МИНОБРНАУКИ РОССИИ

Федеральное государственное автономное образовательное
учреждение высшего образования
«Южный федеральный университет»

Институт математики, механики
и компьютерных наук им. И. И. Воровича

Кафедра информатики и вычислительного эксперимента

Земляк Александр Сергеевич

РЕФАКТОРИНГ И РЕАЛИЗАЦИЯ МНОГОЯЗЫКОВОСТИ СИСТЕМЫ INTELLISENSE В ИНФРАСТРУКТУРЕ PASCALABC.NET

ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА
по направлению подготовки
02.03.02 – Фундаментальная информатика и информационные технологии

Научный руководитель –
доц., к. ф.-м. н. Михалкович Станислав Станиславович

Допущено к защите:
заведующий кафедрой _____ Михалкович С.С.

Ростов-на-Дону – 2025

Оглавление

Введение

Очень важным компонентом современных сред разработки является технология автодополнения кода или по-другому система Intellisense. Наиболее продвинутая версия этой системы представлена в программе Microsoft Visual Studio. Intellisense призвана значительно упростить процесс разработки ПО, а именно в реальном времени предоставлять разработчику описание к используемым им функциям, классам, модулям, переменным и другим сущностям, также подсказывать список методов объекта, то есть в некоторой степени заменять текстовую документацию к программному коду. Обычно для Intellisense требуется версия компилятора языка программирования, которая устойчива к ошибкам в тексте программы и хорошо оптимизирована, поскольку в ходе работы данная система должная собирать метаданные обо всех сущностях в программном коде.

Платформа PascalABC.NET обладает собственной реализацией технологии Intellisense, включающей все присущие ей по стандарту функции. Причем в реализации системы изначально закладывалась возможность поддержки нескольких языков программирования, что видно по разделению на так называемое ядро, универсальное для всех языков и конкретную реализацию «провайдера» PascalABC.NET, специфическую для языка. Однако, некоторые части системы, как выяснилось, требуют переработки, то есть рефакторинга для полноценной поддержки многоязыковости, особенно если учесть принципы чистой архитектуры. Именно это является основной задачей данной выпускной работы.

Помимо реализации многоязыковости и тестировании ее на примере нового языка платформы – SPython в процессе разработки также потребовалось решить проблемы с утечками памяти, возникающими в процессе работы приложения по причине запусков Intellisense. И наконец, завершающим шагом было создание языкового сервера PascalABC.NET, который планируется к использованию в новой кроссплатформенной IDE платформы.

Постановка задачи

Целью работы является рефакторинг кода, связанного с Intellisense в PascalABC.NET и усовершенствование Intellisense с учетом многоязыковости среды PascalABC.NET. Для достижения указанной цели требуется решить следующие задачи:

1. Выявить и устранить основные причины утечек памяти в платформе PascalABC.NET, связанные с системой Intellisense
2. Проанализировать различия в системе Intellisense для различных языков в инфраструктуре PascalABC.NET
3. Провести рефакторинг кода в проекте PascalABC.NET, направленный на реализацию Intellisense для различных языков программирования
4. Реализовать поддержку нового языка платформы – SPython в системе Intellisense
5. Адаптировать систему Intellisense для использования ее в виде LSP-сервера платформы PascalABC.NET, поддерживающего функции автодополнения, подсказки по наведению и подсказки параметров функций

1. Устранение утечек памяти, связанных с работой системы Intellisense

1.1. Устранение утечек памяти, возникающих при компиляции модуля PABCSystem

В процессе работы с системным модулем языка PascalABC.NET PABCSystem было обнаружено, что количество памяти, выделяемое на работу платформы медленно растет и со временем достигает недопустимых размеров. Поскольку консольный компилятор периодически завершает свою работу и запускается заново был сделан вывод, что наиболее вероятная причина неограниченного роста по памяти — это проблема в работе Intellisense.

Чтобы отследить источники утечек памяти были использованы стандартные средства профилирования, предоставляемые Microsoft Visual Studio. С помощью этих средств было замечено, что после нескольких вызовов парсинга системы Intellisense в памяти накапливаются метаданные о сущностях, и, в частности, такие крупные объекты, как объекты типов InterfaceUnitScope и ImplementationUnitScope, содержащие информацию о секции интерфейса и реализации модуля соответственно. (см. Рис. 1)

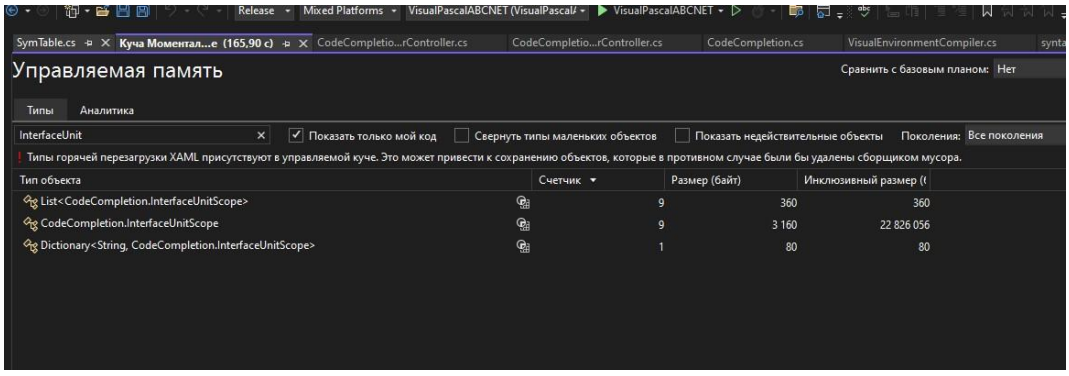

Рис. 1. Накопление объектов InterfaceUnitScope

Сами объекты InterfaceUnitScope и ImplementationUnitScope не сохраняются в статических переменных и не захватываются в анонимных методах, что

означает, что, скорее всего, на них сохраняются ссылки в каких-то других корневых объектах. К счастью, Visual Studio предоставляет средство для отслеживания «путей к корню» от любых объектов в снимке кучи. Правда таковых, как правило, оказывается довольно много, так что требуется выделение и проверка наиболее правдоподобных кандидатов.

Таким образом после нескольких дней дебаггинга наконец были обнаружены истинные переменные, сохраняющие ссылки на метаданные модулей. Ими оказался кэш типов, а также методы расширения для стандартных типов (int32, double, string и т. д.). И переменная кэша, и переменные стандартных типов являются статическими и никогда не очищались. Наверное, это самый типичный вид утечки памяти в .NET.

Исправление этой проблемы было достигнуто достаточно просто, в метод Compile системы Intellisense была добавлена очистка кэша и очистка методов расширения. В результате, после проведения тестирования было установлено, что проблема неограниченного нарастания количества занимаемой платформой памяти в случае модуля PABCSystem была полностью решена – значение последней достигало около 300 мегабайт и уже не увеличивалось.

1.2. Устранение утечек памяти, возникающих при компиляции программ с большим количеством шаблонных типов

Последние несколько лет от пользователей платформы PascalABC.NET поступали жалобы о непомерных тратах памяти в случае написания программ с высокой концентрацией шаблонных типов. В особых случаях это достигало таких масштабов, что приложение занимало всю память системы и переставало быть пригодным к использованию. По таким признакам так же, как и в предыдущем случае, было несложно предположить, что виновником является анализатор кода, ведь именно он занимается обработкой самого большого количества данных в небольшие промежутки времени.

