Правительство Российской Федерации
Федеральное государственное бюджетное образовательное учреждение
высшего профессионального образования
«Санкт-Петербургский государственный университет»

Кафедра Системного программирования

Самойлов Антон Сергеевич

Реализация поддержки платформой JetBrains DataPad сгенерированных специализированных языков и редакторов

Дипломная работа

Допущена к защите.
Зав. кафедрой:
д. ф.-м. н., профессор Терехов А. Н.

Научный руководитель:
ст. преп., магистр информационных технологий Григорьев С. В.

Рецензент:
Разработчик JetBrains s.r.o. Соломатов К. В.

Санкт-Петербург
2015

SAINT-PETERSBURG STATE UNIVERSITY

Department of Software Engineering

Anton Samoylov

Generated DSLs and editors support by JetBrains DataPad

Graduation Thesis

Admitted for defence.
Head of the chair:
professor Andrey Terekhov

Scientific supervisor:
senior lecturer, master of information technology Semen Grigorev

Reviewer:
Software Developer at JetBrains s.r.o. Konstantin Solomatov

Saint-Petersburg
2015

Оглавление

Введение

Высокая доступность информации является ключевым условием деятельности многих современных предприятий. Применение облачных систем хранения данных обеспечивает необходимый уровень доступа и предоставляет возможность использования распределенных вычислений, что позволяет задействовать серверные ресурсы, значительно превышающие по характеристикам пользовательские устройства [28]. Постоянный рост общего объема информации, в том числе за счет автоматически генерируемых данных, ставит перед специалистами новые задачи по их обработке и визуализации. Применение средств визуализации позволяет наглядно представить большие массивы информации и повысить эффективность их анализа.

Помимо обработки данных существует множество задач, решение которых, при наличии правильных инструментов, могут осуществлять специалисты, не владеющие навыками программирования. С подобными задачами лучше всего справляются предметно-ориентированные языки (Domain-Specific Languages, DSL) [19]. В рамках своей области DSL обладают гибкостью, сравнимой с языками общего назначения (C++, Java, Python), при этом DSL гораздо проще для освоения и использования.

Платформа JetBrains DataPad является инструментом, предназначенным для построения облачного сервиса, обеспечивающего как работу с DSL, так и проведение распределенных вычислений.

Разработка предметно-ориентированных языков является трудоемкой и рутинной задачей. Но, так как базовые принципы и блоки для построения языков и редакторов одинаковы, она может быть значительно упрощена за счет автоматической генерации модели DSL и его редактора из простого декларативного описания. В данной работе представлено решение, позволяющее использовать генерируемые DSL, а также их редакторы, в платформе JetBrains DataPad.

1. Платформа JetBrains DataPad

Платформа JetBrains DataPad представляет собой фреймворк для построения и запуска онлайн-сервиса, позволяющего работать с некоторым DSL и проводить вычисления, связанные с назначением этого языка. Кроме того, платформа позволяет строить интерактивные отчеты из результатов вычислений, доступные для демонстрации на индивидуальной веб-странице.

1.1. Общее описание

Платформа JetBrains DataPad базируется на двух фреймворках с открытым исходным кодом: JetBrains Mapper Framework[1] и JetBrains Projectional Editing Framework[2]. Первый фреймворк обеспечивает поддержку реактивного программирования [23]. Второй применяется для построения проекционных редакторов [24].

Интерфейс платформы разделен на две части (рис. 1). В левой части происходит работа с исходным кодом программы на выбранном DSL (далее язык, который используется в левой части будет называться внутренним DSL). В правой части отображаются результаты работы программы.

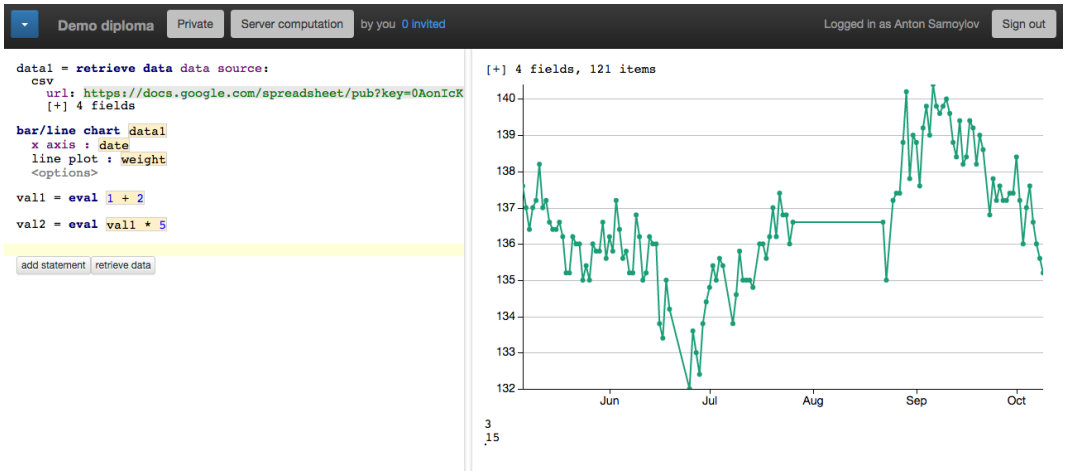

Рис. 1: Внешний вид веб-интерфейса JetBrains DataPad

[1] https://github.com/JetBrains/jetpad-mapper/
[2] https://github.com/JetBrains/jetpad-projectional/

Вычисления выполняются реактивно, то есть результаты пересчитываются по мере изменения кода программы пользователем.

