapper, which
      maps HRC Regions into color data.
      @param rm RegionMapper object to map region values into colors.
  */
  void setRegionMapper(RegionMapper *rm);

  /** Installs specified RegionMapper, which
      is created with ParserFactory methods and maintained internally by this
     handler.
      If no one of two overloads of setRegionMapper is called,
      all work is started without mapping of extended region information.
      @param hrdClass Class of RegionMapper instance
      @param hrdName  Name of RegionMapper instance
  */
  void setRegionMapper(const String *hrdClass, const String *hrdName);

  /** Specifies number of lines, for which parser
      would be able to run continual processing without
      highlight invalidation.
  */
  void setBackParse(int backParse);
```

```
/** Initial HRC type, used for parse processing.
    If changed during processing, all text information
    is invalidated.
*/
void setFileType(FileType *ftype);
/** Initial HRC type, used for parse processing.
    If changed during processing, all text information is invalidated.
*/
void setFileType(const String &fileType);
/** Tries to choose appropriate file type from HRC database
    using passed fileName and first line of text (if available through
   lineSource)
*/
void chooseFileType(const String *fileName);

/** Returns currently used HRC file type */
FileType *getFileType();

/** Adds specified RegionHandler object
    into parse process.
*/
void addRegionHandler(RegionHandler *rh);

/** Removes previously added RegionHandler object.
*/
void removeRegionHandler(RegionHandler *rh);

/** Searches and creates pair match object.
    Returned object could be lately used in pair search methods.
    This object is valid only until reparse of it's line
    occured. After that event information about line region's
    references in it becomes invalid and, if used, can produce
    faults.
    @param lineNo Line number, where to search paired region.
    @param pos Position in line, where paired region to be searched.
           Paired Region is found, if it includes specified position
           or ends directly at one char before line position.
*/
PairMatch *getPairMatch(int lineNo, int pos);

/** Searches and creates pair match object of first enwrapping block.
    Returned object could be used as with getPairMatch method.
    Enwrapped block is the first meeted start of block, if moving
    from specified position to the left and top.
    @param lineNo Line number, where to search paired region.
    @param pos Position in line, where paired region to be searched.
*/
PairMatch *getEnwrappedPairMatch(int lineNo, int pos);

/** Frees previously allocated PairMatch object.
    @param pm PairMatch object
*/
void releasePairMatch(PairMatch *pm);

/** Searches pair match in currently visible text.
    @param pm Unmatched pair match
*/
void searchLocalPair(PairMatch *pm);

/** Searches pair match in all available text, possibly,
    making additional processing.
    @param pm Unmatched pair match
*/
void searchGlobalPair(PairMatch *pm);

/** Return parsed and colored LineRegions of requested line.
    This method validates current cache state
    and, if needed, calls Colorer parser to validate modified block of text.
    Size of reparsed text is choosed according to information
    about visible text range and modification events.
```

```
        @todo If number of lines, to be reparsed is more, than backParse
       parameter,
        then method will return null, until validate() method is called.
    */
    LineRegion *getLineRegions(int lno);

    /** Validates current state of editor and runs parser, if needed.
        This method can be called periodically in background thread
        to make possible background parsing process.
        @param lno Line number, for which validation is requested.
               If this number is in current visible window range,
               the part of text is validated, which is requeried
               for visual painting.
               If this number is not in visible range, or equals to -1,
               all the text is validated.
    */
    void validate(int lno);

    /** Informs BaseEditor object about text modification event.
        All the text becomes invalid after the specified line.
        @param topLine Topmost modified line of text.
    */
    void modifyEvent(int topLine);

    /** Informs about single line modification event.
        Generally, this type of event can be processed much faster
        because of pre-checking line's changed structure and
        cancelling further parsing in case of unmodified text structure.
        @param line Modified line of text.
        @todo Not used yet! This must include special 'try' parse method.
    */
    void modifyLineEvent(int line);

    /** Informs about changes in visible range of text lines.
        This information is used to make assumptions about
        text structure and to make faster parsing.
        @param wStart Topmost visible line of text.
        @param wSize  Number of currently visible text lines.
                      This number must includes all partially visible lines.
    */
    void visibleTextEvent(int wStart, int wSize);

    /** Informs about total lines count change.
        This must include initial lines number setting.
    */
    void lineCountEvent(int newLineCount);

    /** Basic HRC region - default text (background color) */
    const Region *def_Text;
    /** Basic HRC region - syntax checkable region */
    const Region *def_Syntax;
    /** Basic HRC region - special region */
    const Region *def_Special;
    /** Basic HRC region - Paired region start */
    const Region *def_PairStart;
    /** Basic HRC region - Paired region end */
    const Region *def_PairEnd;

