Реферат

Основная идея проекта заключается в том, чтобы предоставить разработчикам программного обеспечения, привыкшим к функционалу для редактирования текста, предоставляемому текстовым редактором GNU Emacs, возможность использовать все преимущества интеллектуальной среды разработки IntelliJ IDEA, не испытывая неудобств.

В результате работы над проектом были исследованы текстовые редакторы GNU Emacs и IntelliJ IDEA на предмет совместимости их моделей поведения, изучен язык программирования Emacs Lisp, проведен анализ, выявивший сходства и различия языков программирования Common Lisp и Emacs Lisp. Также были изучены среда разработки IntelliJ IDEA и процесс разработки расширений для IntelliJ IDEA.

Итогом данной работы является стабильно работающее расширение (плагин) для IntelliJ IDEA, максимально приближающее работу в среде разработки IntelliJ IDEA к работе в GNU Emacs.

Разработка плагина включала в себя реализацию внутренних структур данных, имитирующих соответствующие структуры в GNU Emacs, а также имплементацию встроенных функций, отвечающих за взаимодействие со средой разработки.

Данная работа содержит 37 страниц текстового документа, 8 глав, 9 иллюстраций, 1 листинг кода и 16 использованных библиографических источников.

Содержание

Введение

В настоящее время процесс разработки программного обеспечения разительно отличается того, чем он был 10 и более лет назад. Разработчики хотят максимально автоматизировать процесс своей деятельности, чтобы не тратить время на банальные вещи, такие как: набор длинных имен функций и переменных, поиск простейших ошибок, форматирование кода или мучительную навигацию по большому проекту. Все эти функции, а также многие другие, сейчас выполняют среды разработки программного обеспечения (integrated development environment, IDE). Далеко не последними по важности являются удобство редактирования программного кода и функции для работы с текстом.

Сейчас на рынке представлено достаточно большое количество различных сред разработки программного обеспечения, но большинство из них либо ориентировано на небольшой набор языков программирования, либо не являются дружественными для пользователя, либо разработаны под конкретную операционную систему. Также, среда разработки программного обеспечения не в последнюю очередь характеризуется возможностями предоставляемого текстового редактора.

Данная работа посвящена интеграции текстового редактора GNU Emacs [1] и IDE IntelliJ IDEA [2].

Описание проблемы

Из всех, имеющихся на данный момент на рынке текстовых редакторов, GNU Emacs является одним из самых старых и популярных. Многие пользователи отдают предпочтение GNU Emacs, так как данный текстовый редактор предоставляет широкий спектр функций для редактирования, а так же предоставляет возможности для дополнения данного продукта расширениями, благодаря которым его можно превратить в почтовый клиент, ежедневник и даже IDE. Характерной особенностью GNU Emacs является расширяемость «на лету»:

возможность расширить редактор набором функций и перейти к использованию добавленного функционала без перезагрузки редактора.

Ввиду того, что GNU Emacs позиционируется как настраиваемый и расширяемый программный продукт, прежде чем приступить к работе, пользователю необходимо потратить большое количество времени на его настройку. Если рассматривать GNU Emacs с точки зрения разработки ПО, необходимо также установить расширения, реализующие функционал, характерный для IDE: интеграцию с отладчиком[1], сборку и запуск проекта.

Если подойти к проблеме с другой стороны и рассматривать IDE, то одной из наиболее популярных кроссплатформенных[2] IDE является IntelliJ IDEA. Она предоставляет разработчику большой набор инструментов для оптимизации процесса разработки ПО и тестирования, но обладает меньшим набором функций редактирования текста и является не настолько гибко расширяемой и настраиваемой, как GNU Emacs.

В результате анализа свойств рассматриваемых программных продуктов, было принято решение реализовать функционал редактирования текста, предоставляемый GNU Emacs, в IDE IntelliJ IDEA.

Расширение функционала IDE IntelliJ IDEA доступно внешним разработчикам путем создания соответствующего плагина для IntelliJ IDE. Плагин(англ. plug-in) — независимо компилируемый программный модуль, динамически подключаемый к основной программе, предназначенный для расширения и/или использования её возможностей [4].

Для реализации указанного функционала необходимо выполнить следующие шаги: изучить поведение GNU Emacs и его внутренние структуры данных;

[1] Отладчик (англ. *debugger*) — компьютерная программа, предназначенная для поиска ошибок в других программах.

[2] Кроссплатформенное программное обеспечение — программное обеспечение, работающее более чем на одной аппаратной платформе и/или операционной системе. Типичным примером является программное обеспечение, предназначенное для работы в операционных системах Linux и Windows одновременно [3].

проанализировать то, как он устанавливается; изучить API[3], предоставляемый IntelliJ IDEA разработчикам плагинов; выявить необходимый набор функций, обеспечивающих работу расширений для GNU Emacs; спроектировать и разработать соответствующий стабильно работающий программный продукт.

Структура работы

Далее в данной работе будет более полно описана и сформулирована постановка задачи, исследованы преимущества и недостатки GNU Emacs и IntelliJ IDEA в сравнении друг с другом. Также будет рассмотрена предметная область разработки плагина IntelliJ IDEA для поддержки функций GNU Emacs и представлена модульная структура разработанного плагина. Далее будет описана структура внутренних объектов GNU Emacs, представлены основные графические интерфейсы GNU Emacs и IntelliJ IDEA, а также характерные особенности GNU Emacs, описаны режимы редактирования текста в GNU Emacs, приведены этапы разработки синтаксического анализатора языка Emacs Lisp, пути оптимизации работы разрабатываемого плагина, описан процесс тестирования.

[3] Интерфейс программирования приложений (англ. *application programming interface, API)* — набор готовых классов, процедур, функций, структур и констант, предоставляемых приложением (библиотекой, сервисом) для использования во внешних программных продуктах.

1. Постановка задачи

Основой для данной работы послужили задачи поддержки GNU Emacs в IntelliJ IDEA: задача обеспечения сходства поведения текстового редактора IDE IntelliJ IDEA и поведения текстового редактора GNU Emacs, а также задача предоставления пользователю GNU Emacs привычного расширяемого инструментария для редактирования текста в IntelliJ IDEA.

1.1 GNU Emacs

GNU Emacs — это расширяемый текстовый редактор, предоставляющий широкий спектр возможностей для разметки текста. Он очень популярен среди пользователей, поэтому для него написано большое количество расширений, благодаря которым в нем можно заниматься разработкой, просматривать почту и т. д. Существует интернет-ресурс, посвященный различным аспектам применения GNU Emacs [5]. GNU Emacs широко используется, так как он не только многофункционален, но и предоставляет много возможностей для персональной настройки редактора: можно назначить свои любимые сочетания клавиш, удобно расположить окна редактирования, записать собственные функции для обработки текста и т. п.

Первое, с чем сталкивается пользователь, принявший решение заняться разработкой программного обеспечения в GNU Emacs, это необходимость установить расширения, превращающие GNU Emacs в IDE, и настроить редактор. Для того чтобы выполнить конкретную настройку, которая не входит в число настроек, предусмотренных разработчиками GNU Emacs, пользователю приходится в той или иной степени разбираться в Emacs Lisp – языке программирования, на котором пишутся расширения GNU Emacs (и написана значительная часть GNU Emacs, но об этом чуть позже). Достаточно большое число пользователей отказываются от работы в GNU Emacs после множества неудачных попыток настройки. Тем же, кто успешно с ней справился,

впоследствии становится сложно пользоваться другими IDE, несмотря на их возможные преимущества перед GNU Emacs, по причине невозможности настроить IDE согласно привычной схеме.

