# SDK Pullenti (2.58)

## Обзор

SDK Pullenti (Puller of Entities) предназначено для решения задачи анализа текста и выделения именованных сущностей из неструктурированных русскоязычных текстов в рамках информационных систем, разрабатываемых на .NET Framework 2.0 и выше.

SDK состоит из общей и специализированной части. Общая часть содержит реализацию общих алгоритмов морфологического и синтаксического анализа, а также поддержку модели данных. Специализированная часть состоит из отдельных сборок (анализаторов), реализующих выделение сущностей определённых типов (персоны, организации и др.). Такие сборки подключаются динамически как плагины, благодаря чему система может расширять свой функционал без перекомпилирования. Например, семантический анализ реализован в виде такой отдельной сборки.

Именованная сущность – это объект, содержащий набор значений атрибутов, отличающий его от других объектов этого же типа. Конечно, это не строгое определение. Каждый анализатор сам решает, что считать сущностью, а что нет. Например, в фразе «Вася пошёл гулять» персона не выделится, так как только по одному имени или по фамилии персона не выделяется во избежание потери точности. Хотя если в тексте персона с именем «Вася» в другом месте будет выделена (например, из фразы «Вася Иванов – хороший мальчик»), то и в других местах она будет выделяться по одному имени (если, конечно, в тексте нет других «Вась»).

## Подключение к проектам .NET

Все общие сборки и некоторые специализированные сборки включены в проект EP.Sdk. Для использования во внешнем решении (Solution в MS Visual Studio) нужно добавить в решение этот проект. В конечном проекте (EXE или ASP) нужно установить ссылки на все необходимые DLL, чтобы они попали в исполняемую директорию после компиляции.  Демонстрационный пример решения DemoNer.sln показывает, как это делается.

Все внешние классы и определения находятся в пространстве имён EP, некоторые служебные классы – в EP.Semantix.

Внимание! Если у Вас версия MS Visual Studio раньше 2010, то могут возникнуть проблемы с обратной совместимостью, и студия не загрузит проект EP.Sdk.csproj. В этом случае просто добавьте в свой проект все dll из EP.Sdk.csproj.

## Интеграция с другими языками и на других ОС

Для запуска .NET приложений на не-Windows операционных системах (Linux, Mac, Android) можно использовать открытый проект Mono (http://www.mono-project.com/), который является аналогом java-машины для запусков приложений на Java.

Одним из вариантов интеграции является оформление обработчика как исполняемого модуля .NET, работающего в пакетном режиме и\или реализующий TCP-сервер, взаимодействующий с внешней системой по TCP\IP. Такой модуль является кросплатформенным благодаря Mono.

Пример реализации такого модуля есть в проекте EP.DemoServer из DemoNer.sln. Модуль работает в 2-х режимах и управляется аргументами командной строки.

Если задать ключи `–input ИМЯ_ВХОДНОГО_ФАЙЛА –output ИМЯ_ВЫХОДНОГО_XML`, то это пакетный режим – после обработки файла (текст должен быть в кодировке UTF-8) модуль завершает работу. Неудобство здесь состоит в том, что на запуск и инициализацию тратится несколько секунд, что неэффективно при обработке большого числа небольших файлов, но эффективно при обработке больших файлов.

Если не задавать ключ –input, то модуль работает в серверном режиме, слушая порт, заданный параметром –port командной строки (по умолчанию, 1234). При поступлении по этому порту текста в кодировке UTF-8 модуль его обрабатывает и возвращает XML с результатом. Многопоточность поддерживается. Завершает работу модуль при нажатии в консоли любой клавиши. В качестве демонстрации обращения к такому серверу можно использовать проект EP.DemoClient, ну а на других языках обращение будет аналогичным:

```
WebClient web = new WebClient();
// кодируем текст
byte[] dat = Encoding.UTF8.GetBytes(text);
// запрос серверу
byte[] res = web.UploadData("http://127.0.0.1:1234", dat);
```

## Инициализация для EXE

Перед использованием функционала в EXE-модуле желательно произвести инициализацию. Она занимает несколько секунд, во время которых происходит распаковка словарей, в основном морфологических. Также в ходе инициализации составляется список доступных DLL из основной директории[1] и происходит инициализация их специфических данных.

```
EP.ProcessorService.Initialize();
```

В демонстрационном проекте инициализация делается в Program.cs.

[1] Иногда при .NET4 и 64бит этот список не составляется, анализаторы отсутствуют и сущности не выделяются. В этом случае используйте надёжную Web-инициализацию, описанную далее.

Если инициализацию не сделать, то при первой обработке текста она будет произведена автоматически, поэтому обработка этого текста займёт больше времени, чем остальных.

Внимание! Морфологический словарь для каждого языка занимает в памяти от 100 до 150Мб. По умолчанию, загружаются словари для русского и английского языка. Если требуется загружать другие словари, то их список нужно указать вторым параметром Initialize, например:

```
EP.ProcessorService.Initialize(true, MorphLang.RU | MorphLang.EN | MorphLang.UA);
```

Также можно впоследствии выгружать ненужные словари и загружать нужные через функции морфологического сервиса, например:

```
EP.Text.Morphology.UnloadLanguages(MorphLang.UA);
EP.Text.Morphology.LoadLanguages(MorphLang.BY);
// Morphology.LoadedLanguages – указывает на список загруженных языков
```

## Инициализация для WEB

Для Web-проектов необходим другой метод инициализации, так как стандартная плагинная техника здесь не работает (все DLL копируются и запускаются из временной директории, а не из Bin). Здесь приходится явно инициализировать специфические DLL, которые планируется использовать, например, в Global.cs:

```
// инициализируем морфологию без загрузок DLL
// (можно указать список языков вторым параметром)
EP.ProcessorService.Initialize(false);

(new EP.Semantix.MiscInitializer()).Initialize();
(new EP.Semantix.DateInitializer()).Initialize();
(new EP.Semantix.LocationInitializer()).Initialize();
(new EP.Semantix.OrgInitializer()).Initialize();
(new EP.Semantix.PersonInitializer()).Initialize();
(new EP.Semantix.TechnicalInitializer()).Initialize();
(new EP.Semantix.DecreeInitializer()).Initialize();
(new EP.Semantix.BiblioInitializer()).Initialize();
(new EP.Semantix.BusinessInitializer()).Initialize();
(new EP.Semantix.SemanticInitializer()).Initialize();
```

Пример Web-проекта есть в DemoNer.sln – это DemoWeb.

## Использование

Использование функциональности в коде: создать экземпляр лингвистического процессора (Processor), вызвать его функцию Process() на анализируемом тексте (SourceOfAnalysis) – результат (AnalysisResult) содержит список выделенных сущностей Entities. Обзор основных элементов SDK представлен на диаграмме 3 (все диаграммы находятся в проекте EP.Sdk.csproj в папке Doc).

Пример использования в коде:

```
using EP;

// создаём экземпляр процессора
Processor processor = new Processor();

// запускаем на тексте text
AnalysisResult result = processor.Process(new SourceOfAnalysis(text));

// получили выделенные сущности
foreach (Referent entity in result.Entities)
       Console.WriteLine(entity.ToString());
```

Вот чуть более сложный пример, когда из текста нужно выделить все существительные и нормализовать их:

```
// перебираем токены
for (Token t = result.FirstToken; t != null; t = t.Next)
{
    // нетекстовые токены игнорируем
    if (!(t is TextToken)) continue;
    // несуществительные игнорируем
    if (!t.Morph.Class.IsNoun) continue;
    // получаем нормализованное значение
    string norm = t.GetNormalCaseText(MorphClass.Noun, true);
    Console.Write("\r\nNoun on position {0}: {1} ", t.BeginChar, norm);
}
```

## Модель данных для сущностей

Базовым классом для сущностей является класс Referent (терминология частично взята из UIMA – см. http://uima.apache.org). Тип сущностей задаётся классом ReferentClass, который содержит набор атрибутов (Feature). Значение атрибута в сущности называется слотом, то есть слот – это пара «атрибут, значение». Значение может быть как простым (строка, число), так и ссылкой на другую сущность. Обзор классов для сущностей представлен на диаграмме.

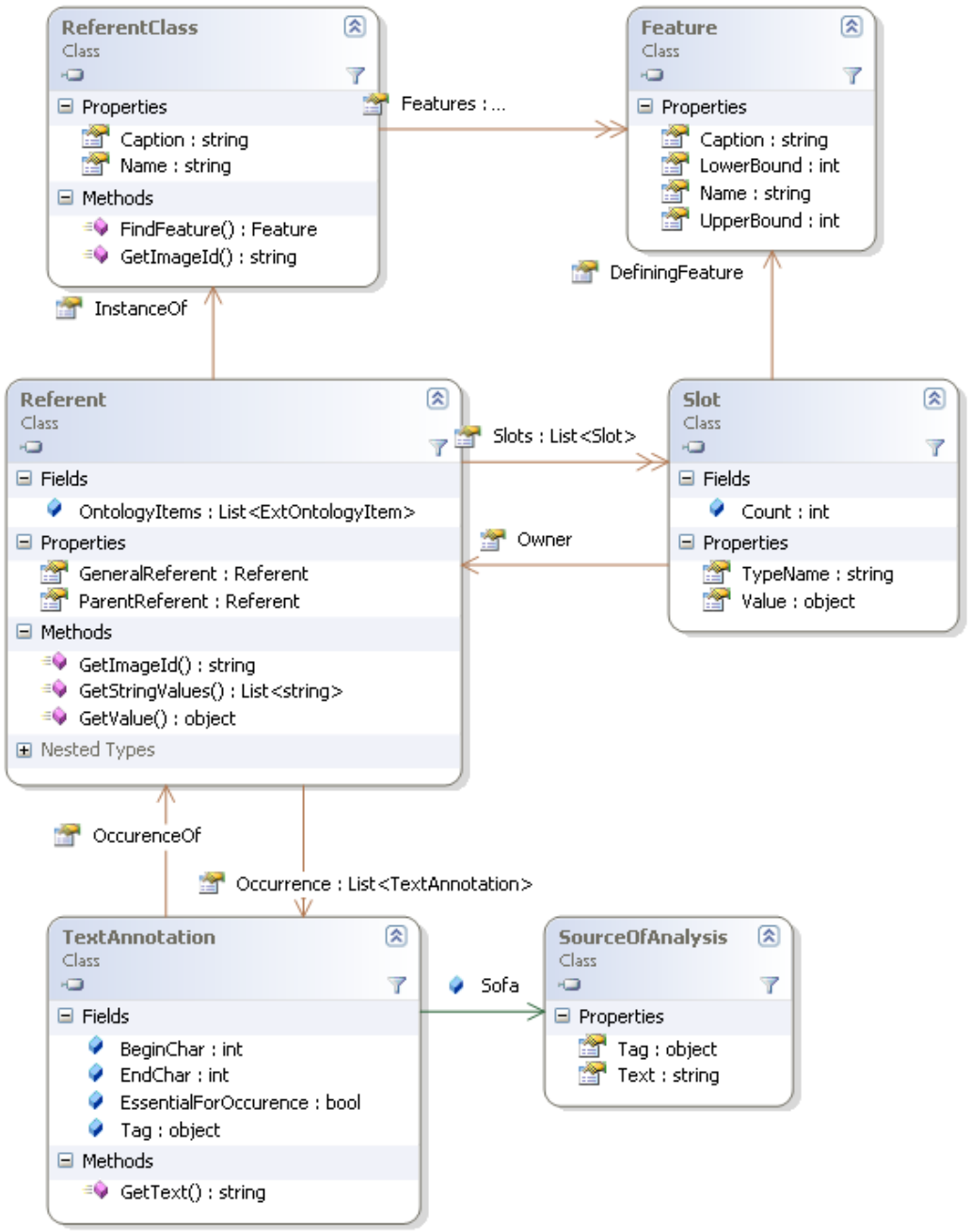



*Рис.1. Базовые классы модели данных*

Помимо значений атрибутов, сущность содержит список ссылок на участки (TextAnnotation) исходного текста, в которых эта сущность располагается. Исходный текст представляется классом SourceOfAnalysis, ссылки на него сокращённо называются Sofa (см. UIMA).