    /** Basic HRC region mapping */
    const RegionDefine *rd_def_Text, *rd_def_HorzCross, *rd_def_VertCross;

    void startParsing(int lno);
    void endParsing(int lno);
    void clearLine(int lno, String *line);
    void addRegion(int lno, String *line, int sx, int ex, const Region *region);
    void enterScheme(int lno, String *line, int sx, int ex, const Region
       *region, const Scheme *scheme);
    void leaveScheme(int lno, String *line, int sx, int ex, const Region
       *region, const Scheme *scheme);

private:
```

```
  HRCParser *hrcParser;
  TextParser *textParser;
  ParserFactory *parserFactory;
  LineSource *lineSource;
  RegionMapper *regionMapper;
  LineRegionsSupport *lrSupport;

  FileType *currentFileType;
  Vector<RegionHandler*> regionHandlers;

  int backParse;
  // window area
  int wStart, wSize;
  // line count
  int lineCount;
  // size of line regions
  int lrSize;
  // position of last validLine
  int invalidLine;
  // no lines structure changes, just single line change
  int changedLine;

  bool internalRM;
  bool regionCompact;
  bool breakParse;
  bool validationProcess;

  inline int getLastVisibleLine();
  void remapLRS(bool recreate);
protected:
  ErrorHandler *feh;
};

#endif
```

OutlineItem.h

```
#ifndef _COLORER_OUTLINEITEM_H_
#define _COLORER_OUTLINEITEM_H_

#include<colorer/Region.h>

/** Item in outliner's list.
    Contans all the information about single
    structured token with specified type (region reference).
    @ingroup colorer_editor
*/
class OutlineItem {
public:
  /** Line number */
  int lno;
  /** Position in line */
  int pos;
  /** Level of enclosure */
  int level;
  /** Item text */
  StringBuffer *token;
  /** This item's region */
  const Region *region;

  /** Default constructor */
  OutlineItem(){
    lno = pos = 0;
    token = null;
  };
  /** Initializing constructor */
  OutlineItem(int lno, int pos, int level, String *token, const Region
     *region){
    this->lno = lno;
    this->pos = pos;
```

```
    this->level = level;
    this->region = region;
    this->token = null;
    if (token != null) this->token = new StringBuffer(token);
  };
  ~OutlineItem(){ delete token; };
};

#endif
```

Outliner.h

```
#ifndef _COLORER_OUTLINER_H_
#define _COLORER_OUTLINER_H_

#include<common/Vector.h>
#include<colorer/LineSource.h>
#include<colorer/RegionHandler.h>
#include<colorer/editor/OutlineItem.h>

/** Used to create, store and maintain
    lists of different special regions.
    These can include functions, methods, fields,
    classes, errors and so on.
    @ingroup colorer_editor
*/
class Outliner : public RegionHandler {
public:
  /** Creates outliner object, that searches stream for
      the specified type of region.
      @param searchRegion Region type to search in parser's stream
  */
  Outliner(const Region *searchRegion);
  ~Outliner();

  /** Returns reference to item with specified ordinal
      index in list of currently generated outline items.
      Note, that pointer is correct only between subsequent
      parser calls.
  */
  OutlineItem *getItem(int idx);

  /** Total number of currently available outline items
  */
  int itemCount();

  /** Static service method to make easy tree reconstruction
      from created list of outline items. This list contains
      unpacked level indexed of item's enclosure in scheme.
      @param treeStack external Vector of integer, storing
             temporary tree structure. Must not be changed
             externally.
      @param newLevel Unpacked level of item to be added into
             the tree. This index is converted into packed one
             and returned.
      @return Packed index of item, which could be used to
              reconstruct tree of outlined items.
  */
  static int manageTree(Vector<int> &treeStack, int newLevel);

  void startParsing(int lno);
  void endParsing(int lno);
  void clearLine(int lno, String *line);
  void addRegion(int lno, String *line, int sx, int ex, const Region *region);
  void enterScheme(int lno, String *line, int sx, int ex, const Region
     *region, const Scheme *scheme);
  void leaveScheme(int lno, String *line, int sx, int ex, const Region
     *region, const Scheme *scheme);

protected:
```

```
  bool isOutlined(const Region*region);

  const Region *searchRegion;
  Vector<OutlineItem*> outline;
  bool lineIsEmpty;
  int curLevel;
};

#endif
```

PairMatch.h

```
#ifndef _COLORER_PAIRMATCH_H_
#define _COLORER_PAIRMATCH_H_

#include<colorer/handlers/LineRegionsSupport.h>

/** Representation of pair match in text.
    Contains information about two regions on two lines.
    @ingroup colorer_editor
*/
class PairMatch{
public:
  /** Region's start position */
  LineRegion *start;
  /** Region's end position */
  LineRegion *end;
  /** Starting Line of pair */
  int sline;
  /** Ending Line of pair */
  int eline;
  /** Identifies initial position of cursor in pair */
  bool topPosition;
  /** Internal pair search counter */
  int pairBalance;