1.2 IntelliJ IDEA

IntelliJ IDEA, в свою очередь, является IDE, ориентированной на производительность разработчика. Она предоставляет набор таких интеллектуальных функций, как:

1. Инспекции кода;

Инспекцией кода (англ. code inspection) называется статический анализ программного кода, выполняемый IDE параллельно с набором текста разработчиком. Результатом этого анализа является набор интерактивных подсказок, позволяющих избежать простейших ошибок на этапе разработки программного продукта.

2. Автоматизированные рефакторинги;

Рефакторингом (англ. refactoring) в программировании называется процесс изменения внутренней структуры программы, не затрагивающий её внешнего поведения и имеющий целью облегчить понимание её работы. В основе рефакторинга лежит последовательность небольших эквивалентных (то есть сохраняющих поведение) преобразований [6].

IDE IntelliJ IDEA предлагает разработчику набор таких часто выполняемых рефакторингов, как: извлечение метода/класса/подкласса/интерфейса; создание новой переменной по коду, возвращающему некоторое значение; безопасное (с анализом использования) удаление методов/классов/файлов/и т. п.; изменение модификаторов доступа и многое другое [7].

3. Навигация по коду.

Навигацией по коду называется набор функций, обеспечивающих переход (т. е. позиционирование каретки редактора) к желаемому месту в тексте программы, например, определению метода или переменной, классу, и т. п.

Существует полное описание набора интеллектуальных функций IntelliJ IDEA [8].

Благодаря своим интеллектуальным функциям, IDE значительно упрощает и ускоряет работу программиста.

Настройки IDE IntelliJ IDEA имеют дружественный графический интерфейс, что позволяет потратить минимум времени на настройку, получив при этом максимум результата.

1.3 Интеграция

По принятию решения реализации интеграции GNU Emacs и IntelliJ IDEA, необходимо также решить, какой из этих программных продуктов следует интегрировать. Для этого был проведен анализ внутренней организации данных в рассматриваемых программных продуктах.

IDE IntelliJ IDEA написана на языке программирования высокого уровня Java, поддерживающем парадигму объектно-ориентированного программирования. В IntelliJ IDEA основной структурой данных для работы с кодом является абстрактное синтаксическое дерево.

Абстрактное синтаксическое дерево — это конечное, помеченное, ориентированное дерево, в котором внутренние вершины сопоставлены с операторами языка программирования, а листья с соответствующими операндами. Таким образом, листья являются пустыми операторами и представляют только переменные и константы. Синтаксические деревья используются в синтаксических анализаторах для промежуточного представления программы между деревом разбора и структурой данных, которая затем используется в качестве внутреннего представления компилятора или интерпретатора компьютерной программы для оптимизации и генерации кода [9].

Абстрактное синтаксическое дерево используется для выполнения статистического анализа кода и подсветки синтаксиса используемого языка программирования. Разработчиками IntelliJ IDEA были реализованы сложные

логические структуры данных с нетривиальной логикой. Таким образом, для реализации интеллектуальных функций, свойственных IDE IntelliJ IDEA, необходимо спроектировать и разработать аналогичную структуру данных, а также сами функции, что является очень трудоемкой задачей.

GNU Emacs, в свою очередь, написан на языках программирования высокого уровня Emacs Lisp и C, поддерживающими процедурную парадигму программирования. Функционал текстового редактора, а также его расширения представляют собой набор определений функций и переменных. Таким образом, для того, чтобы перенести функционал GNU Emacs в IntelliJ IDEA достаточно будет реализовать функции нижнего уровня по работе с редактором.

Принимая во внимание результаты проведенного анализа, а также тот факт, что данный проект ориентирован на разработчиков ПО, было принято решение встраивать функции редактирования GNU Emacs в IntelliJ IDEA, а не наоборот.

1.4 Выбор среды и языка разработки

Следующим вопросом, решенным в процессе подготовки к выполнению поставленной задачи, был вопрос выбора языка программирования и IDE.

Ввиду того, что было принято решение интегрировать GNU Emacs в IntelliJ IDEA путем разработки плагина для IntelliJ IDEA, естественным шагом было вести разработку в IntelliJ IDEA. Это позволило на практике ознакомиться с поведением IDE и работой в ее текстовом редакторе. Дополнительным преимуществом данного решения было наличие в платформе IntelliJ IDEA фреймворка[4] для тестирования плагинов.

Что касается языка разработки, то платформа IntelliJ IDEA позволяет вести разработку плагинов на различных языках программирования высокого уровня

[4] Фреймворк (англ. framework) — в информационных системах структура программной системы; программное обеспечение, облегчающее разработку и объединение разных компонентов большого программного проекта. В отличие от библиотек, которые объединяют набор подпрограмм близкой функциональности, фреймворк содержит в себе большое количество разных по назначению библиотек [10].

(например, Scala, Kotlin, Java и т. п.). Ввиду того, что сама IntelliJ IDEA написана на Java, а также с целью облегчить процесс дальнейшей поддержки плагина, если она будет вестись другим разработчиком, было принято решение вести разработку на Java.

1.5 Задание на выполнение работы

Основная идея проекта заключается в том, чтобы предоставить разработчикам, привыкшим применять функционал GNU Emacs для редактирования текста, возможность воспользоваться всеми преимуществами IDE IntelliJ IDEA, не испытывая неудобств.

Для реализации указанного функционала необходимо выполнить следующие шаги: изучить поведение GNU Emacs и его внутренние структуры данных; проанализировать то, как он устанавливается; изучить интерфейсы, предоставляемые IntelliJ IDEA разработчикам плагинов; выявить необходимый набор функций, обеспечивающих работу расширений для GNU Emacs; спроектировать и разработать соответствующий стабильно работающий программный продукт.

Разработанный программный продукт должен удовлетворять следующим требованиям:

1. Обеспечить поведение таких функций редактирования текста, как вставка, удаление, навигация по тексту, в IntelliJ IDEA аналогично их поведению в GNU Emacs.

2. Предоставить пользователю возможность оперировать в IntelliJ IDEA теми же понятиями графических интерфейсов, что и в GNU Emacs.

3. Обеспечить возможность настройки привязок сочетаний клавиш к произвольным командам из набора Emacs, а также пользовательским командам в IntelliJ IDEA.

4. Предоставить поддержку языка программирования Emacs Lisp в IntelliJ IDEA. Понятие «поддержка языка» включает в себя подсветку синтаксиса и набор функций по работе с текстом данного вида.

5. Обеспечить возможность применения режимов редактирования текста, предоставляемых GNU Emacs, в IntelliJ IDEA.

6. Обеспечить возможность запуска расширений для GNU Emacs из-под IntelliJ IDEA.

2. Предметная область

2.1 Дистрибутив GNU Emacs

GNU Emacs поставляется в виде интерпретатора языка Emacs Lisp на C и наборе исходного кода на Emacs Lisp. Расширения для GNU Emacs так же пишут на Emacs Lisp. Таким образом, для того, чтобы реализовать поддержку GNU Emacs в IntelliJ IDEA, достаточно будет реализовать интерпретатор Emacs Lisp.