В принципе, этой структуры достаточно для работы с сущностями. Но для облегчения дальнейшего анализа каждый тип сущностей оформляется своим классом, наследным от Referent. Такой класс содержит специфические свойства, которые просто оборачивают обращения к значениям соответствующих слотов. Например, сущность DateReferent имеет свойство int Year { get; set; } – это работа с значением слота, ссылающегося на атрибут с именем YEAR. Итак, значения атрибутов всегда хранятся в слотах, доступ к которым происходит на уровне Referent, а специфические классы сущностей просто обрамляют работу со слотами.

На диаграмме показана иерархия некоторых типов сущностей.  Отметим, что множество сущностей и их атрибутов постоянно расширяется, поэтому актуальную спецификацию сущностей следует искать во встроенной xml-документации.

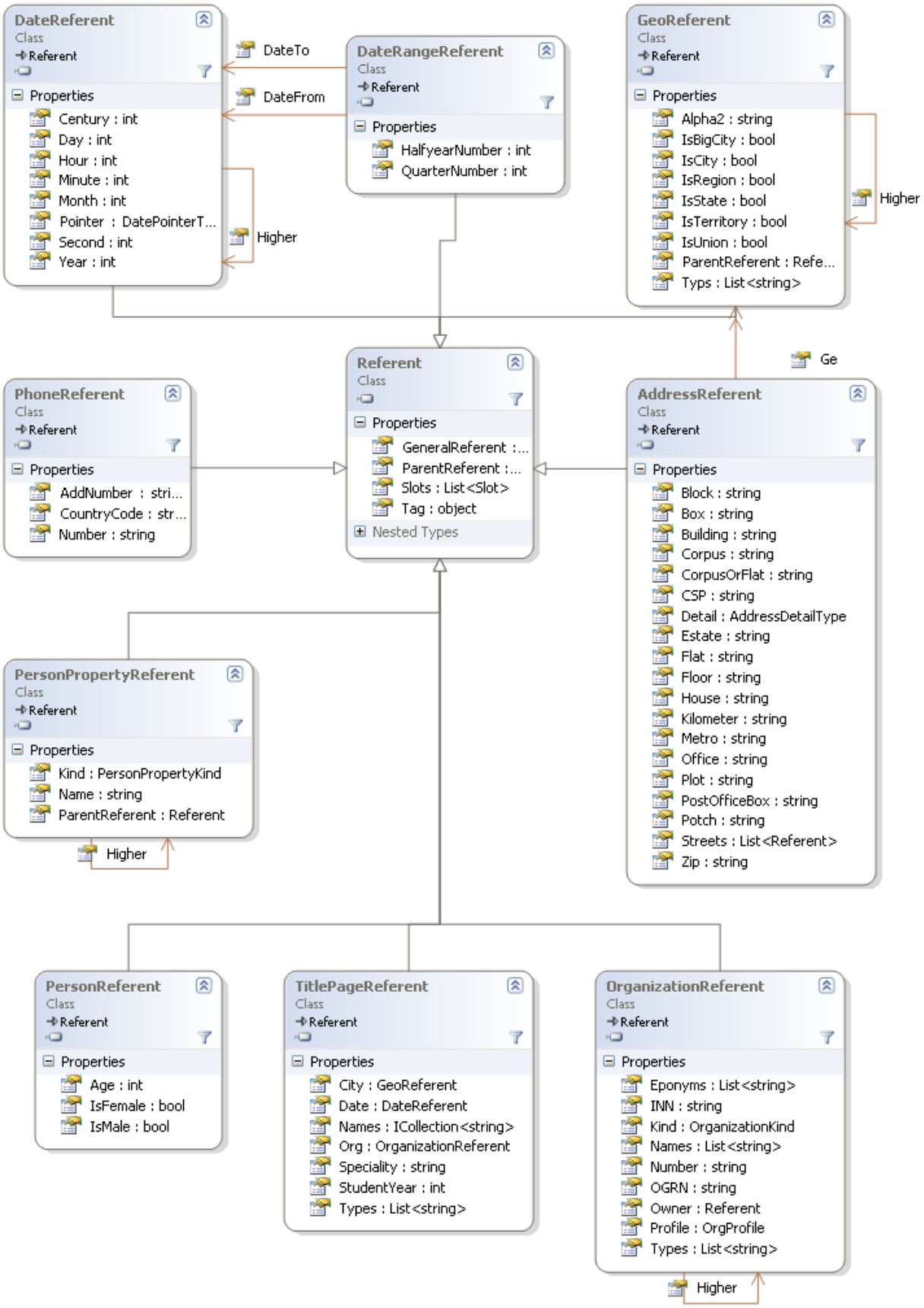



*Рис.2. Классы некоторых сущностей*

## Лингвистический процессор

Лингвистический процессор представлен классом Processor. Он инкапсулирует в себе общие алгоритмы анализа – морфологического и синтаксического. Выделение конкретных типов сущностей происходит в анализаторах, базовым классом которых является Analyzer. Анализатор, в частности, содержит список (TypeSystem) классов сущностей ReferentClass, которые умеет выделять в тексте.  Последовательность обработки процессором определяется массивом Analyzers.

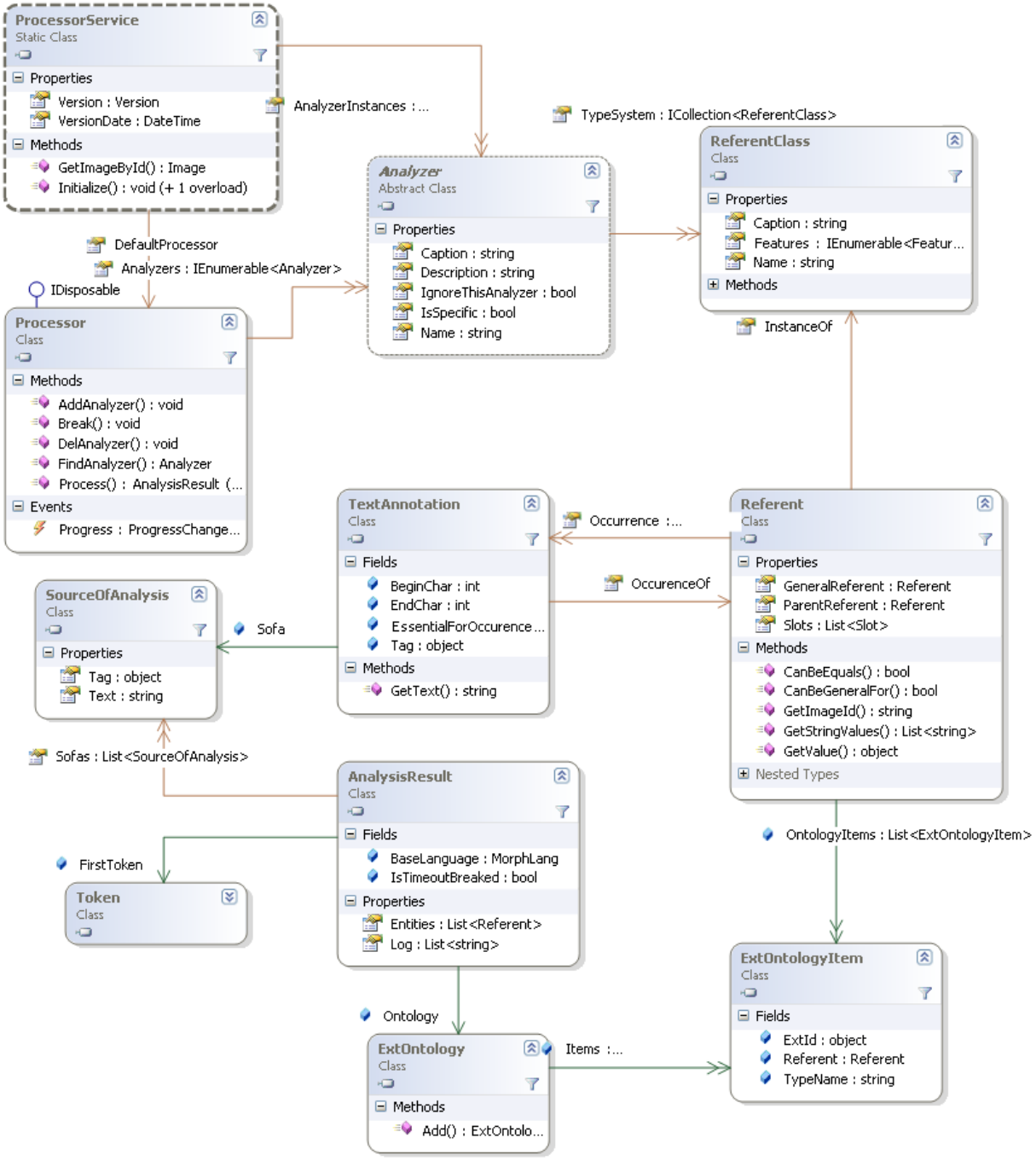



*Рис.3. Основные классы SDK Pullenti*

Статический класс ProcessorService содержит список найденных анализаторов AnalyzerInstances. Поиск производится в сборках директории, в которой располагается базовая сборка EP.Core.dll, и этот список составляется динамически при первом обращении к классу. Если этот способ по какой-либо причине неудобен, то можно регистрировать типы анализаторов вручную функцией RegisterAnalyzerType (например, при использовании SDK на стороне Web-сервера плагинный механизм не работает).

Анализаторы подразделяются на общие и специфические, что задаётся свойством IsSpecific. Общие анализаторы всегда вставляются по умолчанию в последовательность обработки. Включение специфического анализатора нужно осуществлять явно. Это сделано потому, что специфические анализаторы рассчитываются на тексты определённой тематики и структуры. Например, включённый в демо-пример анализатор заголовочной информации (TITLEPAGE).

Чтобы выделить сущности из текста, сначала необходимо создать экземпляр процессора.

При вызове конструктора без параметров в список анализаторов попадают все доступные общие анализаторы:

```
// создаём экземпляр процессора
Processor processor = new Processor();
// последовательность обработки
foreach (Analyzer a in processor.Analyzers)
       Console.WriteLine(a.ToString());
```

Если в дополнение к ним нужно включить специфический анализатор, то его имя нужно указать в конструкторе:

```
// включение специфического анализатора для анализа заголовочной информации
Processor processor = new Processor("TITLEPAGE");
```

При необходимости можно исключить какой-либо анализатор из списка Analyzers стандартным удалением элемента, или установить у него свойство IgnoreThisAnalyzer = true.

Ход анализа можно отслеживать через событие Progress системного типа:

```
// подписываемся на событие "бегунка"
processor.Progress += new ProgressChangedEventHandler(processor_Progress);

// пример обработки бегунка
void processor_Progress(object sender, ProgressChangedEventArgs e)
{
       if (e.ProgressPercentage >= 0)
           toolStripProgressBar1.Value = e.ProgressPercentage;
       else
       {
           // если < 0, то это просто информационное сообщение
       }

       if (e.UserState != null)
       {
           toolStripLabelMessage.Text = e.UserState.ToString();
           toolStrip1.Update();
       }
```

```
}
```

Сам анализ производится функцией Process, которой на вход нужно подать исходный текст, заданный классом SourceOfAnalysis:

