Создание автоматических тестов по текстовым пользовательским сообщениям средствами системы Nalaps.

Г.П. Путилов (putilov@mitme.ru), А.С. Лебедев (andremoniy@gmail.com)

Московский Государственный Институт Электроники и Математики, Москва, Россия

1. Введение. Автоматическое тестирование.

Автоматические тесты (АТ) являются неотъемлемой частью цикла разработки крупных коммерческих и промышленных программных комплексов. Под АТ-ом понимается программа, написанная обычно на скриптовом языке или на том же языке программирования, что и сам программный комплекс. Автоматическое тестирование таким образом – это разработка планов тестирования и программирование самих тестов. (1). Цель написания и работы автоматического теста – проверка критической ситуации, конкретных модулей системы или отдельных участков кода для выявления потенциальных сбоев системы и проверки корректности ее работы в эмулируемых ситуациях. Необходимость создания автоматических тестов обусловлена сложностью современных программ, что влечет за собой побочные и практически неизбежные эффекты разработки, например, когда исправление одной ошибки приводит к возникновению новой в уже ранее отлаженном блоке. Автоматически обнаружить такие ошибки позволяет комплекс мер, называемый системой непрерывного автоматического тестирования, где перед каждой сборкой проекта производится запуск всех АТ-ов.

Вместе с вышесказанным следует указать и на другую, практическую сторону процесса написания АТ-а. Ввиду их специфики они являются с одной стороны довольно простыми по логической структуре, но достаточно трудоемкими по созданию с другой, что обусловлено необходимостью ручного внесения в код программы большого количества шаблонных тестовых данных, а также описание рутинных действий пользователя системы, эмулирующих заданную для проверки ситуацию. Такая задача отнимает у программистов много времени, являясь тяжелым и рутинным занятием, в особенности, если речь идет о тестировании графического пользовательского интерфейса (GUI) (2). Однако, если фундаментальный набор тестов, покрывающий явно очевидные критичные блоки программы необходимо создавать в начале цикла разработки программы, то написание программ для тестирования уникальных ситуаций, в которых могла бы возникнуть (или возникала ранее) ошибка, можно было бы автоматизировать, низведя до уровня простого конструктора объектов метаданных. Данное предположение обосновывается опытом, полученным при написании ряда промышленных и коммерческих программных продуктов – в том числе и опытом поддержки систем, эксплуатируемых в промышленном режиме.

В действительности, большинство замеченных в системе ошибок пользователи описывают в виде обычного текста на естественном языке (ЕЯ), предоставляя такие описания в различные системы для ведения списка дефектов. Иного удобного способа подачи заявок на исправление дефектов кроме текстового описания, в системе общения «пользователь<->разработчик» пока не придумано, или, по крайней мере, не выявлено в качестве реальной альтернативы. Полученная заявка на исправление ошибки проходит проверку: является ли описанная ситуация ошибкой и воспроизводится ли она при описанных условиях? В е если ошибка подтверждается, ее устраняют

и создают тест, проверяющий, что данный дефект при описанных условиях больше не проявляется.

В данном докладе предлагается описание механизма автоматического тестирования, реализуемого с помощью системы концептуального естественно-языкового программирования Nalaps, разработанной на кафедре МОСОИиУ МИЭМ. Предложенный подход позволяет сократить временны**е** затраты аналитиков и разработчиков на написание тестов для ситуаций, имеющих достаточно формализованное (в терминах предметной области) текстовое описание, что достигается за счет автоматизации данного процесса целиком или его основных этапов. При наличии достаточно полной базы знаний предметной области (т.е. той предметной области, к которой относится тестируемый программный комплекс) представляется теоритческая возможность полной автоматизации процесса создания тестов. База знаний в свою очередь накапливается в процессе работы разработчиков с предложенным инструментом.

Важно отметить, что система Nalaps – это реально действующий программный комплекс, который постоянно совершенствуется. Предложенный же в докладе механизм автоматического тестирования – это концепция, опробованная на сравнительно небольшом количестве примеров из промышленной среды, показавших практическую возможность реализации подхода с помощью системы Nalaps.