  /** Default constructor.
      Clears all fields
  */
  PairMatch::PairMatch(){
    sline = eline = -1;
    pairBalance = 0;
    topPosition = true;
  };
};

#endif
```

9. Приложение B. XSLT-модуль XSD2HRC

xsd2hrc.params.xsl

```
<?xml version="1.0" encoding="windows-1251"?>
<xsl:stylesheet version="1.0"
     xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <!-- global type name parameter -->

  <xsl:param name="hrctype" select="'default-type'"/>

  <!-- allows common xml attributes (xml:*, xsi:*, xmlns:*) to appear in any
     context -->

  <xsl:param name="allow-common-attr" select="'yes'"/>

  <!-- allows any attributes to appear in any context -->

  <xsl:param name="allow-any-attr" select="'no'"/>

  <!-- allows unknown elements everywhere -->

  <xsl:param name="allow-unknown-elements" select="'no'"/>

  <!-- allows unknown root elements -->

  <xsl:param name="allow-unknown-root-elements" select="'no'"/>

  <!-- force single root element checks -->

  <xsl:param name="force-single-root" select="'yes'"/>

  <!-- include prototype definition into target file -->

  <xsl:param name="include-prototype" select="'no'"/>

  <!-- path to HRC catalog (colorer.hrc) -->

  <xsl:param name="catalog-path" select="'../../hrc/colorer.hrc'"/>

  <!-- path to custom parser file -->

  <xsl:param name="custom-defines" select="'custom.default.xml'"/>

  <!-- Use specified single top-level element
       If not specified, all global elements could be at top level of file.
  -->
  <xsl:param name="top-level-element" select="''"/>

  <!-- internal anonymous type prefix -->

  <xsl:variable name="anonymous" select="'_hrc_int_'"/>

</xsl:stylesheet>
```

xsd2hrc.typedefs.xsl

```
<?xml version="1.0" encoding="windows-1251"?>
<xsl:stylesheet
     version="1.0"
     exclude-result-prefixes="c hrc xsl"
     xmlns="http://colorer.sf.net/2003/hrc"
     xmlns:c="uri:colorer:custom"
     xmlns:hrc="http://colorer.sf.net/2003/hrc"
     xmlns:xs="http://www.w3.org/2001/XMLSchema"
     xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
```

```
<!-- Mode 'typedefs' searches and defines all possible
complex and simple types: named and anonymous.
-->

<xsl:template match="text()" mode="typedefs">
</xsl:template>


<!-- Basic template, defines schema datatypes.
We have three kinds of datatypes to compose in HRC:
  1. complexType with complexContent
     basic scheme '{$typename}-content' have to be used only
     within element's content and can contain just elements,
     possibly mixed with _untyped_ cdata
  2. complexType with simpleContent
     basic scheme '{$typename}-content' could be used within
     element's content - here we have to allow xml:content.cdata
     and xml:content.other (for PI!?). Simultaneously we must
     check and parse here simpleType definition.
  3. simpleType
     basic scheme '{$typename}-content' could be used within
      a. attribute's content - include just Entities, badChars
         and simpleType definitions.
      b. element's content - as with 2nd variant
-->
<xsl:template match="xs:complexType | xs:simpleType" mode="typedefs">

  <xsl:variable name="typename">
    <xsl:choose>
      <xsl:when test="@name"><xsl:value-of select="@name"/></xsl:when>
      <xsl:otherwise><xsl:value-of select="concat($anonymous, generate-
   id(.))"/></xsl:otherwise>
    </xsl:choose>
  </xsl:variable>

  <xsl:if test="@name and preceding-sibling::*[position()=1 and local-name()
   = 'annotation']">
    <xsl:call-template name='crlf'/>
    <xsl:comment>
    <xsl:value-of select="preceding-sibling::*[position()=1 and local-
   name()='annotation']"/><xsl:call-template name='crlf'/>
    </xsl:comment>
  </xsl:if>
  <xsl:call-template name='crlf'/>
  <xsl:if test="../@name">
    <xsl:comment>
      parent: <xsl:value-of select="../@name"/><xsl:call-template
   name='crlf'/>
    </xsl:comment><xsl:call-template name='crlf'/>
  </xsl:if>
  <!-- Base scheme - creates real type content
  Could be customized in separate file, and included separately.
  This is done mostly for basic xmlscheme types, but you can use it
  to extend your schema abilities.
  Customization is made also for groups definitions
  -->
  <xsl:call-template name="createScheme">
    <xsl:with-param name="name" select="concat($typename, '-content')"/>
    <xsl:with-param name="content">
      <xsl:if test="xs:annotation/xs:documentation">
        <annotation><documentation>
          <xsl:value-of select="xs:annotation"/>
        </documentation></annotation>
      </xsl:if>
      <xsl:apply-templates mode="include-content" select="."/>
      <xsl:if test="self::xs:complexType and not(xs:simpleContent) and
   $allow-unknown-elements = 'yes'">
        <inherit scheme="xml:element"/>
      </xsl:if>
    </xsl:with-param>
```