Простой интерпретатор анализирует и выполняет программу по мере поступления её исходного кода на вход интерпретатора.

Интерпретатор состоит из синтаксического анализатора и набора встроенных функций, с помощью которых обеспечивается выполнение методов более высокого уровня. Синтаксический анализатор выполняет процесс сопоставления линейной последовательности лексем языка с его формальной грамматикой. Результатом такого разбора обычно является дерево разбора (синтаксическое дерево) [11].

Реализация интерпретатора языка Emacs Lisp дает возможность выполнять функции из исходного кода GNU Emacs на Emacs Lisp.

Набор встроенных функций интерпретатора состоит из порядка 1000 методов. В процессе реализации данного набора были выявлены такие примитивные функции, которые в силу расхождений концепций программирования на C и на Java, теряли смысл. Подобные случаи позволили сократить объем разработки.

2.2 Поиск существующих решений

Первый вопрос, который возникает при поиске решения поставленной задачи, заключается в поиске существующего свободного[5] интерпретатора языка Emacs Lisp на Java, который позволил бы только переписать взаимодействие с текстовым редактором IDE IntelliJ IDEA.

Исследование показало, что таких интерпретаторов нет, зато существует большое число интерпретаторов базового языка семейства Lisp: Common Lisp.

В ходе изучения возможности использования какого-либо интерпретатора языка Common Lisp для реализации интерпретатора языка Emacs Lisp, было обнаружено принципиальное различие этих языков программирования: Emacs Lisp использует динамические области видимости, а Common Lisp – статические (иначе, лексические).

Областью видимости переменной в программировании называют контекст выполнения программного кода, в котором ее идентификатор является действительным и может быть использован [12]. Вне области видимости переменная считается неопределенной.

Использование динамической области видимости подразумевает наличие глобального стека привязок идентификаторов переменных к их значениям (англ. global stack binding). То есть, различные переменные обязаны иметь уникальные имена в течение всего времени выполнения кода.

В свою очередь, при использовании статических областей видимости, привязка имени переменной к ее значению существует лишь во время пребывания в этом блоке кода (как текста). Данный тип области видимости получил свое

---

[5] Программа свободна, если ее пользователям предоставлены четыре свободы:

0. Свобода выполнять программу в любых целях.

1. Свобода изучать работу программы и модифицировать программу, чтобы она выполняла ваши вычисления, как вы пожелаете. Это предполагает доступ к исходному тексту.

2. Свобода передавать копии программы.

3. Свобода передавать копии своих измененных версий другим. Это предполагает доступ к исходному тексту.[13]

название благодаря тому, что в этом случае предварительный анализ доступности переменных в точках вызова происходит статически (не требуя ее выполнения).

Приведем пример влияния различий применения динамических и статических областей видимости на поведение программы.

```
x = 1;
void g () { print (x); x = 2; }
void f () { x = 3; g(); }
g();
f();
print (x);
```

Листинг 1. Пример простой программы на псевдокоде с С-подобным синтаксисом.

Рассмотрим вывод программы (листинг 1) при условии использования различных типов областей видимости.

При использовании динамической области видимости, объявление переменной «x» заносится в глобальный стек привязок идентификаторов переменных к их значениям, и все дальнейшие операции над переменной с идентификатором «x» происходят над имеющейся сущностью. Таким образом, в этом случае выводом программы будет «1 3 2».

Использование же статистических областей видимости позволяет создавать внутри функций f и g различные переменные с идентификатором «x». То есть, в этом случае выводом программы будет «1 1 1».

Таким образом, это различие сводит на нет возможные преимущества от использования существующего интерпретатора: для того, чтобы использовать интерпретатор Common Lisp для интерпретации Emacs Lisp, необходимо предварительно обеспечить уникальность имен функций и переменных, а так же реализовать среду исполнения для управления привязками переменных, что по трудоемкости равно самостоятельной реализации интерпретатора, но будет гораздо менее эффективно по производительности.

3. Особенности GNU Emacs

3.1 Представление графических объектов в GNU Emacs

Для реализации задачи приближения работы в IDE IntelliJ IDEA к работе в GNU Emacs были исследованы представления графических интерфейсов в данных программных продуктах.

В GNU Emacs своя терминология для графических интерфейсов.

Буфером называется объект, представляющий какой-либо текст. Большинство буферов соответствуют открытым файлам, но в принципе буфер не обязан быть привязанным к файлу: например, буфер может содержать документацию по функции или переменной, или встроенные подсказки.

Фрейм соответствует окну в обычном понимании этого слова. Каждый фрейм содержит область вывода и одно или несколько окон GNU Emacs.

Окном называется прямоугольная область фрейма, которая отображает один из буферов. Помимо этого, каждое окно имеет свою строку состояния, где выводится название буфера, его основной режим ит.п.

Область вывода — это одна или несколько (по необходимости) строк внизу фрейма, в которой GNU Emacs выводит различные сообщения, а также запрашивает подтверждения и дополнительную информацию от пользователя.

Минибуфер используется для ввода дополнительной информации, например, для чтения параметров команд при интерактивном исполнении. Это такой же буфер, как и все остальные.

На рисунке 1 представлен снимок экрана компьютера с запущенным GNU Emacs.

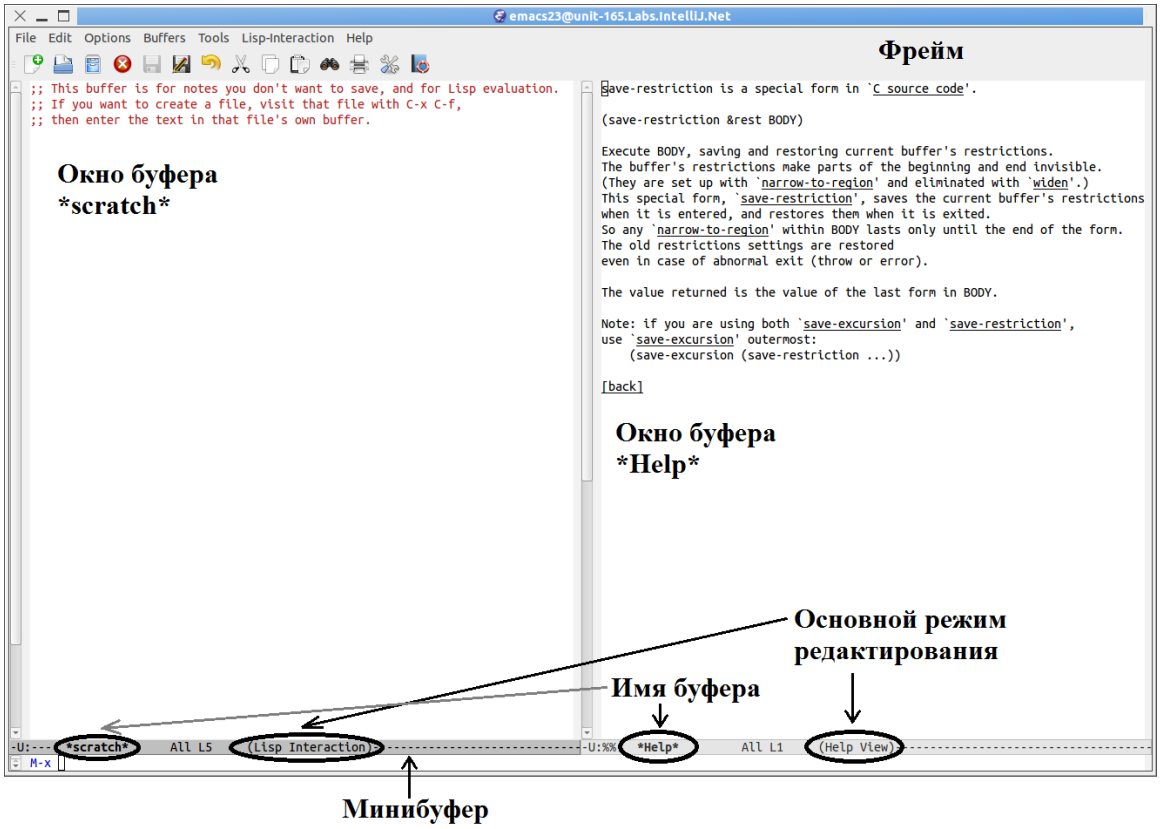

Рис. 1. Графический интерфейс GNU Emacs.