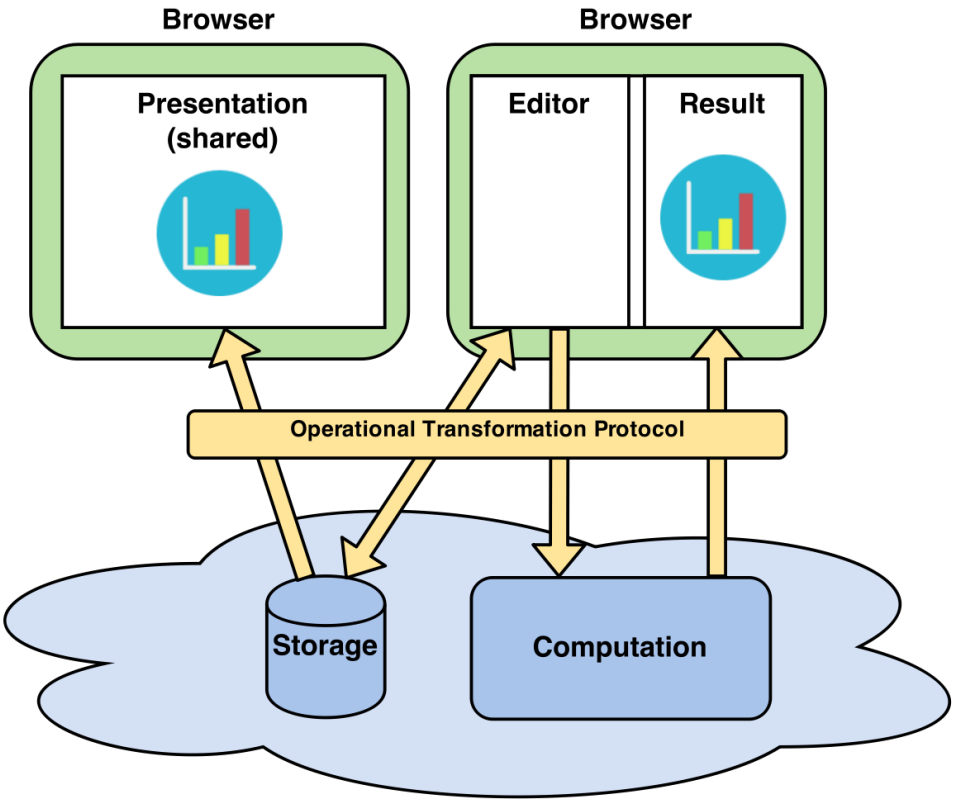

Рис. 2: Базовая схема работы платформы JetBrains DataPad

Работа с программой на внутреннем DSL происходит в порядке, установленном шаблоном проектирования Model-View-Controller [5]. Редактор для языка является представлением, а моделью является абстрактное синтаксическое дерево (Abstract Syntax Tree, AST [3]).

Хранение модельной части редактируемой программы, а также коммуникация клиентской части сервиса с сервером происходит посредством проприетарного протокола (далее OT-протокол), обеспечивающего Operational Transformation [7], метод разрешения конфликтов при совместном редактировании модели. Использование OT-протокола для передачи и хранения данных накладывает ряд ограничений на реализацию внутреннего DSL, в частности, необходимо, чтобы все классы объектов наследовали специальные, определенные в коде протокола базовые классы, а все поля инициализировались при помощи методов этих базовых классов. Благодаря использованию OT-протокола без дополни-

тельных усилий со стороны разработчика доступно совместное редактирование программ несколькими пользователями и поддержка операций undo/redo. Кроме того, использование низкоуровневого формата данных, предоставляемого OT-протоколом, решает задачу сохранения модели программы на диск.

Программный код платформы написан на языке Java. Для обеспечения работы в браузере используется фреймворк Google Web Toolkit[3] (далее GWT), который транслирует программный код на языке Java в код на языке JavaScript для исполнения в браузере. Также, возможен запуск платформы в качестве desktop приложения с помощью Java Swing. Это позволяет производить отладку логики программного кода, не специфичного для браузера, значительно быстрее, чем при использовании транслятора GWT. Вся логика приложений, не специфичная для конкретной платформы, может работать как на сервере, так и в браузере.

Для платформы DataPad командой проекта был разработан базовый язык для анализа и визуализации данных. В рамках этого языка доступна работа с данными из различных источников, построение графиков, наложение информации на карту и другие операции.

1.2. Структура языков

При работе с языками на платформе JetBrains DataPad происходит непосредственное редактирование AST. Такой подход, называемый проекционным редактированием (projectional editing), устраняет этап разбора (parsing) исходного кода программы для получения AST, что упрощает разработку новых языков. Кроме того, данный подход позволяет отображать значительную часть исходного кода на внутреннем DSL статическими блоками, недоступными для редактирования пользователем, которые генерируются автоматически и выступают в качестве опорного интерфейса. Это уменьшает сложность использования и порог входа, т.к. упрощает интерфейс взаимодействия с программой.

[3] `http://www.gwtproject.org/`

Например, как видно на рис. 1, в выражении для построения графика пользователю необходимо выбрать только переменную, хранящую таблицу с данными и два столбца: для осей x и y графика. Остальная часть выражения генерируется при его создании.

Программа на внутреннем DSL состоит из последовательных выражений (statements). Каждое выражение атомарно и имеет четко определенную цель: загрузить данные по ссылке в таблицу, нарисовать график, отфильтровать таблицу, посчитать среднее по выборке и т.д. Результатом работы каждого выражения является результат (output part): таблица, график, текст и т.д. Для некоторых выражений результат может оказаться пустым, в таком случае отображение для него будет опущено в правой части редактора.

1.3. Выполнение вычислений

Для того, чтобы получить результат выражения, нужно произвести некоторые вычисления. Такие вычисления в некоторых случаях могут требовать больших вычислительных ресурсов и машинного времени. В других случаях вычисления могут быть просты и не ресурсоемки. Поэтому платформа JetBrains DataPad предусматривает две стратегии вычислений:

1. на распределенном сервере;
2. в отдельном потоке браузера, используя механизм Web Workers[4].

На данный момент переключение стратегии происходит по нажатию кнопки в пользовательском интерфейсе, в будущем возможны реализации более сложных стратегий.

Выполнение вычислений выражения может зависеть от результатов вычислений для другого выражения. Например, отрисовка графика зависит от результата выражения, загружающего данные с удаленного сервера. Поэтому в платформе JetBrains DataPad вычисления на сервере выполняются с учетом зависимостей. Для этого каждому выра-

[4] `https://html.spec.whatwg.org/multipage/workers.html`

жению присваивается статус: $INVALID$, $WAITING$, $COMPUTING$, $VALID$ и вычисляется список его зависимостей. Статусы определяются текущим состоянием вычисления для соответствующего выражения. Обновление статусов просиходит автоматически по мере выполения вычислений (рис. 3).