```
    </xsl:call-template>

    <!-- This scheme adds errors highlighting into simple type -->
    <xsl:call-template name="createScheme">
      <xsl:with-param name="name" select="concat($typename, '-content-
     error')"/>
      <xsl:with-param name="content">
        <inherit scheme="{$typename}-content"/>
        <inherit scheme="xml:badChar"/>
      </xsl:with-param>
    </xsl:call-template>

    <!-- complexType attributes -->

<!--    <xsl:if test="self::xs:complexType">-->
      <scheme name="{$typename}-Attributes">
        <xsl:apply-templates mode="include-attr"/>
        <xsl:if test="$allow-any-attr != 'no'">
          <inherit scheme="xml:Attribute.any"/>
        </xsl:if>
      </scheme>
<!--    </xsl:if>
    <xsl:if test="self::xs:simpleType">
      <scheme name="{$typename}-Attributes"/>
    </xsl:if> -->

    <!-- We need this scheme to call from attribute definitions.
    So, we create it only for simpleType's
    -->

    <xsl:if test="self::xs:simpleType">
      <scheme name="{$typename}-AttributeContent">
        <inherit scheme="AttributeContent">
          <virtual scheme="xml:AttValue.content.stream" subst-
     scheme="{$typename}-content-error"/>
        </inherit>
      </scheme>
    </xsl:if>

    <!--
    Scheme to support calls from element's creation.
    Active for any kind of type
    -->
    <xsl:if test="self::xs:complexType or (self::xs:simpleType and @name)">
      <scheme name="{$typename}-elementContent">
        <inherit scheme="{$anonymous}elementContent">
          <xsl:if test="self::xs:complexType and not(xs:simpleContent)">
            <virtual scheme="xml:element"  subst-scheme="{$typename}-
     content"/>
            <xsl:choose>
              <xsl:when test="not( xs:complexContent/@mixed and (
                                   string(xs:complexContent/@mixed) = 'true'
                                   or string(xs:complexContent/@mixed) = '1')
                                  or
                                   not(xs:complexContent/@mixed) and
                                   (string(@mixed) = 'true' or string(@mixed)
     = '1')
                             )">
                <virtual scheme="xml:content.cdata" subst-
     scheme="xml:badChar"/>
              </xsl:when>
            </xsl:choose>
          </xsl:if>
          <!-- Allows to parse simpleType content in CDATA sections content --
     >
          <xsl:if test="self::xs:simpleType or xs:simpleContent">
            <virtual scheme="xml:CDSect.content.stream" subst-
     scheme="{$typename}-content-error"/>
            <virtual scheme="xml:content.cdata.stream"  subst-
     scheme="{$typename}-content-error"/>
            <virtual scheme="xml:element" subst-scheme="def:empty"/>
```

```
          </xsl:if>

          <virtual scheme="xml:Attribute.any" subst-scheme="{$typename}-
      Attributes"/>
          <xsl:if test="self::xs:complexType and $allow-common-attr != 'yes'">
            <virtual scheme="xml:Attribute.common" subst-scheme="def:empty"/>
          </xsl:if>
        </inherit>
       </scheme>
    </xsl:if>
    <!--
    Recursively searches any other types (named or unnamed)
    -->
    <xsl:apply-templates mode="typedefs"/>

  </xsl:template>

</xsl:stylesheet>
```

xsd2hrc.xsl

```
<?xml version="1.0" encoding="windows-1251"?>
<xsl:stylesheet
     version="1.0"
     exclude-result-prefixes="c hrc xsl"
     xmlns="http://colorer.sf.net/2003/hrc"
     xmlns:c="uri:colorer:custom"
     xmlns:hrc="http://colorer.sf.net/2003/hrc"
     xmlns:xs="http://www.w3.org/2001/XMLSchema"
     xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:include href="xsd2hrc.params.xsl"/>
  <xsl:include href="replace-string.xsl"/>

