Московский государственный университет имени М.В. Ломоносова
Факультет вычислительной математики и кибернетики
Кафедра автоматизации систем вычислительных комплексов
Лаборатория вычислительных комплексов

Курсовая работа на тему:

# «Однооконный интерфейс среды разработки программ»

Студент 422 группы
Ручкин И.Д.

Научный руководитель
Прус В.В.

Москва 2010

# Аннотация

В данной работе описывается подход к решению проблем сложности графического интерфейса пользователя для интегрированных сред разработки программ, заключающийся в создании однооконного интерфейса. Этот подход предполагает удаление инструментальных окон – дополнительных окон среды разработки – и перенесение их функциональности в окно текстового редактора. Создание однооконного интерфейса включает обзор существующих интегрированных сред разработки программ, построение проекта однооконного интерфейса и частичную реализацию построенного проекта.

Обзор сред разработки описывает организацию инструментальных окон и сами инструментальные окна популярных сред разработки программ. Результатом обзора является модель инструментальных окон, описывающая классы инструментальных окон, обобщенные инструментальные окна и их функциональность. За счет этой модели дальнейшие рассуждения абстрагируются от конкретных сред разработки программ.

Далее в рамках полученной модели инструментальных окон создается проект однооконного интерфейса. Для этого рассматривается каждое обобщенное инструментальное окно и предлагается способ замены его функциональности. Предложенный проект однооконного интерфейса частично реализуется на базе интегрированной среды разработки KDevelop 4.

# Содержание

# 1. Введение

Существует большое число программных средств, используемых разработчиками для написания программного кода. Эти программы могут быть как простыми текстовыми редакторами, предлагающими только базовую подсветку текста (к примеру, Notepad и Kate), так и сложными средами разработки программ (*IDE* – *Integrated Development Environment*) [1], как Eclipse и Microsoft Visual Studio. Такие среды разработки объединяют различные инструменты, используемые в разработке, например отладчик и компилятор, для предоставления пользователю особых возможностей. Например, в IDE может присутствовать автоматическое дополнение кода, проверка синтаксической правильности программы без компиляции, возможность задавать точки останова в коде и так далее [1][2].

Несмотря на эти привлекательные возможности, многие разработчики считают IDE переусложненными и предпочитают пользоваться простыми текстовыми редакторами и дискретными инструментами разработки [2][3]. Причины этого могут быть самыми различными, и, чтобы определить их, требуется отдельное исследование удобства (*usability*) IDE [4][5][6]. Тем не менее, эмпирический опыт показывает, что одной из основных проблем являются инструментальные окна – дополнительные окна среды разработки, отличные от окна редактирования кода [7][8]. Как правило, дополнительные окна постоянно используются в процессе работы, поэтому пользователю приходится либо постоянно держать такие окна открытыми (уменьшая размер экранного пространства, доступного для редактирования кода), либо постоянно активировать инструментальные окна вручную.

Как возможное решение данной проблемы нами предлагается не содержащий инструментальных окон интерфейс среды разработки. Такой интерфейс далее будем называть однооконным. Прежде всего, требуется разработать однооконный интерфейс. После этого необходимо произвести практическую оценку удобства интерфейса [6]. Возможно, что в результате оценки будет выявлено, что часть инструментальных окон не может быть удалена без ухудшения эргономики[1]

[1] Под эргономикой мы будем понимать удобство (*usability*) графического интерфейса [4][5]

интерфейса IDE. Однако мы полагаем, что подавляющее большинство инструментальных окон может быть удалено.

Целью данной работы является собственно разработка однооконного интерфейса. Для проектирования этого интерфейса [5][9] мы будем последовательно убирать инструментальные окна из интерфейса, перенося их функциональность в окно текстового редактора. Очевидным примером удаляемого инструментального окна может служить окно результатов сборки, показывающее ошибки и предупреждения сборки проекта. Для пользователя осмысленно видеть только первую ошибку, так что отображение списка ошибок мы можем заменить показом первой ошибки в тексте программы и возможностью перехода к другим ошибкам [10].