IntelliJ IDEA имеет отличную GNU Emacs иерархию графических интерфейсов.

За работу с текстом отвечает *редактор*, поддерживающий произвольное количество открытых файлов. Для каждого открытого файла создается соответствующая вкладка в области редактирования. Вкладка содержит компонент, предоставляющий функционал для работы с файлом. Понятие вкладки редактора в IntelliJ IDEA соответствует понятию окна в GNU Emacs.

Помимо компонента, отвечающего за редактирование, открытому файлу также соответствует компонент, представляющий данный файл в виде внутренней структуры данных платформы IntelliJ IDEA. Данный компонент соответствует понятию буфера в GNU Emacs.

Понятия *фрейма* являются одинаковыми для рассматриваемых программных продуктов.

Область вывода и минибуфер не имеют графических аналогов в IntelliJ IDEA.

В IntelliJ IDEA реализованы области для отображения специального инструментария (например, инструментарий для отладки и пошагового выполнения программ, результаты поиска, инструментарий реализующий интеграцию с системами контроля версии и т.д.).

На рисунке 2 представлен снимок экрана компьютера с запущенной IDE IntelliJ IDEA.

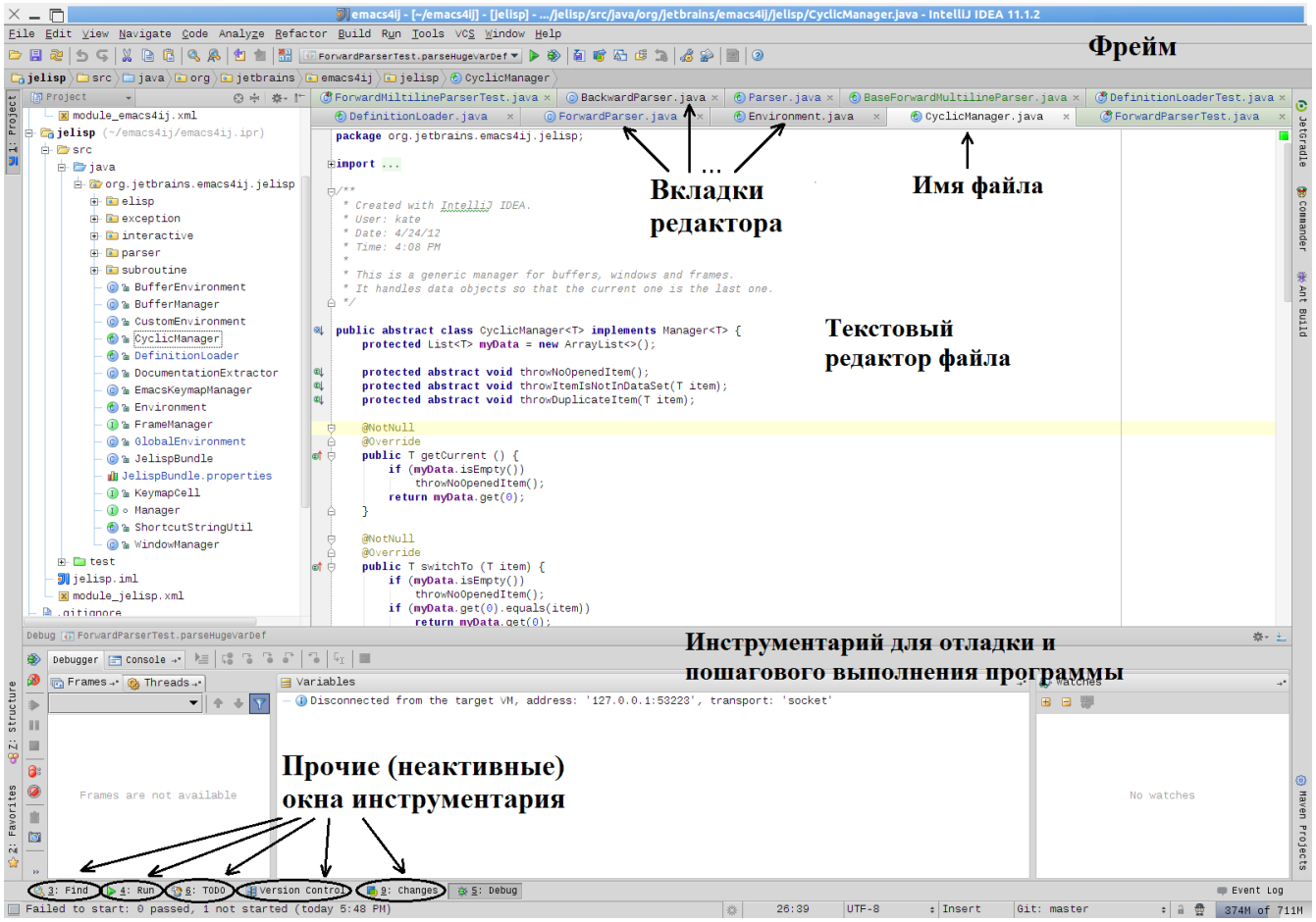

Рис. 2. Графический интерфейс IDE IntelliJ IDEA.

Чтобы обеспечить поведение IntelliJ IDEA, аналогичное поведению GNU Emacs, были спроектированы и разработаны интерфейсы и классы, обеспечивающие работу с соответствующим компонентом IntelliJ IDEA как с объектом GNU Emacs.

Визуализация Emacs окон, отображающих буфера, содержащие файлы, в разработанном плагине реализована при помощи редактора IntelliJ IDEA. Буфера, содержимое которых не является файлом, а также область вывода, не

соответствуют концепции редактора IntelliJ IDEA. Визуализация Emacs окон, отображающих данные сущности, выполнена с использованием компонента платформы IntelliJ IDEA, отвечающего за специальный инструментарий.

Минибуфер реализован в виде всплывающей по требованию строки в верхней части редактора файла.

На рисунке 3 представлен снимок экрана компьютера с запущенной IDE IntelliJ IDEA и активированным плагином, разработанным в рамках данного проекта.