  <xsl:include href="xsd2hrc.callable.xsl"/>
  <xsl:include href="xsd2hrc.include-content.xsl"/>
  <xsl:include href="xsd2hrc.include-attr.xsl"/>
  <xsl:include href="xsd2hrc.typedefs.xsl"/>
  <xsl:include href="xsd2hrc.root.xsl"/>


  <xsl:output indent="yes"
              doctype-public="-//Cail Lomecb//DTD Colorer HRC take5//EN"
              doctype-system="http://colorer.sf.net/2003/hrc.dtd"
     encoding="utf-8"/>

  <!-- ignore spaces -->

  <xsl:strip-space elements="*"/>

  <!-- version -->
  <xsl:variable name="version" select="'0.9'"/>

  <!-- path to prototype catalog -->
  <xsl:variable name="catalog" select="document($catalog-path)"/>

  <!-- path to replace patterns list -->
  <xsl:variable name="replace-patterns" select="document('xsd2hrc.replace-
     patterns.xml')"/>

  <!-- Schema's targetNamespace
    In case of noTargetNamespace schema, transformation results
    may need some manual fixes. Also it is impossible to link
    such a schema with other generated schemas
  -->
  <xsl:variable name="targetNamespace"
     select="string(/xs:schema/@targetNamespace)"/>

  <!-- path to custom defines catalog -->
```

```
<xsl:variable name="custom-type" select="document($custom-
   defines)/c:custom/c:custom-type[@targetNamespace = $targetNamespace]"/>
<xsl:variable name="custom-type-schemes" select="$custom-type/hrc:type/*"/>


<!-- possible used namespace prefixes -->

<xsl:variable name="ns-map" select="$custom-type/c:prefix | $custom-
   type/c:empty-prefix | $custom-type/c:any-prefix"/>
<xsl:variable name="ns-real-prefix">
  <xsl:if test="$ns-map">
    <xsl:text>(?{}</xsl:text>
    <xsl:choose>
      <xsl:when test="$ns-map/self::c:any-prefix">
        <xsl:text>%xml:NCName;</xsl:text>
      </xsl:when>
      <xsl:when test="$ns-map/self::c:prefix">
        <xsl:text>(?{}</xsl:text>
        <xsl:for-each select="$ns-map/self::c:prefix">
          <xsl:value-of select="."/>
          <xsl:if test="position() != last()">|</xsl:if>
        </xsl:for-each>
        <xsl:text>)</xsl:text>
      </xsl:when>
    </xsl:choose>
    <xsl:text>:)</xsl:text>
  </xsl:if>
</xsl:variable>
<xsl:variable name="nsprefix">
  <xsl:if test="$ns-map">
    <xsl:value-of select="$ns-real-prefix"/>
    <xsl:if test="$ns-map/self::c:empty-prefix">?</xsl:if>
  </xsl:if>
</xsl:variable>

<!-- New line character -->
<xsl:template name="crlf">
  <xsl:text>&#10;</xsl:text>
</xsl:template>

<!-- insert comments -->
<xsl:template match="comment()">
  <xsl:call-template name='crlf'/><xsl:copy/><xsl:call-template
   name='crlf'/>
</xsl:template>


<!-- root template
Creates basic defines, schemes,
and calls recursive processing
-->

<xsl:template match="/">
  <xsl:if test="$hrctype = 'default-type'">
    <xsl:message>warning: Default Type Name is not specified (use param
   hrctype=name) </xsl:message>
  </xsl:if>

  <hrc version="take5"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://colorer.sf.net/2003/hrc
   http://colorer.sf.net/2003/hrc.xsd">

    <xsl:if test="$include-prototype = 'yes'">
      <prototype name="{$hrctype}" group="group" description="{$hrctype}"
   targetNamespace="{$targetNamespace}">
        <location link="link-to-{$hrctype}/>"/>
        <filename>/ filemask /ix</filename>
      </prototype>
    </xsl:if>
```

```
      <xsl:if test="$include-prototype = 'no'">
<xsl:call-template name='crlf'/>
<xsl:comment>
  insert this define into HRC base catalog file (colorer.hrc)

  &lt;prototype name="<xsl:value-of select='$hrctype'/>" group="group"
     description="<xsl:value-of select='$hrctype'/>"
     targetNamespace="<xsl:value-of select='$targetNamespace'/>">
    &lt;location link="<xsl:value-of select='$hrctype'/>.hrc"/>
    &lt;filename>/\./ix&lt;/filename>
  &lt;/prototype>
</xsl:comment>
      </xsl:if>

      <type name="{$hrctype}">
        <annotation>
         <documentation>
           XSLT Generated HRC scheme for language '<xsl:value-of
     select="$hrctype"/>'
           from XML Schema with xsd2hrc.xsl version <xsl:value-of
     select="$version"/> (c) Cail Lomecb