Для получения однооконного интерфейса требуется, во-первых, отделить рассуждения от конкретных сред разработки, описав модель [11] инструментальных окон среды разработки на основе обзора наиболее популярных IDE [3][12][13]. Во-вторых, на основе полученной модели необходимо создать проект однооконного интерфейса, в котором возможности инструментальных окон доступны через окно текстового редактора. И, наконец, требуется реализовать полученный проект однооконного интерфейса.

# 2. Цель работы и постановка задачи

Данный раздел описывает цель данной работы и решаемую задачу.

## 2.1 Цель работы

Целью данной работы является создание проекта однооконного графического интерфейса IDE и его частичная реализация.

## 2.2 Постановка задачи

Задачи данной работы:

1. обзор существующих сред разработки программ с точки зрения пользовательской работы с инструментальными окнами в них,
2. создание проекта однооконного интерфейса сред разработки путем переноса функций инструментальных окон в окно текстового редактора,
3. практическая реализация проекта интерфейса на базе KDevelop 4, включающая перенесение функциональности окна навигации по проектам и файлам, окна ошибок и окна стека вызовов.

# 3. Обзор сред разработки программ

В обзоре мы рассмотрим интерфейс нескольких наиболее популярных IDE [3][12][13].

Во избежание обзора вообще всех сред разработки, введем следующие ограничения:

- Рассматриваются только среды для разработки программ на компилируемых объектно-ориентированных языках (а именно C++, C#, Java).
- В случае если среда разработки предоставляет возможность подключения дополнительных модулей, рассматривается только имеющийся по умолчанию набор модулей.

## 3.1 Цель обзора

Целью обзора является построение модели инструментальных окон в средах разработки программ. Данная модель описывает, во-первых, обобщенную организацию инструментальных окон и, во-вторых, набор обобщенных по назначению и использованию инструментальных окон и их функций.

## 3.2 Предмет обзора

Для каждой рассматриваемой среды разработки опишем:

- Организацию работы пользователя с дополнительными окнами: возможность их скрытия, автоматического скрытия и настройки их расположения,
- Набор основных инструментальных окон, используемых в этой IDE, и их основных функций.

## 3.3 Среды разработки программ

Перед тем как перейти к рассмотрению конкретных IDE, кратко опишем основные составляющие их графического интерфейса:

- Главное меню программы;
- Настраиваемая панель команд с иконками (и, возможно, надписями) наиболее часто используемых команд главного меню;
- Окно текстового редактора, в котором происходит написание программного кода. Слева от этого окна, как правило, расположена вертикальная полоса для установки точек останова и закладок;

- Инструментальные окна, располагающиеся по сторонам окна текстового редактора;

- Строка состояния – горизонтальная полоса размером в одну строку текста, расположенная снизу окна среды разработки и отображающая, как правило, информацию о прогрессе текущей операции и текстовые подсказки.

Теперь рассмотрим конкретные среды разработки программ.

### *3.3.1 Visual Studio*

Visual Studio [14] – 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.

Каждое инструментальное окно может находиться в одном из 4 режимов:

- Режим дрейфа (*floating*): инструментальное окно не связано с позициями в окне IDE, является отдельным окном в смысле окон операционной системы;

- Режим стыковки (*dockable*): инструментальное окно находится в одной из описанных выше позиций;

- Режим вкладки (*tabbed*): инструментальное окно перемещается во вкладки окна текстового редактора, отображается на месте редактируемого документа;

- Режим автоматического скрытия (auto hide): инструментальное окно сворачивается до кнопки, когда курсор мыши уходит за его пределы, и разворачивается, когда курсор наводится на кнопку.

*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[1] 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”,

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

“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 [15] – 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. Отражает структуру каталогов, распределяя файлы по категориям, каждая из которых соответствует каталогу ФС. Использование этого окна заключается в обзоре расположения файлов и открытии новых файлов.
- *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 объектов[1] 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.

[1] Под *объектом кода* понимается некоторый идентификатор; например, имя класса, функции или переменной.