2. Система Nalaps

2.1. Концепция системы

Система Nalaps[1] – это программный комплекс, представляющий собой среду концептуального естественно-языкового программирования (3). Учет экстралингвистических знаний реализован путем использования ограничений, накладываемых предметной областью на интерпретацию текста. Концептоцентрический подход, примененный в системе, подразумевает, что в основу семантической классификации единиц ЕЯ положена база знаний о классификации предметов и явлений внеязыковой действительности (4). Такие знания выражают свойства объектов, процессов, явлений через понятия соответствующей предметной области. Для того чтобы описать с позиции концептуального знания некоторое понятие, нужно указать взаимосвязь его с другими понятиями, охарактеризовать его компоненты, а также зависимости последних между собой и между компонентами других понятий (5). Концепт определен в лингвистике как конфигурация знаний, отношения внутри которой суть связи между ее элементами, а последние в свою очередь тоже могут быть концептами (5). Такое понятие очень хорошо описывается таким понятием из объектно-ориентированного программирования (ООП), как класс. В системе Nalaps предлагается описание концептов в виде классов языка Java. Подчеркнем, что концепт является именно знанием, а не набором данных; точно также классы представляют собой только типы, определенные программистом, данными же являются объекты - сущности классов.

В соответствии с обозначенной парадигмой системы, она включает в себя:

- средства лингвиста для настройки проекта на предметную область (программирование лингвистического процессора);

- инструменты разработчика для описания связей между классами проекта и предметной областью;

[1] Nalaps – **NA**tural **LA**nguage **P**rogramming **S**ystem. http://www.nalaps.ru

- интерфейс для обработки текстовой информации, а также программные библиотеки, для встраивания данного интерфейса в сторонние Java приложения.

Система Nalaps реализована на языке Java 1.5, и имеет два режима работы: независимо от среды разработки Java и как надстройка в системе Intellij IDEA[2]. В последнем случае включаются такие функции, как прямая работа с исходными кодами классов, описанных как элементы базы знаний, автоматическая генерация классов и редактирование связей между концептами и классами.

Процесс автоматической генерации программы на языке Java в системе Nalaps состоит из следующих этапов:

1. Естественно-языковой текст поступает на вход лингвистического процессора, на выходе из которого продуцируется синтаксико-семантическое представление.
2. Полученное семантическое представление обрабатывается объектным препроцессором системы, который выделяет из полученной сети концепты (информационные объекты) и создает концептуальный граф.
3. Полученный концептуальный граф рассматривается как «скелет» для создания объектно-ориентированной программы. Каждому информационному объекту ставится в соответствие класс языка Java, найденный в базе данных предметной области, а каждому семантическому отношению ставится в соответствие метод класса, сопоставленного с информационным объектом - агенсом, где в качестве параметра метода используется объект, обладающий семантической ролью пациенса.

2.2. Автоматическая генерация тестов

Поскольку А-тест также является программой, то и его генерация возможна в системе Nalaps. Основной сложностью в данной задаче является «понимание» системой текстового описания поставленной задачи, на основе которой необходимо сгенерировать программу. Предполагается, что вводимые в систему тексты являются довольно хорошо формализованными, что обусловлено спецификой проблемы: предметная область достаточно узка и поддается описанию для конкретного тестируемого программного комплекса.

Рассмотрим перечисленные выше этапы с привлечением простейшего примера для иллюстрации.

Первый этап, связанн с работой лингвистического процессора (ЛП). В системе Nalaps ЛП является прикладным инструментом и поэтому имеет модульную реализацию - он может быть заменен другим, имеющим такой же программный интерфейс. ЛП системы Nalaps состоит из двух компонентов: морфологический анализатор (МорфАн) и синтаксико-семантический анализатор. На вход синтаксико-семантического анализатора поступает результат работы МорфАн-а, где каждой словоформе из входного потока поставлен в соответствие набор выделенных лексем с их грамматическими признаками. В качестве МорфАн-а используется парсер Mystem компании Яндекс[3]. В настоящей реализации синтаксико-семантического анализатора используется аппарат

[2] http://www.jetbrains.com/idea/
[3] http://company.yandex.ru/technology/mystem/ - «О программе mystem».
Официальное разрешение от компании Yandex на включение парсера mystem в состав сборки системы Nalaps получено 2 апр. 2010 года.