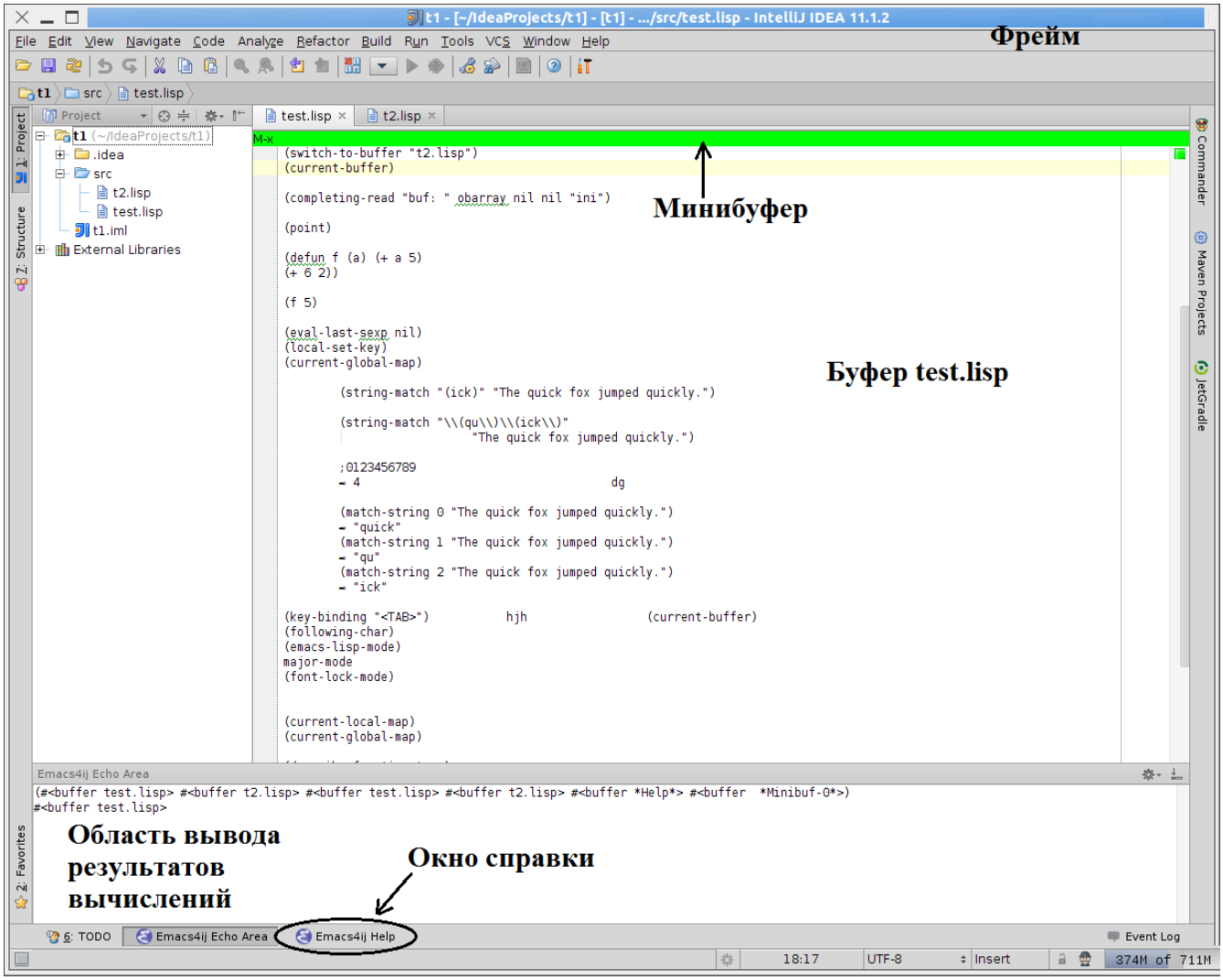

Рис. 3. Графический интерфейс разработанного плагина.

3.2 Интерактивное выполнение команд

Особенностью графического интерфейса GNU Emacs, которая не имеет аналогов в IntelliJ IDEA, является предоставление пользователю возможности интерактивного выполнения команд: пользователь может инициировать интерактивное выполнение команды, после чего редактор считает название команды из минибуфера, а так же все необходимые параметры.

Функция, доступная для интерактивного вызова, называется командой. Каждая команда в теле своего определения имеет специальную форму, носящую название «интерактивной». Интерактивная форма описывает формат параметров, считываемых командой при интерактивном выполнении.

В GNU Emacs вызов команды в интерактивном режиме инициируется другой командой, которая, в свою очередь, считывает имя функции, подлежащей вызову, и передает ей управление. Такое поведение позволило абстрагироваться от поведения конкретной команды и спроектировать общую схему для интерактивного считывания параметров.

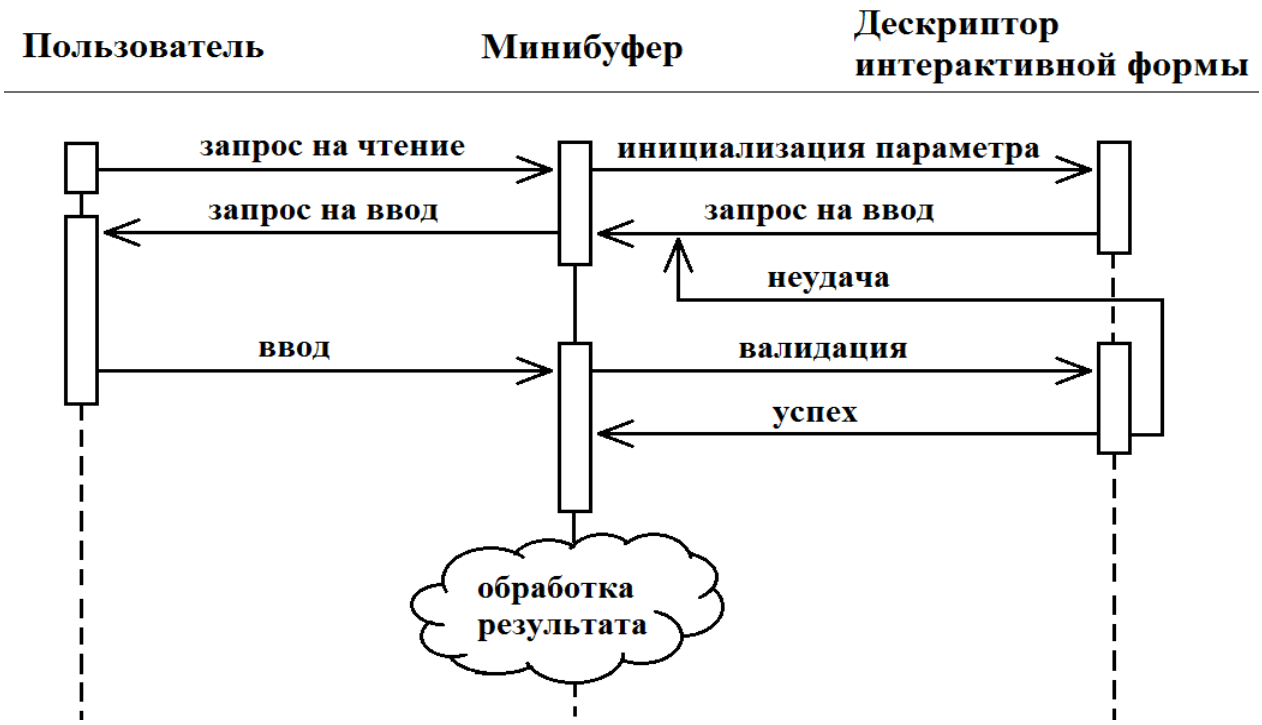

Рис. 4. Диаграмма состояний, демонстрирующая считывание одного из параметров, согласно интерактивной форме.