- *Variables*. Это окно содержит список переменных, наблюдаемых 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 [16] – 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.

*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 [17] – 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 [18] – 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 [19] – независимая от платформы свободная IDE для графической оболочки KDE, написанная на 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, еѐ 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 объекта 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) [20] – 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.

*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 [21] – 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.

## 3.4 Результаты обзора

Итак, в обзоре установлены следующие факты:

- Организация инструментальных окон очень схожа в различных IDE и заключается в наличии окна текстового редактора и настраиваемых инструментальных окон, каждое из которых может находиться в нескольких режима (подробное описание режимов находится в обзоре Visual Studio).

- Рассмотренные инструментальные окна можно разделить по структуре и схеме использования на несколько больших классов.

- Рассмотренные инструментальные окна можно обобщить по функциональности, получив, таким образом, обобщенные инструментальные окна, каждое из которых относится к определенному классу.

На основе рассмотренных сред разработки программ составим модель инструментальных окон.

## 3.5 Модель инструментальных окон

Опишем сначала классы обобщенных инструментальных окон, входящих в модель, а затем сами обобщенные инструментальные окна. Классы инструментальных окон определяют визуальную структуру содержимого окна и основные его функции, а обобщенные инструментальные окна конкретизируют сущности, описываемые окном, и его возможности.

### *3.5.1 Классы инструментальных окон*

1. Окна-деревья навигации по сущностям. Такие окна отображают дерево сущностей и, возможно, дополнительные подокна для просмотра атрибутов или потомков выбранного элемента дерева. Важным критерием окон-деревьев является возможный неограниченный рост глубины дерева в процессе использования IDE. Окна-деревья предоставляют пользователю следующие функции:

   - Просмотр пользователем дерева сущностей.
   - Вызов контекстного меню для произвольной сущности дерева и совершение некоторых действий.
   - Возможно, переход к объекту исходного текста (файлу, части файла, строке), связанному с выбранной сущностью посредством двойного щелчка мыши.

К этому классу относятся окна древовидной навигации по проектам и файлам, по объектам кода, целям Make и другие похожие окна.

2. Окна-списки сущностей, связанных со строками кода. Для данных окон существенна связь один к одному со строками кода разрабатываемой программы. Окно-список, в общем случае, может отображать дерево, каждый элемент которого связан со строкой кода, и, что не менее важно, глубина которого не более 2 (для глубины 2 мы получаем де-факто категоризированный список). Функции окон-списков, используемые разработчиком:

   - Просмотр списка сущностей и их атрибутов.
   - Переход к строкам кода, связанным с сущностями.
   - Возможно, редактирование атрибутов сущностей.

К таким окнам можно отнести списки задач, точек останова, списки ошибок и предупреждений сборки и другие подобные окна.

3. Окна – документы. К данному типу отнесем окна, не являющиеся окнами первых двух типов. Функции окон данного типа, предлагаемые пользователю:

   - Как правило, просмотр простого текстового вывода внешних инструментов с примитивными возможностями копирования текста.
   - Возможно, ввод текстовых пользовательских команд.
   - Некоторая функциональность просмотра и редактирования, не связанная со строками кода (например, для окон-списков, сущности которых не имеют привязки к строкам кода).

К этому типу окон следует отнести окна вывода инструментов сборки, вывода отладчика и другие.

### *3.5.2 Обобщенные инструментальные окна*

Обобщенные инструментальные окна мы получим, выделяя назначение инструментальных окон и сущности, с которыми они оперируют (например, ошибки, файлы или задания). Основные функции обобщенного окна определяются его классом; все специфичные для конкретного окна функции мы будем указывать отдельно.

Обобщенные инструментальные окна-деревья описаны в таблице 1.