Расширенных Сетей Переходов (РСП) (6), где применен ряд эвристических подходов при разработке ATN[4]-анализатора, некоторые из которых мы рассмотрим ниже . Для упрощения программирования РСП в системе реализован визуальный ATN-редактор, который включает в себя все необходимые функции, связанные с обработкой графов: маркировкой дуг и узлов. Подключение к редактору морфологического анализатора позволяет строить шаблоны сетей по заданному предложению, а внутренний поиск отслеживает наличие подобных структур в имеющейся базе сетей (7).Фактически РСП представляет собой направленный граф, где вершиной является нулевой узел, с которого начинается разбор очередного фрагмента текста. В узлах сети могут быть записаны простейшие команды работы с переменными, конструкторы синтаксико-семантических отношений и некоторые команды. Каждое ребро графа маркируются наборами грамматических признаков, которыми должна обладать очередная лексема из входного потока для прохода по нему. Общий принцип работы предложенного ATN-анализатора кратко описывается следующими шагами:

1. Предложения разбираются поочередно.
2. Из входного потока поступает следующее слово (последовательный перебор слов и знаков препинания слева на право).
3. Если нет открытых РСП или нет подходящих дуг для перехода, то открыть РСП соответствующее части речи текущей словоформы.
4. Если у текущего слова из входного потока МорфАн-ом выделено более одной лексемы, т.е. присутствует омонимия, то процесс анализа в п.3. распараллеливается для каждой лексемы.
5. Анализируется число совпадений грамматических признаков текущей словоформы и грамматических признаков дуг, выходящих из текущего узла сети. Совпадение считается равным «-1», если полностью не совпадает хотя бы один грамматический признак (например, у дуги прописан признак «Падеж: им., вин.», а у входного слова однозначно определен предложный падеж).
6. Выбирается дуга с наибольшим числом совпадений, отличных от «0» и происходит проход к следующему узлу. В новом узле выполняются все описанные в нем команды.
7. Если в п.6 не найдено ни одной дуги, то переход к п.3.
8. Если узел помечен признаком конца, то текущая РСП закрывается.
9. Переход к п.3.

Пример работы предлагаемого механизма автоматического тестирования и работы ЛП, в частности, рассмотрим на следующем примере:

В карточке клиента на закладке “счета” в поле “дата платежа” невозможно указать сегодняшний день. Оно подсвечивается красным цветом. (1).

Для разбора примера (1) достаточно иметь 6 простых РСП:

1. РСП «глагол»:

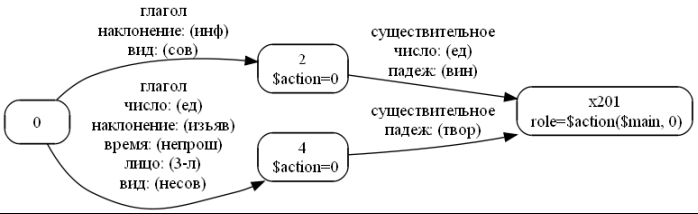

Рис. 1. РСП «глагол»

Данная сеть служит для разбора таких фрагментов предложений, как «*указать день*» и *«подсвечивается цветом».* В данной РСП инициализируется новая переменная “$action”,

[4] ATN – augmented transition network (англ.) – расширенная сеть переходов.

которая хранит в себе инфинитив предыдущего глагола, который будет служить именем семантического отношения, конструируемого в узле x201.

Итогом разбора становится отношение “$action($main, 0)”.

2. РСП «местоимение»:

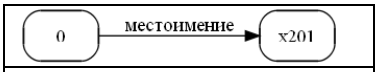

Рис.2. РСП «Местоимение»

Данная простейшая сеть служит только для осуществления корректного прохода по словам из входного потока. В нашем примере, данная сеть «проглотит» местоимение «*оно*» без выявления каких-либо отношений.

3. РСП «наречие»:

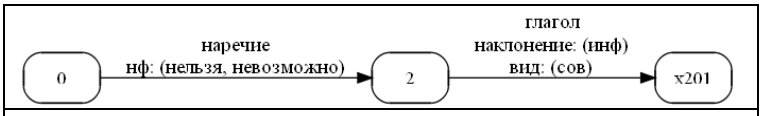

Рис. 3. РСП «Наречие»

Подобно предыдущей РСП в данной сети также отсутствует какой либо синтаксико-семантический разбор. В рамках выбранной предметной области выделение отдельной смысловой нагрузки наречия «нельзя/невозможно» избыточно, т.к. конечная цель – это построение программы-теста, проверяющей описанную ситуацию с точки зрения отсутствия ошибок валидации.