На диаграмме состояний (рис. 4) представлена схема взаимодействия пользователя, минибуфера и внутренней структуры данных плагина, представляющей собой дескриптор интерактивной формы, в процессе считывания параметра в интерактивной форме. Дескриптор является объектом, описывающим данные интерактивной формы: количество и тип ожидаемых параметров.

По запросу пользователя на ввод параметра, происходит открытие минибуфера. Согласно данным об ожидаемом типе параметра, хранящимся в дескрипторе интерактивной формы, выполняется предварительная инициализация значения параметра по умолчанию (если оно предусмотрено). Далее управление передается пользователю. По сигналу пользователя о том, что ввод параметра завершен, управление возвращается минибуферу. Он инициирует верификацию соответствия введенного значения ожидаемому типу, и, в случае если она проходит успешно, считанный параметр передается запросившему его объекту. В случае неудачи верификации, пользователю предлагается повторить ввод параметра.

3.3 Выполнение кода in-place

Еще одной особенностью текстового реактора GNU Emacs является возможность выполнения текущей инструкции кода на Emacs Lisp. Данная функция позволяет быстро получить значение выражения, не прибегая к интерпретации всего файла.

3.4 Настраиваемые сочетания клавиш

Одним из показателей гибкости GNU Emacs является возможность настройки привязок сочетаний клавиш к выполняемым функциям. В базовой версии настроек GNU Emacs представлена карта привязок сочетаний клавиш к выполняемым функциям по умолчанию, но пользователь в любой момент может ее отредактировать.

3.5 Автодополнения

GNU Emacs предоставляет пользователю функции автодополнения имен функций и переменных, загруженных в среду исполнения при запуске редактора. Данный функционал значительно облегчает и ускоряет интерактивный вызов команд, вызов справки и позволяет получить представление о доступных методах и переменных.

Пользовательские функции и переменные, определенные в текущей сессии GNU Emacs, также попадают в списки автодополнения.

3.6 Документация

Одной из отличительных особенностей Emacs Lisp является то, что он «самодокументируемый»: документация функций и переменных предусмотрена в определении соответствующих элементов. Данное свойство позволяет пользователю получить не только документацию по любой функции или переменной, но также и информацию о текущем состоянии объекта.

Документация по пользовательским функциям и переменным, определенным в текущей сессии GNU Emacs, также доступна в общей справке после выполнения инструкций кода, отвечающих за регистрацию данных функций и переменных в среде выполнения.

4. Режимы редактирования текста

В GNU Emacs подсветка синтаксиса, специфические функции для работы с текстом специального вида, задается специальным режимом редактирования текста. Режим редактирования текста состоит из основной моды (англ. major mode) и набора вспомогательных мод (англ. minor mode).

В общем случае, модой в GNU Emacs называется набор функций и переменных, предоставляющих некоторый функционал (например, подсветку синтаксиса). Основная мода помимо указанного набора содержит также набор специальных сущностей, обеспечивающих поддержку текста определенного вида.

Каждый буфер может находиться в своем режиме редактирования.

4.1 Major mode

Основные моды редактирования отвечают за разметку текста определенного вида. Каждая основная мода редактирования задает такие структуры данных, как:

1. Синтаксическая таблица;

Синтаксической таблицей называется таблица соответствия символа и его синтаксического класса. Синтаксический класс символа может принимать значения из предопределенного множества. Данное множество содержит такие элементы, как: «пунктуация», «символ», «слово», «комментарий», «открывающая скобка», «закрывающая скобка» и т. д.

2. Набор функций для работы с текстом конкретного вида;

3. Набор сочетаний клавиш, к которым привязаны указанные выше функции.

Основные моды редактирования являются взаимно исключающими.

Примером основной моды служит emacs-lisp-mode: режим, обеспечивающий поддержку языка Emacs Lisp. Понятие «поддержка языка» включает в себя подсветку синтаксиса и набор функций по работе с текстом данного вида.

4.2 Minor mode

Вспомогательным режимом редактирования в GNU Emacs называется функциональность, объединенная общей целью и вынесенная в отдельный блок кода.

Примерами вспомогательных режимов редактирования являются режим проверки правописания, автосохранение файла через определенный интервал времени, отображения номеров строк и т. д.

4.3 Поддержка режимов редактирования

Режим редактирования является индивидуальным для каждого буфера Emacs, поэтому каждый объект, представляющий собой буфер, содержит поля, отвечающие за представления соответствующих функций и переменных активных мод, а также специальных сущностей, задаваемых основной модой.

По умолчанию при инициализации буфера активируется фундаментальная мода, которая является «пустой», то есть поддерживает текст на естественном языке.

5. Синтаксический анализ

Для обеспечения выполнения кода на Emacs Lisp, необходимо реализовать механизм трансляции текстового представления программного кода во внутренние структуры данных, посредством которых будут производиться заданные программным кодом вычисления.

Таким механизмом трансляции служит синтаксический анализ. Синтаксический анализатор выполняет процесс сопоставления линейной последовательности лексем языка с его формальной грамматикой. В данном случае его результатом является структура языка Emacs Lisp, подлежащая вычислению.

5.1 Несколько определений из теории конечных автоматов

С целью дать определение механизма работы реализованного синтаксического анализатора, приведем несколько базовых определений из теории автоматов.

1. Абстрактный автомат — это математическая абстракция, модель дискретного устройства, имеющего один вход, один выход и в каждый момент времени находящегося в одном состоянии из множества возможных. На вход этому устройству поступают символы одного алфавита, на выходе оно выдаёт символы (в общем случае) другого алфавита [14].

2. Конечным называется абстрактный автомат без выходного потока, число возможных состояний которого конечно [14].

3. Детерминированным конечным автоматом (ДКА) называется такой конечный автомат, в котором для каждой последовательности входных символов существует лишь одно состояние, в которое автомат может перейти из текущего [15].

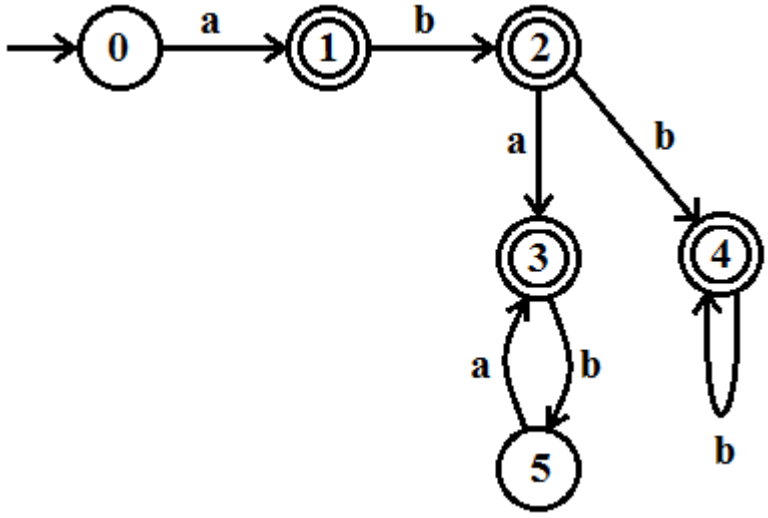

Рис. 5. ДКА.