**Таблица 1. Обобщенные окна-деревья**

| **Название окна** | **Сущность, представляемая листом дерева** | **Возможные элементы дерева помимо листов** | **Окна рассмотренных IDE, являющиеся прототипами** |
|---|---|---|---|
| Окно навигации по проектам | Документ с возможностью редактирования в IDE | Группы документов. Например, исходные коды, заголовочные файлы и др. | Solution Explorer (Visual Studio), Projects (Eclipse, NetBeans, …), … |
| Окно навигации по файловой системе | Файл | Каталоги файловой системы | Files (NetBeans, MonoDevelop), Filesystem (KDevelop) |
| Окно навигации по объектам кода | Метод класса, атрибут класса, макрос, глобальная функция, глобальная переменная | Класс, категория «Глобальные переменные», категория «Глобальные функции», категория «Макросы» | Class View (Visual Studio), Classes (Eclipse), Structure (IntelliJ IDEA), … |

Окно навигации по проектам отображает на верхнем уровне список открытых в IDE проектов, в каждый из которых вложенные некоторые группы документов. Конкретная группировка документов в рамках данной работы нас интересовать не будет.

Обобщенные инструментальные окна-списки сущностей описаны в таблице 2.

**Таблица 2. Обобщенные окна-списки**

| **Название окна** | **Сущность, представляемая элементом списка** | **Строка, связанная с сущностью** | **Атрибуты сущности** | **Окна рассмотренных IDE, являющиеся прототипами** |
|---|---|---|---|---|
| Окно задач | Задача, взятая из комментария в коде программы | Строка комментария со словом TODO, FIXME, HACK, … | Имя файла, номер строки, описание задачи, активна/неактивна | To-Do list (Code::Blocks), Task List (Visual Stuido), … |
| Окно ошибок | Ошибка, предупреждение или | Строка кода, породившая | Имя файла, номер строки, текст, тип | Messages (IntelliJ IDEA, KDevelop), |

| | | | | |
|---|---|---|---|---|
| | сообщение сборки | сущность | (ошибка, предупреждение, сообщение) | Problems (Eclipse) |
| Окно точек останова | Точка останова | Строка, в которой установлена точка останова | Имя файла, номер строки, условие останова, описание, активна/неактивна | Breakpoint List (C++ Builder), Breakpoints (Eclipse, KDevelop), … |
| Окно нитей | Нить отлаживаемого приложения | Текущая точка управления данной нити | Номер нити, имя файла, номер строки | Threads (Visual Studio, C++ Builder) |
| Окно стека | Вызов функции в стеке отлаживаемой нити | Строка-начало описания функции | Имя функции | Call Stack (KDevelop, NetBeans, Visual Studio) |

Также для некоторых обобщенных окон-списков специфичны следующие функции:

- Окно ошибок автоматически активируется при появлении хотя бы одной ошибки/предупреждения/сообщения сборки;
- Окно точек останова отображается автоматически при попадании одной из нитей отлаживаемого приложения на точку останова;
- Окно переменных отображается автоматически при запуске отладки приложения
- В окне стека отображается стек вызовов для нити, выбранной в окне нитей. Оба эти окна используются только при отладке.

Обобщенные инструментальные окна-документы описаны в таблице 3.

**Таблица 3. Обобщенные окна-документы**

| **Имя окна** | **Содержимое окна** | **Окна рассмотренных IDE, являющиеся прототипами** |
|---|---|---|
| Окно вывода системы сборки | Простой текстовый результат работы инструмента, осуществляющего сборку разрабатываемого приложения | Build log (Code::Blocks), Output View (Visual Studio), Build Output (MonoDevelop),… |
| Окно вывода отладчика | результат работы отладчика; позволяет вводить команды для выполнения отладчиком | Debug (IntelliJ IDEA, Eclipse), … |
| Окно просмотра кода | Отображает часть кода, связанную с описанием объекта кода, выбранного в окне текстового редактора. Например, отображает определение класса, на имени которого находится курсор текстового редактора. Более | Code Definition Window (Visual Studio), Code Browser (KDevelop) |

| | | |
|---|---|---|
| | точное описание отображаемого этим окном текста не требуется | |
| Окно свойств | Отображает редактируемый список свойств объектов, выбранных в других инструментальных окнах. Данное окно, хоть и формально отображает список, не имеет привязки к строкам кода, поэтому является окном-документом. | Properties (Visual Studio, NetBeans, Eclipse), … |
| Окно наблюдаемых выражений и локальных переменных | Отображает список наблюдаемых выражений, введенных пользователем, и локальных по отношению к положению курсора переменных. По причине отсутствия прямой связи со строками кода является окном-документом. | Watches (Code::Blocks, IntelliJ IDEA), Watch (MonoDevelop), Variables (KDevelop, Eclipse, NetBeans) |