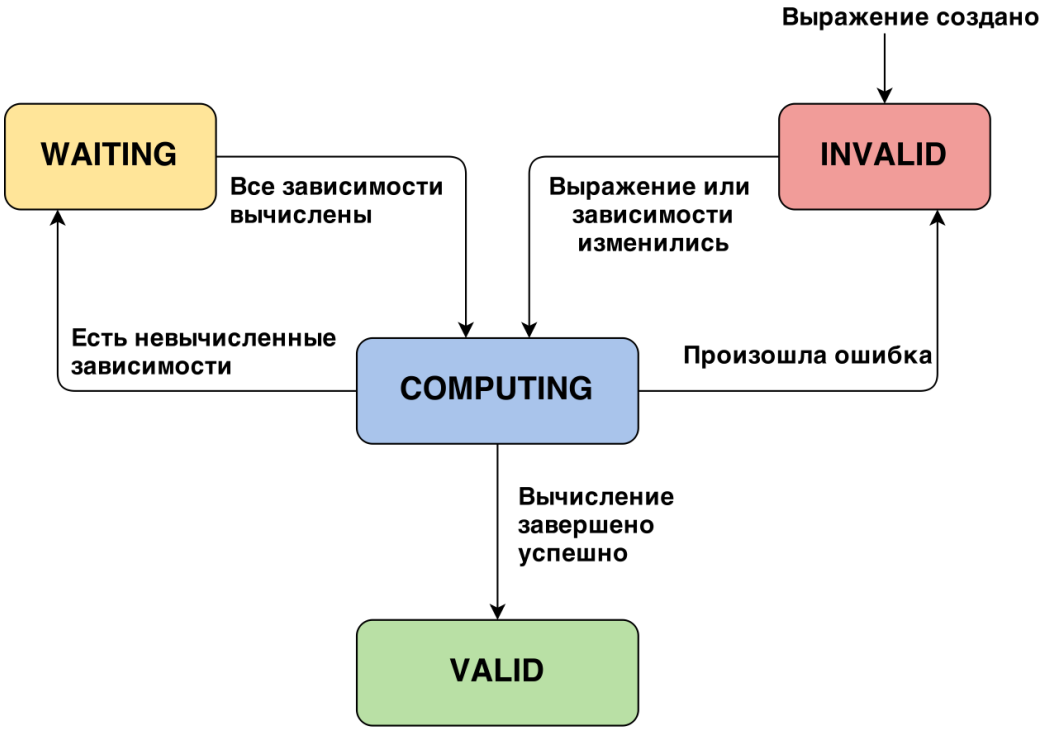

Рис. 3: Статусы вычислений

1.4. Набор вспомогательных компонент платформы

В качестве части платформы JetBrains DataPad доступна библиотека компонент, независимых от классов платформы, использующихся для создания веб-сервиса и работы со внутренним DSL. Эти компоненты могут быть применены как для разработки языка и поддерживающего его программного кода, так и для других задач, не связанных с использованием платформы. В библиотеке содержатся компоненты для:

- построения интерактивных графиков и визуализации данных с помощью векторного формата SVG [29];

- организации вычислений в браузере в отдельном потоке (Web Worker);
- извлечения и обработки данных из OLAP-кубов[5];
- работы с географическими картами, и других задач.

Большинство компонент используются для определения и обеспечения процесса вычислений базового языка.

[5] `http://olap.com/`

2. Обзор средств генерации языков

Для определения языков используются так называемые Language Workbenches [8] (далее LWB), инструменты для проектирования и реализации различных DSL. LWB снижают стоимость разработки новых языков и таким образом позволяют использовать DSL в сферах, где ранее применялись только языки общего назначения. Подход автоматизированного создания DSL давно применяется в индустрии. Примерами продуктов, созданных с применением такого подхода могут служить (в скобках указаны LWB, использованные для разработки):

- ПО для телефонов Nokia [20] (MetaEdit+ [16]);
- RISLA, DSL для финансовых продуктов [1] (ASF+SDF [17]);
- мониторы сердечного ритма и спортивные часы фирмы Polar [13] (MetaEdit+);
- WebDSL [25] и Mobl [10] для создания веб-сервисов и мобильных приложений соответственно (Spoofax [14]);
- mbeddr [27], язык для разработки встраиваемых систем (JetBrains MPS [26]).

Сравнение этих инструментов друг с другом с точки зрения эффективности или выразительной силы трудноосуществимо [15], поэтому ограничимся лишь обзором решений с точки зрения функциональных особенностей генерируемых DSL и редакторов и их применимости к задаче интеграции с платформой JetBrains DataPad.

Почти столь же долго как программисты создают языки, они также создают инструменты, облегчающие разработку языков и увеличивающие продуктивность их использования. Самым первым LWB вероятно был SEM; другие ранние LWB включают MetaPlex, Metaview, QuickSpec и MetaEdit. Для разработки графических языков в современном мире используются MetaEdit+, DOME, GME и QReal [30]. Текстовые нотации появились в таких инструментах, как Centaur, ASF+SDF

Meta-Environment, LRC и Lisa. Эти системы изначально базировались на инструментах для формальной спецификации языков общего назначения [9], тем не менее, многие из них успешно использовались для создания практических DSL [19]. LWB, поддерживающие текстовые нотации: Rascal [18], Spoofax, и Xtext [6]; могут рассматриваться, как переемники этих систем, использующие наработки в сферах развития редакторов и встроенных систем разработки. В то же время, проекционные LWB, такие как MPS и Intentional возрождают и развивают старую идею структурных редакторов [22], предоставляя возможность смешивать произвольные нотации.

Для интеграции с платформой DataPad генерируемый редактор должен быть проекционным, что сильно ограничивает выбор из представленных инструментов. Решение JetBrains MPS[6] позволяет генерировать проекционные редакторы, а также интегрируется со средой разработки Intellij IDEA[21], что позволяет создавать язык и поддерживающий его код, без необходимости переключения между инструментами. Данные характеристики послужили в пользу выбора JetBrains MPS для генерации интегрируемых языков и редакторов в рамках данной работы. Подробное описание решения приведено в главе ниже.

[6] https://www.jetbrains.com/mps/

3. Определение языков и редакторов при помощи JetBrains MPS

Проект JetBrains MPS позволяет работать с различными языками проектирования и реализации DSL (language design languages). В рамках данной дипломной работы интересны два из них: *Concept Language* и *Notation Language*. Они поддерживаются плагином для среды разработки Intellij IDEA [21], что упрощает процесс работы с ними, избавляя от необходимости установки полноценной среды разработки JetBrains MPS. Стоит отметить, что данные языки позволяют определять более широкий класс DSL, чем поддерживается платформой JetBrains DataPad.

Для осуществления генерации программного кода к языкам "подключаются" генераторы, специфичные для типа выходного кода. Таким образом декларативное определение языков и редакторов может производить различный программный код, в том числе код на разных языках, в зависимости от подключенного редактора. В рамках данной работы использовался генератор, производящий код с использованием фреймворка JetBrains Projectional Editing Framework.

3.1. Concept Language

Concept Language — это язык для задания модели внутреннего DSL. Модель состоит из самостоятельных единиц, т.н. концептов. Далее приведен список характеристик, которые возможно указать для каждого концепта.