4. РСП «предлог»:

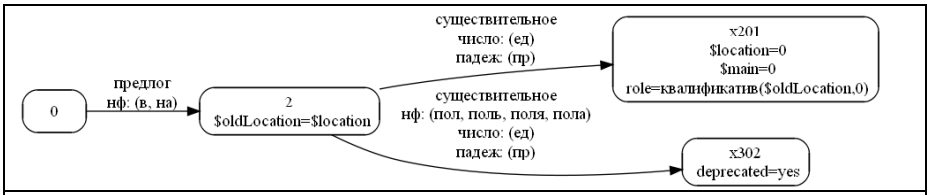

Рис. 4. РСП «Предлог»

Данная сеть позволяет разбирать фрагменты вида *«на столе»* или *«в шкафу»*. Специальная дуга [2 -> x302] является примером подхода к снятию омонимии: в предметной области примера (1) представляется крайне маловероятным использование слов *«пол, поль, Поля, пол**а**»* в конструкции вида *«в поле» («на поле»)*, поэтому данный вариант разбора помечается командой deprecated, что запрещает дальнейший разбор для данного варианта интерпретации словоформы *«поле»*.

Переменные $location и $main сохраняют последнее значимое слово из входного потока. Переменная $location используется в дальнейшем в той же сети, для связи обстоятельств места, данных в цепочке, как в примере (1). Так, при первом проходе по данной сети в $location будет записано слово «карточка», при втором – *«закладка»* и будет сконструировано отношение *«квалификатив(карточка, закладка)».*

Переменная $main будет использована в дальнейшем.

5. РСП «прилагательное»:

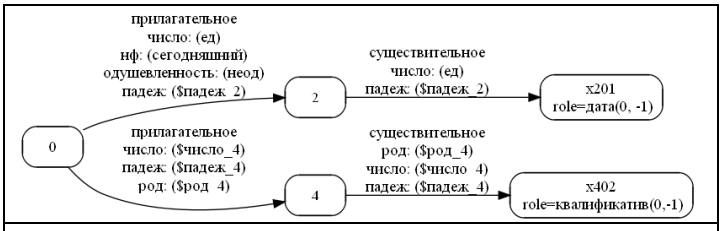

Рис. 5. РСП «прилагательное»

Отдельная ветвь с прилагательным *«сегодняшний»* выделяет семантическое отношение *«дата».* Числовые значения, указываемые в конструкторе отношения, обозначают смещение относительно данного узла. Так, «0» соответствует текущее слово из входного потока, «-1» - слово, сопоставленное с предыдущим узлом данной сети и т.д.

6. РСП «существительное»:

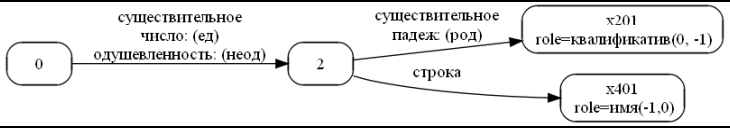

Рис. 6. РСП «существительное»

В данном примере под признаком дуги [2 -> x401] «строка» понимается любая строковая информация, заключенная в двойные кавычки. Данная строка передается в методы классов при конструировании программы как константа типа String.

Остановимся подробнее на механизме разрешения анафоры. Для этой цели используются переменные, вводимые в узлах сети. Их значения сохраняются на протяжении разбора всего текста. В примере (1) последним значением переменной $main будет лексема *«поле»* (им. п.). При разборе следующего предложения в РСП «глагол» будет вызван конструктор «$action($main,0)», из которого будет получено отношение «*подсвечиваться(поле, цвет)*». Предложенный способ позволяет эффективно разрешать большинство несложных анафор, использование которых можно предположить в описываемой задаче разбора сообщений об ошибках.

Полученное в результате работы ATN-анализатора с вышеописанными сетями синтактико-семантическое представление примера (1) имеет вид, представленный на рис.7.