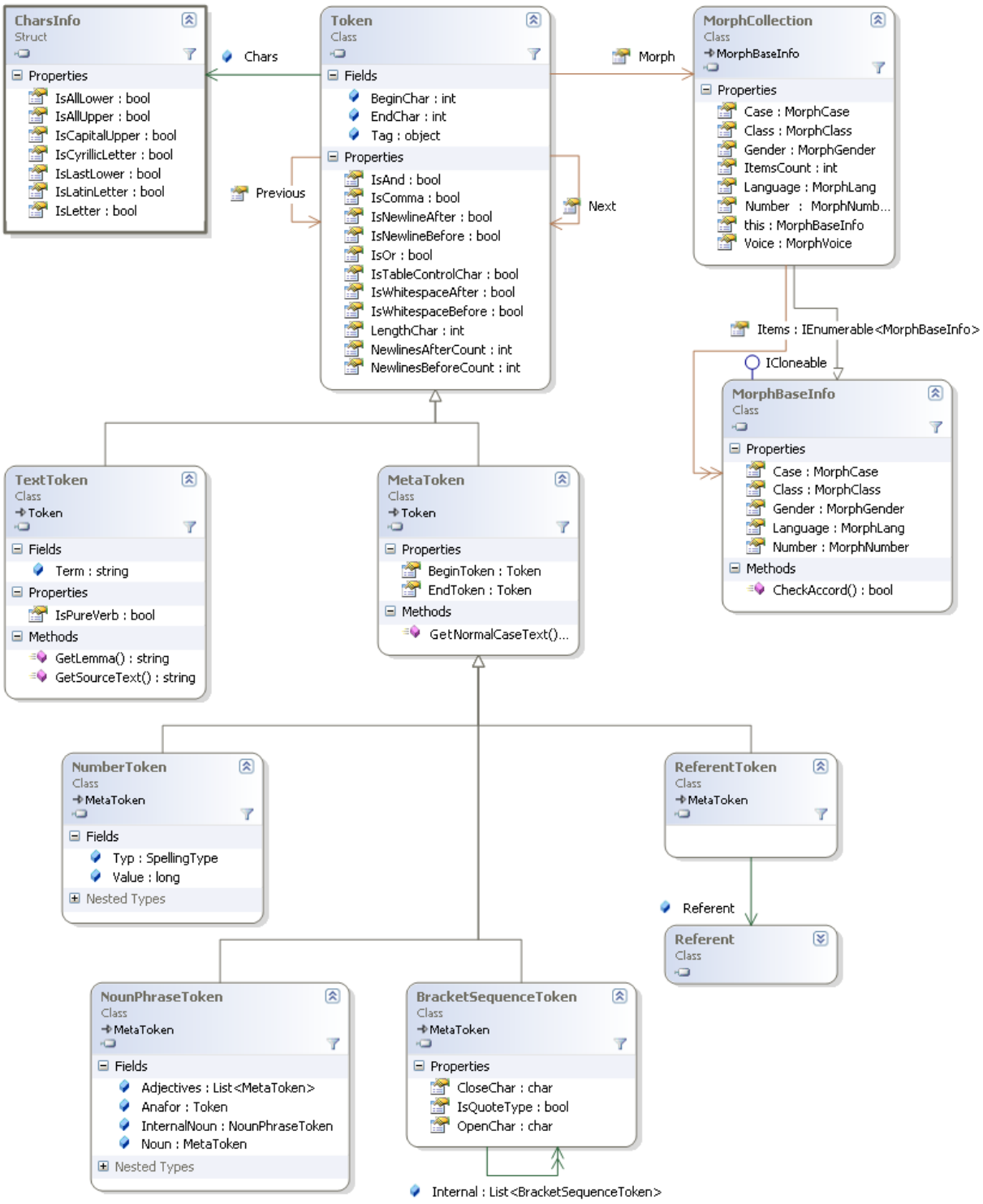
```
string text = "...исходный анализируемый текст...";
// обёртка текста
SourceOfAnalysis sofa = new SourceOfAnalysis(text);
// запускаем анализ
AnalysisResult result = processor.Process(sofa);

// получили выделенные сущности
foreach (Referent entity in result.Entities)
       Console.WriteLine(entity.ToString());
```

Результат содержит список (Entities) сущностей, а также лог с кратким протоколом обработки, и ссылку на первый токен FirstToken (см. далее). Данный токен играет исключительное значение в деле дальнейшего анализа текста, если в этом есть необходимость.

## Токены

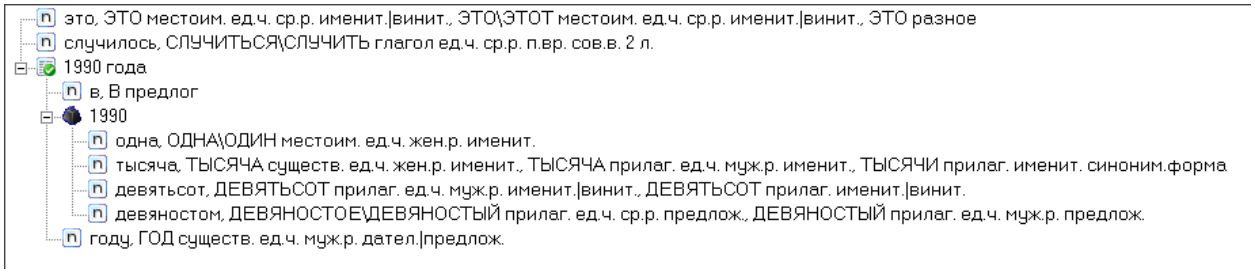
При анализе исходный текст разбивается на токены в виде двунаправленного списка. В дальнейшем токены объединяются в «метатокены», представляющие более крупные конструкции, в частности, сущности. Результирующий класс AnalysisResult содержит ссылку на первый токен FirstToken, через который можно получить и все остальные токены. На диаграмме ниже представлена информация об основных токенах.



Базовым классом является класс Token. Свойства Previous и Next ссылаются на соседние токены, BeginChar и EndChar – позиции токена в символах в исходном тексте, CharsInfo – информация о символах токена (IsLetter – символы, IsAllLower – все символы в нижнем регистре и т.д., см. встроенный help), Morph - морфологическая информация (см. отдельный раздел).

Основными наследными классами Token являются TextToken и MetaToken.
TextToken – это чистый исходный фрагмент текста, содержащий результат морфологического анализа. Ссылается на MorphToken, содержащий все морфологические варианты разбора.
MetaToken – это токен, заменяющий диапазон токенов. Он как бы накрывает сверху другие токены, указывая свойствами BeginToken и EndToken на первый и последний из заменяемых токенов.
К классу метатокенов относятся NumberToken, представляющий число, ReferentToken, представляющий сущность, а также множество других элементов, используемых при анализе (некоторые из них описываются ниже).

Пусть, например, исходный текст такой: «это случилось в одна тысяча девятьсот девяностом году».




Результирующая последовательность токенов будет состоять из 3-х: первые два типа TextToken, и последняя ReferentToken, ссылающаяся на сущность (дату). В свою очередь ReferentToken покрывает 3 токена – текстовой («В»), NumberToken (1990) и текстовой («году»), а NumberToken под собой имеет 4 исходных текстовых токена, интерпретированных как одно число.

В процессе решения задач обработки текстов сложились хелперы и специфические метатокены, обрабатывающие и представляющие те или иные лингвистические аспекты. К таким аспектам относятся работа с числовыми значениями, именные группы (существительные с возможными прилагательными, согласованными по морфологии), скобки и кавычки, концы предложений и пр. Для решения многих задач может оказаться достаточным набор возможностей. Отметим, здесь идут постоянные улучшения и доработки, поэтому актуальное состояние функций следует брать из встроенной xml-документации.  Опишем здесь некоторые.

## Числовые токены

Числовые токены представлены классами NumberToken (для целых чисел) и NumberExToken (для дробных чисел с последующими мерами массы, площади, длины и пр.).

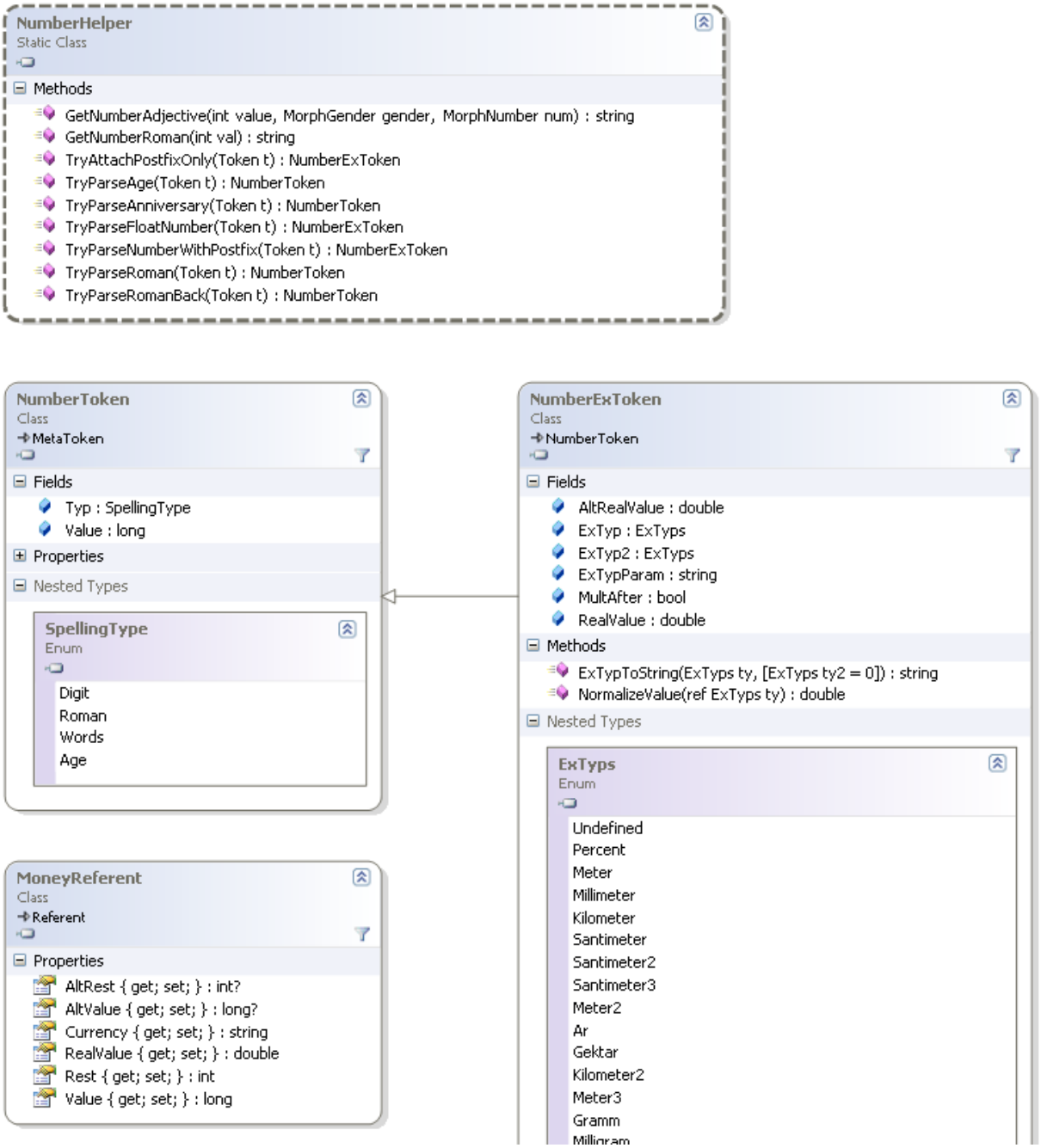
NumberToken имеет целочисленное значение и тип. Типы Digit (явное число арабскими цифрами) и Words (прописью, например, «сто двадцать пять», «1-й») выделяются автоматически и встраиваются в последовательности токенов на начальном этапе обработки. Всё остальное нужно выделять явно через функции NumberHelper.TryParse… Почему так? Да потому, что выделение или невыделение, скажем, римских цифр зависит от контекста. Например, в тексте присутствует «I» (английское ай). Это может быть и инициал, и римская единица, и английское местоимение, и было бы неразумно сразу принимать какое-либо решение. Это делается (или не делается) потом, на соответствующем этапе анализа. Например, анализатор «Персоны» если после I идёт точка, то это инициал, если перед I стоит имя, то это римская цифра (Пётр I), и т.д.

Рассмотрим пример, когда в тексте нужно выделять номера сессий (... на XXI-й сессии …).

```
// перебираем токены
for (Token t = result.FirstToken; t != null; t = t.Next)
{
	// может, номер задан явно цифрами или прописью
	NumberToken num = t as NumberToken;
	Token t1 = null; // ссылка на слово "сессия"
	if (num != null) t1 = t.Next;
	else
	{
		// пробуем выделить римское число
		num = NumberHelper.TryParseRoman(t);
		if (num != null)
		{
	// поскольку токен num не встроен в общую цепочку, а BeginToken\EndToken
	// указывают на первый и последний токены цепочки, то следующий не num.Next,
	// а именно num.EndToken.Next
			t1 = num.EndToken.Next;
		}
	}
	if (t1 == null || num == null)
		continue;
	if (!t1.IsValue("СЕССИЯ"))
		continue;

	// нашли
	Console.Write("\r\nSession {0} on position {1}", num.Value, t.BeginChar);
	t = t1;
}
```

Можно поступить по-другому, сначала проверяя t.IsValue(“СЕССИЯ”), а затем пытаясь выделить римскую цифру в обратном порядке через TryParseRomanBack.