- Родительский концепт, задающий иерархию и позволяющий унаследовать все свойства и дочерние концепты от родительского концепта. Механизм наследования концептов схож с механизмом наследования в объектно-ориентированных языках[7] . Множественное наследование, как и в языке Java, запрещено.

[7] http://en.wikipedia.org/wiki/Inheritance_(object-oriented_programming)

- Родительский класс для соответствующего сгенерированного класса, позволяющий наследовать концепты от классов, описанных в Java-коде. Концепт не может одновременно иметь предка и указанный родительский класс для генерируемого класса.
- Типизированные свойства, имеющие примитивные типы (`string`, `int`, `boolean`). Свойства могут быть так же унаследованы от родительского класса при указании аннотации *@inherit* с указанием имени наследуемого свойства.
- Ссылки на дочерние концепты. Возможны ссылки как на единичные концепты, так и на списки.
- Абстрактность. Данное свойство определяет, что сгенерированный класс не может быть инстанциирован, то есть создание экземпляров этого класса невозможно.

Кроме того, возможна декларация *enum* концепта, каждый экземпляр которого является элементом из списка, заранее заданного в определении.

В результате работы генератора для языка *Concept Language* из определения каждого концепта генерируется отдельный класс на языке Java. В некоторых случаях из одного определения может генерироваться более одного Java-класса. Задание концептов может быть рассмотрено на примере (см. Listing 1).

Listing 1 Задание концепта PrintStatement

```
concept PrintStatment extends <default> {
    string text;
} <no corresponding class>
```

В данном примере определяется простой концепт с одним свойством *text* типа `string`. Он не имеет предков, как не имеют их и сгенерированные им классы.

Все декларируемые свойства и дочерние концепты будут доступны из экземпляра сгенерированного класса через методы доступа, что позволяет использовать его в сопровождающем коде.

3.2. Notation Language

Notation Language — это язык для декларативного определения внешнего вида и поведения редактора для генерируемого DSL. Язык позволяет концепту, определенному с помощью *Concept Language*, сопоставить нотацию, задающую его визуальное представление. Каждой нотации соответствует ровно один концепт. Ниже приведен список элементов и характеристик, которые можно определить для нотации.

- Гибридность редактора. Гибридный редактор совмещает в себе черты проекционного и текстового метода работы. Он может находиться в невалидных состояниях, для него задается грамматика, используемая при разборе и набор текста происходит линейно. При этом структура редактируемого выражения, аналогично проекционному подходу, состоит из отдельных токенов, а не символов.

- Классы аналогичные классам CSS [4]. Для классов могут быть определены соответствующие стилистические свойства, такие как цвет текста или фона, и свойства поведения.

- Статические части, представимые в виде текстовых блоков, недоступных для редактирования пользователем.

- Ссылки на дочерние элементы соответствующего концепта, которые будут отображаться используя собственные нотации.

Визуальное представление для нотации может генерироваться динамически в зависимости от пользовательского ввода. То есть, если у концепта есть несколько наследников, имеющих структурно разные нотации, выбор в пользу одной из них будет осуществляться на основе действий пользователя.

Listing 2 Задание нотации (отображения) PrintStatement

```
1 notation PrintStatement ::
2     "print text:" text ;
```

В примере выше (см. Listing 2) определяется нотация для концепта `PrintStatement` (см. Listing 1). Выражение будет отображаться как статическая (недоступная для редактирования) строка "*print text* : " и поле для ввода текста, соответствующего свойству *text* концепта.

4. Постановка задачи

Целью данной работы является обеспечение возможности использования DSL и их редакторов, сгенерированных при помощи языков *Concept Language* и *Notation Language* в платформе JetBrains DataPad.

Для достижения поставленной цели были сформулированы задачи, приведенные ниже.

- Спроектировать архитектуру интеграции сгеренированных языков и редакторов с платформой JetBrains DataPad.
- Произвести внедрение точек расширения в код платформы, необходимых для обеспечения работоспособности всех возможностей редактора платформы для языков, сгенерированных с помощью *Concept Language* и *Notation Language*.
- Создать демонстрационный язык, полностью определенный с помощью языков *Concept Language* и *Notation Language*. В языке должны быть представлены выражения для демонстрации работоспособности всех возможностей редактора платформы JetBrains DataPad в применении к сгенерированному языку.

Описание решения поставленных задач приведено в последующих главах данной работы.

5. Расширение платформы

Для решения задачи расширения платформы JetBrains DataPad в рамках данной работы были выделены подзадачи, перечисленные ниже.

- Провести структурный рефакторинг платформы для создания модульной структуры и уменьшения связности компонент.
- Реализовать механизм расширений платформы (плагинов), который бы позволил определять точки расширения для различных классов. В генерируемых языках крайне важна модульность [11], то есть возможность составлять язык из нескольких частей, поэтому реализованная система должна позволять модульно расширять языки с помощью плагинов.
- Произвести внедрение точек расширения в программный код платформы для обеспечения работоспособности всех возможностей редактора для подключаемых языков.

Далее приведены подробные описания решений для каждой подзадачи. Общая схема процесса интеграции сгенерированных языков показана ниже (рис. 4).

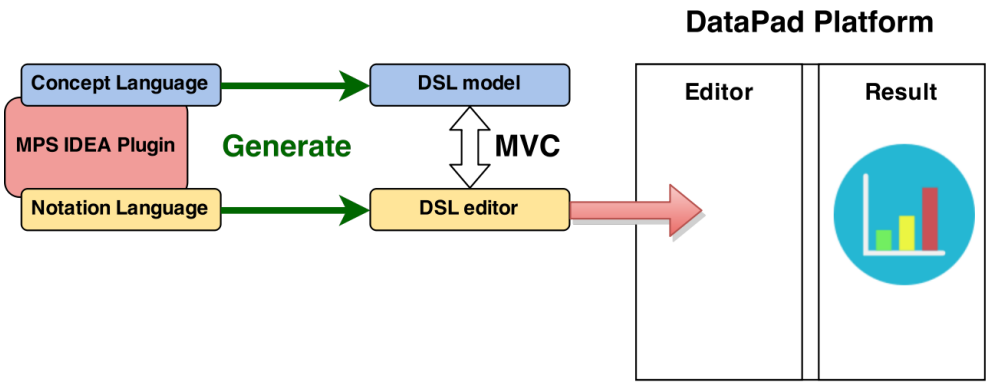

Рис. 4: Схема интеграции сгенерированных языков и редакторов

5.1. Рефакторинг архитектуры платформы JetBrains DataPad

Для упрощения работы с программным кодом платформы JetBrains DataPad в рамках данной дипломной работы был проведен структурный рефакторинг платформы. Далее приведен список модулей, выделенных в результате рефакторинга.