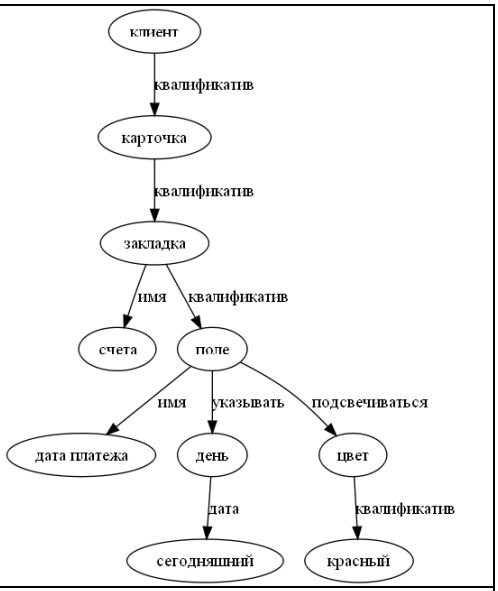

Рис. 7. Синтаксико-семантическое представление для примера 1

В целом, алгоритм первого и второго этапов носит характер модели «мягкого понимания» текста (4). Данная модель сочетает в себе следующие аспекты, отражающие процесс понимания текста человеком:

а) пофразное, поэлементное чтение и понимание ЕТ;

б) отождествление сущностей, упоминаемых в разных фразах;

в) выборка из ЕТ нужных терминологичных элементов;

г) извлечение из ЕТ сюжетов предметной области.

Реализация аспекта а) носит характер получения синтаксико-семантического представления (СинСемП) отдельных предложений ЕТ с учетом связей между ними. Аспекты б), в) и г) отождествляются с процессом выделения информационных объектов и переходом от СинСемП отдельных предложений к семантическому представлению всего текста в виде концептуального графа, что и является **вторым этапом** обработки текста системой Nalaps. Термин ***информационный объект*** (ИнфОб) предложен Кузнецовым И.П. (8) и выражает те сущности, под которыми понимаются встречающиеся в тексте конкретные предметы, лица, явления, термины и пр., которые имеют описание в виде концептов в базе концептуальных знаний.

В примере (1) информационными объектами будут выступать существительные, связанные со своими определениями. В данном случае, это:

Карточка клиента

Закладка «счета»

Поле «Дата»

Сегодняшний день

Красный цвет

Третий этап заключается в создании текста программы на основе полученного концептуального графа. Заметим, что генерация предполагает параллельное создание, как текста программы, так и пошаговое выполнение созданных команд, т.е. система Nalaps выступает также как интерпретатор. Отметим, что описание того, что *поле подсвечивается красным цветом*, избыточно в рамках решаемой задачи, поэтому в базе знаний нет необходимости определять сущность «*цвет*».

Приведем сокращенный код классов, описывающих понятия предметной области.

Для понятия «клиент»:

```
@Semanticable("Клиент")
public class Debtor {
   @Semanticable("Карточка")
   private DebtorCard card;
}
```

Для понятия «карточка»:

```
@Semanticable("карточка")
public class DebtorCard {
   @Semanticable("закладка")
   private Bookmark activeBookmark;
}
```

Для понятия «закладка»:

```
@Semanticable("закладка")
public class Bookmark {
   private String name;
   @Semanticable("поле")
   private FormField activeField;
   @Semanticable("имя")
   public void setName(String name) {
      this.name = name;
   }
}
```

Для понятия «поле»:

```
@Semanticable("поле")
public class FormField {
   private String name;
   @Semanticable("имя")
   public void setName(String name) {
      this.name = name;
   }
   @Semanticable("указывать")
   public void setValue(Object value)  {
      ...
   }
}
```

```
Для понятия «день»
@Semanticable("день")
public class DayT extends DateT {
        @Semanticable("датавремя")
        public void setDatetime(String datetime)  {
                if (datetime.equalsIgnoreCase("сегодняшний"))
                        setTime(new Date().getTime());
        }
}
```

Как видно из кода, в нем присутствует разметка, выполненная с помощью инструмента аннотаций (*@Semanticable)*. Указанная разметка служит вспомогательным инструментов для описания карты настройки класса на предметную область. Подобно описанию дуг в графах РСП, каждому размеченному таким образом элементу класса ставится в соответствие набор грамматических признаков слов (чаще всего с неопределенной формой), которые позволяют сопоставлять выделенные в концептуальном графе понятия с нужными классами.

Основной сложностью при написании тестов в данном примере является необходимость тестирования пользовательского web-интерфейса, т.к. все доступные операции в системе выполняются непосредственно из него. Для этих целей используется система Selenium HQ, представляющая собой набор инструментов для тестирования web-приложений (9). Указанная специфика обуславливает необходимость создания классов-оберток, описывающих способы доступа к тем или иным элементам интерфейса, описываемым в пользовательских сообщениях об ошибках. Не будем останавливаться на программной реализации сценариев их работы с web-интерфейсом. Отметим, что данная операция носит примитивный характер и заключается в перечислении действий в браузере, необходимых для получения страницы с требуемым элементом. В приведенном примере данные сценарии опущены.