Функция TryParseAge выделяет конструкции типа «20-летний», TryParseAnniversary разные годовщины типа «XX-й годовщины», GetNumberAdjective позволяет преобразовать любое число в его словесное представление в нужном роде и числе (например, 34 для ж.р. => «ТРИДЦАТЬ ЧЕТВЕРТАЯ»).

У NumberExToken есть RealValue. AltRealValue представляет возможное второе значение, которое далее идёт в скобках. Например, 24 (двадцать пять) рублей.  ExTyp задаёт возможный постфикс, список которых определяется enum ExTyps. Постфиксы привязываются с учётом возможных способов написания. Например,  25 кв.м, 25 м2, 25м<2>, 25 кв.метров дадут один тип Meter2. ExTyp2 – это когда вторая мера задаётся через дробь. Например,  м/гр.  Функция NormalizeValue позволяет преобразовать значение к некоторому каноническому виду.  Скажем, все меры длины к сантиметрам или метрам, умножая RealValue на соответствующие значения.

Данные токены выделяются через функцию TryParseFloatNumber (без анализа постфикса) и TryParseNumberWithPostfix с полным анализом вместе с постфиксом.

Отметим, что для выделения денег специально введена сущность MoneyReferent, которая автоматически выделяется и встраивается в последовательность токенов. При выделении соответствующий анализатор использует функцию TryParseFloatNumber, и если получает ExTyps.Money, то оформляет экземпляр сущности.

Рассмотрим пример выделения всех денежных рублевых сумм, а также суммарную длину в метрах всех числовых характеристик, задающих меру длины.

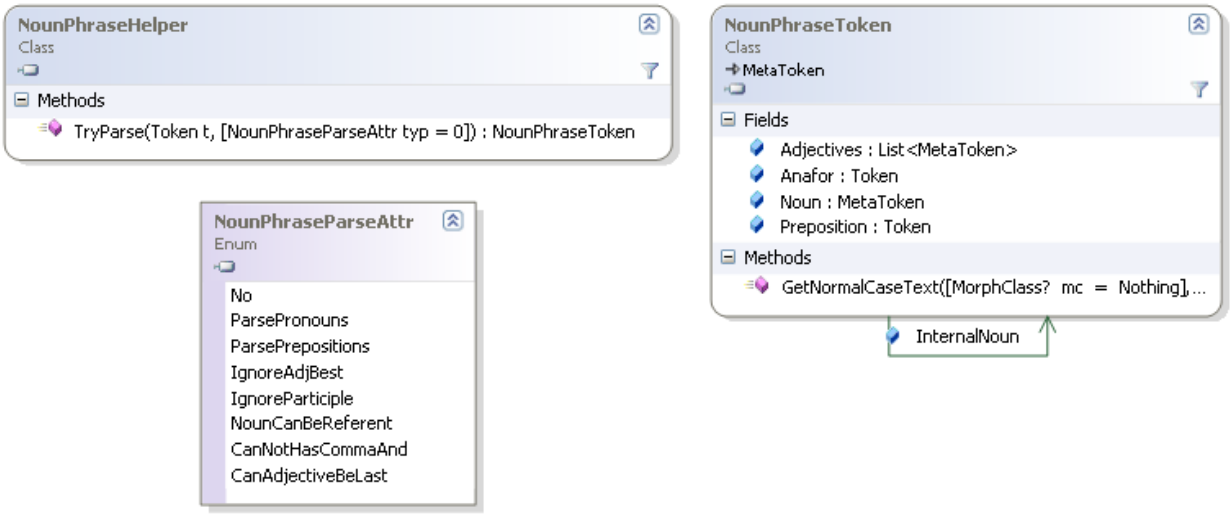
```
double sumLengthMeter = 0;
for (Token t = result.FirstToken; t != null; t = t.Next)
{
	// если это деньги, то они уже встроены
	MoneyReferent money = t.GetReferent() as MoneyReferent;
	if (money != null)
	{
		if(money.Currency == "RUB")
			Console.Write("\r\nRubles {0} on position {1}",
				money.ToString(), t.BeginChar);
		continue;
	}

	NumberExToken num = NumberHelper.TryParseNumberWithPostfix(t);
	if (num == null) continue;
	if ((num.ExTyp == NumberExToken.ExTyps.Kilometer ||
		num.ExTyp == NumberExToken.ExTyps.Meter ||
		num.ExTyp == NumberExToken.ExTyps.Santimeter ||
		num.ExTyp == NumberExToken.ExTyps.Millimeter) &&
	    num.ExTyp2 == NumberExToken.ExTyps.Undefined)
	{
		NumberExToken.ExTyps normTyp = NumberExToken.ExTyps.Meter;
		double normVal = num.NormalizeValue(ref normTyp);
		if (normTyp == NumberExToken.ExTyps.Meter)
			sumLengthMeter += normVal;
	}
	// обязательно нужно перемещаться на конец метатокена, чтобы не делались
	// привязки с середины. Например, пусть "-20м", если взять просто следующий, то
получим "20м" противоположное по знаку число.
	t = num.EndToken;
}
```

## Токены. Именные группы

Именная группа представляется классом NounPhraseToken, который содержит корень (Noun) и некоторое количество прилагательных (Adjective). Эти элементы сами являются метатокенами, так как могут состоять из нескольких текстовых токенов (например, Noun «девочка-подросток» или Adjective «сильно-действующий»). Может быть ссылка на текстовой токен, представляющий анафору, и на предлог («от её старшего брата» - Preposition: «от», Anafor: «её», Adjective: «старший», Noun: «брат»).

Может быть внутренняя именная группа InternalNoun (например, «по настоящим на данный момент представлениям» - базовая группа «НАСТОЯЩЕЕ ПРЕДСТАВЛЕНИЕ», а внутренняя «ДАННЫЙ МОМЕНТ».




Как и у любого токена, NounPhraseToken имеет свойство Morph с уточнённой морфологической информацией. Напомним, что для текстовых токенов морфология вычисляется без учёта контекста, и Morph.Items содержит все морфологические варианты (POS-Tagging). Для именной группы лишние несогласованные варианты отбрасываются, и оставляются только согласованные. Соответственно функция GetNormalCaseText возвращает также нормализованный вариант всей группы, а не отдельных слов. Например, для фрагмента «информационных систем» у каждого токена по отдельности эта функция вернёт «ИНФОРМАЦИОННЫЙ» и «СИСТЕМА», а для всей именной группы GetNormalCaseText(null, false) = «ИНФОРМАЦИОННЫЕ СИСТЕМЫ», GetNormalCaseText(null, true) = «ИНФОРМАЦИОННАЯ СИСТЕМА» (второй параметр говорит о необходимости приведения к единственному числу).

Перечислим атрибуты, которые как битовая маска могут задавать параметры выделения в TryParse:

- ParsePronouns – выделять ли местоимения и анафоры (по умолчанию, нет)
- ParsePrepositions – выделять ли в начале предлог, если он есть (по умолчанию, не выделять)
- IgnoreAdjBest – не выделять прилагательные в превосходной степени;

- IgnoreParticiple – игнорировать причастия, Adjective будут чистыми прилагательными;
- NounCanBeReferent – в качестве Noun может быть сущность (например, немытая Россия);
- CanAdjectiveBeLast – прилагательное может быть после существительного (например, конь педальный);
- CanNotHasCommaAnd – прилагательные не должны разделяться ни союзом «и», ни запятыми;

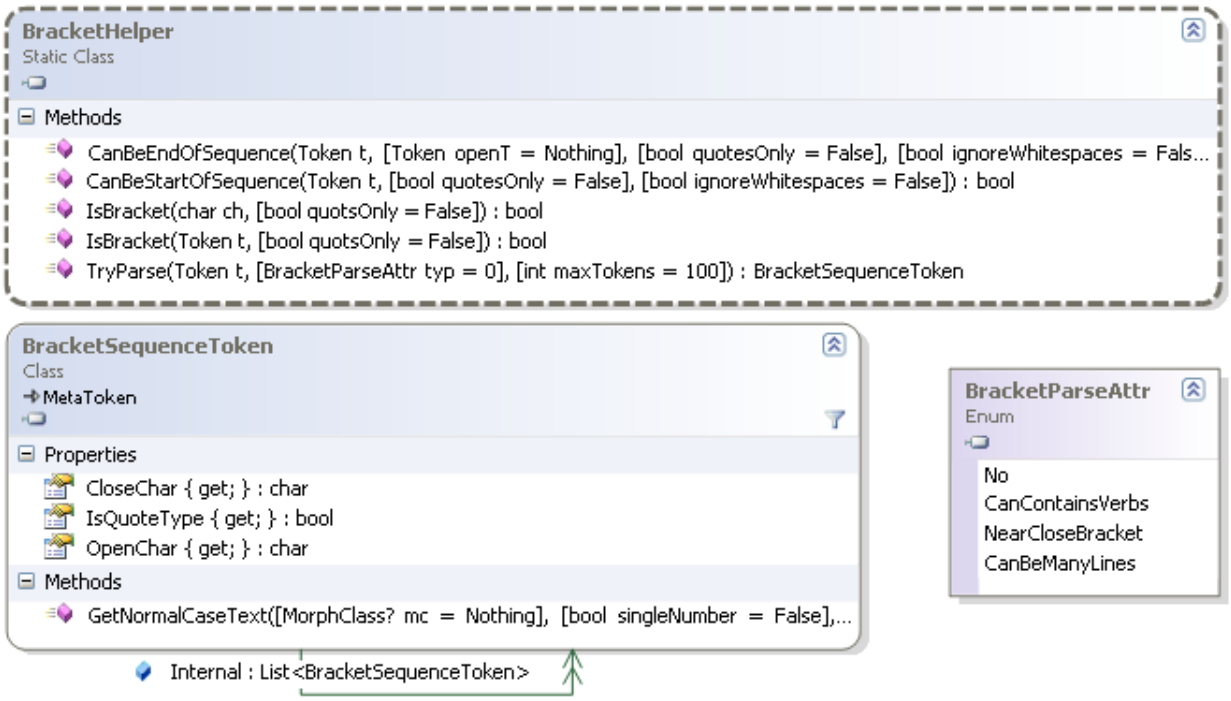
Пример выделения всех именных групп и составления словаря частоты их встречаемости.

```
Dictionary<string, int> stat = new Dictionary<string,int>();
for (Token t = result.FirstToken; t != null; t = t.Next)
{
	NounPhraseToken npt = NounPhraseHelper.TryParse(t);
	if (npt == null) continue;
	// нормализуем к единственному числу
	string normal = npt.GetNormalCaseText(null, true);
	if (stat.ContainsKey(normal)) stat.Add(normal, 1);
	else stat[normal]++;
	t = npt.EndToken;
}
```

Отметим, что если не перемещаться в конец метатокена t = npt.EndToken, то для текста «информационная система» сначала выделится «информационная система», а затем просто «система».

## Токены. Скобки и кавычки

Проблемы выделения последовательностей, обрамляемых кавычками и скобками, возникают, когда забывают ставить закрывающие скобки или несколько закрывающих скобок сливаются в одну при вложенных друг в друга последовательностях. Поскольку кавычки часто используются для задания имён и наименований, то работа с ними выделена в отдельный хелпер.




Хэлпер работает со скобками и кавычками в различных их представлениях. Последовательность BracketSequenceToken может иметь список Internal внутренних последовательностей. Например, ОАО «Компания «Пупкиных» - одна последовательность «Компания Пупкиных» имеет вложенную подпоследовательность «Пупкиных».

Функция GetNormalCaseText возвращает внутренний текст, при этом первая именная группа приводится к именительному падежу (и единственному числу, если 2-й параметр true). Например, для текста:  описание «Турбинных двигателей, вращающих …» нормализация даст «ТУРБИННЫЕ ДВИГАТЕЛИ, ВРАЩАЮЩИЕ …» или для единственного числа «ТУРБИННЫЙ ДВИГАТЕЛЬ, ВРАЩАЮЩИЙ …».