- *lang* — части платформы относящиеся к поддержке моделей языков и редакторов для них. Могут быть использованы независимо от остальной платформы. Содержит следующие подмодули:
  - *model*, описание модельной части языка, т.е. внутренней структуры выражений и их свойств. Не имеет связи с представлением языка и не зависит от подмодуля *editor*;
  - *editor*, описание визуальной части языка, т.е. представления выражений в редакторе и его поведение: реакции на пользовательский ввод, отображение ошибок, навигация и т.д. Зависит от подмодуля *model*, но не зависит от модуля вычислений, *lang−runtime*.
- *lang−runtime* — вычислительная поддержка языков. Здесь определяются правила вычисления выражений для получения результата. Так как программный код совместим с GWT, он может исполняться как на сервере, так и на клиентском компьютере в браузере. Модуль *lang−runtime* не зависит от модуля *editor*.
- *app* — части платформы относящиеся к непосредственно созданию приложения. Здесь находятся стартовые точки для сервера, клиента, desktop-приложения, а также инициализация протоколов, баз данных, управление потоками.

Целью рефакторинга являлось разделение логики на компоненты, которые могут быть использованы независимо друг от друга. Кроме того, новая структура проекта позволяет изолировать код от ненужных зависимостей, например, код модели языка от кода представления.

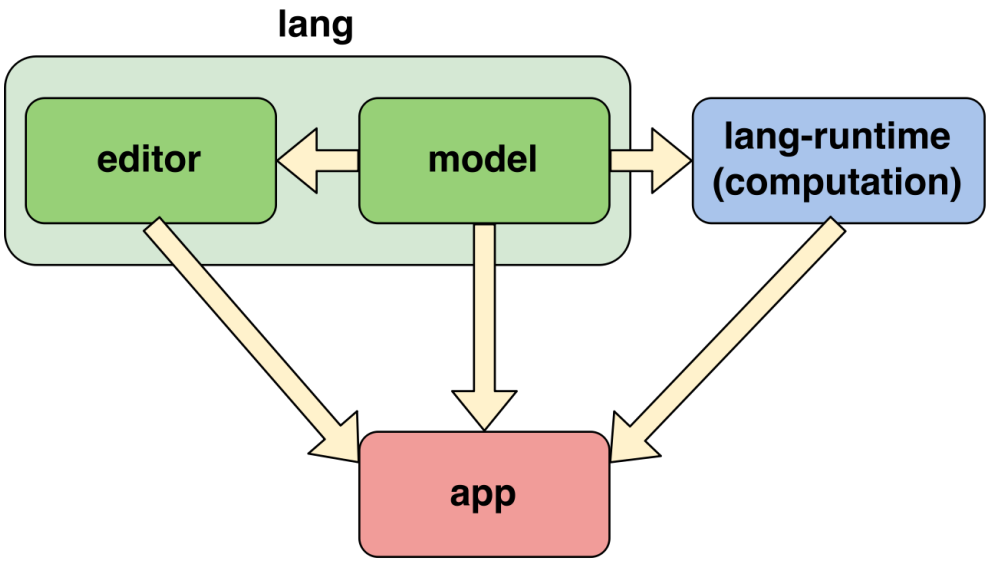

Рис. 5: Взаимное отношение компонент платформы

5.2. Система расширений платформы

Система расширений является инструментом для повышения гибкости платформы, путем внедрения точек расширения.

Архитектура системы расширений платформы схожа с архитектурами расширений сред разработки Intellij IDEA [12] и Eclipse [2].

Центральной частью системы расширений платформы DataPad является интерфейс `PluginConfiguration` (см. Listing 3), который представляет собой объединенный интерфейс доступа ко всем точкам расширения (рис. 6). Точки расширения могут быть множественными (в этом случае возвращается объединение по всем подключенным плагинам) или сингулярными, для которых может быть определен только один экземпляр во всех подключенных плагинах.

Listing 3 Интерфейс PluginConfiguration

```
1 public interface PluginConfiguration {
2   public abstract <ExtT> Collection<ExtT>
3     getExts(ExtensionPoint<ExtT> ep);
4   public abstract <ExtT> ExtT getExt(ExtensionPoint<ExtT> ep);
5 }
```

Точки расширения (`ExtensionPoint`) представляют собой типизированный ключ, который используется для извлечения в коде плат-

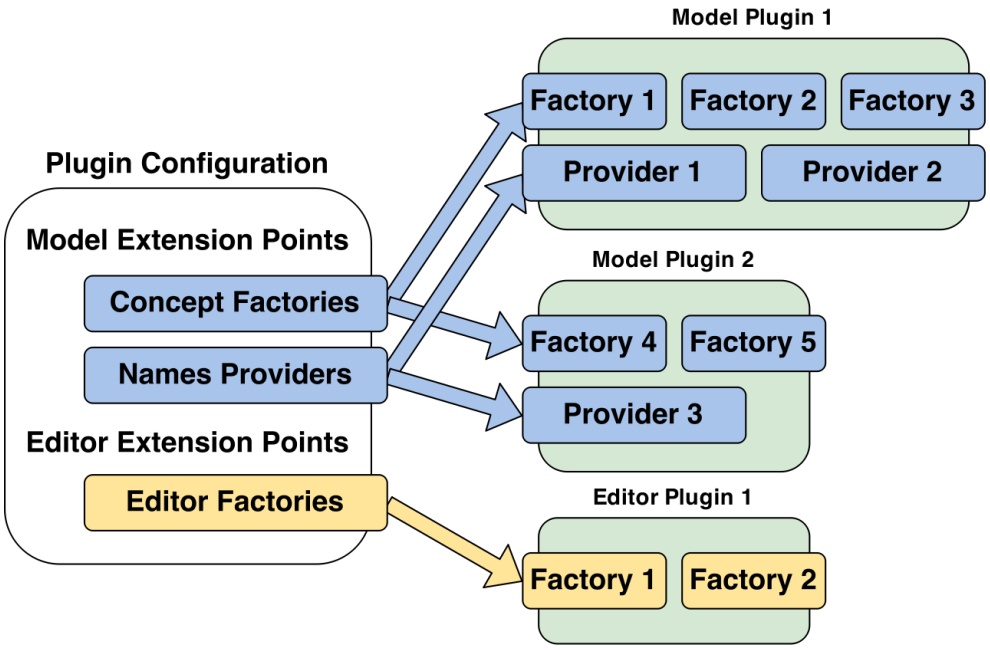

Рис. 6: Архитектура расширений платформы

формы типизированных расширений из конфигурации плагинов. Конфигурация строится из отдельных расширений (см. Listing 4).