Первый этап создания текста программы – автоматическая генерация объектов языка Java. Для каждого объекта система создает латинский акроним, полученный путем транслитерации неопределенных форм соответствующих данному информационному объекту лексем.

```
создаем объект, представляющий сущность «Клиент»
autotests.Debtor klient_1 = new autotests.Debtor();

для понятия «карточка клиента»
autotests.DebtorCard kartochka _1 = new autotests. DebtorCard ();

и т.д.:
autotests.Bookmark zakladka_1 = new autotests.Bookmark(“счета”);
autotests.FormField pole_1 = new autotests.FormField(“дата платежа”);
autotests.DayT den_segodnyashnii = new autotests.DayT();
```

Затем система генерирует код вызова методов на основании связей в концептуальном графе:

```
 klient_1.setDebtorCard(kartochka_1);
kartochka_1.setActiveBookmark(zakladka_1);
```

```
zakladka_1.setActiveField(pole_1);
den_segodnyashnii.setDatetime("сегодняшний");
pole_1.setValue(den_segodnyashnii);
```

Полученный текст программы может быть включен в цикл непрерывного тестирования – в случае возникновения ошибки, тест сгенерирует исключение (exception).

3. Инструменты и элементы интерфейса.

3.1. Связь между концептами и классами

Связка между концептами, описанными с помощью классов языка Java, и ЕЯ понятиями осуществляется с помощью языка XML. Такие связки, называемые *картами настройки на предметную область* и хранимые в XML-файлах, создаются с помощью инструментария с графическим интерфейсом пользователя, входящим в состав системы Nalaps. Данный инструментарий, включающий в себя морфологический процессор, позволяет быстро и удобно связывать классы и его компоненты с терминами и лексемами предметной области.

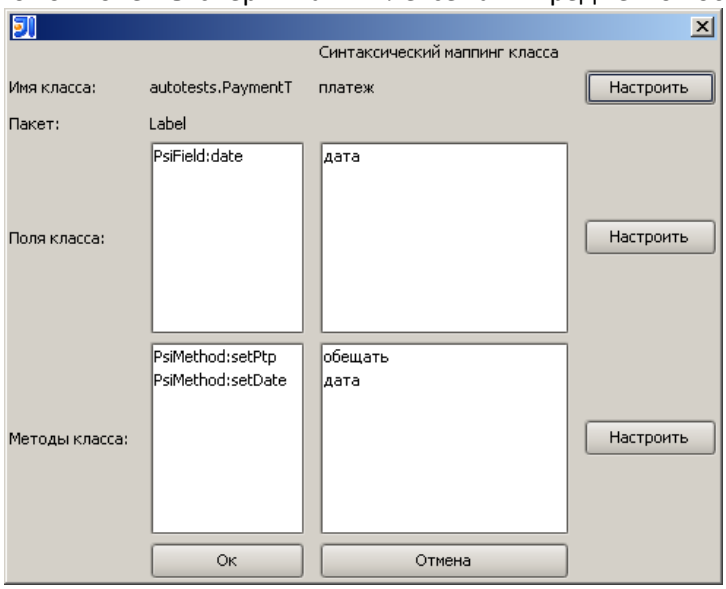

Рис. 8 Инструмент настройки классов на предметную область

3.2. Цветовая подсветка в редакторе и прямой доступ к классам

Обработанный текст получает в редакторе цветовую и схематичную раскраску. Распознанные термины и понятия выделяются полужирным шрифтом. Неизвестные понятия – полужирным красным. Нажатие правой кнопки мыши на выделенном слове открывает выпадающее меню, позволяющее редактировать существующее или создавать новое описание понятия.