Также для некоторых обобщенных окон-документов специфичны следующие функции:

- Окно вывода системы сборки отображается автоматически при начале сборки проекта;
- Окно вывода отладчика отображается автоматически при начале отладки приложения.
- Окно наблюдаемых выражений автоматически отображается при попадании исполнения на точку останова.

Итак, используемая далее в этой работе модель инструментальных окон состоит из перечисленных обобщенных окон, разбитых на 3 класса. Окном текстового редактора будем считать текстовую область с вкладками открытых документов, которую можно разбить на две вертикально или горизонтально, затем можно разбить еще раз и т.д. Инструментальные окна могут быть расположены с любой из четырех сторон от окна текстового редактора и находиться в одном из четырех режимов, описанных выше в обзоре среды Visual Studio.

# 4. Проект однооконного интерфейса

В данной главе создается проект однооконного интерфейса для среды разработки программ. Рассуждения будут происходить на основе модели инструментальных окон, построенной в предыдущей секции.

## 4.1 Замена функциональности инструментальных окон

Опишем, какие способы могут быть использованы для замены функциональности инструментальных окон. Поскольку функциональность некоторых окон (например, навигация по файлам) невозможно заменить, используя лишь окно текстового редактора, мы будем при необходимости вводить новые или использовать существующие в средах разработки визуальные элементы. Принципиальным отличием этих элементов от инструментальных окон является отсутствие необходимости постоянно изменять размер, открывать и закрывать эти элементы.

При удалении инструментального окна любого типа становится невозможным уведомление пользователя о событиях путем открытия этого окна и, возможно, выделения там некоторого элемента. Также пользователю необходим доступ к настройкам отображения перенесенной из инструментальных окон информации. Для этих целей будет введена динамическая строка состояния, подробно описанная ниже.

Способы замены инструментальных окон зависят в большей степени от структуры и схемы использования окон, так что опишем отдельно способы для каждого типа инструментальных окон, полученного при построении модели инструментальных окон.

### *4.1.1 Окна-деревья*

Напомним, окна-деревья предоставляют пользователю возможность осматривать дерево некоторых сущностей, совершать с ними действия посредством контекстного меню и двойного щелчка мыши. Конкретные действия по замене каждого обобщенного окна-дерева опишем в разделе 4.3, а здесь перечислим приемы, которыми можно частично заменить функциональность такого типа окон.

- Замена отображения дерева может осуществляться распространенным механизмом навигации «хлебные крошки» (breadcrumbs) [22]. Об этом механизме будет подробнее рассказано ниже.

- Уведомление пользователя о событиях и предоставление ему возможности настройки отображения при помощи строки состояния

### *4.1.2 Окна-списки*

Окна-списки сущностей, как было сказано выше, отображают список сущностей, каждая из которых связана со строкой исходного кода разрабатываемой

программы. Следующие способы могут быть использованы для замены окон-списков:

- Использование навигации breadcrumbs для отображения всех элементов списка или только текущего элемента.
- Использование навигации по другим сущностям в breadcrumbs для отображения меток, связанных с сущностями окна-списка. Пример: отображение меток точек останова около файлов, которые их содержат, в режиме навигации по файлам в breadcrumbs.
- Метки внутри окна текстового редактора слева или справа от связанной сущностью строки текста.
- Вставка визуальных элементов между или внутри строк программы. Эти визуальные элементы могут отображать атрибуты сущности, расположенной в соседней строке, и кнопки управления. Такие визуальные элементы мы будем называть внутритекстовыми.
- Подчеркивание слов в строке или целых строк; всплывающие подсказки с информацией о строке, связанной с сущностью некоторого окна, при наведении на строку.
- Перенос уведомлений и управления отображением в строку состояния среды разработки.