Основной функцией хелпера является TryParse. Количество токенов можно ограничить, застраховавшись от случая отсутствия закрывающей кавычки. Дополнительные параметры выделения:

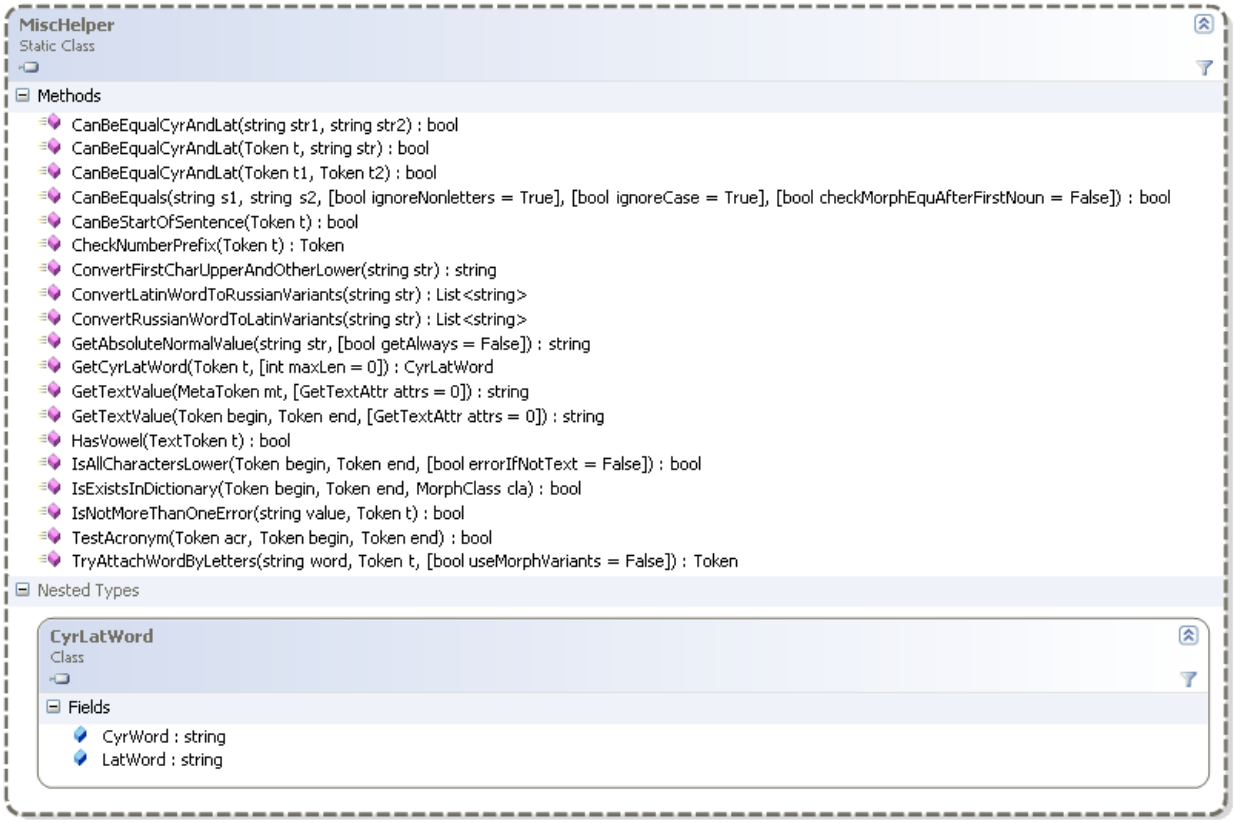
- CanContainsVerbs – по умолчанию, последовательность не может содержать глаголов, потому что в наименованиях именованных сущностей они отсутствуют, как правило. Но если они могут быть, то этот ключ нужно указать;
- NearCloseBracket – брать первую же подходящую закрывающую скобку\кавычку. По умолчанию, анализируются сложные случаи вложенности и сливания кавычек;

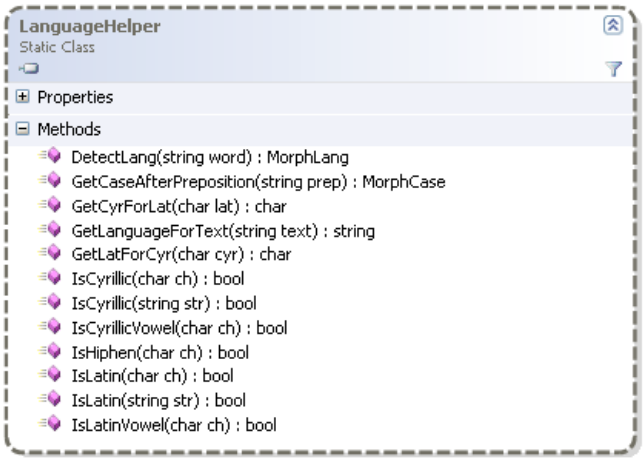
- CanBeManyLines – обычно наименования именованных сущностей являются одно строковыми. Если это не так, то нужно указать этот ключ.

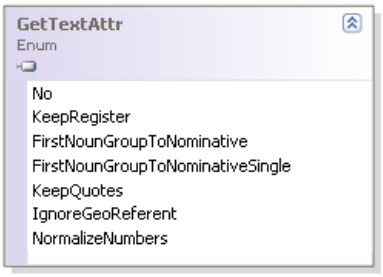
Отметим, что данная функция ориентирована именно на наименования, поэтому по умолчанию и подразумевает отсутствие глаголов и однострочность. Вообще для случаев больших цитат, диалогов и пр. данная функция не очень подходит.  Тут возможна специфическая логика, которую рекомендуется реализовывать самостоятельно, и для этого пригодятся функции CanBeStartOfSequence \ CanBeEndOfSequence.

## Другие полезные хелперы

В ходе решения разных лингвистических задач общие функции анализа выносились в хелперы ядра (EP.Core), где благополучно находятся и совершенствуются. Кратко опишем некоторые из них, подробности во встроенной xml-документации.



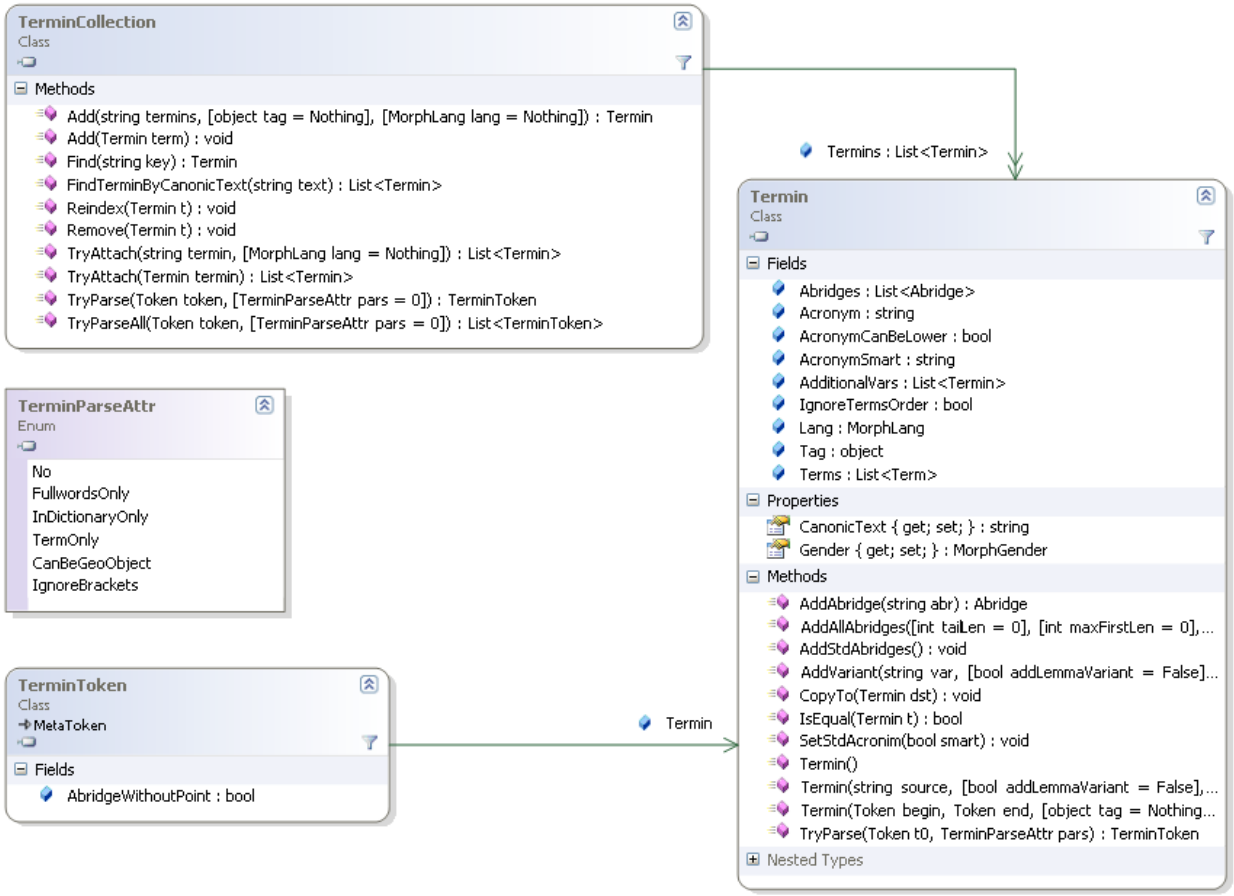

- CanBeStartOfSentence – проверка, что токен может начинать предложение;
- GetTextValue – получение для указанного диапазона токенов текста, лежащего под ним. Чрезвычайно полезная функция, использующая дополнительные параметры GetTextAttr (можно как битовую маску):
    - KeepRegister – сохранять регистр символов (по умолчанию, всё в верхний);

- FirstNounGroupToNominative – первую именную группу приводить к именительному падежу (число сохраняется). Например, для «информационных систем, применяемых в отрасли» получим «ИНФОРМАЦИОННЫЕ СИСТЕМЫ, ПРИМЕНЯЕМЫЕ В ОТРАСЛИ»;
- FirstNounGroupToNominativeSingle – аналогично, только ещё и к единственному числу. Для вышеуказанного примера получим «ИНФОРМАЦИОННАЯ СИСТЕМА, ПРИМЕНЯЕМАЯ В ОТРАСЛИ»;
- KeepQuotes – сохранять ли кавычки (по умолчанию, игнорируются);
- NormalizeNumber – преобразовывать числовые значения в цифровой вид;
- IgnoreGeoReferent – игнорировать географические объекты (иногда бывает полезно);

- CheckNumberPrefix – проверка различных способов написания префиксов номеров (№, N, Ном., Рег.Номер, рег. н-р и пр.), если есть, то вернёт ссылку на последний токен префикса;
- TryAttachWordByLetters – проверка случая, когда слово в тексте задаётся вразбивку по буквам, разделённым пробелами (например, П Р И К А З);
- IsNotMoreThanOneError(value, t) – проверка, что текстовой токен t содержит слово value, которое может быть написано в токене не более чем с одной ошибкой.
- CanBeEquals – проверка эквивалентности строк с учётом игнорирования пробелов, небуквенных символов, регистра и морфологических вариантов.
- CanBeEqualCyrAndLat, GetCyrLatWord – это работа с латинскими и кириллическими написаниями одних и тех же слов. Например, можно проверить, что «IKEA» и «ИКЕЯ» эквиваленты. Можно получить возможные варианты написаний в другой транслитерации. Например, для «IKEA» получить варианты «ИКЕЯ» и «ИКЕА» и наоборот.
- TestAcronym – проверка, что текстовой токен содержит сокращенное написание того, что находится между begin и end токенами (например, ГосЗаказ = государственный заказ, ЛПР = лицо, принимающее решение)

## Словари терминов

Если нужно проверить для токена некоторое слово (с учётом морфологических вариантов), то для этого у токена существует функция IsValue(string). Однако если нужно проверить много ключевых слов, то циклический вызов этой функции сильно снизит производительность. Для таких случаев предназначен класс TerminCollection.




Принцип работы следующий – в коллекцию добавляются термины, а затем можно быстро проверять на их наличие.

Термин – это конструкция, содержащая не только слова и их словосочетания, но и всевозможные сокращения, аббревиатуры и другие варианты написаний.