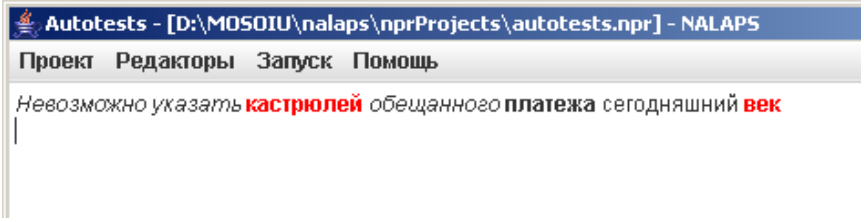

Рис. 9. Пример выделения нераспознанных объектов в тексте редактора

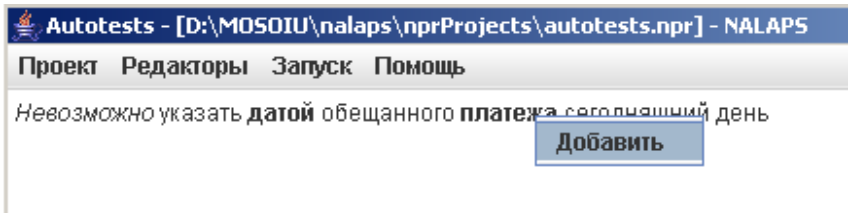

Рис. 10. Пример выделения распознаных объектов в тексте редактора и выпадающее меню.

При нажатии левой кнопки мыши на распознанный информационный объект происходит автоматический переход к коду соответствующего класса.

3.3. Редактор фреймовых структур базы знаний.

Для удобства разработчиков в систему добавлен инструмент редактирования фреймовой структуры, позволяющий описывать понятия предметной области из текстового редактора, не прибегая к прямому созданию классов языка Java – на основании данных структур классы генерируются автоматически. Таким образом, аналитик системы может описать с помощью данного инструмента понятие предметной области, а разработчик дополнить полученный класс необходимым кодом.

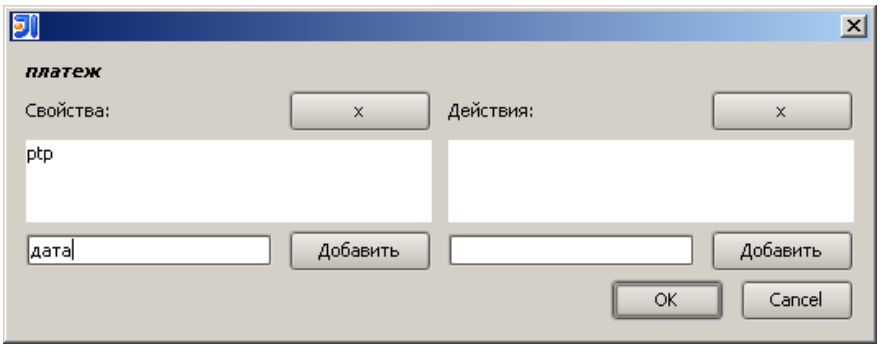

Рис. 11. Окно создания фреймовой структуры класса.

При генерации класса, система пытается автоматически определить тип полей по их названию, производя поиск по базе знаний концептов.

В целом интерфейс редактора Nalaps стремится быть понятным, интуитивным, и обладает функциями схожими с функциями IDE обычных языков программирования.

4. Сравнение с аналогами.

Систем автоматической генерации тестов известно несколько. Однако все они опираются на строго формализованную постановку задачи, представляемую средствами формальных языков для описания структуры и поведения системы (XML, UML, SDL или MSC) [11]. Так, в одних системах (1) требуется формализация задачи с помощью UML/Use Case diagramm с привлечением диаграмм действий (UML/Activity diagram), где на помощь приходят средства вроде Rational Rose. В других, как в системе TAT (Test Automation Training) (10), используется формальное описание тестов в виде MSC диаграмм и конфигурационных файлов в формате XML. Так, в системе TAT для создания цикла тестирования требуется выполнить множество операций:

- описать wrapper для тестируемой системы;
- создать файл конфигурации;

- создать формальное описание тестов в виде MSC-диаграмм;
- нарисовать MSC-диаграммы в MS Visio;
- сгенерировать с помощью макроса тестовый файл в формате MPR;
- настроить в конфигурационном файле проект теста;
- запустить и проанализировать тест.

Как видно, для получения автоматически сгенерированного теста требуется выполнение множества операций, умение работать в указанных прикладных системах, а также знание форматов и правил составления формализованных сценариев для данной системы тестирования. В случае же описания тестов на языке UML требуется хорошо представлять диаграмму классов тестируемого программного комплекса и владеть соответствующими программными инструментами.

Средств же, способных на основании текстового описания проблемы сгенерировать автоматический тест, в настоящий момент авторам неизвестно. Таким образом, система Nalaps в предложенном подходе представляется как возможное дополнение к существующим подобным утилитам, либо как полная их замена.