На рис. 5 приведен пример ДКА. Согласно определению, он имеет конечное число состояний, и все его переходы являются детерминированными. Двойным кружком помечены конечные состояния ДКА.

5.2 Основной синтаксический анализатор языка Emacs Lisp

В данном проекте реализован компонент, выполняющий синтаксический анализ текста, и преобразующий поданный текст в структуру, аналогичную соответствующей структуре языка Emacs Lisp.

Разработанный компонент представляет собой ДКА, входным алфавитом которого служат символы из таблицы ASCII, а также специальные символы из алфавитов естественных языков. Конечными состояниями данного ДКА являются состояния, соответствующие завершению разбора структуры данных, аналогичной соответствующей структуре Emacs Lisp.

5.3 Синтаксический анализатор общего вида

Для поддержки синтаксических таблиц произвольного вида возникла задача в дополнение к основному синтаксическому анализатору языка Emacs Lisp реализовать синтаксический анализатор общего вида, который бы размечал поданный текст в соответствии с заданной синтаксической таблицей.

В процессе подготовки к реализации данного компонента были проанализированы грамматические структуры, характерные для языков программирования. С учетом выявленных закономерностей был разработан компонент, выполняющий синтаксический анализ согласно заданной синтаксической таблице.

Так же, как и основной синтаксический анализатор языка Emacs Lisp, синтаксический анализатор общего вида является ДКА. Входным алфавитом данного ДКА служат символы синтаксической таблицы, а конечными состояниями являются состояния, соответствующие завершению разбора грамматической конструкции.

6. Улучшение производительности: индексация

В целях улучшения производительности разработанного плагина была реализована предварительная индексация файлов с исходным кодом GNU Emacs на Emacs Lisp.

Индексация — это процесс построения ассоциативного списка, в данном случае элементами списка являются пары, состоящие из имени функции (или переменной) и списка имен файлов, содержащих определение указанного объекта, с указанием сдвига по файлу до начала определения. Построение ассоциативного списка выполняется путем сканирования всех файлов исходного кода и записи позиций определений всех представленных в файле функций и переменных во внутреннюю структуру данных.

Индексация выполняется однократно, в фоновом режиме, параллельно с инициализацией окружения среды разработки.

Данная оптимизация позволила сделать поиск необходимого определения незаметным для пользователя.

7. Архитектура проекта

7.1 Модульная структура проекта

Разработанный в результате данной работы проект состоит из двух модулей: модуля Emacs4ij, отвечающего за интеграцию с IDE IntelliJ IDEA, и модуля Jelisp, являющегося реализацией интерпретатора языка Emacs Lisp на Java.

Интеграция с IDE IntelliJ IDEA состоит в реализации методов и структур данных, обеспечивающих взаимодействие с текстовым редактором IDE и привязку сочетаний клавиш к вызову функций из набора GNU Emacs, а также функционала, отвечающего за активацию плагина.

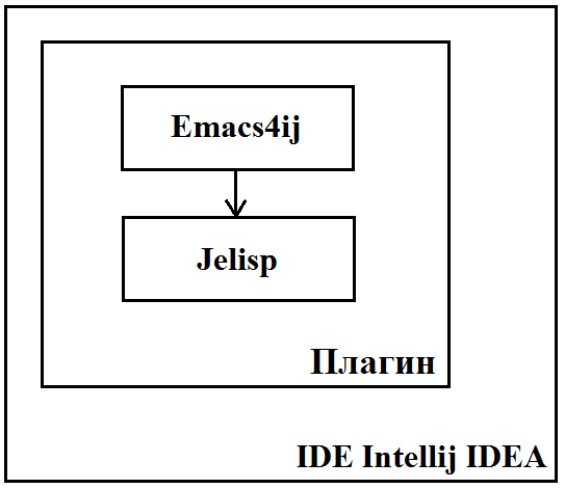

Рис. 6. Модульная структура проекта.

Рисунок 6 демонстрирует зависимости между модулями проекта. Модуль Jelisp является независимым от модуля Emacs4ij в то время как модуль Emacs4ij использует функции и структуры данных из модуля Jelisp, а также содержит имплементацию интерфейсов структур данных, чья реализация тесно связана с платформой IntelliJ IDEA.

Независимость модуля Jelisp от платформы IDE дает возможность использования функционала интерпретатора в любом окружении, при условии реализации имплементаций соответствующих интерфейсов, представляющих собой платформенно-зависимые сущности.

7.1 Модуль Jelisp

Модуль Jelisp содержит такие компоненты плагина, как: синтаксический анализатор языка программирования Emacs Lisp, синтаксический анализатор общего вида, набор встроенных функций языка Emacs Lisp, классы и интерфейсы, представляющие собой внутренние структуры данных Emacs.

Данный модуль не зависит от платформы IntelliJ IDEA, т.к. сущности, имеющие платформенно-зависимую реализацию, представлены интерфейсами, а их реализация находится в модуле Emacs4ij. Такими сущностями являются объекты, имитирующие буфер Emacs, минибуфер, фрейм и окно.

7.2 Модуль Emacs4ij

Модуль Emacs4ij содержит реализацию компонентов, отвечающих за взаимодействие со средой IntelliJ IDEA, по сути, обеспечивающих работу разработанного плагина, а также имплементацию интерфейсов из модуля Jelisp: сущностей, представляющих собой элементы графического интерфейса Emacs.

7.3 Взаимодействие компонентов плагина

Данная глава посвящена описанию основных сценариев взаимодействия компонентов разработанного плагина между собой.

Первый сценарий описывает запуск IDE и инициализацию окружения для выполнения функций из набора GNU Emacs.

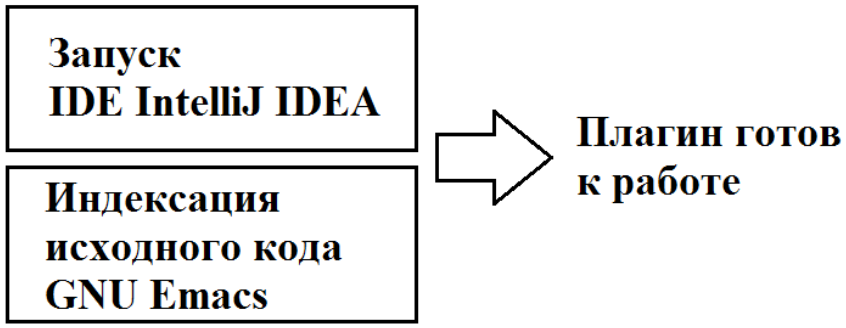

Рис. 7. Сценарий инициализации.

На рисунке 7 представлена схема выполнения указанного сценария. При запуске IDE, параллельно с инициализацией окружения IDE, стартует построение индекса исходного кода GNU Emacs. Индексация файла, содержащего программный код на Emacs Lisp, использует синтаксический анализатор Emacs Lisp для поиска определений функций и переменных. По завершению индексации функционал плагина становится доступным для использования.

Второй сценарий является сценарием для вычисления значения некоторого выражения на Emacs Lisp.