В качестве примера рассмотрим фрагменты кода, как формируется один из внутренних словарей для задания ключевых слов анализатора, выделяющего улицы.

```
TerminCollection m_Ontology = new TerminCollection();
Termin t;
t = new Termin("УЛИЦА") { Tag = StreetItemType.Noun };
t.AddAbridge("УЛ.");
m_Ontology.Add(t);

t = new Termin("ВУЛИЦЯ") { Tag = StreetItemType.Noun, Lang = MorphLang.UA };
t.AddAbridge("ВУЛ.");
m_Ontology.Add(t);
```

Отметим, что здесь возможные сокращения задаются явно. Также для разных языков можно добавлять в один и тот же словарь, но в зависимости от языка текущего текста будет использоваться только соответствующие записи. Поле Tag используется произвольным образом, здесь он задаёт некоторый внутренний классификатор.

```
t = new Termin("ПРОСПЕКТ") { Tag = StreetItemType.Noun };
t.AddAbridge("ПРОС."); t.AddAbridge("ПРОСП."); t.AddAbridge("ПР-Т");
t.AddAbridge("ПР-КТ"); t.AddAbridge("П-Т"); t.AddAbridge("П-КТ");
m_Ontology.Add(t);

t = new Termin("ВОЕННЫЙ ГОРОДОК") { Tag = StreetItemType.Noun };
t.AddAbridge("В.ГОРОДОК"); t.AddAbridge("В/Г"); t.AddAbridge("В/ГОРОДОК");
t.AddAbridge("В/ГОР");
m_Ontology.Add(t);

t = new Termin("ПРОМЫШЛЕННАЯ ЗОНА") { Tag = StreetItemType.Noun };
t.AddVariant("ПРОМЗОНА");
m_Ontology.Add(t);
```

Здесь отметим, что AddVariant задаёт не сокращение, а именно дополнительный способ написания.

```
t = new Termin("ВЕРХНИЙ") { Tag = StreetItemType.StdAdjective };
t.AddAbridge("ВЕРХН."); t.AddAbridge("ВЕРХ.");
t.AddAbridge("ВЕР."); t.AddAbridge("В.");
t.AddVariant("ВЕРХНІЙ");
m_Ontology.Add(t);
```

А здесь украинский способ просто добавляется как вариант к русскому – так тоже можно.

```
t = new Termin("АВТОДОРОГА") { Tag = StreetItemType.Noun };
t.AddVariant("ФЕДЕРАЛЬНАЯ АВТОДОРОГА"); t.AddVariant("АВТОМОБИЛЬНАЯ ДОРОГА");
t.AddVariant("ТРАССА"); t.AddVariant("АВТОТРАССА"); t.AddVariant("ФЕДЕРАЛЬНАЯ ТРАССА");
m_Ontology.Add(t);
```

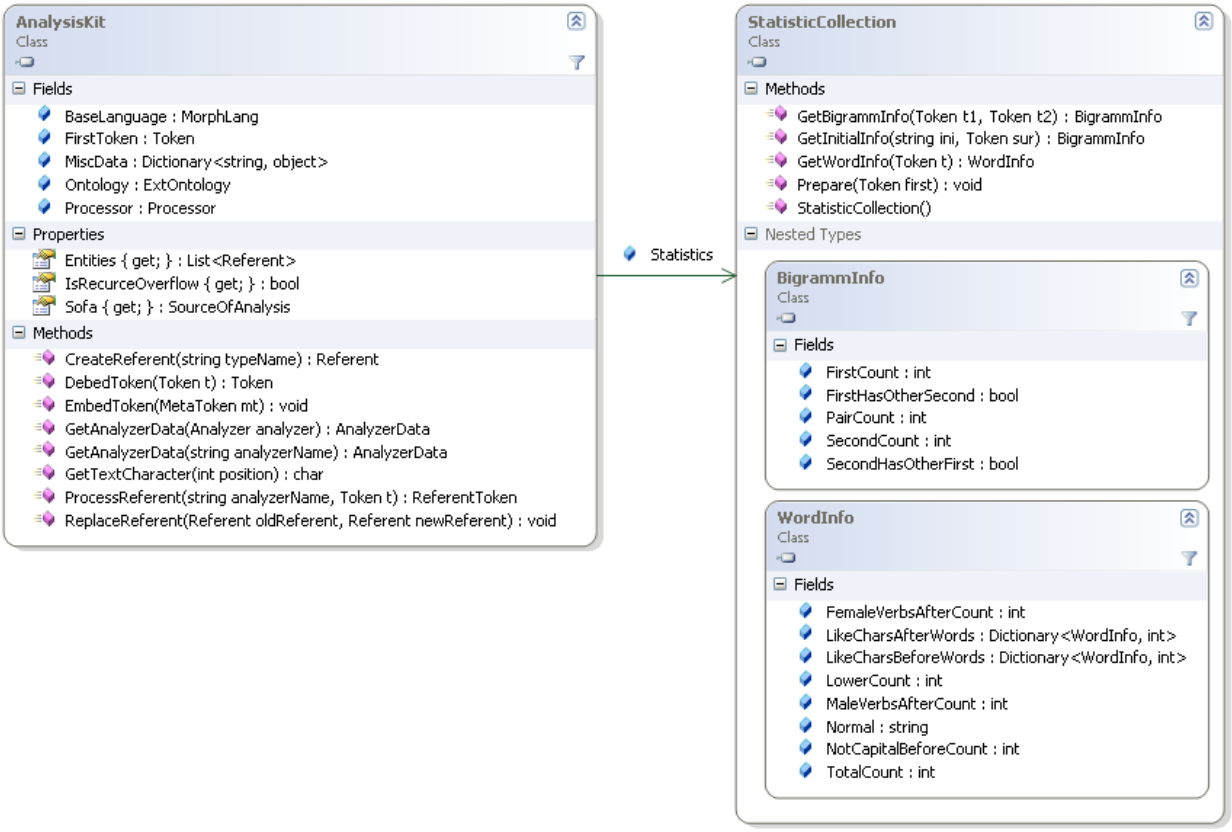
Обратите внимание, то термин добавляется в словарь после полного своего определения. Это важно, так как в момент своего добавления все его способы представления добавляются во внутренние словари TerminCollection, и последующие изменения t не будут учитываться (впрочем, можно вызвать функцию Reindex для термина).

Это пример статического словаря, формируемого один раз при инициализации движка и в дальнейшем не изменяемого. Но можно формировать такие словари динамически на основе анализа текущего текста, добавляя термины динамически. Именно так анализатор выделения персон составляет локальный список фамилий и учитывает эту информацию при выделении.

Основная функция при выделении в потоке токенов – TryParse(t). В случае удачи вернёт TerminToken – метатокен, задающий границы начала и конца термина в тексте (BeginToken \ EndToken) и ссылающийся на термин Termin. Если терминов может быть привязано несколько, то берётся самый длинный из возможных. Если нужно получить все варианты, то используйте функцию TryParseAll, возвращающую список.

## Аналитический контейнер

Для каждого текста на начальной стадии его анализа создаётся так называемый аналитический контейнер AnalysisKit, информацию из которого можно использовать. Каждый токен ссылается на контейнер через свойство Kit. Контейнер содержит все выделяемые сущности, последовательность токенов (FirstToken – ссылка на первый), статистику по токенам (Statistics) и множество вспомогательных данных, используемых внутренним образом.




Через функцию EmbedToken контейнера происходит встраивание метатокенов в последовательность токенов. Например, пусть для удобства анализа нужно встроить именные группы в последовательность.

```
for (Token t = result.FirstToken; t != null; t = t.Next)
{
       NounPhraseToken npt = NounPhraseHelper.TryParse(t);
       if (npt == null) continue;
       // встраиваем
       t.Kit.EmbedToken(npt);
      // теперь вместо npt.BeginToken (=t) ... npt.EndToken в последовательности один npt
       t = npt;
}
```

Рассмотрим текст «на маленьком плоту сквозь». Изначально здесь 4 токена: `FirstToken` = «на» ↔ «маленьком» ↔ «плоту» ↔ «сквозь», стрелками показаны ссылки Previous и Next на соседние токены. После выделения npt до его встраивания npt.Next = npt.Previous = null, а npt.BeginToken = «маленьком» и npt.EndToken = «плоту». После встраивания получаем 3 токена: `FirstToken` = «на» ↔ «МАЛЕНЬКИЙ ПЛОТ» ↔ «сквозь», причём BeginToken и EndToken у встраиваемого npt

продолжают указывать на токены, которые покрыл npt, так что в случае необходимости до них всегда можно добраться.

Если по какой-либо причине нужно провести обратную операцию восстановления покрытых метатокеном токенов, то используется функция DebedToken(MetaToken mt). Она восстанавливает цепочку на месте mt и возвращает указатель на mt.BeginToken.

## Морфология

Вся морфология реализуется в одной сборке EP.Morphology.dll, которая может использоваться абсолютно независимо от SDK Pullenti (но не наоборот).

Здесь морфология производится для каждого токена отдельно, независимо от окружающих его токенов. То есть это POS-Tagger (Part of Speech), выдающий для каждого токена-словоформы всевозможные морфологические варианты. Вопрос интерпретации и уменьшения множества вариантов решается вне POS-Tagger. Например, ядро SDK Pullenti при анализе именных групп оставляет только непротиворечивые корневому слову варианты и т.п.

Отметим следующее:

- Поддерживаются русский, украинский и английский язык;
- Для неизвестных слов выдаются варианты;
- Производится лексикографическая корректировка. Например, если в слове на кириллице одна буква заменена на латинскую, аналогичную по внешнему виду, то производится соответствующая корректировка. Также корректируются буквы с ударениями и т.п., замена апострофа на «ъ» (об`явление), «ё» на «е» и множество экзотических случаев (например, некоторые извращенцы пишут букву Ы через Ь и I);
- Для токена предлагается вариант леммы (нормальная форма);
- Есть функция получения для нормальной формы слова всех вариантов словоформ;
- При первом запуске движок инициализирует в памяти словари, на что требуется несколько секунд. Чтобы избежать задержку при первом анализе, рекомендуется принудительно вызывать Morphology.Initialize() при старте программы с указанием нужных языков, для которых загружать словари[2];
- Словарь для каждого языка занимает в памяти от 100 до 150 Мб, словари можно динамически загружать и выгружать через функции LoadLanguages и UnloadLanguages, однако для каждого текста это может быть неэффективно – процедура занимает несколько секунд;

[2] Если морфология используется вместе с лингвистическим процессором, то этого делать не нужно, так как инициализация морфологии происходит при ProcessorService.Initialize().

Морфанализ[3] производится через функцию Process статического класса Morphology, которая на выходе выдаёт список токенов List<MorphToken>.

Токен содержит смещение в символах (BeginChar и EndChar), информацию о символах (CharsInfo, см. предыдущий раздел), Term – преобразованный исходный фрагмент (верхний регистр, замена некоторых символов), и список морфологических вариантов WordForms.