5. Выводы

Предложенный подход позволяет автоматически строить тесты для автоматизированного тестирования для проверки корректной работы программного комплекса в ситуациях, описываемых пользователями/тестировщиками системы на естественном языке.

Плюсами такого подхода являются:

- ускоренная разработка тестов вплоть до практически полной автоматизации процесса их создания для несложных текстовых описаний проверяемых ошибок;
- отсутствие необходимости предварительной формализации тестовых сценариев средствами UML, XML и др.; не требуется изучение форматов и правил описания сценариев тестов на промежуточных мета-языках и обучения работы в соответствующих редакторах;
- возможность полной автоматизации проверки критичных ситуаций, описание которых приходит в базу данных об ошибках, при достаточно хорошо описанной базе знаний предметной области;
- сравнительно простой механизм наполнения базы знаний о концептах предметной области.

Между тем, отметим и недостатки:

- сложные сценарии тестирования, описываемые на естественном языке, обязательно потребуют уточнения и более четкой формализации;
- практическая невозможность и нецелесообразность применения подхода для написания тестов, проверяющих внутреннюю логику работы отдельных блоков программы;
- необходимость поддержки системы лингвистами, что обусловлено использованием лингвистического процессора; впрочем, данная задача может быть решена путем включения более совершенных реализаций модуля лингвистического процессора.

Предложенный способ применения системы естественно-языкового программирования Nalaps в качестве системы автоматической генерации тестов при его развитии может составить альтернативу имеющимся инструментам, созданным для решения поставленной задачи. Проведенные опыты по использованию системы показали практическую применимость

описанного механизма генерации автоматических тестов на несложных текстовых описаниях ошибок.

Официальный web-сайт проекта: http://www.nalaps.ru

Текущая версия системы (0.3) доступна для загрузки: http://www.nalaps.ru/download

6. Список литературы

1. **Э. Дастин, Д. Рэшка, Д. Пол.** *Автоматизированное тестирование программного обеспечения. Внедрение, управление и эксплуатация.* М. : Изд-во «Лори», 2003.

2. **Калинов А.Я., Косачев А.С., Посыпкин М.А., Соколов А.А.** Автоматическая генерация тестов для графического пользовательского интерфейса по UML диаграммам действий. *Труды Института Системного Программирования РАН.* [В Интернете] [Цитировано: 29 01 2010 г.] http://www.citforum.ru/SE/testing/generation_uml/.

3. *Система естественно-языкового концептуального программирования Nalaps.* **Путилов Г.П., Лебедев А.С.** Киев : б.н., 2009. Материалы международной научной конференции Megaling 2009, 21-26 сентября 2009. стр. 65-66.

4. **Н.Н., Леонтьева.** *Автоматическое понимание текстов: системы, модели, ресурсы: учеб. пособие для студ. лингв. фак. вузов.* М. : Издательский центр «Академия», 2006.

5. *Представление машиностроительных моделей в базах знаний и персональные САПР.* **Тамм Б.Г., Тыугу Э.Х.** 1988 г., Вестник РАН. Журнал №5, стр. 39.

6. **Люгер, Джордж Ф.** *Искусственный интеллект: стратегии и методы решения сложных проблем, 4-е издание.* М. : Издательский дом "Вильямс", 2005.

7. *Редактор расширенных сетей переходов с графически интерфейсом пользователя .* **А.С., Лебедев.** М. : РГГУ, 2009. Компьютерная лингвистика и интеллектуальные технологии: По материалам ежегодной Международной конференции «Диалог 2009» (Бекасово, 27-31 мая 2009 г.). Вып. 8 (15). стр. 284-290.

8. *Средства настройки процессора Semantix на предметную область.* **Кузнецов И.П., Ефимов Д.А.** М. : РГГУ, 2009. Компьютерная лингвистика и интеллектуальные технологии: По материалам ежегодной Международной конференции «Диалог 2009» (Бекасово, 27-31 мая 2009 г.). Вып 8 (15). стр. 262-270.

9. About Selenium. *Selenium HQ Web application testing system.* [В Интернете] [Цитировано: 29 01 2010 г.] http://seleniumhq.org/about/.

10. Автоматическая генерация тестов на основе формального описания. *INTUIT.ru Интернет Университет Информационных технологий.* [В Интернете] [Цитировано: 29 01 2010 г.] http://www.intuit.ru/department/se/testing/21/ .