Наводящим на корень проблемы признаком оказалось огромное количество одинаковых строк, отображаемых в профилировщике. Несложно было понять, что это за строки. Это были описания шаблонных типов данных, хранимые в Intellisense. (см. рис. 2)

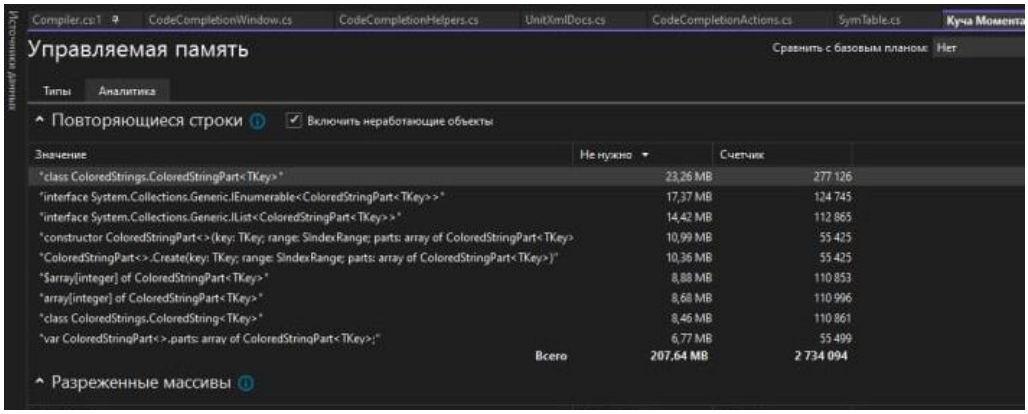

Рис. 2. Дубликаты описаний к шаблонным типам

Дубликаты строк означали дубликаты объектов метаданных о шаблонных типах и инстансах шаблонов. Для решения этой проблемы был введен особый кэш, хранящий в качестве ключей структуры с информацией о контексте создания экземпляров упомянутых ранее объектов. Этим контекстом является оригинальный шаблонный тип данных, а также конкретные типы-подстановки вместо шаблонных параметров. Для подсчета хэш-кода контекста используется хэш-код конкатенации строк-описаний элементов контекста. (см. листинг 1)

```
internal struct InstanceCreationContext : IEquatble<Instance…>
{
    public TypeScope originalType;
    public List<TypeScope> genericArguments;
    public bool exact;

    public InstanceCreationContext(TypeScope originalType,
List<TypeScope> genericArguments, bool exact)
    {
        this.originalType = originalType;
        this.genericArguments = genericArguments;
        this.exact = exact;
    }

    private struct ScopeComparer : IEqualityComparer<TypeScope>
    {
        public bool Equals(TypeScope t1, TypeScope t2) => t1.Is-
Equal(t2);
```

```
        public int GetHashCode(TypeScope t) => t.GetDescrip-
tion().GetHashCode();
    }

    public bool Equals(InstanceCreationContext otherInfo)
    {

        return this.originalType.IsEqual(otherInfo.originalType)
&&    this.genericArguments.SequenceEqual(otherInfo.genericArgu-
ments,
                                            new ScopeComparer())
&&
this.exact == otherInfo.exact;
    }

    public override int GetHashCode()
    {
        return string.Join("", genericArguments.Select(arg =>
 arg.GetDescription())
.Concat(new string[]{ originalType.GetDescription(),
 exact.ToString() })).GetHashCode();

    }
}
```

Листинг 1. Структура InstanceCreationContext

Очень важно было не вызвать новые утечки памяти и очищать новый кэш при каждой компиляции.

Тестирование новой функциональности показало ее эффективность. Теперь на файлах прежде вызывавших, по сути, отказ работы приложения Intellisense штатно справлялся со своей задачей, не увеличивая количество занимаемой памяти со временем.

2. Анализ отличий в реализации системы Intellisense для новых языков в инфраструктуре PascalABC.NET от реализации для основного языка платформы

В архитектуру системы Intellisense изначально были заложены следующие принципы. Система имеет ядро, которое по задумке не требует модификаций при добавлении в инфраструктуру новых языков программирования. Оно представляет из себя некоторый уровень абстракции, содержащий верхнеуровневые функции, такие как формирование подсказки при наведении мыши на токен, подсказки параметров функции, автодополнение, переход к определению символа и другие. Ядро же опирается на конкретные парсеры языков, причем как на основной LR-парсер для построения полного синтаксического дерева и его частей, так и на вспомогательный нисходящий парсер для нахождения в тексте выражений, заголовков методов и подобного. Также ядро использует своеобразную базу данных языка, чтобы брать из нее информацию для текста подсказок. Под базой данных в данном случае подразумеваются конкретные реализации интерфейса ILanguageInformation. Таким образом, по плану авторов системы Intellisense разработчикам нового языка платформы потребуется лишь добавить свою реализацию этого интерфейса.

Однако, некоторые части ядра все еще оставались жестко привязаны к основному языку платформы, что нарушает принцип устойчивости абстракций [1, с. 135]. Так, например, в функции GetPopupHintText, формирующей подсказку при наведении мыши, содержались проверки исходного кода на присутствие конкретных подстрок, специфических для языка PascalABC.NET. Также во многих местах можно было увидеть строку «PABCSystem» и «PABCExtensions», то есть названия стандартных модулей PascalABC.NET. Такой код, конечно же, требовалось обобщить.

Помимо проблемы привязки кода ядра к конкретному языку во время добавления нового языка – SPython обнаружились серьезные концептуальные расхождения в реализации Intellisense. Наиболее значительным из таких был факт того, что анализатору кода может требоваться вызов стандартных преобразователей синтаксического дерева, используемых пока только в основном компиляторе. Эта потребность возникла прежде всего из-за того, что в SPython появились новые синтаксические узлы, которые удаляются из дерева перед семантическим проходом в основном компиляторе. Компилятором Intellisense они также не поддерживаются, соответственно, нужно было или добавлять эту поддержку, или аналогично основному компилятору избавляться от них перед семантической обработкой. Для некоторых из узлов был выбран второй подход, а для некоторых первый, далее будет указано для каких.

Также были замечены несколько фундаментальных проблем при работе с информацией о местоположении функций в программе. Поскольку SPython не содержит операторные скобки, положение тела функции задается от его начала и до первого оператора вне функции. Это вызывает неприятный эффект для функции автодополнения: когда пользователь вводит первую букву некоторого токена ниже тела функции, но выше первого внешнего оператора, Intellisense добавляет в список подсказок в том числе локальные переменные и параметры функции независимо от того, есть ли отступ или его нет.

Вторая проблема состояла в том, что при определенном расположении функции в программе алгоритм поиска необходимой области видимости в Intellisense работал не до конца корректно, завершаясь при локализации в неявной функции MAIN, а не в этой функции. Дело в том, что в силу специфики языка SPython границы функции MAIN во многих случаях охватывают некоторые другие функции, но при этом эти функции не воспринимаются как вложенные. Чтобы устранить этот недостаток необходимо было внести изменения в алгоритм.

В архитектуре семейства классов LanguageInformation также требовались корректировки. Изначально интерфейс ILanguageInformation в случае

языка PascalABC.NET реализовывался напрямую в классе PascalABCLanguageInformation с заделом на то, чтобы аналогичные классы других языков становились наследниками этого. Однако, как известно, наследование — это самый сильный вид зависимости, что означает, что наследования от конкретных классов следует избегать [1, с. 102]. Вполне логичным решением в данном случае будет выделение базового абстрактного класса BaseLanguageInformation, реализующего некоторые общие алгоритмы и оставляющего реализацию специфических действий для наследников в виде PascalABCLanguageInformation, SPythonLanguageInformation и других.

3. Рефакторинг системы Intellisense для многоязыковой поддержки

3.1. Регистрозависимый поиск символов

Новый язык платформы PascalABC.NET SPython является регистрозависимым, соответственно требует поддержки регистрозависимого поиска имен сущностей. Прежде всего это касается поиска в таблице символов. Для реализации такой функциональности нужно было переработать несколько вещей. Во-первых, требовалось хранить имена в таблице в исходном виде, а не в строчном регистре как это было сделано. И во-вторых, нужно иметь возможность искать в таблице двумя способами: с учетом регистра и без учета. Для этого было принято решение создавать две одинаковые хэш-таблицы, но с различным видом поиска.

В классе SymbolsDictionary были модифицированы метод Find и Add, а именно в них была добавлена логика регистрозависимого поведения, как видно в листинге 2 на примере метода Find.

```
public HashTableNode Find(string name, bool inCaseSensitive)
{
    HashTableNode node;

    if (inCaseSensitive)
    {

        DictCaseSensitive.TryGetValue(name, out node);
    }
    else
    {
        dictCaseInsensitive.TryGetValue(name, out node);
    }

    return node;
}
```

Листинг 2. Метод Find таблицы символов

Помимо таблицы символов потребовалось сохранение информации о регистрозависимости в PCU файлы, для этого был создан специальный флаг languageCaseSensitive, была добавлена его сериализация и десериализация.

И последний компонент, требующий модификаций в этой связи — это подсистема поиска файлов модулей, часть компилятора платформы. Для всех ее методов был добавлен параметр caseSensitiveSearch, а сам регистрозависимый поиск был реализован с помощью метода Directory.GetFiles с передачей ему имени файла в качестве паттерна для поиска.

3.2. Модификация кода компиляции стандартных модулей в Intellisense

Компилятор Intellisense так же, как и основной компилятор неявно подключает к программам стандартные модули языка. Для этого, очевидно, он должен брать их имена из базы данных языка. Проблема состояла в том, что обращение к классу LanguageInformation для текущего языка программы не было реализовано, вместо этого вне зависимости от языка подключался модуль PABCSystem и модуль PABCExtensions. Это было исправлено следующим блоком кода (см. листинг 3).

```
foreach (var unitName in
   currentUnitLanguage.SystemUnitNames.Except(usedUnitsNames))
{
    AddStandardUnit(unitName);
}
```

Листинг 3. Добавление стандартных модулей

Как видно, имена стандартных модулей для текущего языка берутся из информации о языке и сравниваются с именами подключенных явно модулей. Если выясняется, что явно стандартные модули не были подключены вызывается функция добавления их в список uses.

Добавление стандартных типов в основной стандартный модуль было реализовано следующим образом (см. листинг 4).

```
string unitName = _unit_module.unit_name.idunit_name.name;

// считаем основной модуль идущим первым в списке
var language = Languages.Facade.LanguageProvider.Instance.Lan-
guages.Find(lang => lang.SystemUnitNames.First() == unitName);

if (language != null)
{
    AddStandardTypes(entryScope, language.LanguageInformation);

    if (language == Languages.Facade.LanguageProvider.In-
stance.MainLanguage)
    {
        AddPascalStandardProcedures();
    }
}
```

Листинг 4. Добавление стандартных типов

Это происходит в функции visit(unit_module _unit_module). Если текущий компилируемый модуль является главным стандартным модулем некоторого языка, то в него добавляются стандартные типы из этого языка. Также, если это PABCSystem, то в него необходимо добавить стандартные Паскалевские процедуры.

3.3. Модификация контроллера парсеров в Intellisense

В ядре Intellisense реализован контроллер парсеров, который выбирает нужный парсер в зависимости от расширения программы. Реализация этого была приемлема, но платформа PascalABC.NET в основном компиляторе перешла на более удобную концепцию выбора языка программирования, то есть класса, содержащего информацию и о парсере этого языка, и о системных модулях, и о других атрибутах. Для удобства и унификации Intellisense нужно было также перестроить на этот подход.

Для достижения указанной цели функция SetParser была переименована на SetLanguage и стала выглядеть так, как показано ниже (см. листинг 5).

```
public static void SetLanguage(string fileName)
{
    currentLanguage =
  LanguageProvider.SelectLanguageByExtensionSafe(fileName);

}
```

Листинг 5. Функция SetLanguage в контроллере Intellisense

Нужный парсер теперь достается из переменной currentLanguage как currentLanguage.Parser, аналогично можно запросить currentLanguage.DocParser (парсер документирующих комментариев), currentLanguage.CaseSensitive (чувствительность к регистру) и т. д.

3.4. Создание класса BaseLanguageInformation для выделения универсальных методов и минимизации дублирования кода

Язык PascalABC.NET использовал класс PascalABCLanguageInformation, как базу данных для Intellisense. При этом многие методы были сделаны виртуальными, чтобы в наследниках можно было их переопределять. Такой подход решено было пересмотреть и для этого был выделен базовый абстрактный класс [2, с. 336] BaseLanguageInformation с общими универсальными методами (в основном методами нисходящего парсера) и создав уже конкретного наследника PascalABCLanguageInformation.

Класс BaseLanguageInformation содержит абстрактные свойства по типу ParameterDelimiter, BodyStartBracket и других для того, чтобы в алгоритмы FindExpression, FindPattern, FindIdentifier и т. д. можно было сделать шаблонными (опираясь на паттерн проектирования Шаблонный метод [3, с. 309]) и универсальными. Пример универсальной реализации можно увидеть на методе GetSimpleDescriptionForProcedure (см. листинг 6).

```
protected string GetSimpleDescriptionForProcedure(IProcScope
scope)
{
    StringBuilder sb = new StringBuilder();
    if (scope.TopScope is ITypeScope)
        sb.Append(GetTopScopeName(scope.TopScope));
```

```
    sb.Append(scope.Name);
    sb.Append(GetGenericString(scope.TemplateParameters));
    sb.Append('(');
    IElementScope[] parameters = scope.Parameters;

    for (int i = 0; i < parameters.Length; i++)
    {
        sb.Append(GetSimpleDescription(parameters[i]));
        if (i < parameters.Length - 1)
        {
            sb.Append(ParameterDelimiter + " ");
        }
    }

    sb.Append(')');
    return sb.ToString();
}
```

Листинг 6. Функция GetSimpleDescriptionForProcedure

Как можно заметить в функции вызываются другие методы GetGenericString и GetSimpleDescription, которые могут быть переопределены в производных классах. ParameterDelimiter это свойство, которое также обычно переопределяется.

Помимо этого, в класс BaseLanguageInformation были добавлены некоторые новые объекты, а именно BaseKeywords (класс, содержащий ключевые слова языка и некоторые методы, связанные с ними) и Dictionary<string, DirectiveInfo> (информация о директивах, используемых в данном языке). Теперь и система Intellisense, и основной компилятор могут брать эти данные из одного места.

3.5. Добавление в ядро Intellisense вызовов конверторов синтаксического дерева

Новый язык платформы SPython был реализован согласно такой архитектуре, что для анализа синтаксического дерева системе Intellisense недостаточно просто вызвать парсер, кроме этого, требуется применить многие из конверторов. Чтобы обеспечить такое поведение первым шагом в интерфейс ILanguage был добавлен флаг ApplySyntaxTreeContvertersForIntellisense, определяющий нужно ли вообще для данного языка вызывать конверторы. Затем в

основной визитор Intellisense (DomSyntaxTreeVisitor) в методы обхода узла program_module и unit_module были добавлены следующие строчки кода (см. листинг 7).

```
if (language.ApplySyntaxTreeConvertersForIntellisense)
{
    var namesFromUsedUnits = CollectNamesFromUsedUn-
its(cur_scope);

    var artifacts = new CompilationArtifactsUsedBySyntaxConvert-
ers(namesFromUsedUnits);

    foreach (ISyntaxTreeConverter converter in language.Syntax-
TreeConverters)
    {
        _program_module = (program_module)converter.Convert-
AfterUsedModulesCompilation(_program_module, in artifacts);
    }
}
```

Листинг 7. Вызов конверторов в Intellisense

Здесь приведен пример вызовов конверторов второго типа, а именно тех, которые срабатывают после компиляции зависимых модулей, так как требуют некоторую информацию об именах из них. Функция CollectNamesFromUsedUnits собирает эту информацию и затем она передается в ConvertAfterUsedModulesCompilation.

4. Реализация работы системы Intellisense для языка SPython

4.1. Создание класса SPythonLanguageInformation

SPythonLanguageInformation так же, как и PascalABCLanguageInformation был унаследован от BaseLanguageInformation. В нем заданы конкретные ключевые слова, разделители и т. п. В методах FindExpression и подобных потребовалось поменять некоторые части алгоритма, чтобы учесть то, что SPython вообще не содержит операторных скобок, поддерживает оба вида кавычек для строковых литералов и наконец не поддерживает многострочные комментарии.

Для нового языка потребовалось добавить метод переименования и удаления из подсказок названий сущностей Intellisense (см. листинг 8).

```
public override void RenameOrExcludeSpecialNames(SymInfo[]
symInfos)
{
    for (var i = 0; i < symInfos.Length; i++)
    {
        if (symInfos[i] == null)
            continue;

        if (renamings.TryGetValue(symInfos[i].name, out var
newName))
        {
            symInfos[i] = new SymInfo(symInfos[i]);
            symInfos[i].name = newName;
        }
        else if (exclutions.Contains(symInfos[i].name))
        {
            symInfos[i] = new SymInfo(symInfos[i]);
            symInfos[i].not_include = true;
        }
    }
}
```

Листинг 8. Метод RenameOrExcludeSpecialNames

Этот метод вызывается в ядре Intellisense перед формированием подсказок при автодополнении.

Еще одним нововведением стал словарь SpecialModulesAliases (см. листинг 9), содержащий соответствие имен особых модулей и имен файлов для этих модулей. Дело в том, что всем известные модули random и time языка Python содержат функции, которые называются ровно так же, как и сам модуль. Такое не поддерживается платформой PascalABC.NET. Разработчики SPython решили эту проблему тем, что сделали имена файлов этих модулей отличающимися от их названий. Доступ к информации о таких особых случаях должна иметь и платформа Intellisense, поэтому словарь SpecialModulesAliases был размещен в общей базе данных языка. Теперь при обработке конструкций uses и import в синтаксическом дереве Intellisense при необходимости получает реальное название файла модуля и только потом запускает поиск.

```
private Dictionary<string, string> specialModulesAliases = new
Dictionary<string, string>
{
    { "time", "time1" },
    { "random", "random1" },
};
```

Листинг 9. Словарь specialModulesAliases

4.2. Реализация вызова части конверторов синтаксического дерева для системы Intellisense

Для того, чтобы корректно обрабатывать синтаксическое дерево программ на SPython Intellisense потребовалось вызывать часть из конверторов синтаксического дерева, применяемых в основном компиляторе платформы. Ниже представлен код их вызова (см. листинг 10).

```
protected override syntax_tree_node ApplyConversions(syn-
tax_tree_node root, bool forIntellisense)
{

    if (!forIntellisense)
        new UnsupportedConstructsVisitor().ProcessNode(root);

    if (!forIntellisense)
        new ImportToUsesVisitor().ProcessNode(root);
```

```
    new TryCatchDecorator(new BitwiseAssignmentDesugarVisitor(),
forIntellisense).ProcessNode(root);

    new TryCatchDecorator(new ReturnDesugarVisitor(), forIntel-
lisense).ProcessNode(root);

    new TryCatchDecorator(new MapDesugarVisitor(), forIntel-
lisense).ProcessNode(root);

    new TryCatchDecorator(new GeneratorObjectDesugarVisi-
tor(root), forIntellisense).ProcessNode(root);

    new TryCatchDecorator(new CompoundComparisonDesugarVisi-
tor(), forIntellisense).ProcessNode(root);

    return root;
}
```

Листинг 10. Вызовы конверторов синтаксического дерева в SPython (1)

Те конверторы, которые используются в Intellisense, оборачиваются в класс TryCatchDecorator (см. листинг 11), который никак не меняет логику в случае запуска из основного компилятора, но оборачивает в блок try … catch для запуска из Intellisense (см. паттерн Декоратор [3, с. 173]). Как известно, система Intellisense должна быть максимально устойчива к ошибкам в программе. В данном случае есть возможность улучшить это качество, так как многие визитеры работают независимо от результатов работы предыдущих.

```
private class TryCatchDecorator
{
    private WalkingVisitorNew internalVisitor;
    private bool forIntellisense;

    public TryCatchDecorator(WalkingVisitorNew internalVisitor,
bool forIntellisense)
    {
        this.internalVisitor = internalVisitor;
        this.forIntellisense = forIntellisense;
    }

    public bool ProcessNode(syntax_tree_node root)
    {
        if (forIntellisense)
        {
            try
            {
```

```
                internalVisitor.ProcessNode(root);
            }
            catch (Exception)
            {
                return false;
            }
        }
        else
        {
            internalVisitor.ProcessNode(root);
        }

        return true;
    }
}
```

Листинг 11. Класс TryCatchDecorator

UnsupportedConstructsVisitor занимается исключительно тем, что бросает ошибки при встрече недопустимых конструкций. Поэтому для Intellisense его вызов нужно вообще отключить. ImportToUsesVisitor заменяет узлы import и from … import на узел uses, чтобы основной компилятор смог корректно обработать синтаксическое дерево. Его вызов также не нужен в случае работы Intellisense, потому что для анализатора кода важно сохранить информацию о том, какой узел был в исходном варианте программы.

Рассмотрим также вызовы конверторов, применяемых после компиляции зависимых модулей (см. листинг 12). Второй применяемый конвертор (NameCorrectVisitor) является критической стадией компиляции, в случае его неуспеха дальнейшие визитеры применять не имеет смысла, поэтому результат его работы отслеживается и в случае чего происходит выход из функции конвертации. Визитер TypeCorrectVisitor меняет свою стратегию в зависимости от того применяется он для Intellisense или нет. Для Intellisense, в отличие от основного компилятора, не требуется замена имен стандартных типов (кроме типов коллекций) на их аналоги из PascalABC.NET.

Также видно, что для нужд Intellisense не применяется визитер EraseSpythonOnlyNodesVisitor.

```
public override syntax_tree_node ConvertAfterUsedModulesCompila-
tion(syntax_tree_node root, bool forIntellisense, in Compila-
tionArtifactsUsedBySyntaxConverters compilationArtifacts)
{
    var ffv = new FindFunctionsNamesVisitor();

    new TryCatchDecorator(ffv, forIntellisense).Process-
Node(root);

    var ncv = new NameCorrectVisitor(compilationArtifacts.Names-
FromUsedUnits, ffv.definedFunctionsNames);

    if (!new TryCatchDecorator(ncv, forIntellisense).Process-
Node(root))
        return root;

    new TryCatchDecorator(new TypeCorrectVisitor(forIntel-
lisense), forIntellisense).ProcessNode(root);

    var binder = new BindCollectLightSymInfo(root as compila-
tion_unit);
    new TryCatchDecorator(binder, forIntellisense).Process-
Node(root);
    new TryCatchDecorator(new NewAssignTuplesDesugarVisi-
tor(binder), forIntellisense).ProcessNode(root);

    new TryCatchDecorator(new AddForwardDeclarationsVisitor(),
forIntellisense).ProcessNode(root);

    new TryCatchDecorator(new KwargsFunctionDesugarVisitor(),
forIntellisense).ProcessNode(root);

    new TryCatchDecorator(new FunctionsWithNamedParame-
tersDesugarVisitor(), forIntellisense).ProcessNode(root);

    new TryCatchDecorator(new RetainUsedGlobalVariablesVisi-
tor(ncv.variablesUsedAsGlobal), forIntellisense).Process-
Node(root);

    if (!forIntellisense)
        new EraseSpythonOnlyNodesVisitor().ProcessNode(root);

    new TryCatchDecorator(new TreeNodesRearrangementVisitor(),
forIntellisense).ProcessNode(root);

    return root;
}
```

Листинг 12. Вызовы конверторов синтаксического дерева в SPython (2)

4.3. Реализация обработки операторов import и from … import в Intellisense

Как было отмечено ранее для Intellisense в синтаксическом дереве программы оставляются узлы import и from … import. Их обработка должна происходить в том же месте, где это сделано для конструкций uses, то есть перед компиляцией программы или модуля в функциях visit(program_module) и visit(unit_module) соответственно.

В функции CompileImportedDependencies (см. листинг 13) узлы import и from … import выделяются из общего списка операторов, а затем проходятся в обратном порядке для компиляции. Если внутри конструкции import несколько модулей, то они также проходятся в обратном порядке.

```
private void CompileImportedDependencies(statement_list state-
mentList, string fileName, Hashtable ns_cache, List<string>
usedUnitNames, bool caseSensitiveSearch)
{
    var importStatements = statementList.list.Where(st => st is
import_statement || st is from_import_statement);

    foreach (var importStatement in importStatements.Reverse())
    {
        if (importStatement is import_statement import)
        {
            foreach (var unitNode in import.mod-
ules_names.as_statements.Select(st => st.real_name).Reverse())
            {
                add_unit_ref_for_import(new ident_list(unit
                    Node), import, import.modules_names,
                    Path.GetDirectoryName(fileName),
                    cur_scope, ns_cache,
                    semantic_options.allow_import_types,
                    unl, caseSensitiveSearch);

                usedUnitNames.Add(unitNode.name);
            }
        }
        else if (importStatement is from_import_statement fro-
mImport)
        {
            add_unit_ref_for_import(new ident_list(
                fromImport.module_name), fromImport,
                fromImport.imported_names,
                Path.GetDirectoryName(fileName),
                cur_scope, ns_cache,
                semantic_options.allow_import_types,
```

```
                unl, caseSensitiveSearch);

            usedUnitNames.Add(fromImport.module_name.name);
        }
    }
}
```

Листинг 13. Функция CompileImportedDependencies

В функции add_unit_ref_for_import происходит компиляция зависимых модулей, а также добавление нужных имен в текущую область видимости. Последнее рассмотрим подробно.

Как видно в функции AddImportedNamesToCurScope (см. листинг 14) вначале происходит поиск в списке используемых модулей модуля с нужным именем. Если такой есть, то новые имена будут добавляться в его область видимости, если же нет, то будет формироваться фиктивный модуль, содержащий нужные имена. Так в случае import в фиктивный модуль добавляется имя и область видимости импортируемого модуля, а в случае from … import добавляются импортируемые имена.

```
private static void AddImportedNamesToCurScope(statement im-
portStatement, as_statement_list asStatementsList, SymScope cur-
rentScope, string importedModuleName, SymScope importedModule-
Scope, bool notFinalNamespaceName = false)
{
    int fictiveUnitIndex = currentScope.used_units.Find-
Index(unit => unit.Name == importedModuleName);

    SymScope fictiveUnit = fictiveUnitIndex > -1 ? cur-
rentScope.used_units[fictiveUnitIndex] : null;

    if (importStatement is import_statement import)
    {
        if (fictiveUnit == null)
        {
            fictiveUnit = new InterfaceUnitScope(new SymInfo(im-
portedModuleName, SymbolKind.Namespace, ""), null);

            currentScope.AddUsedUnit(fictiveUnit);
        }

        string aliasName = asStatementsList.as_state-
ments.Find(st => st.real_name.name == importedModu-
leName).alias.name;

        fictiveUnit.AddName(aliasName, importedModuleScope);
    }
```

```
    else if (importStatement is from_import_statement fromImport
&& !notFinalNamespaceName)
    {
        if (fromImport.is_star)
        {
            if (fictiveUnit != null)
                currentScope.used_units.RemoveAt(fictiveUnitIn-
dex);

            currentScope.AddUsedUnit(importedModuleScope);
        }
        else
        {
            if (fictiveUnit == null)
            {
                fictiveUnit = new InterfaceUnitScope(new
SymInfo(importedModuleName, SymbolKind.Namespace, ""), null);

                currentScope.AddUsedUnit(fictiveUnit);
            }

            foreach (var asStatement in asState-
mentsList.as_statements)
            {
                SymScope nameScope = importedModuleScope.Find-
NameOnlyInType(asStatement.real_name.name);

                if (nameScope != null)
                {
                    fictiveUnit.AddName(asStatement.alias.name,
nameScope);
                }
            }
        }
    }
}
```

Листинг 14. Функция AddImportedNamesToCurScope

4.4. Модификация алгоритма поиска области видимости по положению курсора

Как упоминалось ранее, язык SPython имеет особую специфику задания информации о положении функций в программе. В частности, неявная функция MAIN часто захватывает всю программу, включая реальные функции. При этом механизма вложенных функций не существует и эти реальные функции располагаются на том же уровне, что и MAIN в дереве областей видимости.

Это приводит к тому, что исходный алгоритм, обходя дерево в глубину, может определить, что некоторые локальные переменные функций находятся внутри MAIN и завершить свою работу, хотя настоящая функция, содержащая данную область кода, располагается ниже по списку.

Чтобы решить данную проблему было принято решение изменить алгоритм поиска для языков с такой спецификой. А именно в функцию FindScopeByLocation (см. листинг 15) теперь передается еще один аргумент – minScope. Это минимальная по размеру область видимости, включающая нужное положение в программе. Чтобы определить ее, дерево областей видимости обходится полностью, и подходящие кандидаты сравниваются по количеству занимаемых строк в функции UpdateMinScopeIfNeeded.

```
public virtual SymScope FindScopeByLocation(int line, int col-
umn)
{
    SymScope minScope = null;

    bool findMinScope = CodeCompletionController.Current-
Parser.LanguageInformation.UsesFunctionsOverlappingSourceCon-
text;

    SymScope foundScope = FindScopeByLocation(line, column, ref
minScope, findMinScope);

    return findMinScope ? minScope : foundScope;
}

public virtual SymScope FindScopeByLocation(int line, int col-
umn, ref SymScope minScope, bool findMinScope)
{
    SymScope res = null;

    cur_line = line;
    cur_col = column;

    if (IsInScope(loc, line, column))
        res = this;

    foreach (SymScope ss in members)
        if (this != ss && this.topScope != ss && ss.loc != null)
        {
            if (IsInScope(ss.loc, line, column))
            {
                if (this is TypeScope && ss.loc.end_line_num >
loc.end_line_num)
```

```
                    continue;
                res = ss;

                SymScope tmp = ss.FindScopeByLocation(line, col-
umn, ref minScope, findMinScope);
                if (tmp != null)
                {
                    res = tmp;
                }

                if (!findMinScope)
                    return res;

                UpdateMinScopeIfNeeded(ref minScope, res);
            }
            else if (!(ss is CompiledScope))
            {
                SymScope tmp = ss.FindScopeByLocation(line, col-
umn, ref minScope, findMinScope);
                if (tmp != null)
                {
                    res = tmp;

                    if (!findMinScope)
                        return res;

                    UpdateMinScopeIfNeeded(ref minScope, res);
                }
            }
        }
    return res;
}

private static SymScope UpdateMinScopeIfNeeded(ref SymScope min-
Scope, SymScope scopeToCompare)
{
    if (minScope != null)
    {
        if (scopeToCompare.loc.end_line_num - scopeToCom-
pare.loc.begin_line_num <= minScope.loc.end_line_num - min-
Scope.loc.begin_line_num)
            minScope = scopeToCompare;
    }
    else
    {
        minScope = scopeToCompare;
    }

    return minScope;
}
```

Листинг 15. Функции FindScopeByLocation и UpdateMinScopeIfNeeded

5. Разработка LSP-сервера для платформы PascalABC.NET

Команда разработчиков PascalABC.NET работает над созданием новой кроссплатформенной IDE. Систему Intellisense, так же, как и основной компилятор решено было использовать в рамках обращения к отдельному процессу, то есть языковому серверу. Выбор протокола LSP для общения клиента и сервера объясняется несколькими причинами. Во-первых, это проверенное в промышленности решение и команде PascalABC.NET не придется выдумывать свой велосипед. Во-вторых, LSP-подход позволит за бесплатно реализовать поддержку языков платформы в среде Visual Studio Code и других известных IDE, поскольку LSP-клиент там реализован. И наконец, сообщество OmniSharp предоставляет библиотеку, содержащую готовый низкоуровневый серверный код, а также интерфейсы обработчиков Intellisense, которые можно реализовывать для работы своего языкового сервера.

5.1. Создание отдельного компонента для контроллеров системы Intellisense

LSP-сервер представляет из себя самостоятельную программу, отдельную от оболочки PascalABC.NET. Для работы ему нужно будет обращаться к контроллерам Intellisense. Чтобы сделать это общение более чистым с точки зрения архитектуры, нужно было извлечь контроллеры из компонента VisualPascalABCNET и поместить в отдельный компонент.

Для поддержки основных функций языкового сервера в отдельный компонент были перемещены классы CodeCompletionParserController, CodeCompletionProvider, InsightProvider, TooltipServiceManager и CodeCompletionHelper.

CodeCompletionParserController обеспечивает запуск парсинга Intellisense в фоновом режиме. Стратегия парсинга не менялась, это все так же запуск анализатора каждые две секунды для всех измененных или новых

открытых файлов. На более современный поменялся код завершения потока с Intellisense. Теперь при завершении работы посылается специальный сигнал, для этого используется общепринятая в языке C# техника cancellationToken [4]. Как видно в листинге 16 в SwitchOffIntellisense вызывается функция cancellationTokenSource.Cancel(), после чего бросается исключение OperationCancelException, которое перехватывается в функции BackGroundParsingLoop. В цикле парсинга также перехватываются другие исключения, повторная попытка обработки совершается через пол секунды.

```
public void SwitchOffIntellisense()
{
    cancellationTokenSource.Cancel();
    parseLockEvent.Set();
    backgroundParsingThread.Join();
}


private void BackgroundParsingLoop()
{
while (!cancellationTokenSource.IsCancellationRequested)
    {
        try
        {
            ParseInThread();

            parseLockEvent.Wait(TimeSpan.FromSeconds(2), cancel-
lationTokenSource.Token);
        }
        catch (OperationCanceledException)
        {
            // Выход при отмене
            break;
        }
        catch (Exception e)
        {

            Console.Error.WriteLine("Intellisense exception:");
            Console.Error.WriteLine(e.Message + "\n" + e.Stack-
Trace);

            // ожидание перед повторной попыткой
            Thread.Sleep(500);
        }
    }
}
```

Листинг 16. Функции SwitchOffIntellisense и BackGroundParsingLoop

Остальные классы контроллеров Intellisense перенесены в новый компонент практически без изменений за исключением того, что из них убран код, связанный с WindowsForms и ICSharpCodeTextEditor.

5.2. Реализация функциональности LSP-сервера на основе библиотеки Omnisharp.Extensions.LanguageServer

Для реализации LSP-сервера в качестве основы была взята библиотека Omnisharp.Extensions.LanguageServer версии 0.8.0 под .NET Framework. В качестве потоков ввода и вывода были выбраны StandardInput и StandardOutput. Как видно в коде ниже (см. листинг 17), были реализованы следующие обработчики: TextDocumentSyncHandler для учета изменений файла в редакторе, CompletionHandler для автодополнения, HoverHandler для подсказок при наведении курсора и SignatureHelpHandler для подсказок параметров функции.

```
static async Task Main()
{
    PascalABCCompiler.StringResourcesLanguage.LoadDefaultCon-
fig();

    Languages.Integration.LanguageIntegrator.LoadAllLanguages();

    var loggerFactory = new LoggerFactory();

    var server = new LanguageServer(
            Console.OpenStandardInput(),
            Console.OpenStandardOutput(),
            loggerFactory,
            true
     );

    var documentStorage = new DocumentStorage();

    server.AddHandler(new TextDocumentSyncHandler(documentStor-
age));

    server.AddHandler(new CompletionHandler(documentStorage));

    server.AddHandler(new HoverHandler(documentStorage));
```

```
    server.AddHandler(new SignatureHelpHandler(documentStor-
age));

    await server.Initialize();

    await server.WaitForExit;
}
```

Листинг 17. Точка входа LSP-сервера

5.2.1. Синхронизация изменений в файлах

Обработчик TextDocumentSyncHandler реагирует на открытие, закрытие и изменение файла. Чтобы хранить текст файла в актуальном состоянии созданы классы DocumentStorage и TextBuffer. Для примера рассмотрим обработку открытия файла (см. листинг 18). В первой строке метода достается абсолютный путь к открытому файлу. Он используется как ключ при сохранении документа в DocumentStorage во второй строке. В последней строке файл добавляется в очередь для обработки системой Intellisense.

```
public Task Handle(DidOpenTextDocumentParams request)
{
    string documentPath = LspDataConvertor.GetNormalizedPath-
FromUri(request.TextDocument.Uri);

    documentStorage.AddDocument(documentPath, request.TextDocu-
ment.Text);

    codeCompletionController.RegisterFileForParsing(docu-
mentPath);

    return Task.CompletedTask;
}
```

Листинг 18. Обработка открытия файла

Класс DocumentStorage внутри использует потокобезопасную версию словаря (ConcurrentDictionary) для хранения соответствий путей к файлу и их содержимого. Потокобезопасность здесь важна, поскольку основной поток сервера конкурирует с потоком парсинга при доступе к тексту файлов. Самым интересным методом в классе является метод UpdateDocument (см. листинг 19), остальные методы являются простыми обертками над методами словаря.

При изменении текста файла из буффера при необходимости удаляется старое содержимое на месте изменения и вставляется новое.

```
internal class DocumentStorage : PascalABCCom-
piler.ISourceTextProvider
{
    private readonly ConcurrentDictionary<string, TextBuffer>
documents = new ConcurrentDictionary<string, TextBuffer>();

    public void UpdateDocument(string documentPath, TextDocu-
mentContentChangeEvent change)
    {
        var buffer = documents[documentPath];

        var range = change.Range;

        int startLine = (int)range.Start.Line;
        int startCol = (int)range.Start.Character;
        int endLine = (int)range.End.Line;
        int endCol = (int)range.End.Character;

        if (change.RangeLength > 0)
            buffer.Delete(startLine, startCol, endLine, endCol);

        buffer.Insert(startLine, startCol, change.Text);
    }

    ...
}
```

Листинг 19. Часть класса DocumentStorage

Класс TextBuffer специально спроектирован для эффективных изменений содержимого при условии передачи только измененной части файла (см. листинг 20). При создании экземпляра он сохраняет содержимое файла в виде списка строк. Методы Insert и Delete симметричны. В методе Insert тривиален случай вставки одной строки, он реализуется как обычная вставка в нужную строку файла. Если же вставляемых строк несколько, то Insert работает следующим образом: из исходной строки текста выделяются две части, одна до места вставки, другая – после. Затем формируется список вставляемых строк, первая из которых получается склеиванием первой части исходной строки и первой добавленной строки, далее входят добавленные строки со второй по предпоследнюю и наконец склеенная последняя добавленная строка с концом

исходной строки. В конце собранные строки вставляются вместо строки, которая была в нужном месте.

```
internal class TextBuffer
{
    private List<string> lines;

    public TextBuffer(string initialText)
    {
        UpdateText(initialText);
    }


    public void UpdateText(string text)
    {
        lines = text.Split(new[] { Environment.NewLine },
                           StringSplitOptions.None).ToList();
    }

    public void Insert(int line, int column, string text)
    {
        string originalLine = lines[line];

        string[] insertedLines = text.Split(new[] { Environ
                    ment.NewLine }, StringSplitOptions.None);

        if (insertedLines.Length == 1)
        {
            lines[line] = originalLine.Insert(column, inserted
                                                    Lines[0]);
        }
        else
        {
            string firstPart = originalLine.Substring(0, col
                                                        umn);
            string lastPart = originalLine.Substring(column);

            var newLinesToInsert = new List<string>(){firstPart
                                            + insertedLines[0]};

            newLinesToInsert.AddRange(inserted
             Lines.Skip(1).Take(insertedLines.Length - 2));

            newLinesToInsert.Add(insertedLines[inserted
                                Lines.Length - 1] + lastPart);

            lines.RemoveAt(line);
            lines.InsertRange(line, newLinesToInsert);
        }
        ...
}
```

Листинг 20. Часть класса TextBuffer

5.2.2. Автодополнение

Рассмотрим метод Handle класса CompletionHandler (см. листинг 21). Вначале мы видим обращения к классу DocumentStorage для получения текста файла, а также буффера. Буффер позволяет эффективно получить текст измененной строки файла. Далее выясняется символ, который вызвал функцию автодополнения. Если предыдущий относительно текущего символа в строке является пробельным символом, то считаем, что был использован явный вызов автодополнения при помощи комбинации клавиш ctrl + space (переменной triggerChar в этом случае присваивается символ нижнего подчеркивания).

Затем при помощи буффера эффективно получаем индекс triggerChar в тексте программы. В блоке try … catch для случая подсказки по точке, либо ctrl + space вызываем функцию GenerateCompletionDataWithKeyword класса CompletionProvider, а для случая автодополнения после ввода первого символа функцию GenerateCompletionDataByFirstChar того же класса.

В конце конвертируем список подсказок в нужный формат, используя при этом поле InsertText для случаев, когда вставляемый текст не совпадает с отображаемым в подсказке.

```
internal class CompletionHandler : ICompletionHandler
{
    public Task<CompletionList> Handle(TextDocumentPositionPar-
ams request, CancellationToken cancellationToken)
{
    var documentPath = LspDataConvertor.GetNormalizedPath-
FromUri(request.TextDocument.Uri);

    Position pos = request.Position;

    var docText = documentStorage.GetDocumentText(documentPath);

    var buffer = documentStorage.GetDocumentBuffer(docu-
mentPath);

    var changedLineText = buffer.GetLine((int)pos.Line);

    int col = (int)pos.Character;

    char triggerChar = col > 0 ? changedLineText[col - 1] : '_';

    if (char.IsWhiteSpace(triggerChar))
```

```
    {
        triggerChar = '_';
    }

    UserDefaultCompletionData[] completionList = null;

    int symbolIndex = buffer.GetOffsetFromPosi-
tion((int)pos.Line, (int)pos.Character) - 1;

    try
    {
        if (specialTriggerCharacters.Contains(trigger-
Char.ToString()) || triggerChar == '_')
        {
            completionList = completionProvider.GenerateComple-
tionDataWithKeyword(documentPath, docText,
                    symbolIndex,
                    (int)pos.Line, (int)pos.Character, trigger-
Char, PascalABCCompiler.Parsers.KeywordKind.None);
        }
        else
        {
            completionList = completionProvider.GenerateComple-
tionDataByFirstChar(documentPath, docText, symbolIndex,
(int)pos.Line, (int)pos.Character, triggerChar, keyw);
        }
    }
    catch (Exception e) {}
    if (completionList != null)
    {
        var resultList = new CompletionList(completionList.Se-
lect(item => new CompletionItem() {
            Label = item.Text,
            Detail = item.Description,
            InsertText = GetInsertedTextForComple-
tionItem(item.Text)
        }));

        return Task.FromResult(resultList);
    }

    return Task.FromResult(new CompletionList());
}

...
}
```

Листинг 21. Метод Handle класса CompletionHandler

5.2.3. Подсказка при наведении

Перейдем к методу Handle класса HoverHandler (см. листинг 22). В нем так же, как и в Handle из CompletionHandler для работы требуются полный текст файла и буффер. С помощью буффера мы получаем индекс символа на позиции курсора в тексте. Для получения подсказки метод обращается к TooltipServiceManager.GetPopupHintText. Для формирования результата работы метод оборачивает подсказку в класс MarkedStringsOrMarkupContent, а также обращается к утилите LspDataConverter для получения границ токена, на который был наведен курсор.

```
internal class HoverHandler : IHoverHandler
{
    Task<Hover> IRequestHandler<TextDocumentPositionParams,
Hover>.Handle(TextDocumentPositionParams request, Cancella-
tionToken token)
    {
        var documentPath = LspDataConvertor.GetNormalizedPath-
    FromUri(request.TextDocument.Uri);

        Position pos = request.Position;

        var docText = documentStorage.GetDocumentText(docu-
    mentPath);

        var buffer = documentStorage.GetDocumentBuffer(docu-
    mentPath);

        int offset = buffer.GetOffsetFromPosition((int)pos.Line,
    (int)pos.Character);

        string hintText = null;

        try
        {
            hintText = TooltipServiceManager.Get-
    PopupHintText(offset, (int)pos.Line, (int)pos.Character,
    docText, documentPath);
        }
        catch (Exception) { }

        if (hintText != null)
        {
            var markedContent = new MarkupContent()
            {
                Kind = MarkupKind.Plaintext,
                Value = hintText
```

```
                };

                return Task.FromResult(
                    new Hover()
                    {
                        Contents = new MarkedStringsOrMarkupCon-
    tent(markedContent),
                        Range = LspDataConvertor.GetRangeForSymbo-
    lAtPos(buffer.GetLine((int)pos.Line), pos)
                    }
                    );
            }

            return Task.FromResult(new Hover());
        }
}
```

Листинг 22. Метод Handle класса HoverHandler

5.2.4. Подсказка параметров функции

Последний из созданных обработчиков, предоставляющий подсказку параметров функции реализован в классе SignatureHelpHandler. Рассмотрим его метод Handle (см. листинг 23). Как и раньше, для работы ему требуется текст программы и буффер. Переменная triggerChar в данном случае может быть символом скобки для вызова функции, либо символом скобки индексатора. Класс DefaultInsightDataProvider реализует необходимую функциональность. После вызова метода SetupDataProvider из переменной methods этого класса можно получить строки подсказок параметров. Чтобы сформировать итоговый список подсказок строковое представление парсится в функции LspDataConvertor.GetLspMethodsInfoFromStringDescriptions и преобразуется в список объектов SignatureInformation. Переменная activeSignature, обозначающая текущую активную сигнатуру метода (которых может быть несколько, если у метода есть перегрузки) также достается из класса DefaultInsightDataProvider. Немного больше манипуляций требуется для задания значения переменной activeParameter, то есть индексу текущего вводимого параметра. Переменная insightProvider.num_param содержит реальный номер текущего параметра, вводимого пользователем, однако ее нужно ограничить сверху количеством параметров функции, а также отдельно рассматривается случай,

когда функция позволяет передавать неограниченное количество параметров определенного типа.

```
internal class SignatureHelpHandler : ISignatureHelpHandler
{
    public Task<SignatureHelp> Handle(TextDocumentPositionParams
request, CancellationToken token)
    {
        var documentPath = LspDataConvertor.GetNormalizedPath-
    FromUri(request.TextDocument.Uri);

        Position pos = request.Position;

        var docText = documentStorage.GetDocumentText(docu-
    mentPath);

        var buffer = documentStorage.GetDocumentBuffer(docu-
    mentPath);

        var changedLineText = buffer.GetLine((int)pos.Line);

        char triggerChar = changedLineText.Substring(0,
    (int)pos.Character).Last();

        var insightProvider = new DefaultInsightDataProvider(-1,
    triggerChar);

        int symbolIndex = buffer.GetOffsetFromPosi-
    tion((int)pos.Line, (int)pos.Character) - 1;

        try
        {
            insightProvider.SetupDataProvider(documentPath, doc-
    Text, symbolIndex, (int)pos.Line, (int)pos.Character);
        }
        catch (Exception) { }

        if (insightProvider.methods == null)
        {
            return Task.FromResult(new SignatureHelp());
        }

        List<SignatureInformation> resultMethods = LspDataCon-
    vertor.GetLspMethodsInfoFromStringDescriptions(insightPro-
    vider.methods, triggerChar == '[');

        int activeSignature = insightProvider.DefaultIndex;

        int activeParameter = -1;

        var paramsOfActiveSignature = resultMethods[activeSigna-
    ture].Parameters;
```

```
            if (paramsOfActiveSignature.Count() > 0)
            {
                var lastParam = resultMethods[activeSignature].Pa-
        rameters.Last().Label;

                if (lastParam.EndsWith("...") || lastParam.Trim-
        Start().StartsWith("params"))
                {
                    activeParameter = Math.Min(insightPro-
        vider.num_param, paramsOfActiveSignature.Count()) - 1;
                }
                else
                {
                    activeParameter = insightProvider.num_param <=
        paramsOfActiveSignature.Count() ? insightProvider.num_param
        - 1 : -1;
                }
            }

            return Task.FromResult(new SignatureHelp()
            {
                ActiveParameter = activeParameter,
                ActiveSignature = activeSignature,
                Signatures = resultMethods
            }
            );

        }
    }
```

Листинг 23. Метод Handle класса SignatureHelpHandler

6. Использование генеративного ИИ при написании работы

В процессе написания работы генеративный ИИ, а именно Deepseek, использовался несколько раз. Первое применение состояло в поиске источников по теме утечек памяти в .NET приложениях, а также поиске средств для их исправления. DeepSeek подсказал несколько вариантов, которые были реально существующими и вполне адекватными. Также при реализации LSP-сервера DeepSeek использовался для того, чтобы узнать принятые практики по написанию класса TextBuffer, он подсказал подход с разделением текста на массив строк и предложил примерную реализацию методов Insert и Delete. Его советы использовались с небольшими правками.

Заключение

В ходе дипломной работы были решены все поставленные задачи, а именно:

- Выявлены и устранены основные причины утечек памяти в платформе PascalABC.NET, связанные с системой Intellisense
- Проанализированы различия в системе Intellisense для различных языков в инфраструктуре PascalABC.NET
- Проведен рефакторинг кода в проекте PascalABC.NET, направленный на реализацию Intellisense для различных языков программирования
- Реализована поддержка нового языка платформы – SPython в системе Intellisense
- Система Intellisense адаптирована для использования ее в виде LSP-сервера платформы PascalABC.NET, поддерживающего функции автодополнения, подсказки по наведению и подсказки параметров функций

Таким образом, достигнуты значительные результаты по переработке и усовершенствованию системы Intellisense. Устранены критические уязвимости в виде утечек памяти, улучшена поддержка многоязыковости, проделана работа по построению новой архитектуры в виде LSP-сервера.

Изменения сформированы в «пулл реквесты» и залиты в репозиторий платформы PascalABC.NET.

Список литературы

1. Мартин Р. Чистая архитектура. Искусство разработки программного обеспечения. / Роберт Мартин. – Пер. с англ. – СПб.: Питер, 2018. – 352 с.: ил. – (Серия «Библиотека программиста»).
2. Фаулер М. Рефакторинг: улучшение существующего кода. / Мартин Фаулер. – Пер. с англ. – СПб: Символ Плюс, 2006. – 432 с., ил.
3. Приемы объектно-ориентированного проектирования. Паттерны проектирования. / Гамма Э., Хелм Р., Джонсон Р., Влиссидес Дж. – Пер. с англ. – СПб: Питер, 2001. – 368 с.: ил. – (Серия «Библиотека программиста»).
4. Отмена в управляемых потоках. – URL: https://learn.microsoft.com/ru-ru/dotnet/standard/threading/cancellation-in-managed-threads (дата обращения 28.05.2025)