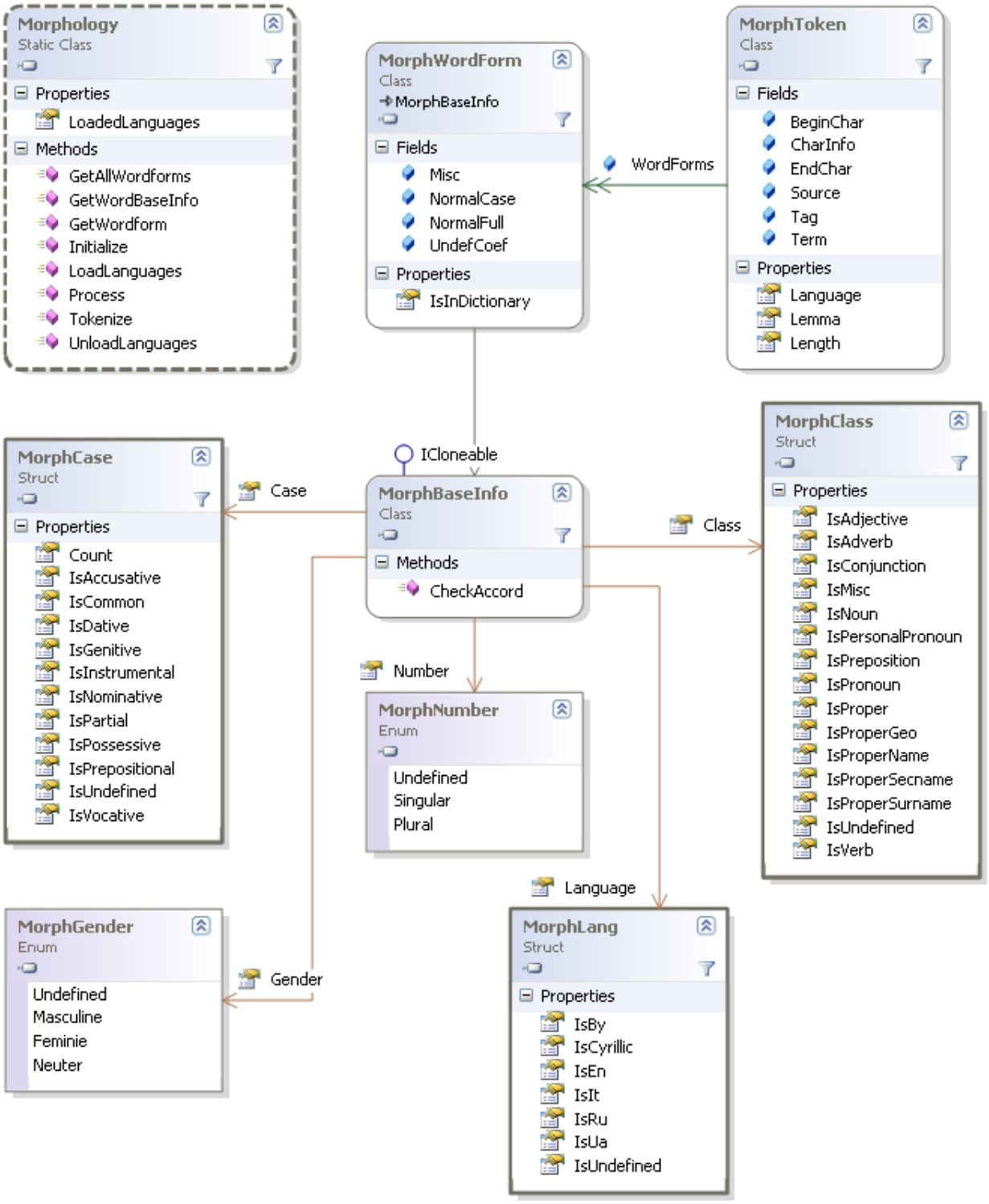
Морфологический вариант MorphWordForm имеет тип (Class словоформы: существительное, прилагательное и т.п.), род (Gender), падеж (Case), число (Number) и язык (Language), а также:

- IsInDictionary – признак того, есть ли словоформа в словаре, или этот вариант был создан для неизвестного слова;
- NormalCase – вариант словоформы в именительном падеже;
- NormalFull – вариант в именительном падеже и единственном числе;
- Misc – дополнительная морфологическая информация (см. help);

Токен MorphToken имеет свойство Lemma, которое выбирает из всех словоформ нормальную форму по фиксированному алгоритму. Правил довольно много, например, существительное имеет преимущество перед глаголом, у глаголов обрезается «-СЯ» и т.п. Например, для исходного слова «стекла» с его двумя словоформами (стекло и стечь) леммой будет «СТЕКЛО». В принципе, для поисковых систем такого рода нормализации вполне достаточно. Если предлагаемый алгоритм формирования лемм не устраивает, то можно реализовать свой на основе анализа списка WordForms.

На диаграмме указаны основные классы и структуры, задействованные в морфологии и расположенный в пространстве имён EP.Text:

[3] Если работа идёт через Processor, то явно из Morphology ничего запускать не нужно – процессор это делаем сам на 1-й фазе анализа перед применением анализаторов.

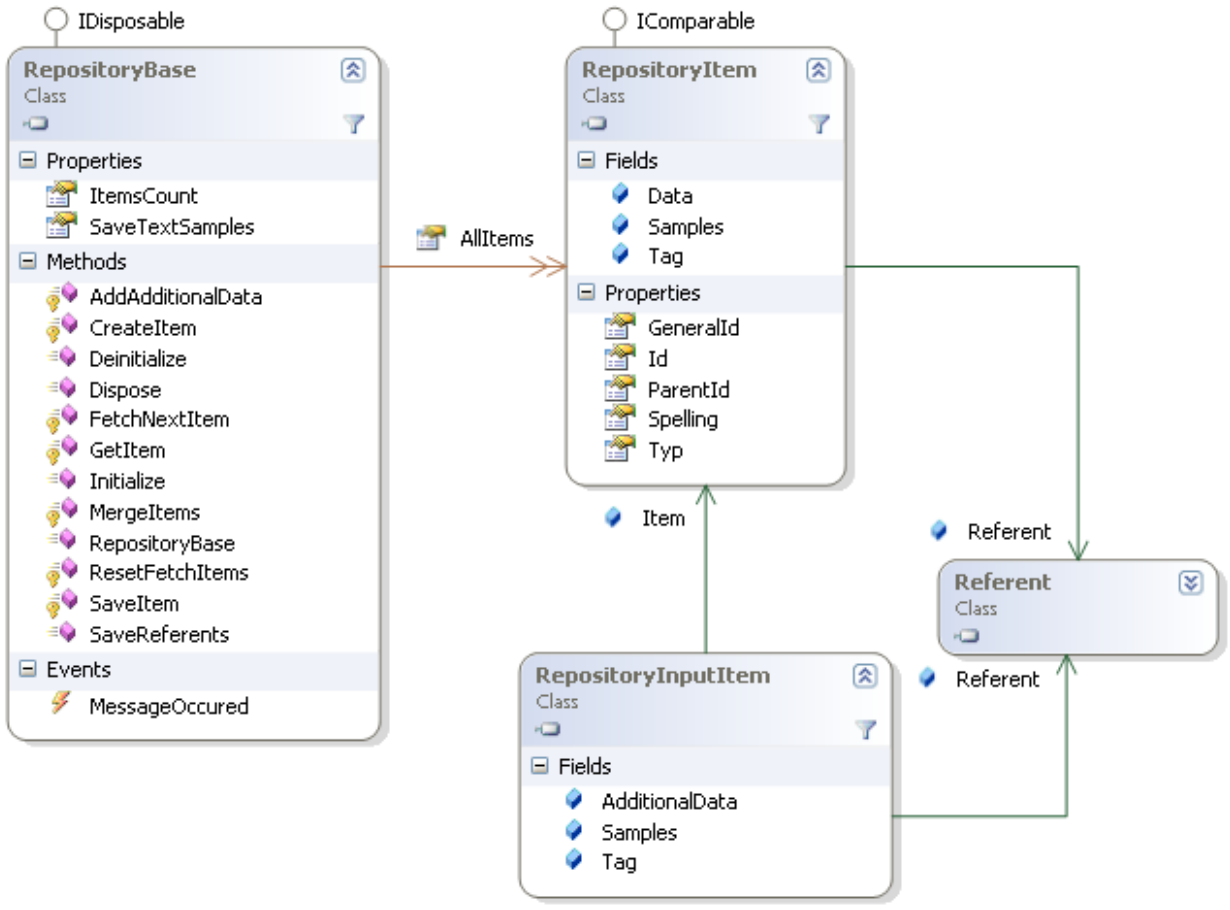

Morphology
Static Class
Properties
LoadedLanguages
Methods
GetAllWordforms
GetWordBaseInfo
GetWordform
Initialize
LoadLanguages
Process
Tokenize
UnloadLanguages
MorphWordForm
Class
MorphBaseInfo
Fields
Misc
NormalCase
NormalFull
UndefCoef
Properties
IsInDictionary
WordForms
MorphToken
Class
Fields
BeginChar
CharInfo
EndChar
Source
Tag
Term
Properties
Language
Lemma
Length
ICloneable
MorphCase
Struct
Properties
Count
IsAccusative
IsCommon
IsDative
IsGenitive
IsInstrumental
IsNominative
IsPartial
IsPossessive
IsPrepositional
IsUndefined
IsVocative
Case
MorphBaseInfo
Class
Methods
CheckAccord
Class
MorphClass
Struct
Properties
IsAdjective
IsAdverb
IsConjunction
IsMisc
IsNoun
IsPersonalPronoun
IsPreposition
IsPronoun
IsProper
IsProperGeo
IsProperName
IsProperSecname
IsProperSurname
IsUndefined
IsVerb
Number
MorphNumber
Enum
Undefined
Singular
Plural
Language
MorphGender
Enum
Undefined
Masculine
Feminie
Neuter
Gender
MorphLang
Struct
Properties
IsBy
IsCyrillic
IsEn
IsIt
IsRu
IsUa
IsUndefined

## Сериализация сущностей и массовая обработка данных

Для задач, где требуется обрабатывать множество текстов и хранить получаемые сущности, SDK предлагает специальный базовый класс RepositoryBase, облегчающий реализацию хранилища сущностей. Место хранения сериализованных данных от сущностей определяется в наследном классе (например, это может быть реляционная СУБД или файловая система). RepositoryBase берёт на себя функции отождествления новых данных со старыми данными и поддержки непротиворечивости семантической сети – всё множество выделяемых сущностей представляет собой ориентированный граф, так как значениями атрибутов могут выступать другие сущности.

*Внимание! Данный класс рассчитан на максимальное количество сущностей порядка миллиона, для бОльшего количества разрабатывается другой класс, который не держит в памяти всю семантическую сеть, а только необходимый фрагмент. Это сказывается на скорости обработки.*

Для реализации собственного хранилища сущностей необходимо переопределить в наследном от RepositoryBase классе ряд виртуальных методов, занимающихся непосредственным сохранением и получением «плоских» данных – экземпляров класса RepositoryItem. Сохраняемые сущности «оборачиваются» экземплярами класса RepositoryInputItem.



Опишем функции, которые необходимо переопределить для собственного репозитория, затем покажем пример использования репозитория.

Функции для переопределения:

```
/// <summary>
/// Инициализировать извлечение всех элементов
/// </summary>
protected virtual void ResetFetchItems()

/// <summary>
/// Извлечь очередной элемент
/// </summary>
/// <returns></returns>
protected virtual RepositoryItem FetchNextItem()
```

Эти функции используются для внутреннего перебора всех элементов. Например, если элементы хранятся в таблице СУБД, то ResetFetchItem должен открывать курсор, а FetchNextItem извлекать элемент из очередной записи таблицы.  У класса RepositoryItem хранятся следующие поля:

```
        /// <summary>
        /// Уникальный идентификатор внутри репозитория
        /// </summary>
        public int Id { get; set; }
        /// <summary>
        /// Это ToString() от сущности
        /// </summary>
        public string Spelling { get; set; }
        /// <summary>
        /// Это тип сущности (поле TypeName)
        /// </summary>
        public string Typ { get; set; }
        /// <summary>
        /// Идентификатор сущности-обобщения ("общее-частное")
        /// </summary>
        public int GeneralId { get; set; }
        /// <summary>
        /// Идентификатор сущности-контейнера ("часть-целое")
        /// </summary>
        public int ParentId { get; set; }

        /// <summary>
        /// Это строка, представляющая сериализацию сущности
        /// </summary>
        public string Data;
        /// <summary>
        /// Это строка, в которой сериализуются примеры встречаемости сущности в текстах
        /// (для десериализации используйте класс RepositoryItemSample
        /// </summary>
        public string Samples;
```

Как видно, здесь присутствуют только поля простых типов, легко отображаемые на поля реляционных таблиц. Генерацию и уникальность Id поддерживается на уровне наследного класса RepositoryBase.

```
/// <summary>
/// Получить элемент по его идентификатору
/// </summary>
```

```
        /// <param name="id"></param>
        /// <returns></returns>
protected virtual RepositoryItem GetItem(int id)
```

Здесь наследный класс должен вернуть экземпляр для указанного Id.

```
/// <summary>
/// Сохранить изменения. Если у элемента нулевой идентификатор, то это новый
/// элемент, и новое значение нужно записать в поле Id.
/// </summary>
/// <param name="item"></param>
protected virtual void SaveItem(RepositoryItem item)
```

Для элемента, который уже существует и требуется сохранить обновления, item.Id > 0. Для нового элемента нужно сгенерировать для него уникальный идентификатор, сохранить элемент и записать в item.Id этот новый идентификатор.

```
/// <summary>
/// Объединить сущности. Необходимо сохранить baseItem, удалить mergedItems,
/// а также предпринять усилия по обеспечению целостности информации, если
/// кто-либо извне ссылается на удаляемые элементы.
/// </summary>
/// <param name="baseItem"></param>
/// <param name="mergedItems"></param>
protected virtual void MergeItems(RepositoryItem baseItem, List<RepositoryItem>
mergedItems)
```

Данная функция вызывается при возникновении ситуации, когда нужно объединить существующие сущности. Рассмотрим пример. Пусть в одном тексте встретилось: ООО «Промышленный банк» (ПромБанк), в другом: ООО «СвязьБанк», соответственно были выделены 2 сущности, которые сохранены. Затем встретился текст: ООО «СвязьБанк» (бывший «ПромБанк»), и система понимает, что новой сущности соответствуют 2 существующие сущности. Объединение сводится к тому, что в одну сущность (baseItem) добавляется недостающая информация, а остальные сущности нужно удалить.

```
/// <summary>
/// Добавить в элемент дополнительную информацию (которая поступает из
RepositoryInputItem.AdditionalData)
/// </summary>
/// <param name="item"></param>
/// <param name="additionalData"></param>
protected virtual void AddAdditionalData(RepositoryItem item, object additionalData)
```

Эту функцию придётся переопределить, если вместе с загружаемыми сущностями поступает некоторая дополнительная информация, выходящая за рамки SDK. В этом случае RepositoryBase просто переадресует эту информацию в наследный класс через эту функцию.

```
/// <summary>
/// Создать экземпляр элемента (по умолчанию создаётся RepositoryItem)
/// </summary>
/// <returns></returns>
protected virtual RepositoryItem CreateItem()
```

Эту функцию можно переопределить, если по каким-либо причинам в наследном классе удобней оперировать экземплярами с некоторыми дополнительными полями. Поскольку все экземпляры создаются через эту функцию, то вместо RepositoryItem наследный класс RepositoryBase будет оперировать переопределённым классом.