           Scheme parameters:
             targetNamespace            : <xsl:value-of
     select="$targetNamespace"/>
             hrctype                    : <xsl:value-of select="$hrctype"/>
             allow-common-attr          : <xsl:value-of select="$allow-
     common-attr"/>
             allow-any-attr             : <xsl:value-of select="$allow-any-
     attr"/>
             allow-unknown-elements     : <xsl:value-of select="$allow-
     unknown-elements"/>
             allow-unknown-root-elements : <xsl:value-of select="$allow-
     unknown-root-elements"/>
             force-single-root          : <xsl:value-of select="$force-
     single-root"/>
             default prefixes           : <xsl:value-of select="$nsprefix"/>
               you can change them with entity 'nsprefix'

         </documentation>
         <documentation>
          Schema documentation:<xsl:value-of
     select="xs:schema/xs:annotation/xs:documentation"/>
         </documentation>
         <contributors>
          <xsl:choose>
           <xsl:when test="$custom-type/c:contributors">
            <xsl:value-of select="$custom-type/c:contributors"/>
           </xsl:when>
           <xsl:otherwise>
            <xsl:text>None</xsl:text>
           </xsl:otherwise>
          </xsl:choose>
         </contributors>
        </annotation>
        <xsl:call-template name='crlf'/>

        <import type="def"/>
        <xsl:call-template name='crlf'/>

        <region name="element.start.name"
     parent="xml:element.defined.start.name"/>
        <region name="element.end.name"
     parent="xml:element.defined.end.name"/>
        <region name="element.start.lt"   parent="xml:element.start.lt"/>
        <region name="element.start.gt"   parent="xml:element.start.gt"/>
        <region name="element.end.lt"     parent="xml:element.end.lt"/>
        <region name="element.end.gt"     parent="xml:element.end.gt"/>
        <region name="element.nsprefix"   parent="element.start.name"/>
        <region name="element.nscolon"    parent="xml:element.nscolon"/>
```

```
   <region name="Attribute.name"
parent="xml:Attribute.defined.name"/>
   <region name="Attribute.nsprefix" parent="xml:Attribute.nsprefix"/>

   <region name="AttValue"           parent="xml:AttValue.defined"/>
   <region name="AttValue.start"
parent="xml:AttValue.defined.start"/>
   <region name="AttValue.end"       parent="xml:AttValue.defined.end"/>

   <region name="Enumeration"        parent="xml:Enumeration"
description="Enumerated type values"/>

   <xsl:for-each select='$custom-type/c:outline/c:element[@name]'>
     <region name="{@name}Outlined" parent="def:Outlined"
description="{@description}"/>
   </xsl:for-each>
   <xsl:call-template name='crlf'/>
   <entity name="ns-real-prefix" value="{$ns-real-prefix}"/>
   <entity name="nsprefix" value="{$nsprefix}"/>
   <xsl:call-template name='crlf'/>

   <!-- These schemes were cloned from xml.hrc
   and allow to reassign new regions to these constructs
   Normally, we should provide mechanism to virtualize region (like
schemes)
   -->
   <scheme name="{$anonymous}elementContent">
     <block start="/~( (&lt;) (  ((%xml:NCName;) (:) )? (%xml:Name;) ) \M
&gt;?   )/x"
            end="/( (&lt;\/) (\y3\b)?= ( (%xml:NCName;) (:) )?
(%xml:Name;) \b \M \s* (&gt;?)
             | (\/ \M &gt;) )/x"
            region01="PairStart" region02="element.start.lt"
            region05="element.nsprefix" region06="element.nscolon"
region07="element.start.name"
            region11="PairEnd" region12="element.end.lt"
region15="element.nsprefix" region16="element.nscolon"
            region17="element.end.name" region18="element.end.gt"
region19="element.start.gt"
            scheme="xml:elementContent2"/>
     <inherit scheme="xml:badChar"/>
   </scheme>
   <scheme name="{$anonymous}AttValue">
     <block start="/(&quot;)/" end="/(\y1)/"
            region00="PairStart" region10="PairEnd"
            region01="AttValue.start" region11="AttValue.end"
            scheme="xml:AttValue.content.quot" region="AttValue"/>
     <block start="/(&apos;)/" end="/(\y1)/"
            region00="PairStart" region10="PairEnd"
            region01="AttValue.start" region11="AttValue.end"
            scheme="xml:AttValue.content.apos" region="AttValue"/>
   </scheme>
   <!-- this one is simple internal service scheme -->
   <scheme name="AttributeContent">
     <inherit scheme="xml:AttributeContent">
       <virtual scheme="xml:AttValue" subst-
scheme="{$anonymous}AttValue"/>
     </inherit>
   </scheme>