Listing 4 Интерфейсы Plugin и PluginContext

```
public interface Plugin {
  void install(PluginContext context);
}

public interface PluginContext {
  public <ExtT> void add(ExtensionPoint<ExtT> ep, ExtT... exts);
}
```

Создание конфигурации плагинов осуществляется с помощью статического метода класса `PluginConfigurationBuilder` (см. Listing 5), реализующего шаблон проектирования `Builder` [5].

Типичное определение плагина представлено ниже (см. Listing 6). На листинге плагин добавляет расширения, используя точки расширения, определенные в статическом классе `CompuationEPs`, как ключи.

Использование расширений в коде платформы может быть продемонстрировано на примере (см. Listing 7). Цикл перебирает все расширения (в данном случае фабрики), определенные в подключенных плагинах и совершает над каждой некоторые действия.

Listing 5 Класс PluginConfigurationBuilder

```
public class PluginConfigurationBuilder {
  public static PluginConfiguration build(Plugin... plugins) {
    final Map<ExtensionPoint<?>, List<?>> context = new HashMap<>();
    for (Plugin p : union(CORE_PLUGINS, plugins)) {
      p.install(new PluginContext() {
        @Override
        public <ExtT> void add(ExtensionPoint<ExtT> ep,
                               ExtT... exts) {
          if (exts.isEmpty()) return;
          if (!context.containsKey(ep))
            context.put(ep, new ArrayList<ExtT>());
          context.get(ep).addAll(exts);
        }
      });
    }
    return new PluginConfiguration() {
      @Override
      public <ExtT> Collection<ExtT> getExts(
                                          ExtensionPoint<ExtT> ep) {
        if (!context.containsKey(ep)) return EMPTY_LIST;
          return context.get(ep);
      }
      @Override
      public <ExtT> ExtT getExt(ExtensionPoint<ExtT> ep) {
        Collection<ExtT> exts = getExts(ep);
        assert exts.size() == 1;
        return exts.iterator().next();
      }
    };
  }
}
```

Ниже перечислены характеристики реализованной системы расширений.

- Модульность: расширения не определяются в одном централизованном месте, а собираются вместе только на этапе создания конфигурации, что позволяет переиспользовать плагины и создавать более гибкие архитектуры. В частности это свойство используется непосредственно в коде самой платформы, где определены плагины для частей, общих для всех языков, что значительно упрощает код интеграции точек расширения.

Listing 6 Типичное определение плагина

```
public class CoreComputationPlugin implements Plugin {
  @Override
  public void install(PluginContext context) {
    context.add(ComputationEPs.EVALUATOR_FACTORIES,
      EvaluatorFactory.CORE_EVALUATOR_FACTORY);
    context.add(ComputationEPs.COMPUTATION_FACTORIES,
      ComputationFactory.CORE_COMPUTATION_FACTORY);
    context.add(ComputationEPs.OUTPUT_FACTORIES,
      OutputFactory.CORE_OUTPUT_FACTORY);
  }
}
```

Listing 7 Типичное использование расширений

```
for (ComputationFactory factory :
        pluginConf.getExts(ComputationEPs.COMPUTATION_FACTORIES)) {
  Computation<Statement> comp = factory.create(/*args*/);
  // do something with comp
}
```

- Изолирование контекстов для точек расширения. Например, ключи точек расширения для редактора недоступны в контексте описания модели, что позволяет избежать ошибок при написании кода. Обычно точки расширения, действующие в одном контексте определяются централизовано (см. Listing 8).

Listing 8 Пример определения точек расширения

```
public class ComputationEPs {
  public static final
    ExtensionPoint<EvaluatorFactory> EVALUATOR_FACTORIES =
    new ExtensionPoint<>("EvaluatorFactories");
  // other EPs definition
}
```

- Отдельные плагины могут поставляться в качестве независимых компонент к платформе, сразу готовых к подключению.

- Расширение языка новыми выражениями возможно осуществить за счет добавления плагинов для описания новых выражений и

переиспользования плагинов для расширяемого языка. В сочетании с предыдущим пунктом это позволяет организовать библиотеку расширений для некоторого базового языка и собирать новые языки из частей подобно конструктору.

- Возможно определение новых точек расширения на уровнях ниже уровня платформы. Это позволяет реализовывать более гибкие языки. Например, если есть язык, имеющий выражение для загрузки данных с удаленного источника, разработчики могут определить точку расширения для видов источника данных (URL для csv-таблицы, Google SpreadSheet, OLAP-куб и т.д.), которой не существовало в оригинальной платформе.

6. Интеграция сгенерированных языков и редакторов

В данной главе описываются точки расширения, которые были интегрированы в код платформы JetBrains DataPad для обеспечения работоспособности возможностей редактора при работе со сгенерированными языками. Для большинства из точек расширений, представленных в секциях ниже, существуют стандартные экземпляры расширений, общие для всех языков, например пустое выражение (`EmptyStatement`). Они определяются в коде платформы, и обычно описываются в виде исключительных случаев в программном коде (hardcoded). После внедрения системы расширений в рамках данной работы были созданы стандартные (core) плагины, подключаемые по умолчанию, которые позволяют изолировать определение таких объектов от их использования. Кроме того, такой подход позволяет унифицировать работу с расширениями, избавившись от исключительных случаев.

6.1. Интеграция моделей и редакторов языков

Так как данные между клиентом и сервером передаются по низкоуровневому протоколу, существует необходимость восстановления оригинальных данных по низкоуровневому представлению. Для этого на клиенте и сервере определяется две точки расширений для фабрик (шаблон проектирования `Factory` [5]) выражений (`Statments`) и результатов (`OutputParts`), восстанавливающих оригинальные классы. Эти точки расширений находятся в контексте модели языка.

Для обеспечения автодополнения в рамках данной работы были решены две задачи: предоставление списка имен выражений для их создания и контекстнозависимое автодополение внутри выражения. Для решения первой задачи была внедрена точка расширения для списка выражений и их отображаемых имен. Вторая задача была выполнена путем реализации класса `Resolver`, позволяющего по экземпляру выражения и роли (role), определяющей контекст внутри выражения, по-

лучить список автодополнения. Передавая различные роли, такие как *data* или *xVar* в случае `BarLineChartStatement`, можно получить различные списки автодополнения.

Кроме того, для обеспечения работоспособности различных возможностей редактора, таких как "Go to definition" и проверки валидности введенных данных, необходимо решать задачу разрешения имен (*Name resolving*). Для этого был реализован класс, позволяющий по имени переменной найти соответствующее выражение. Этот класс активно применяется в поддерживающем коде и коде вычислений.