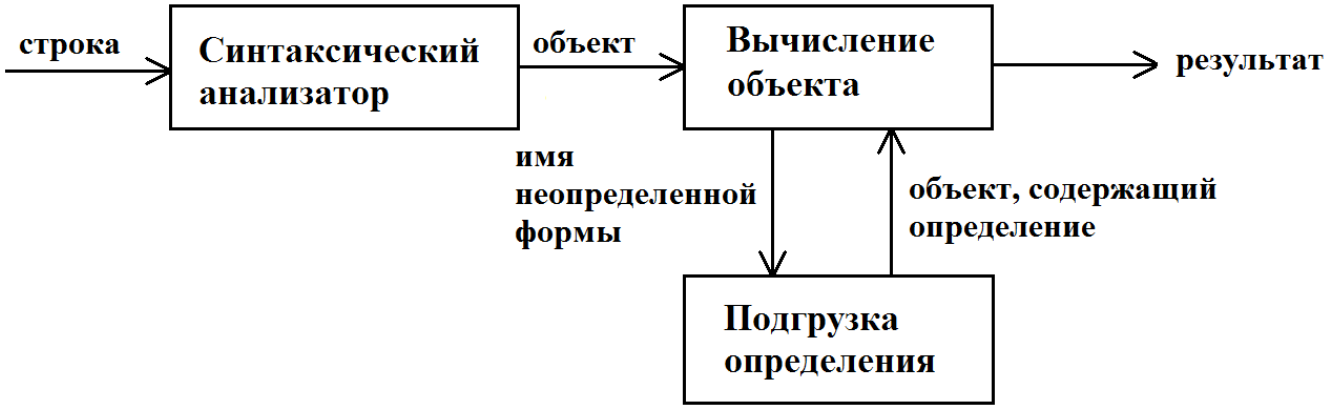

Рис. 8. Сценарий вычисления выражения на Emacs Lisp.

Приведенный рисунок (рис. 8) демонстрирует этапы вычисления выражения на Emacs Lisp. Синтаксический анализатор языка Emacs Lisp принимает на вход строку, содержащую программный код на Emacs Lisp, а на выходе возвращает объект внутренней структуры, являющийся представлением проанализированного выражения. Определения каждого класса, соответствующего структуре из множества структур языка Emacs Lisp, содержит инструкции по вычислению своего значения. Если во время вычисления значения полученного объекта происходит обнаружение формы (функции или переменной), не определенной в среде выполнения плагина, выполняется извлечение определения этой формы, и ее регистрация в среде выполнения плагина. Извлечение определения и регистрация соответствующей формы выполняется следующим образом (рис. 9):

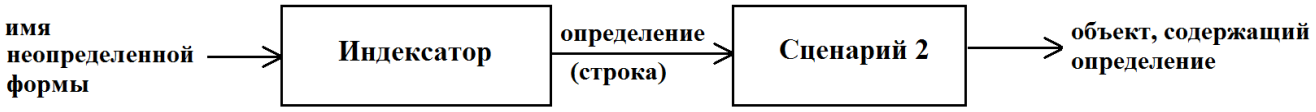

Рис. 9. Загрузка определения Emacs Lisp формы.

Имя требуемой формы подается на вход индексатора, который возвращает строку, содержащую необходимое определение. Далее выполняется вычисление значения данного определения, согласно сценарию вычисления выражения на Emacs Lisp. Результатом этого вычисления является объект, являющийся представлением определения запрошенной формы в виде соответствующих внутренних структур плагина.

8. Тестирование

В настоящей работе для тестирования различных компонент проекта применялась техника модульного тестирования (англ. unit-testing, [16]). Использовались библиотеки JUnit4 и JUnit3.

Для всех модулей проекта были написаны юнит-тесты, покрывающие основную функциональность плагина, в частности, встроенные функции, внутренние структуры и взаимодействие разработанного плагина с IDE.

Платформа IntelliJ IDEA предоставляет разработчикам плагинов удобный фреймворк для юнит-тестирования. Он обеспечивает создание среды окружения IDE IntelliJ IDEA в тестовом режиме, что позволяет автоматизировать процесс тестирования плагина.

Заключение

В ходе работы был разработан стабильно работающий плагин, максимально приближающий работу с текстовым редактором в среде разработки IntelliJ Idea к работе в текстовом редакторе GNU Emacs. Разработка плагина включала в себя реализацию внутренних структур данных, имитирующих соответствующие структуры в GNU Emacs, а также имплементацию набора встроенных функций по работе с редактором.

В рамках подготовки к разработке было изучено поведение GNU Emacs и его внутренние структуры данных; проанализирован процесс его установки; изучены программные интерфейсы, предоставляемые IntelliJ IDEA разработчикам плагинов, и выявлен набор функций, обеспечивающих работу расширений для GNU Emacs.

Для обеспечения работы с кодом на Emacs Lisp был реализован синтаксический анализатор языка Emacs Lisp. Для обеспечения возможности выполнения кода на Emacs Lisp реализован набор встроенных функций языка Emacs Lisp. Для обеспечения возможности интерактивного выполнения команд был реализован минибуфер.

Также в рамках поставленной задачи были реализованы: справка по языку программирования Emacs Lisp, возможность гибкой настройки привязок произвольных функций GNU Emacs к различным сочетаниям клавиш, основной режим редактирования кода на Emacs Lisp, включающий в себя подсветку синтаксиса и специфические функции для работы с текстом данного вида. Также была реализована частичная поддержка большей части расширений для GNU Emacs.

Исходный код разработанного программного продукта доступен на https://github.com/JetBrains/emacs4ij.

Библиографический список использованных литературных источников

[1] Домашняя страница проекта GNU Emacs.

http://www.gnu.org/software/emacs/

[2] Домашняя страница проекта IDE IntelliJ IDEA.

http://www.jetbrains.com/idea/

[3] Страница про кроссплатформенное программное обеспечение.

http://en.wikipedia.org/wiki/Cross-platform

[4] Статьи о разработке плагинов для IntelliJ IDEA.

http://confluence.jetbrains.net/display/IDEADEV/PluginDevelopment

[5] Интернет-ресурс, посвященный различным аспектам применения GNU Emacs. http://emacswiki.org/emacs/

[6] Понятие рефакторинга.

http://c2.com/cgi/wiki?WhatIsRefactoring

[7] Рефакторинг кода в IntelliJ IDEA.

http://www.jetbrains.com/idea/features/refactoring.html

[8] Интеллектуальные функции IDE IntelliJ IDEA.

http://www.jetbrains.com/idea/features/code_analysis.html

[9] Рефакторинг. Улучшение существующего кода [Текст]: /М. Фаулер [и др.]. – СПб.: Символ Плюс, 2003. – С. 404.

[10] Понятие фреймворка.

http://docforge.com/wiki/Framework

[11] Статья про интерпретаторы.

http://en.wikipedia.org/wiki/Interpreter_%28computing%29

[12] Статья про области видимости.

http://en.wikipedia.org/wiki/Lexical_scoping#Lexical_scoping_and_dynamic_scoping

[13] Статья об определении свободного программного обеспечения

http://www.gnu.org/philosophy/free-sw.ru.html

[14] Теория конечных автоматов: абстрактный автомат, конечный автомат.

http://www.intuit.ru/department/algorithms/mathformlang/2/

[15] Теория конечных автоматов: детерминированный конечный автомат.

http://www.intuit.ru/department/algorithms/mathformlang/2/4.html#sect7

[16] Статья про модульное тестирование.

http://c2.com/cgi/wiki?UnitTest