   <xsl:copy-of select="$custom-type-schemes"/>

   <!-- Schema datatypes: -->

   <xsl:apply-templates mode="root"/>
   <xsl:apply-templates mode="typedefs"/>

   <xsl:call-template name='crlf'/><xsl:call-template name='crlf'/>

   <scheme name="{$hrctype}-root" access="public">
     <xsl:choose>
```

```
            <xsl:when test="$custom-type/c:top-level">
              <xsl:for-each select="$custom-type/c:top-level/*">
                <xsl:if test="self::c:element">
                  <inherit scheme="{.}-element"/>
                </xsl:if>
                <xsl:if test="self::c:group">
                  <inherit scheme="{.}-group"/>
                </xsl:if>
              </xsl:for-each>
            </xsl:when>
            <xsl:when test="$top-level-element">
              <inherit scheme="{$top-level-element}-element"/>
            </xsl:when>
            <xsl:otherwise>
              <annotation><documentation>
               You can replace these elements with needed single root element
               with customizing HRC generation process.
              </documentation></annotation>
              <xsl:for-each select="/xs:schema/xs:element">
                <inherit scheme="{@name}-element"/>
              </xsl:for-each>
            </xsl:otherwise>
          </xsl:choose>
          <xsl:if test="$allow-unknown-root-elements = 'yes'">
            <inherit scheme="xml:element">
              <virtual scheme="xml:element" subst-scheme="{$hrctype}-root"/>
            </inherit>
          </xsl:if>
        </scheme>

        <scheme name="{$hrctype}" access="public">
          <xsl:choose>
            <xsl:when test="$force-single-root = 'yes'">
              <inherit scheme="xml:singleroot">
                <virtual scheme="xml:element" subst-scheme="{$hrctype}-root"/>
              </inherit>
            </xsl:when>
            <xsl:otherwise>
              <inherit scheme="xml:xml">
                <virtual scheme="xml:element" subst-scheme="{$hrctype}-root"/>
              </inherit>
            </xsl:otherwise>
          </xsl:choose>
        </scheme>

      </type>
    </hrc>
    <xsl:call-template name='crlf'/>

  </xsl:template>

</xsl:stylesheet>
```

10. Приложение C. Примеры работы библиотеки

XML Schema:

```
<schema targetNamespace="http://www.altova.com/IPO"
            xmlns="http://www.w3.org/2001/XMLSchema"
            xmlns:ipo="http://www.altova.com/IPO">
  <annotation>
    <documentation>
     International Purchase order schema for Example.com
     Copyright 2000 Example.com. All rights reserved.
    </documentation>
    No text here!
    <no-such-element in="schema"/>
  </annotation>
  <!-- include address constructs -->
  <include schemaLocation="address.xsd"/>
  <!-- xml validation: -----  -->

  <element name="purchaseOrder" type="ipo:PurchaseOrderType"/>
  <element name="comment" type="string"/>
  <complexType name="PurchaseOrderType">
    <sequence>
      <element name="shipTo" type="ipo:Address"/>
      <element name="billTo" type="ipo:Address"/>
      <element ref="ipo:comment" minOccurs="0"/>
      <element name="Items" type="ipo:Items"/>
    </sequence>
    <!-- can't use it here -->
    <element name="billTo" type="ipo:Address"/>
    <attribute name="orderDate" type="date"/>
  </complexType>
</schema>
```

PHP и HTML:

```
<form action='demos.php' method="get">
  <h3>Demo File:</h3>
  <select name="filename">
<?
exec("ls -1 demos", $list);
for($i = 0; $i < count($list); $i++){
  $fname = $list[$i];
  $cur = ($fname == $filename)?"selected":"";
  ?><option <?=$cur?> value='<?=$fname?>'><?=$fname?></option><?
};
?>
  </select>
  <h3>Coloring style:</h3>
</form>
```

XSLT преобразование:

```
<xsl:template match="xs:complexType[@name and
            xs:*[1][self::xs:annotation]]" mode="xsdoc">
 <xsl:variable name='ann' select='xs:*[1][self::xs:annotation]'/>

 <anchor id='type{@name}'/>
 <para role='xsdocwrap'>
  <para role='xsdocdecl'>Element Name:
   <literal><xsl:value-of select='@name'/></literal>
   <xsl:if test='@name'>, type:
   <link linkend='xsid{@name}'><literal><xsl:value-of
            select='@name'/></literal></link>
   </xsl:if>
  </para>
  <para role='xsdoc'>
    <xsl:value-of select='normalize-space($ann/xs:documentation)'/>
  </para>
  <xsl:apply-templates select='.//xs:attribute[@name]' mode='xsdoc'/>
  <xsl:if test='.//xs:element[@name]'>
    <para role='xsdocdecl'>Content:</para>
    <xsl:apply-templates select='.//xs:element[@name]' mode='xsdoc'/>
  </xsl:if>
 </para>
</xsl:template>