6.2. Интеграция вычислительных частей языков

Для каждого выражения в модельной части языка необходим вычислитель (evaluator), который сможет преобразовать данные, указанные внутри выражения, в результат. Так как вычислители определяются на уровне конкретного языка, а не платформы, в код платформы была интегрирована точка расширения, позволяющая предоставлять фабрики для вычислителей через механизм плагинов.

Вычислители могут возвращать произвольные объекты, в том числе непригодные для передачи по OT-протоколу без дополнительных преобразований. Кроме того, во время вычислений могут возникать ошибки и исключения. Для упрощения работы с такими ситуациями в код платформы была интегрирована точка расширения, позволяющая определить фабрики для обработки результатов или исключений вычислителей.

Некоторые вычисления, например вызов к удаленному серверу с целью получения данных, требуют асинхронного выполнения. Для управления асинхронностью и исполнения программы в отдельных потоках, в код платформы была внедрена точка расширения с фабрикой оберток (шаблон `Wrapper`), которые решают задачу определения стратегии выполнения вычислений.

Все перечисленные в данном разделе точки расширения находятся в контексте вычислений.

7. Демонстрационный язык

Для демонстрации работоспособности расширения платформы в рамках данной дипломной работы был разработан демонстрационный DSL, далее — демо-язык.

7.1. Генерируемые компоненты

Демо-язык по большей части совпадает с базовым языком DataPad, за исключением вычислительной части, которая присутствует только в объеме, необходимом для демонстрации работоспособности возможностей редактора платформы. Далее приведен список реализованных выражений языка с описанием демонстрируемых возможностей платформы. Описание выражений, не демонстрирующих ранее не обозначенных возможностей, опущено.

- `RetrieveDataStatement` — выражение для загрузки данных, демонстрирует следующие возможности:
  - Обновление мета-информации, полученной после выполнения вычислений на сервере вычислений, которая может быть использована для работы редактора выражений. В данном случае, мета-информация о полях таблицы после загрузки используется для добавления элементов в список автодополнения.
  - Динамическая структура, зависящая от пользовательского ввода: в зависимости от выбора типа источника (*csv* или *other*) в поле *type*, меняется структура оставшейся части выражения.
- `BarLineChartStatement` (рис. 7) — выражение для визуализации данных в виде графиков. Демонстрируемые возможности:
  - Go to declaration: возможность сфокусироваться на определении переменной при нажатии специальной комбинации клавиш на ссылке на эту переменную.

– Отображение и редактирование списка дочерних концептов, а именно перечисления столбцов таблицы, которые нужно отобразить на графике.

– Выбор типа графика из списка заранее заданных вариантов, то есть демонстрируется отображение и работа с enum-концептом.

– Автодополнение, в том числе для элементов, добавленных после проведения вычислений на сервере вычислений (см. `RetrieveDataStatment`).

– Список элементов для автодополнения зависит от контекста вызова: в списке автодополнения для поля *data* отображаются только переменные полученные как результат работы `RetrieveDataStatement`, в то время как для поля *x variable* отображаются названия столбцов из соответствующей таблицы.

```
bar/line chart data1
  x axis : brand
  line plot : starting_price, units_sold
```

Рис. 7: Пример `BarLineChartStatement`

- `EvalStatement` — выражение для вычисления математического выражения. Поддерживаются базовые операции ($+$, $-$, $\times$, $/$), с корректными приоритетами, скобки, функции (*sin* и *cos*) и обращение к результатам вычисления других `EvalStatements`. Данное выражение демонстрирует возможности гибридного редактора, который в отличии от проекционного может находиться в невалидных состояниях и для которого можно задать грамматику.

Модель демо-языка и редактор для него полностью генерируются. Определение языка и редактора для него на языках *Concept Language* и *Notation Language* занимают примерно 500 строк, размер генерируемого кода составляет около 17000 строк кода на языке Java.

7.2. Java компоненты

Для обеспечения работоспособности возможностей редактора не во всех случаях достаточно сгенерированного кода. Поэтому, кроме декларативного определения демо-языка на языках *Concept Language* и *Notation Language*, в рамках данной дипломной работы были реализованы компоненты поддержки на языке Java, приведенные ниже.

- Плагины, добавляющие в конфигурацию элементы для точек расширения:
  - `ModelPlugin`,
  - `EditorPlugin`,
  - `ComputationPlugin`,
  - `ServerPlugin`.
- Демонстрационные вычислители: для большинства выражений это просто текст, показывающий работоспособность вычислительного сервера, для `EvalStatement` — полноценный вычислитель, рассчитывающий и присваивающий переменной результат выражения с учетом применения функций и ссылок на другие переменные.
- Модуль для разрешения (resolving) ссылок на переменные. Используется для валидации кода программы, добавления элементов в автодополнение и определения зависимостей при вычислении.
- Intentions actions — контекстные действия: проставление заранее заданных ссылок в поле *url* в `RetrieveDataStatement`.

Заключение

В рамках данной дипломной работы были решены задачи по обеспечению работоспособности сгенерированных языков и редакторов на платформе JetBrains DataPad.

- Спроектирована архитектура интеграции сгенерированных языков и редакторов с платформой.
- Произведена интеграция точек расширения в код платформы. Для достижения этой цели был также проведен структурный рефакторинг программного кода фреймворка с целью создания модульной структуры и уменьшения связности компонент и спроектирована система расширений. Реализованная система расширений обладает такими преимуществами как:
  - модульность,
  - изолированность контекстов для точек расширения,
  - возможность поставлять отдельные плагины в качестве независимых компонент,
  - возможность дополнения языка новыми выражениями,
  - возможность определения новых точек расширения на уровнях ниже уровня платформы.
- Создан демонстрационный язык показывающий работоспособность всех возможностей редактора платформы. Модель языка и редактор для него полностью генерируются из декларативного описания на языках *Concept Language* и *Notation Language*.

Таким образом, в рамках работы была создана инфраструктура для интеграции различных DSL с платформой JetBrains DataPad. В данный момент результаты используются для разработки внутренних инструментов компании JetBrains, а в будущем планируется более широкое их применение.

Список литературы

[1] Arnold B.R.T, Deursen A. Van, Res M. An Algebraic Specification of a Language for Describing Financial Products // ICSE-17 Workshop on Formal Methods Application in Software Engineering. — IEEE, 1995. — P. 6–13.

[2] Bolour Azad. Notes on the Eclipse Plug-in Architecture. — 2003. — https://eclipse.org/articles/Article-Plug-in-architecture/plugin_architecture.html.

[3] Compilers: Principles, Techniques, and Tools / Alfred V. Aho, Monica S. Lam, Ravi Sethi, Jeffrey D. Ullman. — Second edition. — Addison Wesley, 2006. — ISBN: 978-0321486813.

[4] Consortium World Wide Web. Cascading Style Sheets Level 2 Revision 1 (CSS 2.1) Specification. — 2011. — http://www.w3.org/TR/CSS2/.

[5] Design Patterns, Elements of Reusable Object-Oriented Software / Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides. — Addison-Wesley Professional, 1994. — ISBN: 0201633612.

[6] Eysholdt Moritz, Behrens Heiko. Xtext: Implement Your Language Faster Than the Quick and Dirty Way // Proceedings of the ACM International Conference Companion on Object Oriented Programming Systems Languages and Applications Companion. — OOPSLA '10. — New York, NY, USA : ACM, 2010. — P. 307–309.

[7] Formal Techniques for Networked and Distributed Systems - FORTE 2005 / Abdessamad Imine, Pascal Molli, Gérald Oster, Michaël Rusinowitch / Ed. by Farn Wang. — Springer Berlin Heidelberg. — P. 411–427.

[8] Fowler Martin. Language workbenches: The killer-app for domain specific languages? — 2005. — http://www.martinfowler.com/articles/languageWorkbench.html.

[9] Heering Jan, Klint Paul. Semantics of Programming Languages: A Tool-Oriented Approach // CoRR. — 1999. — Vol. cs.PL/9911001.

[10] Hemel Zef, Visser Eelco. Declaratively Programming the Mobile Web with Mobl // SIGPLAN Not. — 2011. — . — Vol. 46, no. 10. — P. 695–712.

[11] Implementing Modular Domain Specific Languages and Analyses / Daniel Ratiu, Markus Voelter, Zaur Molotnikov, Bernhard Schaetz // Proceedings of the Workshop on Model-Driven Engineering, Verification and Validation. — MoDeVVa '12. — New York, NY, USA : ACM, 2012. — P. 35–40.

[12] Jemerov Dmitry. IntelliJ IDEA Plugin Structure // JetBrains s.r.o. — 2015. — https://goo.gl/bsoXzq.

[13] Kärnä Juha, Tolvanen Juha-Pekka, Kelly Steven. Evaluating the use of domain-specific modeling in practice // Proceedings of the 9th OOPSLA workshop on Domain-Specific Modeling. — 2009.

[14] Kats Lennart C.L., Visser Eelco. The Spoofax Language Workbench: Rules for Declarative Specification of Languages and IDEs // SIGPLAN Not. — 2010. — . — Vol. 45, no. 10. — P. 444–463.

[15] Kelly Steven. Empirical Comparison of Language Workbenches. — 2013. — http://www.dsmforum.org/events/dsm13/Papers/Kelly.pdf.

[16] Kelly Steven, Lyytinen Kalle, Rossi Matti. MetaEdit+ A fully configurable multi-user and multi-tool CASE and CAME environment // Advanced Information Systems Engineering / Ed. by Panos Constantopoulos, John Mylopoulos, Yannis Vassiliou. — Springer Berlin Heidelberg, 1996. — Vol. 1080 of Lecture Notes in Computer Science. — P. 1–21.

[17] Klint P. A meta-environment for generating programming environments // Algebraic Methods II: Theory, Tools and Applications / Ed. by J.A. Bergstra, L.M.G. Feijs. — Springer

Berlin Heidelberg, 1991. — Vol. 490 of Lecture Notes in Computer Science. — P. 105–124.

[18] Klint Paul, van der Storm Tijs, Vinju Jurgen. EASY Meta-programming with Rascal // Proceedings of the 3rd International Summer School Conference on Generative and Transformational Techniques in Software Engineering III. — GTTSE'09. — Berlin, Heidelberg : Springer-Verlag, 2011. — P. 222–289.

[19] Mernik Marjan, Heering Jan, Sloane Anthony M. When and how to develop domain-specific languages // ACM Computing Surveys (CSUR). — 2005. — December. — Vol. 37, no. 4. — P. 316–344.

[20] MetaEdit+ revolutionized the way Nokia develops mobile phone software // MetaCase. — 2007. — http://www.metacase.com/cases/nokia.html.

[21] Pech Vaclav. Using MPS inside IntelliJ IDEA. — 2013. — https://goo.gl/JkzCxC.

[22] Programming environments based on structured editors: The MENTOR experience : Rep. / DTIC Document ; Executor: Veronique Donzeau-Gouge, Gerard Huet, Gilles Kahn, Bernard Lang : 1980.

[23] Salvaneschi Guido, Mezini Mira. Towards Reactive Programming for Object-oriented Applications. — 2014. — ISBN: 978-3-642-55099-7. — http://www.guidosalvaneschi.com/attachments/papers/2014_Towards-Reactive-Programming-for-Object-Oriented-Applications_pdf.pdf.

[24] Voelter Markus, Siegmund Janet, Berger Thorsten, Kolb Bernd. Towards User-Friendly Projectional Editors. — 2014. — http://mbeddr.com/files/projectionalEditing-sle2014.pdf.

[25] Visser Eelco. WebDSL: A Case Study in Domain-Specific Language Engineering // Generative and Transformational Techniques in Software Engineering II / Ed. by Ralf Lämmel, Joost Visser,

João Saraiva. — Springer Berlin Heidelberg, 2008. — Vol. 5235 of Lecture Notes in Computer Science. — P. 291–373.

[26] Voelter M., Pech V. Language modularity with the MPS language workbench // Software Engineering (ICSE), 2012 34th International Conference on. — 2012. — June. — P. 1449–1450.

[27] mbeddr: instantiating a language workbench in the embedded software domain / Markus Voelter, Daniel Ratiu, Bernd Kolb, Bernhard Schaetz // Automated Software Engineering. — 2013. — Vol. 20, no. 3. — P. 339–390.

[28] A view of cloud computing / Michael Armbrust, Armando Fox, Rean Griffith et al. // Communications of the ACM. — 2010. — April. — Vol. 53, no. 4. — P. 50–58.

[29] Самойлов А.С. Реализация поддержки SVG в фреймворке JetBrains JetPad. — 2014. — http://goo.gl/DRKLcG.

[30] Т.А. Брыксин, Ю.В. Литвинов. Технология визуального предметно-ориентированного проектирования и разработки ПО QReal // Материалы второй научно-технической конференции молодых специалистов «Старт в будущее», посвященной 50-летию полета Ю.А. Гагарина в космос. — 2011. — P. 222–225.