Использование хранилища

RepositoryBase имеет следующие прикладные функции:

```
/// <summary>
/// Инициализация репозитория (необходимо вызывать перед первым использованием)
/// </summary>
public void Initialize()

/// <summary>
/// Вызывать в конце работы
/// </summary>
public void Deinitialize()

/// <summary>
/// Сохранить сущности
/// </summary>
/// <param name="input">список обёрток над сущностями</param>
public void SaveReferents(ICollection<RepositoryInputItem> input)
```

Через эту функцию происходит сохранение сущностей, выделяемых семантическим процессором из текстов. Напомним, что процессор в качестве результата работы возвращает экземпляр AnalysisResult, одним из членов которого является List<Referent> Entities – список сущностей. Этот список необходимо преобразовать в список элементов RepositoryInputItem, оставив нужные для сохранения сущности (некоторые типы сущностей могут оказаться ненужными, например, даты):

```
        /// <summary>
        /// Это должно быть установлено на входе
        /// </summary>
        public Referent Referent;
        /// <summary>
        /// Здесь могут быть ранее подготовленные примеры вхождений
        /// (если нет, то будут вычисляться из Referent.Occurence)
        /// </summary>
        public string Samples;
        /// <summary>
        /// Некоторые дополнительные данные
        /// </summary>
        public object AdditionalData;
        /// <summary>
        /// Используется произвольным образом
        /// </summary>
        public object Tag;

        /// <summary>
        /// Это будет установлено после сохранения
        /// </summary>
        public RepositoryItem Item;
```

После успешной отработки SaveReferents в значения Item будут записаны элементы хранилища (новые и существующие), так что для текста получается соответствие между его сущностями и сущностями репозитория.

```
/// <summary>
/// Перечисление всех элеменов
/// </summary>
public IEnumerable<RepositoryItem> AllItems
```

Эта функция перебирает все элементы хранилища, причём у RepositoryItem устанавливается свойство Referent, указывающее на экземпляр сущности.

```
/// <summary>
/// Найти для сущности существующие в хранилище элементы
/// </summary>
/// <param name="referent"></param>
/// <param name="includeGenerals">при true будет включать в список сущности с учётом отношения обобщения</param>
/// <returns>список (null - если нет аналогов)</returns>
public List<RepositoryItem> FindItems(Referent referent, bool includeGenerals)
```

Эта функция может оказаться полезной для обработки поисковых запросов пользователя, из которых выделены сущности. В этом случае в хранилище ничего не добавляется, а результатом является список элементов, которые могут быть эквивалентны.

Пример реализации загрузки в хранилище

```
RepositoryBase rep = new наследный класс от RepositoryBase();
rep.Initialize();
// цикл обработки текстов
while (true)
{
     string text = ...; // извлекаем очередной текст для обработки
     // обрабатываем его
     AnalysisResult ar = rep.Processor.Process(new SourceOfAnalysis(text));
     // формируем список сохраняемых сущностей
     List<RepositoryInputItem> inputList = new List<RepositoryInputItem>();
     // пусть будем сохранять только персон и их свойства (должности и т.п.)
     foreach (var e in ar.Entities)
         if (e.TypeName == "PERSON" || e.TypeName == "PERSONPROPERTY")
             inputList.Add(new RepositoryInputItem() { Referent = e });

     // сохраняемся в репозитории
     rep.SaveReferents(inputList);

     // вот и всё
     foreach(var i in inputList)
         Console.WriteLine("Entity {0} -> Id={1}", i.Referent, i.Item.Id);

 }
rep.Deinitialize();
```

Разное

Есть возможность сохранять фрагменты исходных текстов, из которых выделились сущности. Для этого нужно установить свойство RepositoryBase.SaveTextSamples = true, и в RepositoryItem.Samples будут фигурировать фрагменты текстов. Сущность может определяться по разному, Samples содержит только уникальные  варианты задания, исключая дубликаты. Для работы со строкой Samples можно воспользоваться классом `RepositoryItemSample:`

```
        /// <summary>
         ///  Фрагмент перед
         /// </summary>
         public string HeadPeace;
         /// <summary>
         /// Сам фрагмент
         /// </summary>
         public string BodyPeace;
         /// <summary>
         /// Фрагмент после
         /// </summary>
         public string TailPeace;
         /// <summary>
         /// Признак того, что этот текстовой фрагмент был использован для "первого"
выделения сущности,
         /// а не привязки к ранее выделенному.
         /// </summary>
         public bool IsEssential;
```

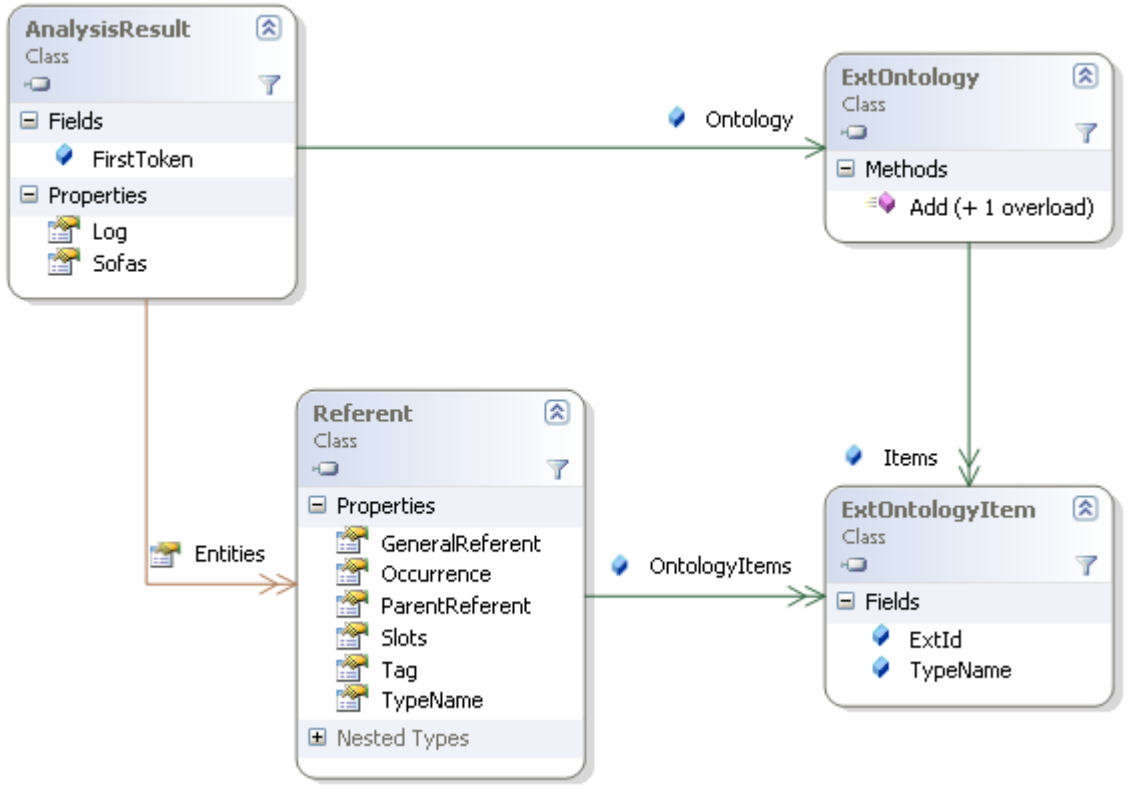
Для десериализации списка фрагментов из строки предназначен метод:

```
public static List<RepositoryItemSample> Deserialize(string samples)
```

## Поддержка внешних онтологий (словарей)

Выделение сущностей основано на правилах. Однако для некоторых типов сущностей можно подгружать внешние словари (онтологии), содержащие описания существующих сущностей, и тогда система при выделении будет пытаться привязываться к внешним сущностям, устанавливая поле OntologyItems у экземпляров Referent.

Словарь реализуется классом ExtOntology, содержащим список элементов ExtOntologyItem, которые нужно добавить функцией Add(…). Словарь может содержать элементы любых поддерживаемых типов.




Рассмотрим пример.

Пусть есть список сотрудников, который нужно искать в текстах.

```
// создаём словарь-онтологию
ExtOntology personOntos = new ExtOntology();
```

Каждого сотрудника можно добавить 2-мя способами: в виде текстового описания и в виде готовой сущности. Если у нас не выделены по отдельности Имя-Фамилия-Отчество, то добавляем неструктурированное описание (точкой с запятой разделяются несколько описаний, если таковые имеются):

```
string desciption = "Иванов Иван Иванович;Ivanov Ivan";
ExtOntologyItem it = personOntos.Add("любой Id", "PERSON", desciption);
```

Если же информация заранее структурирована, то можно сразу создать сущность и добавить её в словарь:

```
PersonReferent person = new PersonReferent();
string firstName = "ИВАН", lastName = "ИВАНОВ", middleName = "ИВАНОВИЧ";
person.AddSlot(PersonReferent.ATTR_FIRSTNAME, firstName, false);
person.AddSlot(PersonReferent.ATTR_MIDDLENAME, middleName, false);
person.AddSlot(PersonReferent.ATTR_LASTNAME, lastName, false);
ExtOntologyItem it = personOntos.Add("любой Id", person);
```

Полученный словарь нужно подать вторым параметром на вход процессору:

```
Processor proc = new Processor();
AnalysisResult res = proc.Process(new SourceOfAnalysis(...), personOntos);
```

В результирующем списке сущностей Entities те из них, которые удалось привязать к элементам внешней онтологии, будут содержать непустые списки на эти элементы (теоретически привязаться может более чем к одному элементу словаря).

## Включение SDK в пользовательский проект

Демонстрационное решение Demo.sln сделано на MS Visual Studio 2010 для версии .NET 2.0.

Все модули SDK собраны в проекте EP.Sdk.csproj, который нужно добавить в пользовательский sln и указать на него ссылку (Add Reference) из главного модуля. Это обеспечит автоматическое копирование в исполняемую директорию всех необходимых файлов. Фактически для развёртывания достаточно скопировать все dll-сборки директории проекта EP.Sdk.csproj в исполняемую директорию. При необходимости вставки в Setup-проект нужно добавить EP.Sdk.csproj в режиме “Content Output”, так как все необходимые файлы помечены атрибутом “Build Action” = “Content”.

В пользовательский проект csproj, в котором непосредственно будет использоваться процессор, нужно добавить ссылку (Add Reference) на сборку EP.Core.dll, в которой реализована поддержка модели данных и базового функционала. При необходимости использовать специфические классы сущностей нужно добавлять ссылки на соответствующие сборки (например, для типа PersonReferent нужна сборка EP.Analyzer.Person.dll).