<xsl:template match="xs:element[@name and not(ancestor::xs:element)]"
            mode="xsdoc">
  <xsl:variable name='ann'>
    <xsl:variable name='anni'
          select='//xs:complexType[@name = current()/@type] /
                  xs:*[1][self::xs:annotation]/xs:documentation'/>
    <xsl:choose>
     <xsl:when test='xs:*[1][self::xs:annotation]'>
       <xsl:value-of
            select='xs:*[1][self::xs:annotation]/xs:documentation'/>
     </xsl:when>
     <xsl:when test='$anni'>
       <xsl:value-of select='$anni'/>
     </xsl:when>
    </xsl:choose>
  </xsl:variable>

  <para role='xsdochead'>Element:
   <literal><xsl:value-of select='@name'/></literal>
   <xsl:choose>
    <xsl:when test='@type'>, type:
    <link linkend='type{@type}'><literal><xsl:value-of
            select='@type'/></literal></link>
    </xsl:when>
    <xsl:when test='@name'>, type:
    <link linkend='xsid{@name}'><literal><xsl:value-of
            select='@name'/></literal></link>
    </xsl:when>
   </xsl:choose>
  </para>
  <para role='xsdoc'>
    <xsl:value-of select='normalize-space($ann)'/>
  </para>
</xsl:template>
```

XML на японском языке:

```
<?xml version="1.0"?>
<!DOCTYPE 予定項目リスト SYSTEM "weekly-utf-16.dtd">
<予定項目リスト>
  <見積もり工数>120</見積もり工数>
  <実績工数>6</実績工数>
  <当月見積もり工数>32</当月見積もり工数>
  <当月実績工数>2</当月実績工数>
  <予定項目 xmlns='当月実績工数'>
    <P><A href="http://www.goo.ne.jp">の機能を調べてみる</A></P>
  </予定項目>
</予定項目リスト>
```

Unicode и XHTML:

```
<?xml encoding="UTF-8"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
    "xhtml/DTD/strict.dtd">
<html>
  <head>
    <meta http-equiv="Content-Type" content="text/html;" />
    <title>When the world wants to talk, it speaks Unicode</title>
  </head>
  <body>
    <h1>When the world wants to talk, it speaks Unicode</h1>
    <p><b>Arabic: </b>عندما يريد العالم أن يتكلّم, هو يتحدّث بلغة يونيكود.</p>
    <p><b>Catalan: </b>Quan el món vol conversar, parla Unicode</p>
    <p><b>Chinese: </b>当世界需要沟通时，请用Unicode！</p>
    <p><b>Danish: </b>Når verden vil tale, taler den Unicode</p>
    <p><b>Dutch: </b>Als de wereld wil praten, spreekt hij Unicode</p>
    <p><b>English: </b>When the world wants to talk, it speaks Unicode</p>
    <p><b>Esperanto: </b>Kiam la mondo volas paroli, ĝi parolas Unicode</p>
    <p><b>Finnish: </b>Kun maailma haluaa puhua, se puhuu Unicodea</p>
    <p><b>French:</b>Quand le monde veut communiquer, il parle en Unicode</p>
    <p><b>German:</b>Wenn die Welt miteinander spricht, spricht sie Unicode</p>
    <p><b>Hebrew: </b>כאשר העולם רוצה לדבר, הוא מדבר ב-Unicode</p>
    <p><b>Hungarian: </b>Ha a világ beszélni akar, azt Unicode-ul mondja</p>
    <p><b>Italian: </b>Quando il mondo vuole comunicare, parla Unicode</p>
    <p><b>Japanese: </b>世界的に話すなら、Unicode です。</p>
    <p><b>Korean: </b>세계를 향한 대화, 유니코드로 하십시오</p>
    <p><b>Norwegian: </b>Når verden vil snakke, snakker den Unicode</p>
    <p><b>Portugese: </b>Quando o mundo quer falar, fala Unicode</p>
    <p><b>Romanian: </b>Când lumea vrea să comunice, vorbeşte Unicode</p>
    <p><b>Russian: </b>Если мир хочет общаться, он общается на Unicode</p>
    <p><b>Slovenian: </b>Ko se želi svet pogovarjati, govori Unicode</p>
    <p><b>Spanish: </b>Cuando el mundo quiere conversar, habla Unicode</p>
    <p><b>Swedish: </b>När världen vill tala, så talar den Unicode</p>
  </body>
</html>
```

Среда разработки Eclipse: