Московский Государственный Университет имени М.В. Ломоносова
Факультет Вычислительной Математики и Кибернетики
Кафедра Автоматизации Систем Вычислительных Комплексов

ДИПЛОМНАЯ РАБОТА

Исследование интерфейса инструментальных окон среды разработки программ

Выполнил **Ручкин И.Д.**

Научный руководитель **Прус В.В.**

Москва 2011

Аннотация

Данная дипломная работа посвящена исследованию интерфейса интегрированных сред разработки программ и подхода к решению проблемы перегруженности интерфейса сред инструментальными окнами. Сначала проводится обзор предметной области и выделение сценариев работы программиста и списка обобщенных инструментальных окон. На основе полученных данных проектируется новый интерфейс без инструментальных окон. Далее проводится реализация этого интерфейса.

Также эта работа включает эмпирическое исследование реализованного интерфейса. Это исследование на реальных пользователях состоит в выполнении типичных задач программиста при помощи нового предлагаемого интерфейса. Оно дает оценку удобства предлагаемого в работе подхода к проектированию интерфейса сред разработки.

Содержание

1. Введение

Существует большое число программных средств, используемых программистами для создания программ. Как правило, программисты используют либо простые текстовые редакторы (к примеру, Notepad и Kate) с набором заранее написанных инструментов, либо сложные среды разработки программ (*IDE – Integrated Development Environment* [1]), как Eclipse и Visual Studio. Такие среды объединяют различные средства, например отладчик и компилятор, для предоставления пользователю удобных возможностей, недоступных в случае использования обычного текстового редактора и отдельных инструментов. Например, в IDE может присутствовать автоматическое дополнение кода, проверка синтаксической правильности программы без компиляции, возможность задавать точки останова в коде и так далее [2] [3].

Несмотря на эти привлекательные функции, многие программисты считают IDE переусложненными и предпочитают пользоваться простыми текстовыми редакторами и отдельными инструментами разработки [3] [4]. Полная картина причин этого не ясна, и для ее выяснения может потребоваться масштабное исследование удобства[1] многочисленных IDE [5] [6].

Однако эмпирический опыт показывает, что одной из заметных проблем интерфейса IDE являются инструментальные окна – дополнительные окна среды разработки, отличные от окна редактирования кода [7] [8], находящиеся либо внутри основного окна, либо в виде самостоятельных окон. Эти дополнительные окна ставят пользователя среды перед альтернативой: либо постоянно держать их открытыми, уменьшая размер экранного пространства, доступного для редактирования кода, либо постоянно отвлекаться на их открытие и закрытие вручную. Эти действия отвлекают и утомляют программиста [9]. Более того, современные IDE в качестве основного своего механизма расширения предлагают добавление инструментальных окон, а увеличение их числа делает использование интерфейса еще сложнее. Указанные трудности использования IDE мы будем называть *проблемой инструментальных окон.*

В средах разработки существуют подходы к сглаживанию проблемы инструментальных окон. Иногда окна показываются автоматически (например, список ошибок сборки при появлении первой ошибки), что избавляет пользователя от необходимости активировать их вручную. Также в некоторых IDE есть понятие перспектив (*perspective* или *area* [10]) – они определяют наборы и расположение инструментальных окон, позволяя мгновенно переключаться между ними. К примеру, можно определить разные перспективы для отладки и написания кода.

[1] *Удобство (usability)* [5] – эргономичность и легкость использования графического интерфейса.

Эти попытки решить проблему инструментальных окон, по мнению автора, малоэффективны, поскольку они игнорируют следующий факт: информация, находящаяся в дополнительных окнах может быть нужна пользователю по отдельности, а инструментальное окно монолитно и может быть либо показано на экране, либо нет. К примеру, если разработчику нужно отредактировать условие текущей точки останова, то вызов инструментального окна со списком всех точек останова и последующее его закрытие – это лишние действия, мешающие разработчику. Следовательно [11], первым шагом должен быть анализ существующих инструментальных окон и использования информации в них программистом. Благодаря такому анализу мы сможем выделить элементарные части функциональности каждого из дополнительных окон.

После уточнения функциональности инструментальных окон мы сможем убрать из интерфейса традиционные инструментальные окна, требующие открытия и закрытия и оставить только постоянные области, при этом стараясь отвести текстовому редактору как можно больше экранного пространства. Также мы будем размещать части функциональности дополнительных окон внутри текстового редактора. Наконец, в этом интерфейсе мы будем стараться уменьшить число переключений[2] между основным окном IDE и всплывающими окнами. Поскольку измерить реальное число переключений при всевозможных взаимодействиях с IDE не представляется возможным, для подсчета переключений будем использовать сценарии использования сред разработки, описывающие типичные действия программиста. В таком случае можно надеяться создать интерфейс, вообще не требующий переключений на всплывающие окна.

Эта работа заключается в проектировании [5] и оценке такого интерфейса. Для проектирования мы, во-первых, рассмотрим интерфейсы современных IDE и обобщим наиболее встречающиеся типы инструментальных окон. Также мы построим модель инструментальных окон IDE, описывающую закономерности поведения и использования инструментальных окон. На основе этой модели и эвристик удобства [12] мы построим новый интерфейс IDE. Далее следует эмпирически оценить данный интерфейс [13]. Широко распространенной методикой оценки интерфейсов является тестирование на пользователях [6] [14]. Для этого следует реализовать вариант созданного интерфейса и протестировать на основной целевой аудитории – профессиональных программистах. Тестирование покажет наиболее серьезные проблемы [15], возникающие при использовании полученного интерфейса, а они позволят оценить исследуемое возможное решение проблемы инструментальных окон.

[2] Переключение – это осмысленное действие разработчика по открытию или переводу фокуса в инструментальное окно или его закрытию. Например, щелчок мышью на пиктограмме окна или щелчок по текстовому редактору с целью закрытия дополнительного окна – это переключения.

2. Постановка задачи

В данной главе описывается цель и задача данной работы.

2.1. Цель работы

Целью данной работы является исследование применимости подхода к проектированию графического интерфейса среды разработки без инструментальных окон.

2.2. Постановка задачи

Задачи данной работы:

1. Провести обзор предметной области, выделить обобщенные инструментальные окна и основные сценарии работы программиста с инструментальными окнами;
2. спроектировать интерфейс среды разработки программ, руководствуясь следующими целями:
    - области внутри окна среды разработки должны иметь постоянный размер;
    - окно текстового редактора занимает наибольшую возможную площадь;
    - при работе с интерфейсом пользователь делает как можно меньше переключений;
    - в рамках выделенных основных сценариев работы пользователь не делает переключений;
3. реализовать спроектированный интерфейс в среде разработки KDevelop,
4. провести эмпирическое исследование реализованного интерфейса на пользователях.

3. Обзор предметной области

Данная глава содержит описание основных понятий, связанных со средами разработки программ, описание сценариев работы программиста с IDE и обзор популярных современных IDE.

3.1. Основные понятия

Графический/визуальный элемент (widget, control) [9] – элементарная часть графического интерфейса, представляющая собой законченную точку взаимодействия пользователя и некоторых данных. Например, кнопка или надпись – это графические элементы.

Среда разработки программ [1] – это программный комплекс, объединяющий инструменты, использующиеся программистами при написании и поддержке программ. Как правило, такие инструменты включают текстовый редактор, компилятор/интерпретатор, систему сборки, отладчик и систему контроля версий. Реже встречаются инструменты работы с базами данных, разработки пользовательских интерфейсов и управления тестированием.

В большинстве современных IDE интерфейс представляет прямоугольное окно, содержащее в себе одно или несколько окон-потомков [16] для редактирования исходных кодов и для работы с инструментами разработки.

Интерфейс среды разработки позволяет группировать несколько окон-потомков (как инструментальных, так и с исходным кодом) в одной области и переключать их при помощи вкладок. В таком случае в области отображается только окно выделенной вкладки.

Окно текстового редактора (с исходным кодом) – это окно, в котором осуществляется редактирование исходных кодов программы. В данной работе такие окна выделяются как более приоритетные для пользователя по отношению к инструментальным окнам.

Инструментальное окно – дополнительное по отношению к окну текстового редактора кода окно для управления дополнительными инструментами разработки. Например, очень распространены (как показано ниже) окна для отображения ошибок сборки.

Программист может перемещать инструментальные окна на экране, изменять их размер и режим. Будем выделять три режима: *плавающий*, *стыкующийся* и *автоматического скрытия* [17].

Плавающий режим (floating) предполагает нахождение инструментального окна «поверх» окна IDE. В таком режиме инструментальное окно закрывает часть информации в основном окне, может иметь любой размер и быть расположено в любом месте экрана.

Режим стыковки (docked) предполагает расположение инструментального окна внутри окна IDE. Так, окна в режиме стыковки граничат друг с другом и конкурируют за место внутри объемлющего окна среды разработки. Такие окна, как правило, могут быть свернуты и раскрыты обратно кликом по их заголовку.

Окна в р*ежиме автоматического скрытия (auto-hide, pin)* сворачиваются, когда пользователь уводит указатель мыши за их пределы (и, возможно, делает щелчок), и возвращаются при наведении мыши на их заголовок.

Перспектива [10] – это набор окон и настройки их расположения (размеры и режимы). Среды разработки позволяют переключаться между несколькими перспективами, мгновенно меняя отображаемые в данный момент окна и их расположение. Как правило, перспективы по умолчанию соответствуют различным активностям разработчика, например, чтение кода, написание кода и отладка. Перспективы могут быть добавлены, удалены или изменены пользователем.

Помимо окна редактора и инструментальных окон, интерфейс сред разработки содержит рассмотренные ниже графические элементы.

Панель меню (menu bar, главное меню) [9] – традиционный элемент современного интерфейса, горизонтальная полоса с надписями, каждая из которых используется для вызова меню со связанными командами. Панель меню находится на самом верху окна, под его заголовком (за исключением использования «общего меню», как в Mac OS X). Часто можно увидеть категории «Файл», «Правка» и другие. Меню используется для отображения полных возможностей интерфейса, поэтому предполагается, что через меню пользователь может найти неизвестные ему функции, а наиболее частые действия он выполняет, например, при помощи панели инструментов.

Панель инструментов (*toolbar*) [9] – горизонтальная тонкая полоса кнопок с иконками и/или текстом часто используемых команд, обычно располагаемая над окном текстового редактора. Панель инструментов – средство быстрого доступа к часто используемым командам.

Строка состояния (status bar) [9] – горизонтальная полоса, обычно располагающаяся в самой нижней части окна и отображающая текущее состояние: например, номер строки и столбца курсора. Также строка состояния иногда используется для отображения подробных данных о выделенной пользователем команде.

3.2. Предмет обзора

Поскольку данная работа исследует решение одной из проблем графического интерфейса, мы будем рассматривать только IDE с графическим интерфейсом [9]. Популярные [18] сегодня среды разработки обладают графическим интерфейсом [3] [19], поэтому такое предположение не ограничивает общности рассуждений.

Возможности, предоставляемые средой разработки, и взаимодействие программиста с ней существенно зависят от языка программирования. Далее будем рассматривать IDE для языка C/C++. Такое ограничение позволит разрабатывать интерфейс на высоком уровне детальности. Например, мы сможем учитывать особенности компиляции и линковки программ на C/C++ для работы с дополнительными окнами, в которых отображаются ошибки сборки.

3.3. Обзор сред разработки программ

Теперь рассмотрим конкретные среды разработки программ.

3.3.1. Visual studio

Microsoft Visual Studio [17] – популярная коммерческая линейка сред разработки компании Microsoft для семейства ОС Windows. Рассматриваемая версия Visual Studio 2008 обладает широкой функциональностью и поддерживает разработку на Visual Basic, Visual C++, Visual C#, Java.

Организация инструментальных окон

По центру окна Visual Studio находится текстовое окно, которое может быть горизонтально или вертикально разделено на несколько окон с документами. Относительно текстового окна существует 4 позиции переменного размера для инструментальных окон: снизу, сверху, справа и слева. Позиция может содержать ряд инструментальных окон в своих вкладках. Позиция инструментального окна, как и окно редактора текста, может быть разделена на несколько частей для показа нескольких инструментальных окон.

Помимо описанных выше трех режимов инструментальных окон, в Visual Studio также имеется режим вкладки (*tabbed*), при котором инструментальное окно находится в одной из вкладок текстового редактора.

Набор инструментальных окон

Solution Explorer. Это окно отображает дерево проектов текущего решения (*solution*) и принадлежащих им документов. Solution Explorer используется для создания новых файлов, открытия существующих и обзора структуры проектов. В

дальнейшем будем называть окна, отображающие дерево элементов и позволяющие переходить к коду, связанному с каждым элементом, окнами *древовидной навигации* или *древовидного отображения*.

Bookmarks. Данное инструментальное окно содержит список закладок в тексте программы с указанием файла и строки, в которых расположена закладка. Использование данного окна ограничивается отключением, переименованием и переходом по закладке.

Class View. Двусоставное окно древовидной навигации по классам. В верхней части находится список классов, а в нижней – содержимое (методы и атрибуты) выбранного в верхней части класса. Используется для обзора структуры классов и переходу к классу или его члену.

Breakpoints. Данное окно отображает список существующих точек останова с указанием имени файла и строки, где расположена точка. Допустимые действия: редактирование свойств точек останова и переход к ним в тексте программы.

Server Explorer. Окно древовидного отображения сведений об операционной системе: история событий, информация о файловой системе, устройства, системные службы, счетчики производительности и т.п. Использование окна заключается в просмотре информации.

Output View. В данном инструментальном окне находится текстовый вывод инструмента сборки, включающий список ошибок и предупреждений для каждого проекта. Окно используется для обзора истории сборки, ошибок/предупреждений и перехода к ошибочному коду.

Code Definition Window. Данное окно используется для просмотра кода, определяющего текущую выбранную в текстовом редакторе сущность. Например, при установке курсора на функцию, данное окно отображает код с определением этой функции.

Error List. Данное окно содержит список ошибок, предупреждений и сообщений сборки с возможностью фильтрации. Используется для обзора этих трех сущностей и перехода к коду, вызвавшему их.

Object Browser. Инструментальное окно, отображающее дерево доступных в проекте компонент, как системных, так и пользовательских. Туда включаются видимые классы, функции и константы, сгруппированные по проектам и наборам компонент.

Call Stack. Данное окно отображает стек вызовов в процессе отладки приложения. Может использоваться для перехода к коду функций-вызовов стека.

Autos, Locals, Threads, Watch. Данные окна-списки отображают сущности, относящиеся к отладке: переменные (автоматические[3] и локальные соответственно), нити и введенные пользователем выражения. Могут быть использованы для перехода к коду и редактирования значений переменных.

Task List. Это окно отображает один из двух списков заданий: либо отмеченные в тексте комментарии (с ключевыми словами “TODO”, “FIXME”, “HACK” и др.), либо добавленные пользователем через это окно. Используется для обзора заданий и перехода к комментариям.

Properties. Окно отображения и редактирования списка свойств объектов, выбранных в окне текстового редактора или инструментальных окнах. Например, при выборе файла в Solution Explorer в Properties отображаются его свойства: имя, тип, относительный путь от текущего файла и прочие.

3.3.2. NetBeans

NetBeans 6.8 [20] – популярная кроссплатформенная IDE с открытым исходным кодом, широко используемая для разработки на Java, а также используемая для других языков, включая C++.

Организация инструментальных окон

В NetBeans инструментальные окна организованы так же, как в Visual Studio, за исключением того, что в NetBeans невозможно поместить инструментальное окно во вкладки текстового редактора.

Набор инструментальных окон

Projects. Окно древовидной навигации по открытым проектам, в которых файлы разделены на категории (заголовочные файлы, файлы реализации и др.). Использование аналогично использованию Solution Explorer в Visual Studio.

Files. Окно древовидной навигации по файловой системе. Отображает структуру каталогов, распределяя файлы по категориям, каждая из которых соответствует каталогу ФС. Использование этого окна заключается в обзоре расположения файлов и открытии новых файлов.

[3] *Автоматические переменные* в данном контексте – это переменные, используемые в строке исходного кода, на которой стоит текстовый курсор.

Classes. Окно отображения классовой структуры проектов, которое в отличие от Class view из Visual Studio отображает методы и члены классов как их прямые потомки в дереве.

Tasks. Окно отображения заданий из кода с возможностью фильтрации и группировки. Используется так же, как и окно Task list в Visual Studio.

Properties. Аналогично одноименному окну в Visual Studio.

Output. Данное инструментальное окно содержит простой текстовый вывод инструментов сборки и не предоставляет возможности открытия кода, вызвавшего какую-либо запись в выводе.

Navigator. Это окно отображает доступные в коде имена артефактов[4] в виде дерева. Так отображаются имена классов, пространств имен и заголовочных файлов. Используется аналогично окну Objects из Visual Studio.

Call Stack. Данное окно отображает список вложенных вызовов функций при отладке программы с возможностью перехода к любому из вызовов.

Variables. Это окно содержит список переменных, наблюдаемых[5] при отладке с возможностью редактирования их значений.

Breakpoints. Отображение списка точек останова, как и в одноименном окне Visual Studio, с возможностью отключения и редактирования условия.

3.3.3. Eclipse

Eclipse [21] – кроссплатформенная среда разработки программ с открытым исходным кодом, используемая в основном для C++ (рассматривалась Eclipse CDT 4.0) и Java (рассматривалась Eclipse JDT 3.5).

Организация инструментальных окон

Инструментальные окна в Eclipse организованы почти так же, как и в Visual Studio. Открытие инструментального окна в режиме автоматического скрытия в Eclipse требует клика мыши, а не просто наведения, как в Visual Studio и NetBeans. В Eclipse можно сохранять и загружать настройки организации инструментальных окон, называемые в этой IDE перспективами.

---

[4] Под *артефактом* или *объектом кода* (*symbol* или *object*) понимается некоторый идентификатор и связанная с ним сущность; например, класс, функция или переменная.

[5] *Наблюдаемое выражение/переменная* (*watch, auto* или *variable*) – это переменная или выражение, текущее значение которой отображается средой разработки в процессе отладки.

Набор инструментальных окон

Navigator. Это окно древовидной навигации по проектам и группированным файлам используется так же, как и окно Solution Explorer в Visual Studio.

Include Browser. Данное окно показывает дерево включений заголовочных файлов: потомками файла являются файлы, включенные в него. Это окно используется для просмотра включений и открытия представленных файлов.

Make Targets. Данное окно отображает дерево целей и подцелей системы сборки Make. Используется для просмотра структуры целей и перехода по ним.

Outline. Это окно аналогично окнам Navigator в Eclipse и Objects в Visual Studio.

Problems. Данное инструментальное окно отображает список ошибок и предупреждений сборки, как Errors list Visual Studio.

Properties. Аналогично одноименным окнам Visual Studio и NetBeans.

Breakpoints. Содержит список точек останова, аналогично одноименному окну в Visual Studio.

Debug. Дерево-историю записей о сессиях отладчика. Элемент каждого запуска является предком нитей, вызываемых в этой сессии отладки.

Variables. Данное инструментальное окно содержит список наблюдаемых при отладке переменных и их значений. Возможно редактирование значений и добавление/удаление наблюдаемых переменных.

Call Hierarchy. Это окно отображает дерево вызовов выбранной функции, статически анализируя код. Есть возможность перехода к любой из вызывающих функций.

3.3.4. Code::Blocks

Code::Blocks [22] – кроссплатформенная IDE с открытыми исходными кодами, разработанная для удобства программирования на C++ с использованием библиотек GTK+, Qt, OpenGL, FLTK, wxWidgets, Lightfeather. Ниже приведены данные для версии 8.02.

Организация инструментальных окон

Существует всего 4 основных контейнера инструментальных окна: Management, Logs & others, Open files list, To-Do list. Контейнер содержит инструментальные окна как вкладки. Существует два режима контейнеров:

- дрейфующий режим: окно контейнера становится отдельным окном в смысле менеджера окон;
- стыкующийся режим: инструментальное окно может находиться в одной из 4 позиций: сверху, снизу, справа и слева от текстового редактора.

Для окон, размещенных в одной позиции, она делится на части настраиваемого размера. Окно текстового редактора может быть разбито на два (горизонтально или вертикально), но в его частях отображается всегда текст одного и того же файла. Наборы настроек расположений окон можно сохранять и загружать (аналогично перспективам Eclipse). Этим ограничиваются возможности организации инструментальных окон в Code::Blocks.

Набор инструментальных окон

Management – Projects. Данное окно по назначению и структуре полностью эквивалентно одноименному окну из NetBeans.

Management – Symbols. Это окно отображает классы, функции, препроцессорные директивы и глобальные константы. Оно организовано, как и инструментальное окно Class view в Visual Studio.

Logs & others – Code::Blocks. Данное окно отображает лог событий среды разработки (создание/открытие проектов/файлов, загрузка подгружаемых модулей и др.). Не допускает переходов и действий, связанных с логом.

Logs & others – Build messages. Это окно отображает список ошибок и предупреждений сборки с атрибутами, полностью аналогично, например, окну Problems из Eclipse.

Logs & others – Build log. Содержит текстовый вывод инструмента сборки проекта. Не допускает переходов к коду или иного сложного взаимодействия.

Logs & others – Search results. Это окно отображает список результатов поиска с указанием имени файла, строки и найденного фрагмента текста. Предоставляется возможность перейти к любому из найденных результатов.

Logs & others – Script console. Это окно принимает текстовые команды пользователя и выводит текстовый результат этих команд.

Logs & others – Debugger. Данное инструментальное окно содержит текстовый вывод отладчика, предоставляя пользователю лишь вводить команды для отладчика. Через это окно также выводятся данные о попадании на точки останова.

Open files list. Данное инструментальное окно отображает список файлов, открытых в Code::Blocks. Используется для обзора открытых файлов и переключения на вкладку любого из этих файлов.

To-Do list. Показывает список задач, содержащихся в коде текущего проекта. Использование аналогично окну Tasks из NetBeans.

Call Stack. Данное окно отображает стек функций в виде списка. Позволяет по двойному щелчку на верхней функции стека переходить к точке исполнения в ней.

Watches. Данное окно отображает дерево наблюдаемых переменных. Наблюдаемая переменная несоставного типа представляет элемент дерева, а потомками каждой переменной-класса является набор значений ее атрибутов.

Running threads. Это инструментальное окно отображает список нитей отлаживаемого приложения. Позволяет перейти к точке выполнения любой нити.

3.3.5. MonoDevelop

MonoDevelop 2.2 [23] – среда разработки с открытым исходным кодом, предназначенная для написания приложений на C/C++, C#, Java, Visual Basic и других языках. Является частью проекта Mono по созданию свободной реализации платформы .NET.

Организация инструментальных окон

Инструментальные окна MonoDevelop подчиняются тем же принципам организации, что и окна Visual Studio за следующими исключениями:

- Возможно сохранение/загрузка перспектив так же, как в Eclipse.
- Окно текстового редактора не может быть разделено на два независимых окна.

Набор инструментальных окон

Solution. Данное окно полностью эквивалентно окну Solution Explorer в среде разработки Visual Studio.

Files. Это типичное окно древовидной навигации по файловой системе, которые мы уже рассмотрели в IDE NetBeans.

Classes. Древовидно отображает принадлежность классов и пространств имен классам из открытого решения (*solution*). Использование аналогично окну Classes из NetBeans.

Errors list. Данное окно аналогично уже рассмотренным окнам отображения списка ошибок.

Task list. Окно отображения одного из списков задач пользователя: списка из комментариев и списка, созданного при помощи этого инструментального окна. Таким образом, внешний вид и использование, как у окна Task list в Visual Studio.

Properties. Данное окно полностью аналогично одноименному окну из Visual Studio и NetBeans.

Document Outline. Отображает функции и классы из текущего документа, повторяя функции окна Outline из Eclipse.

Build Output. Содержит простой текстовый вывод инструмента сборки, доступный только для выделения и копирования.

Application Output. Содержит простой текстовый вывод запускаемого приложения, доступный только для выделения и копирования.

Breakpoints. Отображает список точек останова, аналогично одноименным окнам рассмотренных ранее IDE.

Watch. Отображает список пользовательских выражений и их значений, вычисляемых во время отладки.

Locals. Содержит дерево элементов, каждый из которых соответствует локальной переменной в точке исполнения программы. Потомками локальных переменных являются их базовые классы и члены-данные, для которых отображается текущее значение.

Call Stack. Отображает стек вызовов, как и одноименное окно NetBeans.

Threads. Отображает список нитей, аналогично окну Running threads в Code::Blocks.

3.3.6. KDevelop

KDevelop 4.0 [24] – независимая от платформы свободная IDE для графической оболочки KDE, написанная на языке C++ с помощью библиотек Qt и KDE. Поддерживает программирование на многих языках, в том числе C/C++ и Java.

Организация инструментальных окон

Инструментальные окна реализованы в KDevelop менее гибко, чем в Visual Studio, NetBeans и Eclipse. Возможен единственный режим окон, при котором они

сгруппированы в одной из четырех позиций (сверху, снизу, слева, справа от окна текстового редактора). Для каждой позиции допускается отображение нуля (позиция скрыта, ее место занято текстовым редактором) или одного инструментального окна, все остальные окна находятся во вкладках позиции. Окно текстового редактора может разбиваться на два окна по горизонтали или вертикали неограниченное число раз.

Набор инструментальных окон

Breakpoints. Отображает список точек останова, аналогично одноименному окну в Eclipse

Call Stack. Это окно отображает список нитей и стек вызовов для каждой нити при использовании отладчика. Функционирует и используется аналогично одноименному окну в NetBeans.

Code Browser. Выполняет ту же функцию, что и Code Definition Window в Visual Studio – показ свойств артефакта кода, на котором находится курсор.

Filesystem. Окно древовидной навигации по файлам. Использование аналогично таким окнам, рассмотренным ранее.

Projects. Данное окно древовидной навигации по открытым проектам отличается от, например, окна Projects из NetBeans тем, что позволяет редактировать список собираемых проектов, расположенных в этом же инструментальном окне.

Documents. Окно древовидной навигации по проектам и связанным с ними документам (редактируемым файлам). Использование аналогично другим окнам древовидной навигации.

Messages. Отображение списка ошибок и предупреждений, полученных в результате сборки, с указанием файла, в котором возникла ошибка, линии ошибки и описания. Использование данного окна заключается в просмотре сообщений и переходе к коду, вызвавшему их.

Problems. Окно, аналогичное Messages, но показывает ошибки, полученные в результате фонового разбора кода. Используется аналогично Messages.

GDB. Содержит простой текстовый вывод отладчика GDB, доступный для выделения и копирования.

Variables. Окно с древовидной структурой, отображающее две категории наблюдаемых переменных: автоматические и локальные. Переменные сложных

типов содержат потомков-атрибутов, а переменные простых типов представлены одним элементом дерева.

3.3.7. IntelliJ IDEA

IntelliJ IDEA Community Edition 9.0 [25] – многоплатформенная среда разработки программ с открытым исходным кодом, основной пользовательской аудиторией которой являются Java-разработчики.

Организация инструментальных окон

Инструментальные окна IDEA обладают возможностями настройки, схожими с окнами Visual Studio. Именование режимов отличается, однако их выразительные способности совпадают: для любого расположения инструментальных окон в Visual Studio можно создать аналогичное расположение в IDEA, и наоборот. Единственное исключение – в IDEA нельзя поместить инструментальное окно во вкладку текстового редактора.

В IntelliJ IDEA имеется навигационная панель *breadcrumbs* (*«хлебные крошки»*) [26], расположенная сверху текстового редактора. Она отображает путь до текущего файла и позволяет открывать новые файлы и переключаться между открытыми.

Набор инструментальных окон

Project. Отображает дерево модулей проекта и внешние библиотеки, тем самым совмещая функции окон Solution explorer и Object Browser из Visual Studio.

Messages. Отображает сообщения сборки, включая ошибки и предупреждения. Аналогично другим окнам с сообщениями сборки.

Commander. Двусоставное окно навигации по файловой системе. Верхняя часть отображает структуру каталогов, а нижняя – файлы выбранного в первой части каталога.

Run. Простой текстовый вывод запущенного приложения. Переход к строкам кода и другое сложное взаимодействие с пользователем не предусмотрено.

Debug. Простой текстовый вывод отладчика. Переход к строкам кода и другое сложное взаимодействие с пользователем не предусмотрено.

Frames. Отображает список вызовов в стеке приложения. По щелчку на функцию осуществляется переход либо (в случае щелчка по верхней функции) к точке останова, либо к точке выбранной функции, где управление перешло к следующей по вложенности функции.

Varuables. Отображает дерево локальных переменных. Для простых типов это окно отображает просто элемент верхнего уровня, для сложных типов отображает потомков-членов класса.

Watches. Данное окно отображает список наблюдаемых пользователем выражений и их значений.

TODO. Это окно содержит 2 вкладки: с заданиями проекта и данного файла. Задания берутся из кода. Есть возможность перейти к заданию в коде.

Structure. Древовидное отображение классов, их предков и членов. Аналогично окну Navigator в Eclipse.

3.3.8. C++ Builder

C++ Builder 6.0 [27] – коммерческая среда разработки от компании Borland, предназначенная для разработки на C++ с использованием библиотек CLX, VCL, MFC.

Организация инструментальных окон

Инструментальные окна в C++ Builder организованы схожим образом с окнами Visual Studio за тем исключением, в C++ Builder недоступен режим автоматического скрытия и режим вкладки. Всегда присутствует окно редактора формы, не стыкующееся ни с какими инструментальными окнами. Окно текстового редактора не может быть разделено на несколько окон. В C++ Builder есть возможность сохранения/загрузки настроек расположения инструментальных окон (*save/load desktop*), аналогично перспективам в Eclipse.

Набор инструментальных окон

Project Manager. Древовидное отображение файлов, входящих в проекты. Аналогично Solution Explorer в Visual Studio.

Class Explorer. Отображение дерева классов с их атрибутами и предками. Аналогично окну Classes из NetBeans.

Components. Список визуальных компонентов, доступных для добавления на форму. Используется для поиска и добавления.

Messages. Содержит список сообщений сборки, аналогично одноименным окнам IntelliJ IDEA и KDevelop.

Object TreeView. Это окно отображает дерево принадлежности компонентов на выбранной форме. Может быть использовано для обзора принадлежности и выбора компонента.

Object Inspector. Отображает редактируемый список свойств объекта формы.

Breakpoint List. Редактируемый список точек останова. Используется так же, как и рассмотренные ранее окна точек останова.

Call Stack. Отображает список функций-вызовов в текущем состоянии программы. Может быть использовано для перехода к любой из функций.

Threads. Отображает список нитей отлаживаемого приложения. Для каждой нити имеется возможность перейти к точке ее исполнения.

3.3.9. XCode

XCode 3.2.3 [28], бесплатная среда разработки компании Apple, предназначена для разработки под Mac OS на C, C++, Objective-C и других языках программирования. Поддерживается разработка приложений для Mac OS X и Apple iOS.

Организация инструментальных окон

Интерфейс среды XCode отличается от рассмотренных ранее сред: в нем нет возможности встраивать инструментальные окна в окно программ. Единственный режим дополнительных окон – это плавающий. Это ограничение разработчики XCode обходят так при помощи нескольких решений. Во-первых, основное окно программы содержит две универсальные взаимосвязанные области в режиме стыковки: дерево элементов Overview (слева от редактора) и список (выше редактора). Эти области нельзя переместить или изменить в размере. В дереве показываются категории файлов, целей сборки, ошибок сборки, точек останова, закладок и собственно эти сущности. В списке отображаются сущности, принадлежащие выбранным в дереве категориям. Эти две области можно формально считать универсальными инструментальными окнами. Упомянутые области могут быть отделены от окна редактора кода в отдельные плавающие инструментальные окна.

В XCode имеется тонкая навигационная панель над редактором текста для отображения и смены текущего просматриваемого файла и символа кода. Внутри текстового редактора могут быть отображены данные об ошибках, точках останова и переменных отладки.

Набор инструментальных окон

Overview. Первая из двух фиксированных областей. Древовидное отображение файлов, точек останова, исполняемых файлов, закладок, результатов поиска.

Вторая фиксированная область область со списком файлов, точек останова и др.

Breakpoints. Список точек останова с возможностью редактирования и открытия текстового редактора.

Errors. Список ошибок и сообщений сборки.

Expressions. В этом окне отображается список локальных переменных и наблюдаемых выражений.

Debugger. В этом составном окне отображается список нитей, стек вызовов, значения регистров и исполняемый в данный момент машинный код.

Globals. Показывает имена и значения глобальных констант и переменных.

3.4. Обобщенные инструментальные окна

Во-первых, данным обзором было установлено, что большинство современных IDE с графическим интерфейсом используют инструментальные окна для предоставления доступа к своим инструментам. Это подтверждает актуальность проблемы инструментальных окон, описанной во введении работы.

Во-вторых, многие инструментальные окна схожи своими функциями и структурой. Обобщим рассмотренные выше инструментальные окна для дальнейшей разработки интерфейса. Обобщенные инструментальные окна мы получим, выделяя назначение инструментальных окон и сущности, с которыми они оперируют (например, ошибки, файлы или задания).

3.4.1. Окно навигации по проекту

Данное окно является обобщением инструментальных окон Solution Explorer (Visual Studio) и Projects (Eclipse, NetBeans, …). В этом окне показывается дерево файлов. Дерево включает в себя как объекты файловой системы (файлы и каталоги), так и сущности среды разработки (например, цели сборки, подключаемые библиотеки, ресурсы и др.). По контекстному меню в этом окне доступны стандартные действия над представленными сущностями. Те сущности, которым соответствуют файлы, строки/диапазоны строк в файлах могут быть активированы в редакторе кода двойным щелчком мыши.

3.4.2. Окно навигации по артефактам кода

Данное окно является обобщением инструментальных окон Class View (Visual Studio), Classes (Eclipse) и Structure (IntelliJ IDEA). В этом окне отображается дерево вложенности объектов кода, таких как классы, структуры, пространства имен, функции и т.д. Для всех этих объектов возможен переход в редакторе и стандартные действия контекстного меню.

3.4.3. Окно задач

В окне задач находится список задач, взятых из исходного кода. Прототипом данного обобщенного окна являются окна To-Do list (Code::Blocks), Task List (Visual Studio) и другие. Отображены атрибуты задачи: файл, номер строки и текст задачи. В этом окне можно добавлять/удалять задачи, редактировать текст задачи, а также активировать ее в текстовом редакторе.

3.4.4. Окно ошибок сборки

В окне ошибок сборки отображается список ошибок и предупреждений, возникших в результате сборки проекта. Прототипы - Messages (IntelliJ IDEA, KDevelop), Problems (Eclipse), Errors (XCode). Для каждой ошибки/предупреждения указывается имя файла, строка и текст ошибки. Окно ошибок сборки предоставляет пользователю возможность перейти к любой ошибке и предупреждению. В некоторых рассмотренных IDE ошибки отображаются в выводе системы сборки в соответствующих окнах (Build log в Code::Blocks, Output View в Visual Studio, Build Output в MonoDevelop) с возможностью перехода к ошибкам. Такой тип окон тоже будем считать окнами ошибок сборки.

3.4.5. Окно точек останова

Это окно содержит список точек останова. Для каждой точки отображается файл, номер строки, ее состояние (установлена отладчиком или нет, активирована пользователем или нет), условие останова (возможно, пустое), число попаданий на данную точку. Через данное окно можно добавлять, удалять и переходить к коду, где выставлена точка останова. Прототипом его послужили окна Breakpoint List (C++ Builder) и Breakpoints (Eclipse, XCode, KDevelop).

3.4.6. Окно нитей и стека вызовов

Это окно содержит список нитей отлаживаемой программы и для каждой нити отображает стек вызовов. Информация доступна только при остановленном отладчике. Для каждого из вызовов, соотнесенного со строкой кода, можно перейти к этой строке. Прототипы: Threads (Visual Studio, C++ Builder), Call Stack (KDevelop, NetBeans, Visual Studio), Debugger (XCode).

3.4.7. Окно наблюдаемых выражений и переменных

В данном окне находится список локальных (для текущей точки исполнения программы) переменных и наблюдаемых программистом выражений. Информация о переменных доступна только при остановленном отладчике. В данном окне можно менять значения переменных и добавлять/удалять произвольные выражения для слежения за их значением. Прототипы: Watches (Code::Blocks, IntelliJ IDEA), Watch (MonoDevelop), Variables (KDevelop, Eclipse, NetBeans).

3.4.8. Окно вывода запущенной программы

Это окно отображает результат работы запущенной (как с отладчиком, так и без) разрабатываемой в IDE программы. Это окно содержит простой текстовый вывод, не предполагающий интерактивного взаимодействия с пользователем. Прототипом этого окна послужили окна Output View (Visual Studio), Output (NetBeans), Run (IntelliJ IDEA) и прочие.

3.5. Сценарии работы программиста

Ниже будут описаны основные сценарии работы программиста с инструментальными окнами, полученные рассмотрением функций инструментальных окон и исследований рабочего процесса программиста.

Сценарий использования [29] *IDE* – это совокупность связанных активностей программиста, направленных на выполнение ряда близких задач. Например, сценарий сборки включает в себя запуск/остановку/настройку процесса сборки и анализ его результатов для сборки разрабатываемой системы. Сценарий можно рассматривать как контекст взаимодействия программиста, обладающего локальными целями разработки, с IDE для достижения этих целей.

Выделяются [30] [31] различные этапы работы программиста, например: написание нового кода, поддержка уже написанного кода, чтение кода с целью получения информации, чтение документации к коду, устранение ошибок сборки, отладка, тестирование и работа с системами контроля версий. Мы будем основывать наше рассуждение на взаимодействии человека с инструментальными окнами, поэтому рассмотрим следующие сценарии:

- чтение кода,
- редактирование кода,
- сборка проекта,
- отладка программы.

Одна из особенностей работы программиста [32] заключается в том, что он может часто и частично переходить от одного сценария к другому, поэтому мы не будем проводить четкую границу между ними. К примеру, для исправления ошибок сборки приходится редактировать исходный код.

В описание каждого сценария будем включать список действий, специфичных для сценария, локальную цель программиста в этом сценарии и изменения в разрабатываемой программе в результате этого сценария. Действия, зависимые друг от друга, объединим в цепочки.

Критерием включения некоторого действия в сценарий является выполнение хотя бы одного из условий:

- действие может быть выполнено через инструментальное окно;
- действие может изменить информацию, отображаемую в каком-либо инструментальном окне.

Если действия одинаково изменяют отображаемую информацию, мы «склеиваем» их в одно действие, поскольку с точки зрения использования инструментальных окон они эквивалентны.

3.5.1. Сценарий чтения кода

Цель программиста: получение информации о коде программы.

Действия программиста независимы друг от друга и могут быть следующими:

- открыть/закрыть текстовый файл;
- перейти к определению/реализации функции;
- найти все вхождения строки в код; перейти к любому вхождению;
- найти все использования идентификатора (вызовы функции, обращения к переменной и т.д.); перейти к использованию;
- просмотреть иерархию классов программы: какие наследуют, от каких наследуют, какие члены содержат; перейти к классу;
- просмотреть иерархию включения заголовочных файлов; перейти к заголовочному файлу.

Изменения в разрабатываемой программе: отсутствуют.

3.5.2. Сценарий редактирования кода

Цель программиста: изменение кода.

Действия программиста:

- Создать новый файл/удалить файл;

- Создать новый класс/удалить класс;
- Создать новый атрибут или метод/удалить атрибут или метод;
- Изменить существующий код[6].

Изменения в разрабатываемой программе: внесенные программистом.

3.5.3. Сценарий сборки

Цель программиста: успешная сборка разрабатываемой системы.

Действия программиста зависимы друг от друга. Они показаны на диаграмме на рисунке 1.

Рисунок 1. Действия программиста при сборке.

Изменения в разрабатываемой программе: возможно, исправление ошибок сборки.

3.5.4. Сценарий отладки

Цель программиста: исправление ошибок исполнения.

Действия программиста могут быть зависимы друг от друга. Они указаны на рисунке 2.

[6] С точки зрения работы с инструментальными окнами различные возможности отредактировать код можно не учитывать.

Рисунок 2. Действия программиста при отладке.

Изменения в разрабатываемой программе: изменение точек останова.

4. Модель интерфейса инструментальных окон

В этой главе строится модель инструментальных окон IDE, используемая далее для проектирования интерфейса.

4.1. Основные понятия

Рассмотрим следующую модель интерфейса IDE. Пусть внутри прямоугольного *окна IDE* задана двумерная координатная сетка. Разобьем прямоугольник окна произвольным образом на *прямоугольные области – окна IDE*. Пусть области целиком покрывают окно и не перекрываются между друг другом. Пусть среди областей выделена одна – *окно редактора исходного кода*. Остальные области будем считать дополнительными по отношению к выделенной. *Переключаемыми областями* назовем те, которые могут быть отключены (и тогда их место будет разделено между остальными областями по некоторому закону) и включены обратно, с возвращением на предыдущее место. *Постоянными областями* будем называть области, не являющиеся переключаемыми.

Определим *всплывающие окна* – это прямоугольные области, блокирующие взаимодействие с окнами IDE с равными им координатами в тот момент, когда они активны. Всплывающие окна могут быть активированы и деактивированы. Всплывающее окно не является областью внутри окна IDE, однако к ним применимы характеристики окон, описанные в следующем разделе.

Модель инструментальных окон – это разбиение окна IDE на прямоугольные области с выделенной областью редактора исходного кода и множество всплывающих окон.

Переключение – это активация или деактивация переключаемой области или всплывающего окна, выполненная пользователем (далее будем отличать действия пользователя от автоматических изменений интерфейса). Переключение происходит мгновенно.

4.2. Характеристики окон

Минимальным размером полезности окна называется такой размер (как вертикальный, так и горизонтальный), при уменьшении размера окна ниже которого у пользователя возникают существенные затруднения при использовании этого окна (или же использование становится вообще невозможным). Вообще говоря, любая область имеет минимальный размер

полезности, отличный от нуля, поскольку взаимодействие пользователя с областями размером несколько пикселей практически невозможно.

Максимальный размер полезности – размер области, при увеличении выше которого прироста полезности не происходит. Для некоторых областей он может быть бесконечен, то есть от любого увеличения размера полезность окна увеличивается. Окно с вертикальным списком некоторых сущностей достигает своего максимального размера полезности, когда все сущности становятся видны на экране, потому что дальнейшее увеличение не даст пользователю никакого преимущества.

Будем считать, что вертикальный максимальный размер полезности окна текстового редактора равен бесконечности, поскольку всегда пользователь заинтересован видеть наибольшее число строк кода. Горизонтальный максимальный размер полезности окна текстового редактора положим равным некоторой константе, превышающей среднюю длину строк кода.

Введем понятие *оптимальности размера окна* – непрерывного неубывающего отображения размера окна (то есть двух координат) в отрезок [0, 1] такого, что:

1. максимальный размер полезности отображается в точку 1 (возможно, асимптотически);
2. минимальный размер полезности отображается в точку, отличную от нуля;
3. если хотя бы вертикальный или горизонтальный размер меньше соответствующего минимального размера полезности, то оптимальность размера окна равна нулю.

Вообще, определение точного вида оптимальности размера окна – это трудоемкая задача, требующая принятия всех деталей реализации данного конкретного инструментального окна. Поэтому мы не будем пытаться выразить ее в явном виде.

Назовем окно *критичным к вертикальному (горизонтальному) размеру,* если либо окно имеет высокий вертикальный (горизонтальный) минимальный размер полезности, либо оптимальность размера окна начинает существенно расти при больших значениях вертикального (горизонтального) размера.

4.3. Работа пользователя с моделью интерфейса

Обратимся к моделированию работы пользователя с моделью интерфейса инструментальных окон.

Пусть пользователь может удерживать свое внимание не более чем на одной области или всплывающем окне в каждый момент времени. Окно, на котором пользователь удерживает свое внимание, будем называть *фокусным*. Пользователь может активировать и деактивировать любую область или всплывающее окно (такое действие, как уже упоминалось, мы будем называть переключением). Пользователь также может переносить свой *фокус* между любой открытой областью на экране или открытым инструментальным окном – и это не будет требовать переключения.

Таким образом, *работа* пользователя состоит из последовательности:

1) смен фокуса между областями;

2) активаций и деактиваций окон, допускающих это;

3) взаимодействий с фокусной областью.

Пусть теперь пользователь совершил некоторую работу. Из проблемы инструментальных окон, описанной выше, можно сделать вывод, что *критериями удобства модели интерфейса* для пользователя служат:

I. минимизация числа переключений;
II. максимизация величины $\sum t_i * o_i$, где t_i – время работы с i-той областью, o_i – оптимальность i-той области.

Соотношение важности этих двух критериев может меняться и не может быть установлено в явном виде, поэтому далее мы будем учитывать оба критерия независимо.

4.4. Проблема инструментальных окон

Опишем проблему инструментальных окон в терминах построенной модели. В современных IDE при большом числе инструментальных окон размеры полезности и оптимальность размера определены так, что пользователь не может достичь желаемых значений для критериев I и II одновременно, потому что они вступают в противоречие. Действительно, при желании уменьшить число переключений, приходится держать множество окон открытыми, следовательно, невозможно задать им всем размеры с высокой оптимальностью, так что получается низкое значение величины в критерии II. И, наоборот, при желании постоянно работать с областями с высокой оптимальностью размера, приходится часто активировать и деактивировать области, что приводит к высокому числу переключений.

Данная проблема особенно остро проявляется при использовании окон, критичных к размеру: поскольку они требуют большое количество экранного

пространства при использовании, сохранение их на экране почти всегда не выгодно пользователю, поэтому каждое использование критичного к размеру окна приводит к одному или двум переключениям. Далее при проектировании мы будем стараться использовать меньше окон, критичных к двух размерам сразу.

4.5. Использование модели в проектировании

Чтобы гарантировать, что во время проектирования интерфейса будут учтены оба критерия, нам нужно ограничить множество рассматриваемых интерфейсов. Для этого мы не будем рассматривать активируемые инструментальные окна, а их функциональность предоставим через постоянные области. Тогда единственный источник переключений – это всплывающие окна. Поскольку полностью избежать их использования нельзя, мы будем оптимизировать интерфейс для основных сценариев работы программиста. Можно надеяться, что для выполнения действий в этих сценариях не придется использовать всплывающие окна вообще, и число переключений будет равно нулю.

Таким образом, при выполнении условия неиспользования всплывающих окон, при проектировании нужно заботиться о соблюдении только критерия удобства II. Поскольку нам известно, что вертикальный размер полезности окна текстового редактора неограничен, а горизонтальный может быть сравнительно большим (даже при длине строк в 80 символов), то необходимо проектировать постоянные области так, чтобы они имели высокую оптимальность даже при малых размерах.

5. Подход к проектированию интерфейса

В данной главе описывается подход к проектированию интерфейса, решающего заявленную проблему инструментальных окон. Для этого будет использоваться модель инструментальных окон, описанная в предыдущей главе, и эвристики удобства интерфейсов, описанные ниже. Модель в данном случае направляет проектируемый интерфейс в сторону решения поставленной задачи, а эвристики задают рамки применимости этой модели и не позволяют проектируемому интерфейсу стать неудобным для пользователей.

5.1. Эвристики удобства

При построении графических интерфейсов использование эвристик является стандартом де-факто [15]. Поскольку априорные методы точного построения интерфейсов не эффективны, процесс создания интерфейсов выглядит так [5]: анализируются и документируются потребности пользователя, его модель предметной области (ментальная модель [9]); по потребностям пользователя предлагается ряд вариантов интерфейса, удовлетворяющих эти потребности; при помощи анализа этих вариантов с точки зрения эвристик удобства интерфейсные решения фиксируются и детализируются. Такой способ как не дает возможность надежно оценить качество результата, так и не гарантирует получение наиболее удобного интерфейса. Поэтому, как правило, эвристическое построение интерфейса дополняется тестированием интерфейса на пользователях. Такое тестирование будет проведено и для разрабатываемого интерфейса. Более подробно оно описано в главе 8 «Исследование».

Существует большое число работ по взаимодействию человека и компьютера (*human-computer interaction* [33]), рекомендующих схожие эвристики удобства графических интерфейсов. Обобщим и опишем ниже те эвристики, которые будут использоваться при построении интерфейса. Для каждой эвристики кратко опишем рамки применения. Мы будем применять эвристики как с явной отсылкой к ним, так и неявно применяя их к внешнему виду визуальных элементов.

Итак, в данной работе явно и неявно будут применены следующие эвристики удобства:

1. Минимизация нагрузки на память пользователя [14]

В каждый момент времени пользователь должен видеть основные возможности интерфейса, которые определяют его пути к решению локальных задач. Следует избегать запоминания пользователем сложных последовательностей действий. Например, давая визуальные подсказки.

2. Целостность интерфейса [5]

Схожие задачи в интерфейсе должны быть решены схожими способами, чтобы сформировать у пользователя представление об идиомах интерфейса – шаблонах взаимодействия с пользователем [9]. Например, похожие всплывающие окна должны иметь одинаковое поведение на экране и одинаковые способы закрытия. Зачастую взгляды пользователя на интерфейс формируются под впечатлением аналогичных программ, существовавших ранее. Например, нарушением целостности – это отсутствие сочетания клавиш Ctrl+C и Ctrl+V в текстовом редакторе для копирования и вставки.

3. Предоставление обратной связи пользователю [34]

Пользователь должен не только управлять программой через графический интерфейс, но и получать информацию о ее текущем состоянии. Это критично для выбора пользователем дальнейших действий и перекликается с другими эвристиками. Один из классических примеров обратной связи – строки состояния, показывающие текущие выполняемые задания и возможные последствия действий.

4. Проектирование для наиболее частых ситуаций [5]

Для того чтобы разделить все функции приложения на более доступные и менее доступные анализируется частота их использования. Общий принцип таков: чем более часто и большей группой пользователей используется функциональность, тем более видимым и простым должен быть доступ к ней.

5. Приоритет целей пользователя [35]

Как правило, разработки и использование программного продукта оказывает влияние на ряд категорий лиц: заказчики, руководители разработки, программисты, собственно пользователи. Утверждается, что цели непосредственного пользователя (как правило, это удобство в решении повседневных задач) должны иметь приоритет над интересами других категорий людей, например, над удобством реализации и коммерческой выгодой.

6. Простой и естественный вид интерфейса [9]

Надписи в графическом интерфейсе должны точно и кратко отражать сущность отображаемой информации. Любые диалоги и визуальные элементы не должны отображать то, что может редко понадобиться пользователю или вообще не относится к его текущим задачам.

7. Уменьшение накладных расходов в интерфейсе [9]

Не все действия пользователя продвигают его к цели. Некоторые действия необходимы для подготовки интерфейса к действию, поддержания его в рабочем

состоянии и т.д. Такие действия называют «интерфейсным налогом». При проектировании нужно стараться уменьшить этот интерфейсный налог.

8. Соответствие пользовательскому взгляду на задачу [34]
Зачастую программисты наполняют сообщения программы данными, представляющими ценность для них самих, а не для пользователя. Более того, программисты смотрят на программу с точки зрения реализации, поэтому иногда спроектированный ими интерфейс не понятен пользователю.

9. Отсутствие режимов в работе пользователя [9]
Данная эвристика означает следующее: нельзя заставлять пользователя делать какие-то специальные действия для активации некоторой функциональности (т.н. «перехода в отдельный режим»), потому что затраты на переключение режимов со временем сильно утомляют пользователя.

5.2. Построение интерфейса

Ниже опишем возможные альтернативы визуальных элементов и последовательность действий при разработке интерфейса.

5.2.1. Используемые решения

Итак, необходимо представить выделенную в результате обзора функциональность инструментальных окон так, чтобы на экране в каждый момент времени находились только постоянные области, занимающие как можно меньше места – более точно, чтобы их максимизировать сумму в критерии II предыдущей главы. Также требуется минимально использовать всплывающие окна, по возможности избегая обращения пользователя к ним при действиях сценариев, описанных в главе 3 «Обзор предметной области».

Для решения этой задачи нам понадобятся некоторые обобщенные интерфейсные механизмы, которые могут занимать как можно меньшую фиксированную область хотя бы в одном измерении (вертикальном или горизонтальном) и отображать разнородную информацию, раньше находившуюся в инструментальных окнах.

В ходе обзора были обнаружены следующие визуальные элементы, которые могли бы использоваться в качестве фиксированных областей:

- Панель меню использовалась во всех рассмотренных IDE для представления полного перечня доступных действий для данного выделенного документа и позиции курсора. Однако большая часть действий не относилась к этому контексту и была доступна всегда.

- Панель инструментов также имелась во всех рассмотренных IDE. Она содержала кнопки основных действий, например: начать сборку, запустить программу, начать отладку, остановить сборку/запуск/отладку, а также детальные команды управления отладкой.
- Строка состояния имелась в большинстве рассмотренных сред разработки. Строка состояния использовалась для отображения текущей позиции курсора (строка и столбец), отображения описания команды, которую выделил пользователь, отображения состояния динамической памяти, занятой средой разработки и строки прогресса текущей операции (*progress bar*) [9].
- Навигация breadcrumbs (*«хлебные крошки»*) [26] – горизонтальная тонкая полоса, отображающая путь от директории проекта до текущего открытого файла. Каждый элемент пути (директория или сам файл) представлен кнопкой, при нажатии на которую активируется всплывающее окно, через которое можно открыть новый файл, находящийся на одном уровне каталогов с представленным кнопкой элементом пути. Этот графический элемент был представлен в средах IntelliJ IDEA и XCode.

Также нетрудно заметить, что многие сущности, отображаемые в инструментальных окнах, относятся к определенным строкам кода (например, ошибки, задачи, точки останова и др.). Поэтому будем некоторую информацию отображать в текстовом редакторе, отображая там графический элемент. Такой интерфейсный механизм будем называть *вставкой графических элементов в окно текстового редактора* или *использованием внутритекстовых графических элементов.*

5.2.2. Процесс проектирования

Для каждого обобщенного инструментального окна его функции и отображаемая информации анализируются в контексте действий тех сценариев, в которых оно может использоваться. Далее анализируются варианты представления каждой из частей функциональности, описываются их достоинства и недостатки. В результате выбирается лучший из проанализированных вариантов. К сожалению, невозможно детально провести такой анализ, не принимая во внимания детали реализации. Например, конкретные функции оптимальности окон, минимальный и максимальный размеры оптимальности не представляется возможным выяснить в общем виде. Поэтому в следующей главе опишем только основополагающие решения.

Различные варианты интерфейса могут достигаться комбинацией следующих путей:

- добавление произвольной команды в панель инструментов;

- отображение некоторой информации в breadcrumbs, возможность навигации по этой информации;
- отображение информации о состоянии в строке состояния, привязка произвольных действий к строке состояния (будем называть такое использование *добавлением визуального элемента в статическую часть строки состояния*);
- отображение прогресса операции в строке состояния;
- отображение события в строке состояния (будем называть такое использование события *добавлением в динамическую часть строки состояния*);
- отображение произвольного визуального элемента (произвольная информация и произвольные доступные действия) в окне текстового редактора (то есть в тексте программы);
- отображение произвольного всплывающего окна.

Критерии оценки вариантов интерфейса:

- вертикальный размер окна текстового редактора в данном варианте должен быть как можно больше; горизонтальный размер не может быть меньше некоторой константы, которая определяется в каждой конкретной реализации отдельно;
- число переключений (то есть использований всплывающих окон) в данном варианте должно быть как можно меньше;
- число переключений в данном варианте равно нулю на каждом сценарии;
- на данном варианте выполнены указанные выше эвристики удобства.

Очевидно, что два различных варианта могут оказаться несравнимыми по этим критериям. Например, в первом варианте может быть меньше переключений, зато большая площадь окна редактора, чем во втором варианте. К сожалению, такая ситуация типична [11] для проектирования графических интерфейсов, и есть возможность допустить ошибку. Для обнаружения таких ошибок проводится тестирование полученного интерфейса на пользователях, описанное в главе 8 «Исследование».

6. Проект интерфейса

В этой главе находится рассмотрение инструментальных окон и определение окончательного интерфейса IDE, решающего проблему инструментальных окон.

6.1. Рассмотрение инструментальных окон

Рассмотрим варианты реализации функциональности, предоставляемой каждым из обобщенных инструментальных окон, в соответствии с критериями, определенными выше.

6.1.1. Окно навигации по проекту

Рассмотрим сначала варианты реализации отображения файлов и папок.

- Использование всплывающего окна с деревом элементов. Данный вариант неприменим, поскольку для открытия файлов, входящего в состав сценария «Чтение кода» разработчику придется пользоваться всплывающим окном.
- Использование постоянного окна с деревом элементов. Окно с деревом сущностей критично к обоим размерам, поскольку при раскрытии элементов требуется вертикальное пространство во избежание использования прокрутки, а при раскрытии большого числа элементов вглубь требуется горизонтальное пространство.
- Использование горизонтальной навигации breadcrumbs. Такая навигация критична к горизонтальному размеру, отнимает вертикальный размер у окна редактора кода. Плюс данного варианта – отображение текущего редактируемого файла.
- Использование вертикальной навигации breadcrumbs. Такая навигация критична к вертикальному размеру, отнимает горизонтальный размер у окна редактора кода. Плюс данного варианта – отображение текущего редактируемого файла.

Итак, для реализации отображения файлов и папок выберем вертикальную навигацию breadcrumbs, поскольку она не уменьшает вертикальный размер окна кода, к которому оно критично. В дальнейшем будем использовать вертикальную полосу breadcrumbs.

Операции над файлами, доступные через контекстное меню в инструментальном окне навигации по проекту и ФС, переносятся без изменений в breadcrumbs.

6.1.2. Окно навигации по артефактам кода

Варианты отображения дерева объектов кода описаны ниже.

- Использование всплывающего окна с деревом элементов по активации элемента в статической части строки состояния. Данный вариант требует одного переключения для отображения всего пути в дереве. Позволяет отображать текущий элемент кода, выбранный в редакторе
- Использование уже выбранной в рассмотрении окна навигации по проектам навигации breadcrumbs. К навигационной полоске файла добавляется один или несколько элементов для отображения текущего артефакта кода.
- Использование постоянного окна с деревом артефактов кода. Данный вариант плох тем, что, как уже показано, окна с деревьями сущностей критичны к обоим размерам.

Для навигации по артефактам кода выберем использование breadcrumbs, поскольку, во-первых, он уже выбран и занимает место на экране и, во-вторых, для поддержания целостности интерфейса (см. эвристики удобства).

Все операции контекстного меню и возможность перехода полностью переносятся в breadcrumbs.

6.1.3. Окно задач

Заметим, что каждая задача связана со строкой исходного кода программы. Варианты представления функциональности окна задач описаны ниже.

- Использование статического элемента в строке состояния для отображения количества задач в открытом проекте.
- Использование всплывающего окна со списком задач. Отображать это инструментальное окно можно при активации элемента в строке состояния.
- Использование внутритекстовых вставок. Каждая из таких вставок содержит кнопки перехода к следующей и предыдущей задаче.

Очевидно, что отображать задачи в навигации breadcrumbs бессмысленно – требуемый размер вырастет так, что breadcrumbs будет занимать слишком много места. Поскольку три предложенных выше варианта не противоречат друг другу, будем использовать их одновременно.

Поместим в строку состояния элемент, отображающий число задач, и по активации его будем вызывать всплывающее окно со списком задач. Для уменьшения числа переключений на это всплывающее окно будем также отображать кнопки перехода к следующей и предыдущей задаче внутри кода. В случае если пользователь хочет пройти по всем задачам последовательно, это поможет ему не открывать список задач каждый раз. Также во внутритекстовый

элемент можно добавить кнопку закрытия задачи, которая быстро удаляет комментарий с задачей.

6.1.4. Окно ошибок сборки

Окно ошибок сборки содержит несколько функций: уведомление о появлении ошибки (далее под ошибкой понимаем и предупреждение тоже), отображение списка ошибок, возможность перехода к произвольной ошибке. Опишем варианты отображения функций и информации окна ошибок сборки.

- Добавление событий «Сборка начата», «Сборка завершена успешно», «Сборка завершена неуспешно».
- Отображение статического элемента с числом ошибок сборки.
- Добавление внутритекстовых элементов с текстом ошибки и кнопками перехода к следующей и предыдущей ошибке.
- Использование всплывающего окна со списком текущих ошибок.

Предложенные варианты не противоречат друг другу, поэтому будем комбинировать их: всплывающий список будет отображаться при нажатии на статический элемент. В соответствии со сценарием «сборка», программисту не всегда обязательно смотреть на список ошибок – достаточно последовательно переходить к ним и вносить правки, возможно запуская заново сборку проекта. Следовательно, для использования функциональности окна в рамках этого сценария достаточной кнопок перехода к соседним ошибкам, отображаемых во внутритекстовом элементе.

6.1.5. Окно точек останова

Окно точек останова позволяет редактировать свойства точек останова и переходить к произвольной точке. Варианты отображения описаны ниже.

- Добавление внутритекстовых элементов с текстом ошибки и кнопками перехода к следующей и предыдущей ошибке.
- Отображение статического элемента с числом точек останова.
- Использование всплывающего окна со списком точек останова и их свойствами.

Как и в случае с окном ошибок сборки, мы получили независимые варианты, поэтому в качестве решения выбирается их комбинация. Добавим статический элемент в строку состояния и будем показывать всплывающее окно со списком точек останова при его активации.

Также добавим внутритекстовый элемент для каждой точки останова с возможностью редактировать свойства (активна/неактивна и условие останова) и переходить к соседним точкам останова.

6.1.6. Окно нитей и стека вызовов

Данное окно используется для навигации по стеку вызовов при остановленном отладчике. Рассмотрим следующие варианты реализации.

- Всплывающее окно со списком нитей и стека вызовов, активируемое из строки состояния. Данный вариант плох тем, что и для просмотра всего стека, и для перехода к отдельному вызову требуется совершить переключение.
- Отображение списка вызовов внутри текста. Такая реализация займет много места внутри текстового редактора, что противоречит соображениям максимизации размера окна текстового редактора.
- Отображение стека вызовов в навигации breadcrumbs. Плюс этого варианта в том, что место для breadcrumbs уже отведено на экране.

Выберем вариант с breadcrumbs, поскольку он оптимальнее в отношении занимаемого места.

6.1.7. Окно наблюдаемых выражений и переменных

Варианты альтернативной реализации окна наблюдаемых переменных описаны ниже.

- Вставка списка переменных и выражений в код.
- Отображение списка переменных во всплывающем окне.
- Добавление отдельной постоянной области для отображения списка переменных.

В данном случае оптимальным решением является вставка списка переменных и выражений в код в момент остановки отладчика, поскольку во время сценария «Отладка» программист может посмотреть переменные только в этот момент, и этот вариант выигрывает и в переключениях, и в занимаемом месте. Также добавим события «Отладчик запущен» и «Отладчик остановлен» в динамическую часть строки состояния.

6.1.8. Окно вывода запущенной программы

Данные в окне вывода разрабатываемой программы полностью лишены связи с исходным кодом программы, поэтому единственным возможным вариантом является добавление статического элемента в строку состояния и показ вывода запущенной программы во всплывающем окне. Автоматическая активация этого окна в момент запуска программы экономит одно переключение, и программист вынужден делать переключение только для деактивации этого окна. Такая реализация нарушает выдвинутые требования по отсутствию переключений на

всплывающие окна во время основных сценариев, однако других вариантов реализации найдено не было. Более подробно результат этого решения будет описан главе 8 «Исследование».

6.2. Результат проектирования

Итак, все обобщенные инструментальные окна были рассмотрены и для них выбран вариант реализации, соответствующей заявленной стратегии проектирования интерфейса. Подведем итог:

- все обобщенные инструментальные окна были заменены на обобщенные интерфейсные механизмы,
- навигация breadcrumbs отображает файлы, артефакты кода и стек вызовов при отладке,
- в окно текстового редактора вставляются графические элементы заданий, ошибок сборки, точек останова и переменных и выражений отладки,
- в строке состояния находится ряд статических элементов, активирующих всплывающие окна, и ряд событий, заменяющих функцию уведомления инструментальных окон.

Следующая глава описывает реализацию построенного интерфейса в среде KDevelop.

7. Реализация

Реализация спроектированного интерфейса выполнена на базе среды разработки KDevelop 4 с открытым исходным кодом и включает:

- Механизм навигации breadcrumbs и его использование для навигации по файловой системе и стеку вызовов функций,
- Механизм показа произвольных визуальных элементов в тексте программы и его использование для отображения ошибок и предупреждений сборки, точек останова, задач и переменных во время отладки.
- Строку состояния, показывающую статусную информацию и сообщения от инструментов IDE.

7.1. Архитектура KDevelop

Архитектура среды KDevelop состоит из трех уровней [24]:

1. *Sublime*. Предоставляет минимальные средства построения интерфейса IDE, такие как инструментальные окна и окно редактирования. Классы данного уровня находятся в пакете kdevplatform и пространстве имен Sublime.

2. *Shell*. Управляет подключаемыми модулями, документами, проектами, интерфейсом, отладкой. Содержит ряд подключаемых модулей (*plugins*). Классы данного уровня находятся в пакете kdevplatform и пространстве имен KDevelop.

3. *KDevelop*. Содержит документацию, шаблоны проектов и наиболее специфичные подключаемые модули, а именно: средства сборки для конкретных инструментов (Make, CMake), поддержку отладчика (GDB) и т.д. Классы данного уровня находятся в пакете kdevelop и пространстве имен KDevelop.

Опишем подробнее эти уровни.

7.1.1. Уровень Sublime

На рисунке 3 показаны части графического интерфейса уровня Sublime и классы, реализующие эти части интерфейса. Далее следует текстовое описание основных классов.

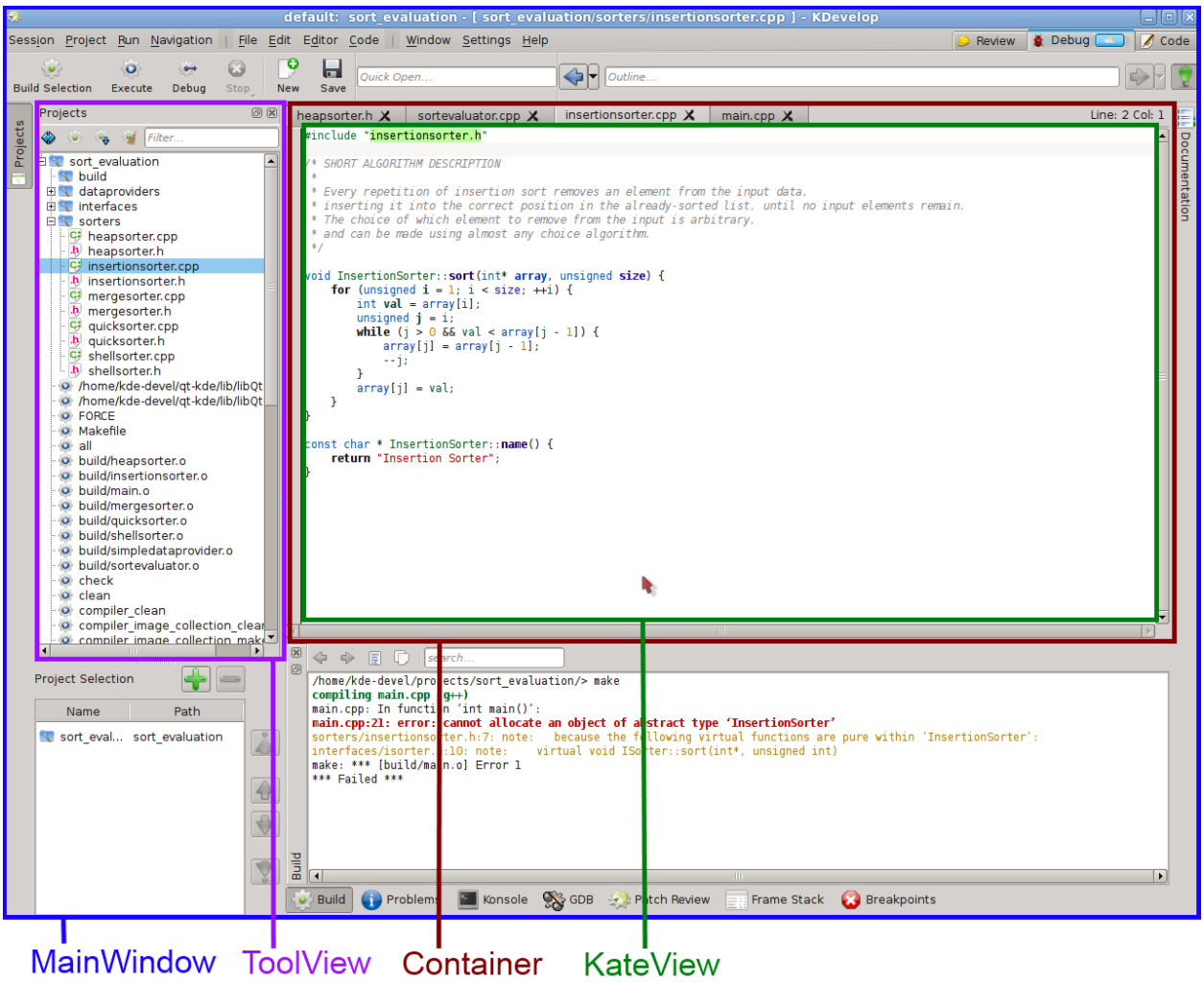

Рисунок 3. Структура графического интерфейса KDevelop.

- *MainWindow* – основное окно интерфейса Sublime. Внутренняя логика реализуется классом Sublime::MainWindowPrivate.
- Инструментальные окна реализуются классом *Sublime::ToolView*, наследующим View, а текстовое окно – классом KateView, также наследующим View. При отображении документа его View ставится в однозначное соответствие QWidget [36], собственно отображающий редактор документа (класс-предок произвольного графического элемента в Qt).
- *Document* – класс документа Sublime. Каждому документу может соответствовать несколько View, предназначенных для его редактирования.
- *Container* – графический элемент, отображающий View документов (текстовый редактор) с возможностью переключения между View с помощью вкладок.
- *View* – класс, определяющий некоторое окно (как инструментальное, так и текстовое) в одном из Container. Является предком класса KateView, реализующим текстовый редактор Kate.

- *Controller* – центральная часть уровня Sublime. Содержит области, документы и инструментальные окна и управляет ими.

Также на уровне Sublime находятся все классы-интерфейсы сущностей, реализуемых в Shell и KDevelop, работы с которыми необходимо провести в Sublime. К таким классам-интерфейсам относится, например, IUiController – предок KDevelop::UiController, определяющий операции, используемые классами Sublime.

7.1.2. Уровень Shell

Структура уровня Shell проще структуры уровня Sublime: основой уровня Shell является класс Shell::Core, содержащий различные контроллеры и управляющий ими. Также Shell содержит несколько подгружаемых модулей.

Классы Shell:

- *Core* – ядро, центральный класс KDevelop. Служит для доступа к контроллерам и хранит информацию о них.
- *DocumentController* реализует контроллер документов, который управляет всеми документами, содержит информацию о них, испускает сигналы об открытии/закрытии.
- *UiController* – контроллер интерфейса. Отвечает за создание интерфейса KDevelop и обработку его изменений.
- *RunController* – контроллер задач. Управляет запуском отладки, запуском скомпилированных программ и т.д.
- *PluginController* – контроллер подгружаемых модулей. Управляет загрузкой и работой этих модулей.
- *SessionController* – контроллер сессий, служащий для управления запущенными процессами.
- *LanguageController* – управление языками кода пользователя. Также осуществляет проверку синтаксиса кода в фоновом режиме.
- *MainWindow* описывает главное окно интерфейса KDevelop. Содержит Sublime::MainWindow в качестве члена класса.
- *OpenProjectDialog, Shell::SaveDialog* – диалоги открытия проекта, сохранения файла.
- *ProblemReporterPlugin* – подгружаемый модуль, реализующий фоновую проверку синтаксиса и предоставляющий окно для отображения результатов этой проверки.

- *BreakpointWidget* – класс, реализующий инструментальное окно точек останова.
- *FrameStackModel* – Qt-модель [36], хранящая данные о нитях и вызовах отлаживаемого приложения.

7.1.3. Уровень KDevelop

Уровень KDevelop содержит реализацию некоторых специфичных классов, строящуюся на базе Sublime и Shell.

При реализации были использованы следующие классы KDevelop:

- *DebuggerPlugin* – подключаемый модуль для отладчика GDB. Используется собственно для управления отладчиком и создания инструментального окна отладчика. В реализации был использован для получения данных о нитях и вызовах.
- *GdbFrameStackModel* – наследник Shell::FrameStackModel, содержащий специфичные для GDB данные о нитях и вызовах.
- *MakeBuilder* – подгружаемый модуль сборки, связанный с инструментом Make. Используется в реализации для получения данных об ошибках сборки.
- *MakeJob* – класс, хранящий информацию об ошибках сборки инструментом Make.

7.2. Механизм навигации breadcrumbs

Этот раздел описывает реализацию механизма навигации breadcrumbs. Вид реализации этого механизма (для файлов и для стека вызовов) в KDevelop показан на рисунках 4 и 5. Данная реализация данного механизма основана на архитектуре Model/View, поэтому детальному описанию этой реализации предшествует краткий обзор архитектуры Model/View.

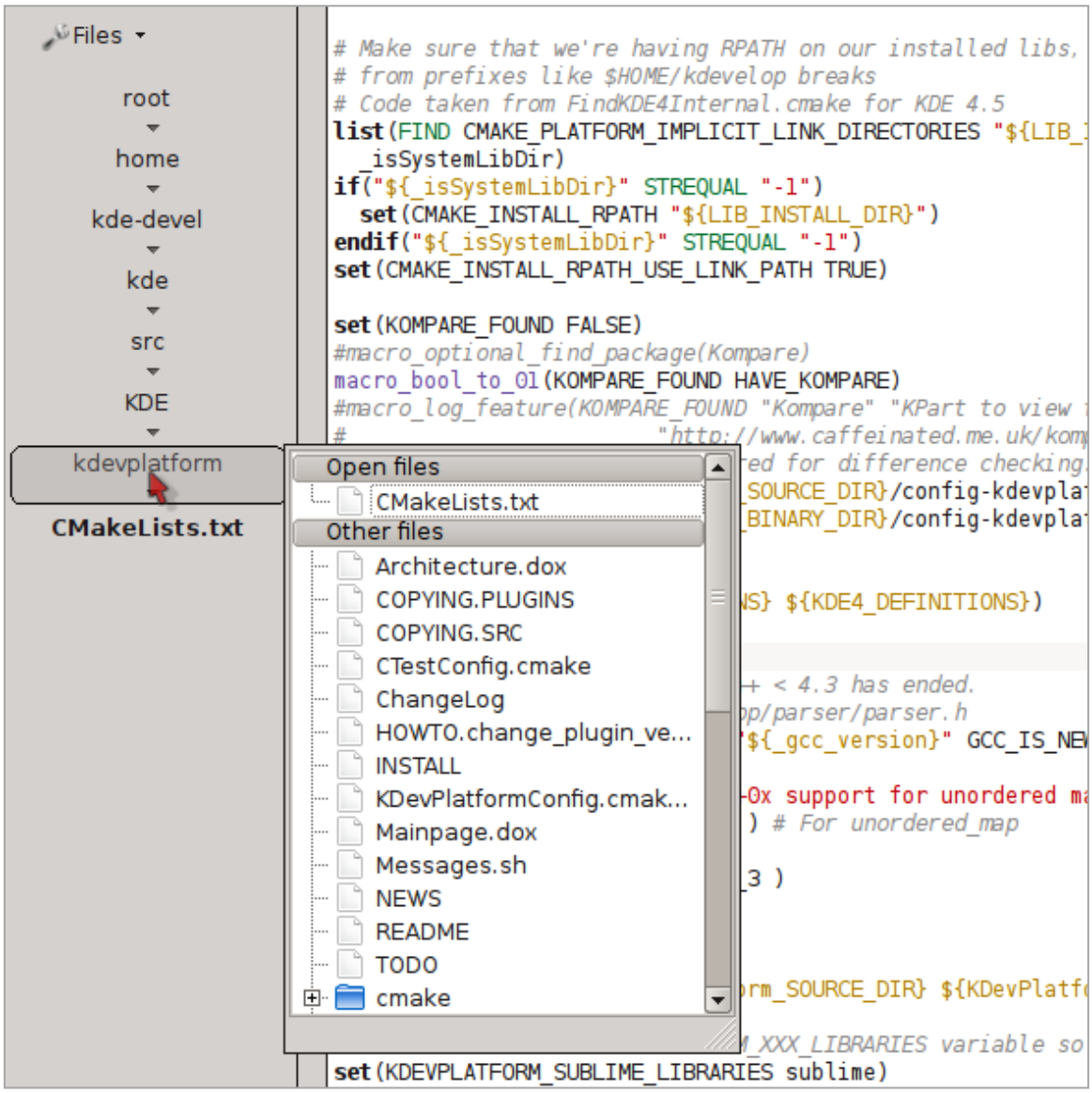

Рисунок 4. Реализованный механизм breadcrumbs для отображения файлов в KDevelop.

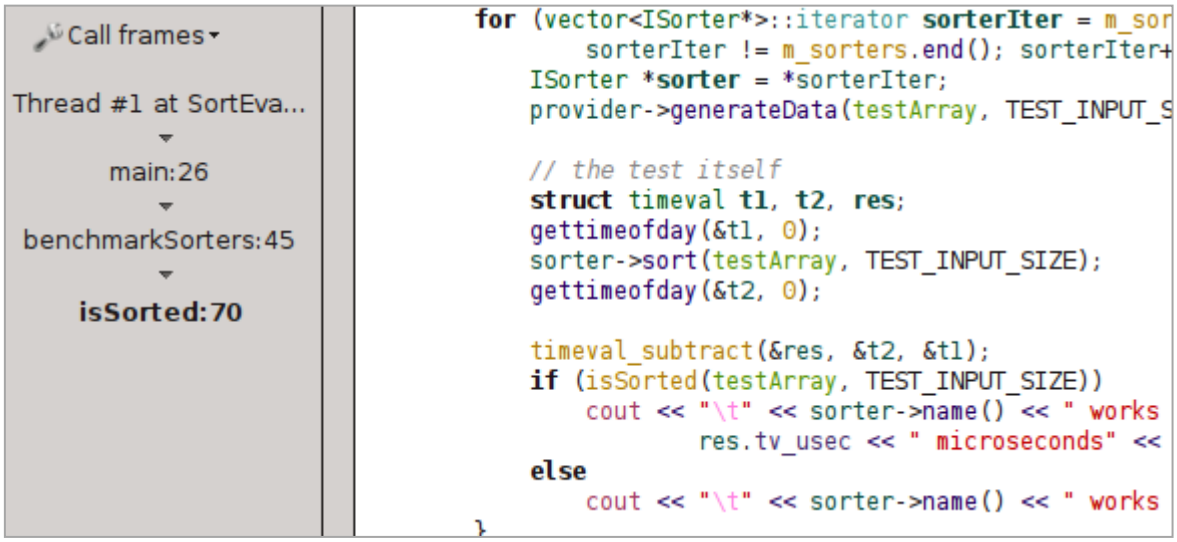

Рисунок 5. Реализованный механизм breadcrumbs для отображения стека вызовов в KDevelop.

7.2.1. Архитектура Model-View

Архитектура Model-View [37] – шаблон объектно-ориентированной архитектуры в разработке программного обеспечения, заключающийся в разделении данных приложения и его интерфейса с пользователем. Этим шаблоном предлагается использовать model-часть как хранилище данных и view-часть как их отображение. Достоинства архитектуры Model-View:

- изоляция данных и их представления,
- независимость модификации данных и их представления,
- возможность использовать одно представление для разных данных,
- возможность использовать одни данные для разных представлений.

Достоинства данной архитектуры применимы к решаемой задаче благодаря тому, что отображение средства навигации для каждого открытого документа должно производиться на уровне Sublime, на котором ничего не известно о данных, по которым возможно придется производить навигацию. Такие данные доступны, в большинстве случаев, на уровнях Shell и KDevelop.

Использование Model/View архитектуры упрощается тем, что в Qt есть поддержка реализации приложений с такой архитектурой [36]. В Qt модель (model) [36] – произвольное дерево элементов, где у каждого элемента есть несколько колонок данных, содержащих различную информацию. Корень модельного дерева не содержит информации. Потомки корня называются элементами верхнего уровня. Также в Qt есть ряд View-классов, отображающих модели (например, в виде таблицы, списка, дерева). Далее под словом модель будем понимать Model-класс Qt.

Реализуем view-часть средства навигации (показ навигационной полоски) на уровне Sublime, а model-части (хранение данных) – на уровнях Shell и KDevelop. В качестве model-части будет выступать класс, содержащий модель навигации и обработчик выбора элементов из модели навигации. Такие классы будем далее называть *навигационными движками*. Передачу из Shell в Sublime и кэширование моделей навигации будет осуществлять Sublime::Controller.

7.2.2. Механизм навигации: структура классов

Навигационная полоска. Графическим отображением навигационного механизма является полоска, состоящая из кнопок со стрелками, находящаяся слева от окна текстового редактора. Полоска создается классом Sublime::Container при создании новой вкладки для документа.

Использование произвольного навигационного движка. Каждый элемент навигационной модели определяет кнопку в полоске навигации. Потомки элемента верхнего уровня образуют дерево, отображаемое при нажатии соответствующей элементу кнопки. Также модель может предоставлять данные о том, нуждается ли данный элемент в выделении цветом, в специальном рисовании и специальной обработке нажатия на него.

При выборе пользователем элемента модели управление передается навигационному движку, который обладает необходимыми данными для выполнения связанной с моделью операции (например, открытие нужного файла или переход к нужной функции).

Было реализовано два навигационных движка:

- для навигации по файлам и артефактам кода;
- для навигации по нитям и стеку вызовов.

Классы, относящиеся к навигационной полоске breadcrumbs, описаны ниже. Диаграмма классов находится на рисунке 6.

- *GenericNavigator* – собственно класс навигатора. Наследует себя от класса QWidget. Для использования навигатора достаточно вставить этот визуальный элемент в интерфейс и предоставить ему движок навигации.
- *INavigationEngine* – класс-интерфейс движка навигации. Определяет операции задания/получения навигационной модели и обработки выбора элемента модели.
- *FileNavigationEngine* – навигационный движок для файлов и артефактов кода. Наследует класс Sublime::INavigationEngine.

- *FrameStackNavigationEngine* – навигационный движок по нитям и стеку вызовов. Наследует класс Sublime::INavigationEngine.

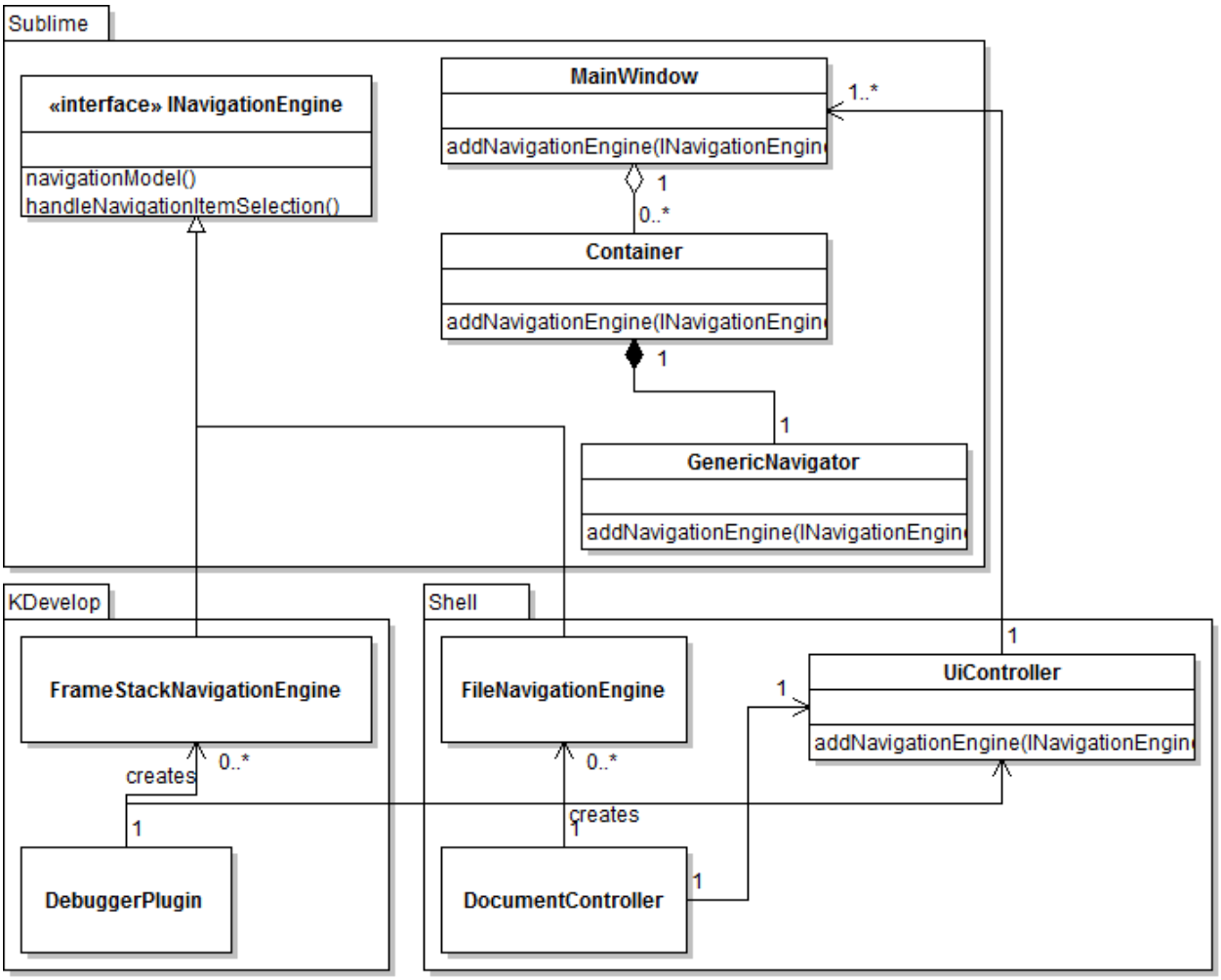

Рисунок 6. Диаграмма классов навигации.

7.2.3. Механизм навигации: передача движков

Движок навигации по файловой системе создается на уровне Shell при открытии нового документа, после чего передается навигатору через класс UiController. Навигационный движок по нитям и стеку вызовов создается классом DebuggerPlugin при начале сессии отладки. Диаграмма последовательности при передаче движков навигатору изображена на рисунке 7.

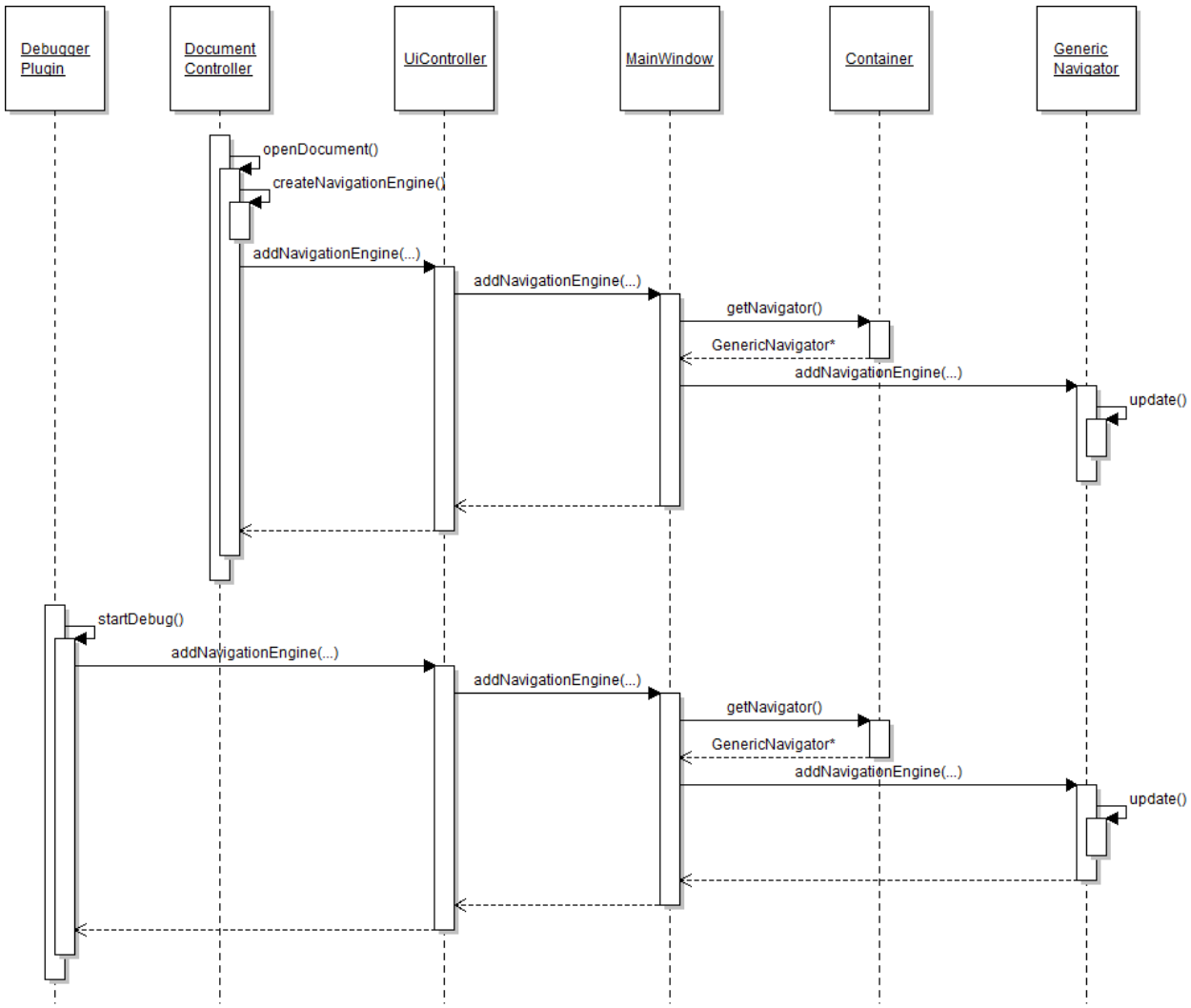

Рисунок 7. Последовательная диаграмма для добавления навигации.

7.3. Показ визуальных элементов в тексте программы

Ниже будет описана реализация механизма показа произвольных визуальных объектов между строк текстового редактора Kate и применение этого механизма для показа ошибок, точек останова, в тексте программы. Вид внутритекстовых элементов (ошибка, точка останова, переменные) в KDevelop изображен на рисунках 8-10.

```
evaluator.addSorter(new InsertionSorter());
evaluator.addSorter(new MergeSorter());
evaluator.addSorter(new HeapSorter());
evaluator.addSorter(new QuickSorter());
evaluator.addSorter(new ShellSorter());
error: no matching function for call to 'SortEvaluator::addSorter(SimpleDataProvider*)'
evaluator.addSorter(new SimpleDataProvider());
```

Рисунок 8. Внутритекстовый элемент ошибки сборки, реализованный в KDevelop.

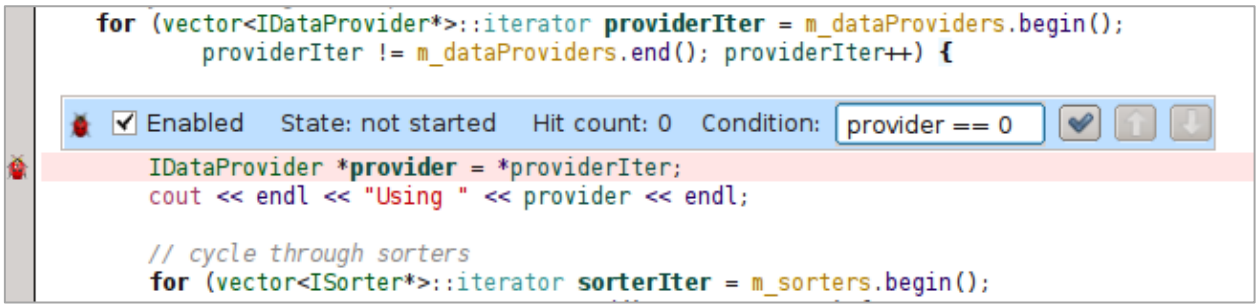

Рисунок 9. Внутритекстовый элемент точки останова, реализованный в KDevelop.

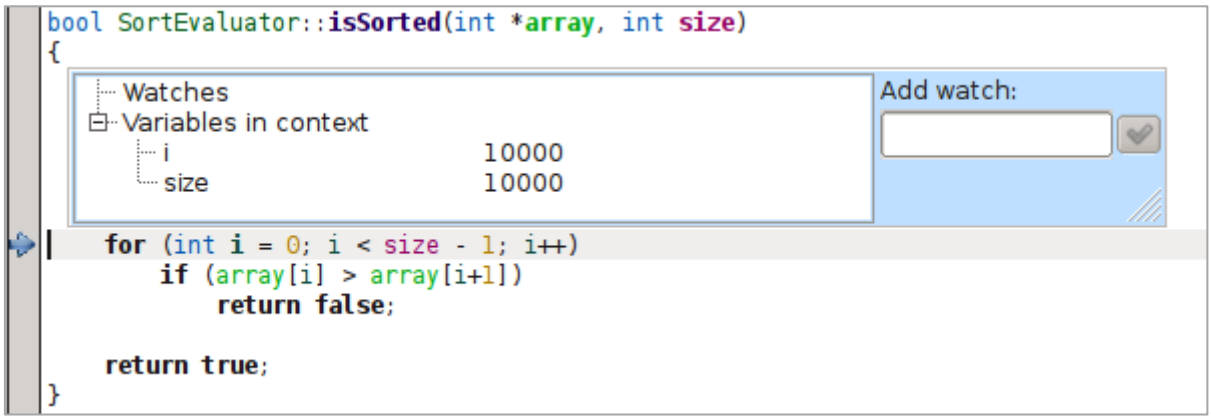

Рисунок 10. Внутритекстовый элемент переменных, реализованный в KDevelop.

7.3.1. Архитектура Kate

Kate – текстовый редактор среды KDE [38]. Окно редактирования текста Kate может быть использовано в другом приложении при помощи технологии KParts [39], так что для показа произвольных визуальных элементов в IDE KDevelop необходима модификация кода окна редактирования Kate.

Отображение окна текстового редактора осуществляется классом KateView, использующим для реализации внутренней логики класс KateViewInternal. Подробное описание основных классов Kate:

- *KateView*. Реализует окно текстового редактора (вместе с прокруткой - скроллингом). Реализует интерфейс KTextEditor::View и ряд других, через которые KDevelop обращается к этому окну. KateView использует KateViewInternal для реализации редактирования текста.
- *KateViewInternal*. Реализует редактирование текста: от отображения на экране и прокрутки до ввода символов и перемещения курсора. Используется только KateView, не реализует интерфейсов. KateViewInternal также взаимодействует с другими классами, предоставляющими

возможность разбиения текста на экранные строки, кэширования и низкоуровневой отрисовки текста.

- *KateDocument*. Класс документа Kate. Наследует интерфейс KTextEditor и ряд других, через которые KDevelop взаимодействует с этим документом. Одному документу может соответствовать один или более отображений KateView.

7.3.2. Механизм показа произвольных визуальных элементов в тексте

Ниже описывается механизм, позволяющий показать набор визуальных элементов между строками текста в окне текстового редактора Kate. Визуальный элемент представлен классом-наследником QWidget.

Для передачи визуального элемента в Kate был выбран шаблон абстрактная фабрика [40], поскольку он позволяет отложить и повторить создание объекта в любой момент независимо от первоначального создателя фабрики.

Для передачи визуального элемента из кода KDevelop в код Kate был разработан интерфейс InlineWidgetInterface, реализуемый KateDocument. Данный интерфейс содержит метод вставки визуального элемента с указанием номера реальной строки документа (KateLineLayout), перед которой необходимо вставить визуальный элемент, и уникальный идентификатор этого визуального элемента. Данный метод также получает указатель на IInlineWidgetFactory – интерфейс абстрактных фабрик. Также интерфейс InlineWidgetInterface имеет метод удаления визуального элемента по его уникальному идентификатору и номеру строки.

Класс KateDocument запоминает переданные ему фабрики и создает визуальные элементы для своих текстовых представлений (KateView). KateView, в свою очередь, передает визуальные элементы в KateViewInternal для их хранения и отображения.

7.3.3. Показ ошибок, точек останова, переменных в тексте программы

Описанный выше механизм используется для отображения визуальных элементов ошибок, переменных и точек останова в тексте программы. Фабрика MakeBuilderInlineWidgetFactory визуальных элементов ошибок (и предупреждений) сборки передается из модуля MakeBuilder. Фабрики визуальных элементов точек останова BreakpointInlineWidgetFactory и переменных VariableInlineWidgetFactory

передаются из подмодуля Debugger уровня Shell. Диаграмма классов показана на рисунке 11.

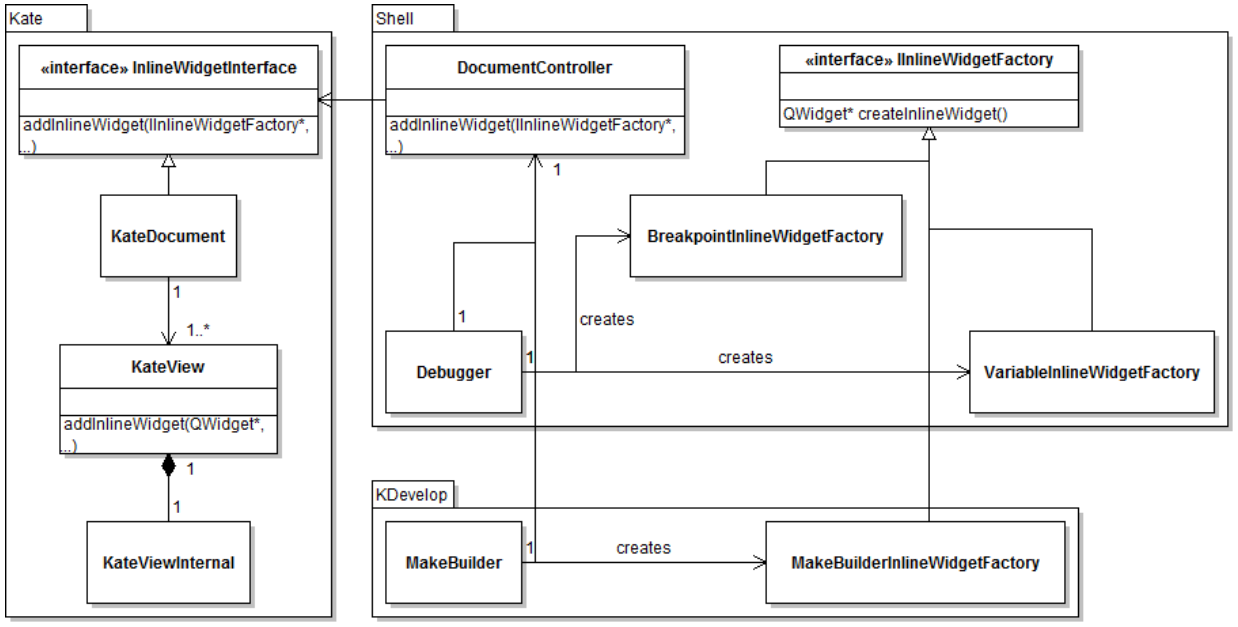

Рисунок 11. Диаграмма классов внутритекстовых элементов.

Диаграмма последовательности показа элемента ошибки (остальные происходят аналогично) показана на рисунке 12.

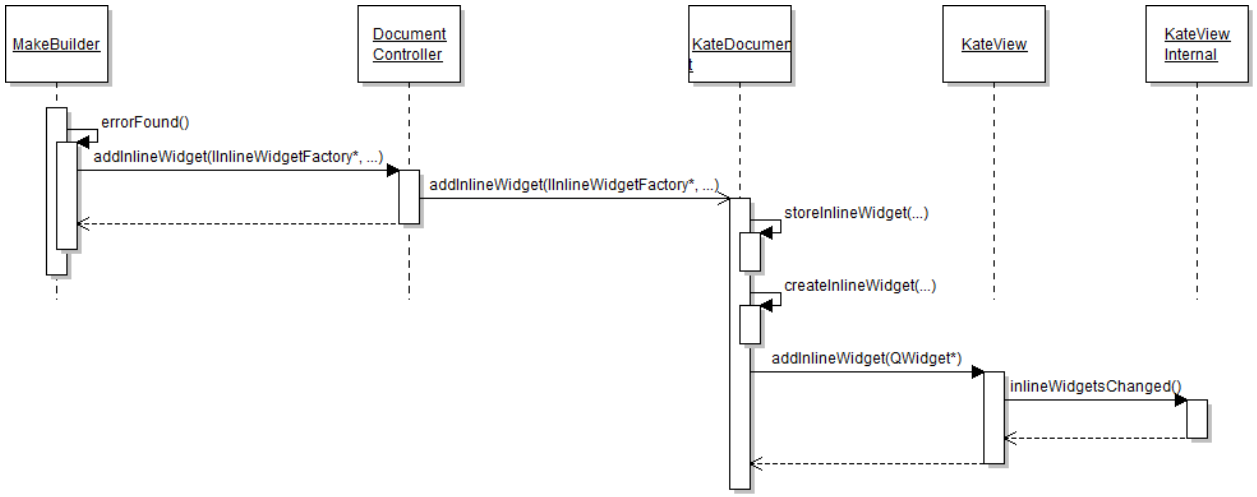

Рисунок 12. Диаграмма последовательности внутритекстовых элементов.

7.4. Строка состояния

Строка состояния, необходимая для построения интерфейса, уже была описана в главе 6 «Проект интерфейса». Отдельно опишем архитектуру, реализацию отображения, а также реализацию динамической и статической частей строки состояния.

7.4.1. Строка состояния: архитектура

Как уже отмечалось, классы KDevelop, создающие основу пользовательского интерфейса, находятся на уровне Sublime. Для наибольшей абстрактности и расширяемости реализации строки состояния опишем ее обобщенный внешний вид на уровне Sublime, а конкретное заполнение строки будет создаваться в том коде, где имеется для этого необходимая информация (например, число ошибок). Такой код, как правило, расположен на уровнях Shell и KDevelop. Достоинством такой архитектуры является легкая расширяемость обеих частей строки состояния без необходимости модификации ее кода на уровне Sublime.

Для статической части реализуем следующие визуальные элементы.

- Число ошибок сборки. При активации этого элемента происходит переход на очередную ошибку сборки. Также можно вызвать всплывающее окно со списком ошибок сборки.
- Текущее состояние отладчика. Активация открывает точку текущей остановки отладчика, если такая имеется. Также можно вызвать всплывающее окно с выводом отладчика и отлаживаемой программы.
- Число точек останова. Активация этого элемента приводит к переходу по точкам останова. Есть возможность вызвать список точек останова.
- Текущее состояние запущенной программы. К активации этого элемента не привязано действий. Возможен вызов вывода запущенной программы.

Для динамической части реализуем следующие сообщения:

- окончательная загрузка среды KDevelop,
- начало и конец сборки проекта,
- конец фоновой проверки синтаксиса,
- изменение состояния отладчика (запущен, работает, приостановлен, остановлен).

Данные для строки состояния будем передавать через UiController – класс уровня Shell. UiController имеет указатель на строку состояния, являясь наследником класса Sublime::Controller. Для этого создадим два класса-интерфейса:

- *IStatusWidgetFactory* – интерфейс фабрики [40] графических элементов статической части. После передачи указателя на этот интерфейс строке состояния она сможет в любой момент создать соответствующий графический элемент, а также вернуть его уникальный идентификатор.
- *IStatusMessage* – интерфейс сообщения для динамической части. Данный интерфейс позволяет получить текст сообщения и выполнить связанное с ним действие при выборе (активации) пользователем этого сообщения.

В UiController добавляются методы для передачи ссылок на указанные два класса строке состояния, так что теперь в любой точке Shell и KDevelop можно передать фабрику статического элемента или динамическое сообщение строке состояния.

Другие классы, относящиеся к строке состояния, перечислены ниже.

- *EnhancedStatusBar* – класс самой строки состояния. Имеет методы для добавления и удаления статусных элементов и сообщений. Находится на архитектурном уровне Sublime.
- *GDBStatusWidget (Factory)* – классы статусного элемента отладчика.
- *MakeBuilderStatusWidget (Factory)* – классы статусного элемента системы сборки.
- *BreakpointStatusWidget (Factory)* – классы графического элемента с числом точек останова.
- *ExecuteStatusWidget (Factory)* – класс статического элемента состояния запущенной программы.
- *StandardStatusMessage* – класс универсальных сообщений, содержащих только текст и никакой реакции на их активацию. Через него реализовано сообщение о загрузке IDE и фоновой проверке синтаксиса.
- *DebugStatusMessage* – класс сообщений отладчика, открывающий всплывающее окно при активации (из соответствующего статического элемента отладчика).
- *BuildStatusMessage* – класс сообщений системы сборки, открывающий окно ошибок при активации.

Диаграмма этих классов показана на рисунке 13.

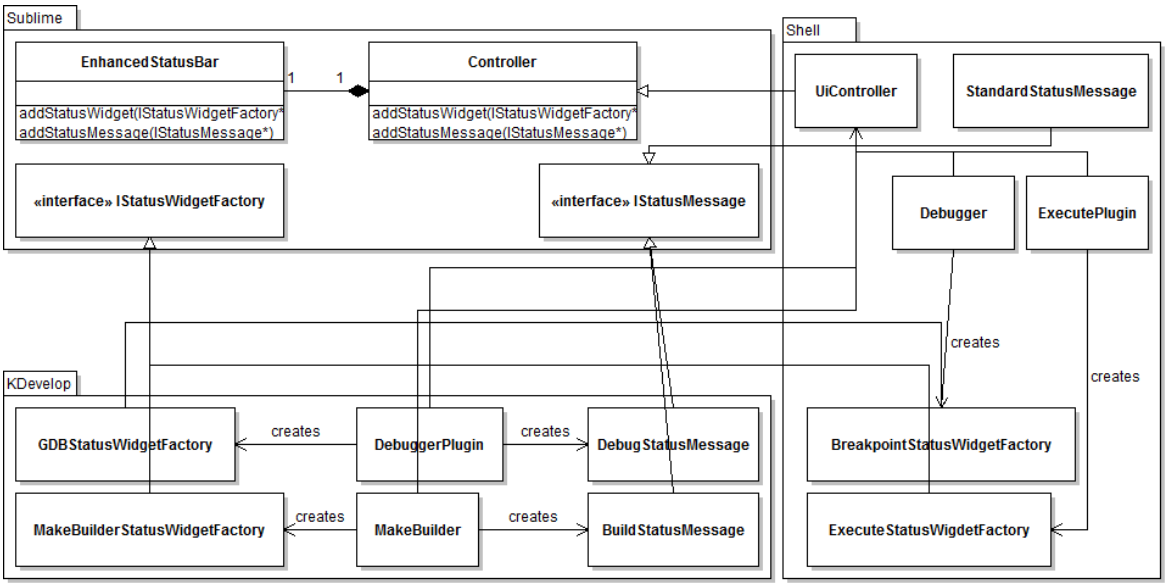

Рисунок 13. Диаграмма классов строки состояния.

7.4.2. Строка состояния: интерфейс

Строка состояния состоит из места для статических элементов (слева), строки прогресса (progress bar), отображающей процент выполнения текущей задачи (она уже была реализована в KDevelop) и области сообщений о событиях (справа). Есть возможность просмотра списка пришедших всех сообщений, реализуемая классом EventDialog. Реализованная строка состояния изображена на рисунках 14 и 15.

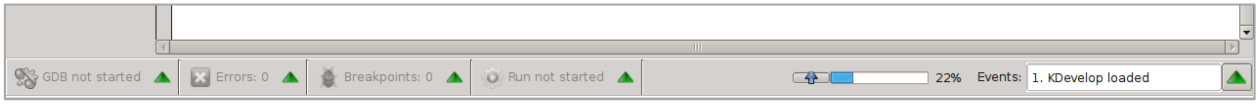

Рисунок 14. Строка состояния с индикатором прогресса, реализованная в KDevelop.

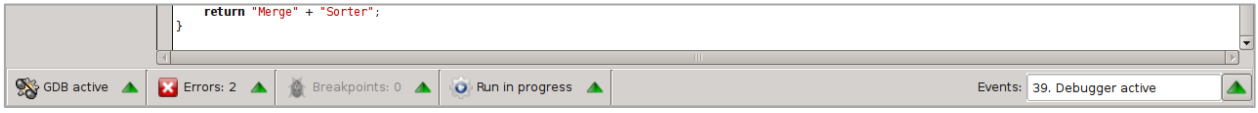

Рисунок 15. Строка состояния без индикатора прогресса, реализованная в KDevelop.

7.4.3. Строка состояния: передача данных

Все данные (и фабрики элементов, и сообщения) передаются в EnhancedStatusBar через UiController. Диаграмма последовательности на примере статусного элемента точек останова и сообщения о завершении сборки показана на рисунке 17.

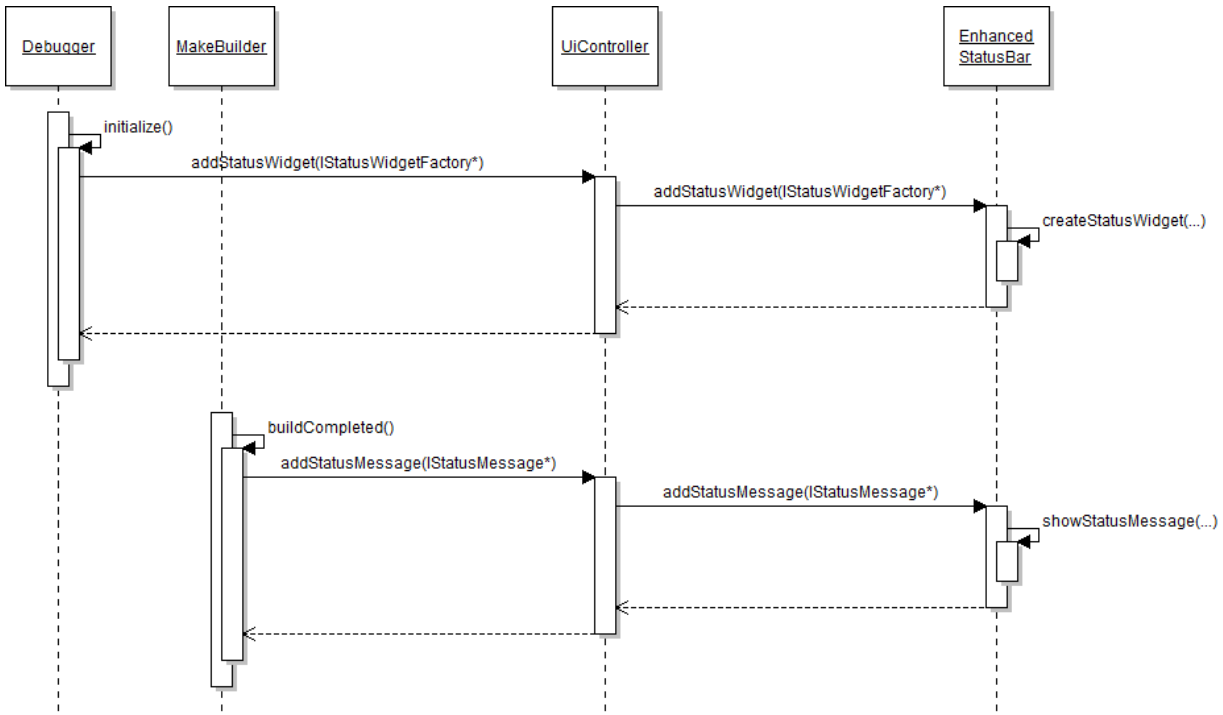

Рисунок 16. Последовательная диаграмма строки состояния.

8. Исследование

Данная глава описывает исследование реализованного интерфейса на пользователях.

8.1. Цель исследования

Целью данного исследования является поиск *проблем* (или *ошибок*) *удобства* (*usability problems/errors* [14]) реализованного интерфейса для получения выводов о подходе к интерфейсу IDE, исследуемому в данной работе, и возможных направлений улучшения разработанного интерфейса.

8.2. Методика исследования

Методика тестирования графического интерфейса на пользователях была адаптирована для данной работы из [5] и [41]. Ниже описаны основные аспекты проведенного тестирования.

Поскольку при тестировании на малых количествах пользователей, принадлежащих разным группам (например, по опыту работы с IDE) анализ численных результатов не будет обладать корректностью и переносимостью [42] [33], мы будем использовать качественные данные для получения выводов об интерфейсе. В соответствии с [43], достаточно протестировать всего шесть пользователей для нахождения более 80% проблем удобства интерфейса.

Для полного тестирования интерфейса необходимо наблюдать проведение пользователями действий, которые ранее проводились через инструментальные окна. Набор этих действий мы возьмем из сценариев работы программиста с инструментальными окнами, описанных в главе 3 «Обзор предметной области». К сожалению, невозможно гарантировать выполнение программистами всех действий, указанных там, задавая некоторые задачи, однако можно гарантировать, что в процессе тестирования будут задействованы сценарии работы.

Выборка пользователей состояла из программистов с различным стажем и предпочтениями среди инструментов разработки. В выборку вошли программисты, использующие в своей повседневной деятельности как интегрированные среды, так и отдельные инструменты разработки.

В качестве тестовых данных было решено взять единую программу, а не набор тестовых заданий, поскольку тестирование должно максимально

аппроксимировать реальное использование IDE [42], в течение которого один сценарий, как уже отмечалось, может перетекать в другой.

Программа, взятая в качестве тестовой, оценивает скорость следующих алгоритмов сортировки: вставками, слиянием, Шелла, быстрой, пирамидальной [44]. В эту программу было внесено три ошибки сборки и две ошибки исполнения так, чтобы гарантировать участие пользователя в четырех упомянутых выше сценариях. Тестовое задание для программистов было такое: собрать проект и исправить ошибки исполнения так, чтобы все сортировки работали корректно. Определить, какие сортировки работают корректно, пользователи могли из вывода программы.

Перед непосредственным тестированием проводилась демонстрация основных элементов интерфейса и их использования. После каждой сессии тестирования делалась попытка исправить обнаруженные проблемы удобства (при условии, что это возможно в кратчайшие сроки), чтобы следующие пользователи уже не сталкивались с обнаруженными проблемами и выявляли новые. Такой подход является условием при использовании оценки количества найденных проблем удобства в [43].

8.3. Результаты исследования

Реализованный интерфейс был протестирован на семи пользователях. Все успешно выполнили задание. Сессии тестирования длились от 15 до 20 минут, включая первоначальную демонстрацию интерфейса. Пользователи успешно использовали все реализованные графические элементы для выполнения действий, ранее производившихся через инструментальные окна.

Результаты тестирования опишем по следующим категориям:

- ошибки анализа поведения программиста – это ошибки, допущенные в процессе обзора предметной области и составления сценариев работы программиста;
- ошибки собственно интерфейса – это ошибки проектирования интерфейса в данной работе и взаимодействия с пользователем;
- ошибки реализации интерфейса – это ошибки реализации верного интерфейсного решения;
- предложения по улучшению интерфейса, выдвинутые автором или пользователями безотносительно проблем удобства.

8.3.1. Ошибки анализа поведения программиста

- Тестирование показало, что не все программисты исправляют ошибки последовательно, начиная с первой. Возможна ситуация, когда

программист открывает историю сборки и пытается устранить сначала кажущиеся ему простыми для исправления ошибки, а потом браться за сложные.

8.3.2. Ошибки интерфейса

- Вставка визуальных элементов в текст и их удаление вызывает сдвиг строк в текстовом редакторе, что вначале запутывает пользователей, и они полагают, что текст программы изменился. Эту проблему можно исправить, отображая визуальные элементы без раздвигания строк, помещая их в пустом пространстве справа от текста.
- Визуальные элементы внутри текста могут мешать программистам модифицировать текст программы, но в то же время программисты могут захотеть обратиться к ним после редактирования. Для исправления этой ошибки следует реализовать сворачивание внутритекстовых элементов до иконки в вертикальном столбце слева от текста и открытие по щелчку на этой иконке.
- Помещение списка контекстных переменных и наблюдаемых выражений в текст занимает, по мнению протестированных пользователей, слишком много места и мешает восприятию текста. Возможные решения: либо отображать переменные справа от текста в пустом пространстве без сдвига строк, либо добавлять визуальный элемент в свернутом до иконки виде и давать пользователю возможность открыть его при необходимости. Эти соображения основываются на том, что не при каждой остановке отладчика в тексте пользователь заинтересован в значениях ошибок.
- Вертикальная полоса навигации breadcrumbs в случае с небольшими проектами имеет много свободного места. Возможные использования этого места – это отображение нескольких навигационных путей подряд (например, для файлов и стека вызовов) или перенос вкладок текстового редактора в эту пустую область. Один из вариантов, предложенных тестируемыми пользователями, – это отображение там списка недавно посещенных или недавно отредактированных файлов.
- Поскольку отобразить вывод разрабатываемой программы удалось лишь во всплывающем окне, пользователи были вынуждены открывать его даже в рамках сценариев. Решить эту проблему без переключений, к сожалению, не удалось. Другие варианты реализации – это открытие лога программы как текстового файла. Следует также рассмотреть вариант использования программистом двух мониторов. В таком случае можно использовать другой монитор для отображения вывода программы без переключений.
- Строка состояния может быть слабозаметной пользователю, поэтому отображение событий необходимо сделать привлекающим больше

внимания. Например, можно показывать всплывающую подсказку на несколько секунд после события, а потом автоматически скрывать ее.

8.3.3. Ошибки реализации интерфейса

- Компилятор gcc, используемый при тестировании, может разбивать одну фактическую ошибку в коде программы на несколько формальных ошибок (например, нарушение прав доступа к функции класса порождает две ошибки в смысле make: одну в точку определения функции и одну в точке ее использования). Из-за этой детали реализации make вставка визуальных элементов происходила в оба места, и текст ошибки разбивался на две части, которые невозможно было прочитать одновременно, а пользователю приходилось переходить от одной ошибки к другой, тратя на это дополнительные усилия.
- Реализация отладчика в KDevelop не отличает различные причины остановки, поэтому пользователю может быть неясно, почему программа была остановлена: в момент старта, по причине ошибки выполнения, точки останова или успешного выполнения. Из-за этого пользователям приходилось открывать всплывающее окно с выводом отладчика. Этой проблемы можно избежать, визуально отображая причину остановки отладчика.

8.3.4. Предложения по улучшению интерфейса

- Интерфейс показа ошибок в тексте удобней использовать, когда сборка не останавливается на первой ошибке, а проходит все файлы проекта. В таком случае пользователь может последовательно переключаться между ошибками и исправлять их.
- Тестируемыми пользователями было предложено добавить клавиатурные сочетания клавиш для управления визуальными элементами. Поскольку число необходимых сочетаний сократилось по сравнению с традиционным интерфейсом инструментальных окон, сочетания клавиш будут легче запоминаемы пользователями.
- Следует интегрировать навигацию по файлам в breadcrumbs с другими сущностями среды разработки, например с ошибками и точками останова. Для этого в кнопках breadcrumbs предлагается сделать цветовые индикаторы, отображая, есть ли в данном поддереве каталогов (или же вызове функции) ошибки или точки останова.
- Во внутритекстовом элементе ошибки можно отображать число раз, которое ошибка возникла при компиляции. Так программист сможет отличать только что возникшие ошибки от старых. Возможно, накопление IDE таких данных об ошибках позволит улучшить навигацию по ним.

8.4. Выводы исследования

Исследование показало, что функциональность большинства инструментальных окон действительно может быть разделена на части и более удобно представлена в IDE. В частности, ошибки, точки останова и задачи могут быть показаны внутритекстовыми элементами, что уменьшает число переключений. С точки зрения модели интерфейса среды разработки, это увеличивает пространство, используемое редактором кода, и уменьшает число переключений между окнами внутри основного окна IDE.

Программисты могут успешно использовать использованные обобщенные интерфейсные механизмы (навигация breadcrumbs, внутритекстовые графические элементы, строка состояния) для достижения своих целей в процессе разработки. Эти факты дают основание утверждать, что исследуемый в данной работе подход применим в большинстве ситуаций и может если не решить, то сильно облегчить проблему инструментальных окон.

Однако наш подход оказался неприменим для инструментального окна, чье содержимое не обладает заранее известной структурой и вынуждено восприниматься IDE как текст. Это окно, содержащее текстовый вывод запускаемой программы, мы вынуждены были показывать в виде всплывающего окна. Несмотря на то, что известен момент, когда нужно показывать это окно (старт программы), пользователю приходилось самостоятельно закрывать это окно для перехода к тексту программы.

9. Заключение

В данной работе был проведен обзор существующих интегрированных сред разработки программ; на основе этого обзора составлен список обобщенных инструментальных окон. Также был описан набор сценариев работы с инструментальными окнами.

Была создана модель инструментальных окон IDE, описывающая их расположение, особенности требований инструментальных окон к занимаемому ими пространству и формализующая требования пользователя к уменьшению взаимодействия с инструментальными окнами.

Был разработан проект интерфейса без инструментальных окон, содержащий постоянные области. В этом интерфейсе размер текстового окна максимален, а число переключений на всплывающие окна минимально.

В рамках этой работы на базе KDevelop 4 были реализованы:

- обобщенный механизм навигации и его использование для навигации по файловой системе и стеку вызовов функций,
- механизм показа произвольных визуальных элементов в тексте программы и его использование для показа ошибок и предупреждений сборки, точек останова и переменных,
- строка состояния, показывающая информацию о состоянии и сообщения о событиях от различных инструментов IDE.

Реализованный интерфейс был исследован на пользователях. По результатам тестирования были оформлены детальные выводы о применимости исследуемого подхода и возможных улучшениях спроектированного интерфейса. Выводы свидетельствуют о применимости подхода к проектированию графического интерфейса среды разработки без инструментальных окон с незначительными ограничениями.

Итак, проделанная работа соответствует поставленной задаче и решает ее.

10. Список литературы

1. Susan Dart, Robert Ellison, Peter Feiler, Nico Habermann Software Development Environments // Computer. 1987. 20. N 11. P. 18-28.

2. P.B. Henderson, D. Notkin Guest Editors' Introduction: Integrated Design and Programming Environments // Computer. 1987. 20. N 11. P. 12-18.

3. Rex Bryan Kline, Ahmed Seffah Evaluation of integrated software development environments: challenges and results from three empirical studies // J. International Journal of Human-Computer Studies. 2005. N 63. P. 607-627.

4. Developpez LLC Les meilleurs environnements de developpement [HTML] (http://general.developpez.com/edi/).

5. Jakob Nielsen Usability Engineering. Boston, USA: Academic Press, 1993. 352 p.

6. Dennis G. Jerz Usability Testing: What is it? [HTML] (http://jerz.setonhill.edu/design/usability/intro.htm).

7. Mark Szymczyk Reducing XCode’s Window Clutter [HTML] (http://meandmarkpublishing.blogspot.com/2007/06/reducing-xcodes-window-clutter.html).

8. M. Stephens 10 Things NetBeans Must Do to Survive [HTML] (http://www.softwarereality.com/soapbox/netbeans.jsp).

9. A. Cooper About Face: The Essentials of User Interface Design. New York: Wiley Publishing, 2007. 651 p.

10. Dave Springgay Using Perspectives in the Eclipse UI [HTML] (http://www.eclipse.org/articles/using-perspectives/PerspectiveArticle.html).

11. Alan Cooper Inmates Are Running the Asylum. Indianapolis, USA: Sams Publishing, 2004. 288 p.

12. Andreas Holzinger Usability engineering methods for software developers // Communications of the ACM - Interaction design and children. 2005. 48. N 1. P. 71 - 74.

13. P. Kokol, I. Rozman, V. Venuti User Interface Metrics // ACM SIGPLAN Notices. 1995. 30. N 4. P. 36-38.

14. Steve Krug Don’t Make Me Think, A Common Sense Approach to Web Usability. Indianapolis, USA: New Riders, 2000. 195 p.

15. Jakob Nielsen, Robert L. Mack Usability Inspection Methods. New York: John Wiley and Sons, 1994. 407 p.

16. MSDN Multiple Document Interface [HTML] (http://msdn.microsoft.com/en-us/library/ms632591%28v=vs.85%29.aspx).

17. Microsoft Visual Studio, MSDN [HTML] (http://msdn.microsoft.com/ru-ru/vstudio/default.aspx).

18. Maxime Caron Survey on Usability of Integrated Development Environment [HTML] (http://docs.google.com/present/view?id=addqfjnjc3d6_108gj3w67c3).

19. Janel Garvin Software Development Platforms - 2009 Rankings, Evans Data Corporation [HTML] (http://www.evansdata.com/reports/viewRelease_download.php?reportID=19).

20. NetBeans Community NetBeans Official Website [HTML] (http://www.netbeans.org).

21. Eclipse Foundation Eclipse Website [HTML] (http://www.eclipse.org).

22. Code::Blocks Website [HTML] (http://www.codeblocks.org).

23. Mono Project MonoDevelop Website [HTML] (http://monodevelop.com).

24. KDevelop team KDevelop Website [HTML] (http://www.kdevelop.org).

25. IntelliJ IDEA Website [HTML] (http://www.jetbrains.com/idea).

26. Bonnie Lida, Spring Hull, Katie Pilcher Breadcrumb Navigation: An Exploratory Study of Usage // Usability News. 2005. 5. №1. [HTML] (http://www.surl.org/usabilitynews/51/breadcrumb.asp).

27. IBM C++ Builder Website [HTML] .

28. Apple XCode Website [HTML] (http://developer.apple.com/xcode).

29. Mary Beth Rosson, John M. Carroll Usability Engineering: Scenario-Based Development of Human-Computer Interaction. San Diego, USA: Morgan Kauffman, 2002. 448 p.

30. Janice Singer, Timothy C. Lethbridge, et al. An Examination of Software Engineering Work Practices // Proceedings of the 1997 conference of the Centre for Advanced Studies on Collaborative research. Toronto, Canada: CASCON, 1997. P. 21.

31. Thomas D. Latoza Maintaining Mental Models: A Study of Developer Work Habits // Proceedings of the 28th international conference on Software engineering. New York, USA: ACM, 2006. P. 492-501.

32. Ko, A.J., Myers, B.A., Coblenz, M.J., Aung, H.H. An Exploratory Study of How Developers Seek, Relate, and Collect Relevant Information during Software Maintenance Tasks // Journal IEEE Transactions on Software Engineering. 2006. 32. N 12. P. 971-987.

33. Martin G. Helander, Thomas K. Landauer, Prasad V. Prabhu Handbook of Human-Computer Interaction. Amsterdam, Netherlands: Elsevier Science, 1997. 1396 p. .

34. Jakob Nielsen Ten Usability Heuristics [HTML] (http://www.useit.com/papers/heuristic/heuristic_list.html).

35. Donald A. Norman The Design of Everyday Things. New York, USA: Doubleday, 1989. 261 p.

36. Бланшет Ж, Саммерфилд М. QT 4: программирование GUI на C++. Москва, РФ: Кудиц-Пресс, 2007. 628 стр.

37. John Deacon Model-View-Controller Architecture [PDF] // Software Development Training Courses in the UK and Europe .

38. The Kate Team Kate Editor Official Website [HTML] (http://kate-editor.org).

39. Philippe Fremy KDE Technology: KParts Components [HTML] (http://phil.freehackers.org/kde/kpart-techno/kpart-techno.html) .

40. Grady Booch Object-Oriented Analysis and Design with Applications. Santa Clara, USA: Addison-Wesley, 2007. 608 p.

41. Joseph S. Dumas, Janice C. Reddish A Practical Guide to Usability Testing. London, UK: Intellect Ltd, 1999. 404 p.

42. Jakob Nielsen Quantitative Studies: How Many Users to Test? [HTML] (http://www.useit.com/alertbox/quantitative_testing.html).

43. Jakob Nielsen Why You Only Need to Test with 5 Users [HTML] (http://www.useit.com/alertbox/20000319.html).

44. Donald Knuth Art of Computer Programming Volume 3: Sorting and Searching. New York, USA: Addison-Wesley, 1998. 802 p.