### *4.1.3 Окна-документы*

Окна-документы отображают некоторый текст и, возможно, позволяют пользователю вводить свои команды. Такие окна не допускают осмысленной привязки к строкам кода и не имеют фиксированной структуры, а действия над ними напоминают работу с обычными текстовыми файлами (просмотр, ввод данных). Поэтому окна-документы мы будем переносить во вкладки текстового редактора, рассматривая их как обычные документы. Это позволяет нам удалить их как инструментальные окна.

## 4.2 Дополнительные визуальные элементы

Данная часть работы содержит описание дополнительных визуальных элементов, используемых при создании проекта однооконного интерфейса.

### *4.2.1 Навигационный механизм breadcrumbs*

Данный механизм [22] в общем случае отображает путь в некотором дереве сущностей до текущей просматриваемой сущности. Визуально breadcrumbs представляет собой горизонтальную полоску визуальных элементов, расположенных в порядке от корня дерева (слева) к текущей сущности (справа). Каждый визуальный элемент этой полоски представляет собой уровень в дереве. Так, например, при навигации по файловой системе в POSIX-системах первым

элементом будет «/», а последним – имя текущего открытого файла. Элементы пути интерактивны и позволяют при взаимодействии либо перейти к непосредственно к ним самим, либо просмотреть вложенные в них элементы и перейти к одному из них. Во втором случае при нажатии на один из элементов пути всплывает окно с отображением списка одноуровневых по отношению к выбранному элементу пути объектов. В этом дереве есть возможность добавлять метки для каждого элемента или выделять их цветом (например, для индикации того, что в файле есть хотя бы одна ошибка).

Преимущества данного навигационного механизма по сравнению с древовидной навигацией:

- Отображение текущей выбранной сущности.
- Соответствие принципу локальности [5]: пользователь чаще хочет переходить к объектам, менее удаленным от текущего выбранного объекта.
- Возможна оптимизация выбора элемента: делать более видимыми и доступными [23] те элементы, которые чаще выбираются пользователем (здесь может быть также любой другой алгоритм оптимизации).

Поскольку данный механизм навигации будет использоваться для переноса нескольких окон, то и режимов навигации может быть несколько. Для смены режимов и отображения текущего будет использоваться специальная кнопка слева от собственно навигационного пути.

### *4.2.2 Внутритекстовые визуальные элементы*

Сущности, отображаемые инструментальными окнами-списками, связаны со строками кода программы. С учетом этого факта, удаление таких инструментальных окон требует механизма отображения информации, связанной со строками кода. В качестве такого механизма будем использовать вставку графических элементов между и внутри строк текста в окне текстового редактора.

Вставка визуального элемента может происходить двумя способами: либо строки текста раздвигаются вертикально, и между ними вставляется графический элемент высотой в несколько строк текста, либо символы строки раздвигаются горизонтально, и между ними вставляется графический элемент высотой в одну строку текста. Вставляемый элемент может содержать текстовую, возможно редактируемую информацию, а также кнопки для управления отображением и переходами к другому коду.

### *4.2.3 Строка состояния*

Строка состояния используется в современных средах разработки программ для отображения информации об их состоянии. В рамках данной работы ее роль будет расширена в силу того, что требуется кратко отображать информацию из удаляемых инструментальных окон и информировать пользователя о событиях в этих окнах. Во-первых, требуется обеспечить управление отображением информации в окне текстового редактора и дополнительных визуальных

элементах. Во-вторых, требуется отображение уведомляющей информации, ранее отображавшейся в инструментальных окна.

Для выполнения обеих требований разделим строку состояния на две части: статическую и динамическую. Первая часть будет содержать графические элементы, кратко отображающие статическую информацию и служащие для настройки отображения. Например, статический элемент «Точки останова» может показывать число точек останова и позволять пользователю включать и отключать отображение точек останова во внутритекстовых элементах. Вторая часть будет содержать оповещения о событиях IDE.

## 4.3 Перенесение функциональности инструментальных окон

Данный подраздел работы описывает способы замены функциональности каждого обобщенного инструментального окна, входящего в рассматриваемую модель инструментальных окон, окном текстового редактора и дополнительными визуальными элементами.

### *4.3.1 Окно навигации по проектам*

Данное окно предоставляет пользователю типичный для окна-дерева сервис: просмотр структуры документов, открытие и другие операции над выбранным документом. Как уже было сказано в подразделе 4.2.1, такой сервис может быть предоставлен при помощи навигационной полоски breadcrumbs, которую и будем использовать для замены данного окна. Самым левым элементом пути будет имя проекта, по нажатию на которое пользователь получает список всех проектов, открытых в IDE. Самый правый элемент – текущий файл во вкладках текстового редактора.

Функции просмотра структуры и выбора элемента доступны из breadcrumbs, а дополнительные операции над документами доступны через контекстное меню элементов, так же, как и в окне древовидной навигации по проектам.

### *4.3.2 Окно навигации по файловой системе*

Окно навигации по файлам заменяется также с помощью breadcrumbs. Для этого вводится дополнительный режим навигации, в котором самым левым элементов является корень файловой системы (или имя логического диска для Windows-систем). Самым правым элементом так же будет имя открытого файла.

### *4.3.3 Окно навигации по объектам кода*

Данное окно предоставляет возможность навигации по классам, макросам глобальным переменным и функциям проекта. Перенесем данную функциональность в breadcrumbs, добавив новый режим навигации. Навигация будет осуществляться по дереву, соответствующему обобщенному окну навигации по объектам кода: первым элементом будет проект, затем категория, затем собственно текущий элемент.

### *4.3.4 Окно задач*

Редактирование задач осуществляется пользователем в окне текстового редактора. Требуется организовать просмотр структуры заданий без помощи инструментального окна. Для этого произведем следующие изменения в проекте интерфейсе IDE:

- Добавим в breadcrumbs по проектам, файлам и объектам кода метки задач, показывающие наличие задач в файле или в файлах некоторого каталога или проекта или объекте кода, который может содержать комментарии (класс, метод класса или глобальная функция).
- Добавим внутритекстовый графический элемент шириной в несколько символов, появляющийся после ключевого слова (“FIXME”, “TODO”, …) каждого задания внутри кода программы, с кнопками, позволяющими перейти к следующему и предыдущему заданию.
- Добавим в строку состояния статический элемент «Количество заданий», отображающий число активных заданий и общее число заданий. По щелчку на этот элемент включаются метки breadcrumbs и внутритекстовые вставки, осуществляется переход к первому заданию. Через контекстное меню этого элемента осуществляется включение/выключение меток в breadcrumbs и внутритекстовых вставок.

Таким образом, функциональность окна задач была заменена.

### *4.3.5 Окно ошибок*

Заменим окно ошибок аналогично окну задач:

- Добавим отображение сообщений сборки в качестве меток в навигацию по проектам, файлам и объектам кода.
- Добавим отображение меток слева от строк кода.
- После первой ошибки добавим внутритекстовый визуальный элемент с описанием ошибки и с кнопками перехода к следующей и предыдущей ошибке. Добавление лишь для первого элемента оправдано тем, что пользователь всегда смотрит на первую ошибку, поскольку последующие ошибки могут быть наведены первой.
- Для предупреждений и сообщений сборки предусмотрим возможность внутритекстовых визуальных элементов, как и для ошибок, однако отключим ее по умолчанию. Также в визуальный элемент добавим пункт постоянного игнорирования ошибки.
- Добавим статический элемент в строку состояния, отображающий число ошибок, предупреждений и сообщений сборки. Через этот статический

элемент пользователь может включать и отключать метки в breadcrumbs и внутритекстовый показ ошибок и предупреждений.

- Добавим динамическое сообщение о завершении сборки в динамическую часть строки состояния для замены активации инструментального окна ошибок при неуспешном завершении сборки.

### *4.3.6 Окно нитей и окно стека*

Окна нитей и стека используются для получения информации об отлаживаемой программе. Каждая нить имеет свой стек вызовов, так что выбор нити определяет рассматриваемый набор функций стека. Будем заменять функциональность этих двух окон одновременно следующим образом:

- Добавим в breadcrumbs режим навигации «Нити и стеки», используемый только при отладке. В этом режиме полоска навигации состоит из двух элементов: первый отображает имя нити, а второй – имя функции на вершине стека. По щелчку на каждом элементе можно получить доступ к списку нитей и вызовов соответственно. При выборе нити происходит переход к точке ее исполнения, при выборе вызова – к описанию соответствующей вызову функции.
- Добавим статический элемент в строку состояния, отображающий текущее количество нитей (количество вызовов менее осмысленно для пользователя).
- Добавим метки слева от строк кода, в которых находится управление каждой из нитей.

### *4.3.7 Окно точек останова*

Отметим, что во многих рассмотренных средах разработки программ добавление, активация/деактивация и снятие точек останова происходит через область слева от строк кода. В нашем однооконном интерфейсе организуем доступ аналогично. Далее, требуется обеспечить обзор точек останова, редактирование их атрибутов и переход по ним.

- Добавим в навигацию breadcrumbs по проектам, файлам и объектам кода метки точек останова, означающие, что отмеченный объект содержит точку останова.

- Добавим около каждой точки останова внутритекстовый элемент с редактируемым условием и описанием, а также кнопками перехода к следующей и предыдущей точке останова. Отображение внутритекстовых элементов может быть включено для всех точек останова или для тех точек, на которых остановилась одна из нитей.

- Добавим в строку состояния статический элемент, отображающий число активных и неактивных точек останова. Через этот статический элемент доступно включение и отключение меток в breadcrumbs и внутритекстовых

элементов, а также операции над всеми точками останова – их отключение и удаление

- Для замены активации окна точек останова при попадании одной из нитей на точку останова добавим динамическое сообщение в строку состояния, по нажатию на которое пользователь переходит к точке останова, вызвавшей сообщение.

### *4.3.8 Окно вывода системы сборки*

Окно вывода системы сборки, будучи окном-документом, в соответствии с подразделом 4.1.3, переносится во вкладки текстового редактора без изменений. Активация окна производится открытием соответствующей ему вкладки.

### *4.3.9 Окно вывода отладчика*

Окно вывода отладчика переносится аналогично окну вывода системы сборки.

### *4.3.10 Окно просмотра кода*

Окно просмотра кода переносится во вкладки текстового редактора, как и любое другое окно-документ. Однако использование окна кода возможно лишь при параллельно открытом окне текстового редактора, так что по умолчанию при открытии данного окна текстовый редактор разбивается пополам.

### *4.3.11 Окно свойств*

Данное окно отображает редактируемый список свойств объектов, выбранных в других инструментальных окнах. Поскольку другие инструментальные окна удаляются, то единственным разумным вариантом является перенесение окна свойств во вкладки текстового редактора и отображение в нем свойств объектов, выбираемых в breadcrumbs, внутритекстовых элементах, собственно тексте программы и строке состояния.

### *4.3.12 Окно наблюдаемых выражений и локальных переменных*

Данное окно отображает введенные пользователем выражения во время отладки, а также переменные, локальные по отношению к текущей позиции курсора. Перенесем данное окно во вкладки текстового редактора. При попадании исполнения на точку останова будем отделять его от текстового редактора для обеспечения параллельного просмотра кода, наблюдаемых выражений и значений локальных переменных.

# 5. Реализация

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

- 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 отображения ошибок и предупреждений сборки,
- Строку состояния, показывающую сообщения от удаленных инструментальных окон и статическую информацию о состоянии IDE.

## 5.1 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.

### *5.1.1 Уровень Sublime*

На рисунке 1 показаны части графического интерфейса уровня Sublime и классы, реализующие эти части интерфейса. Далее следует текстовое описание основных классов.

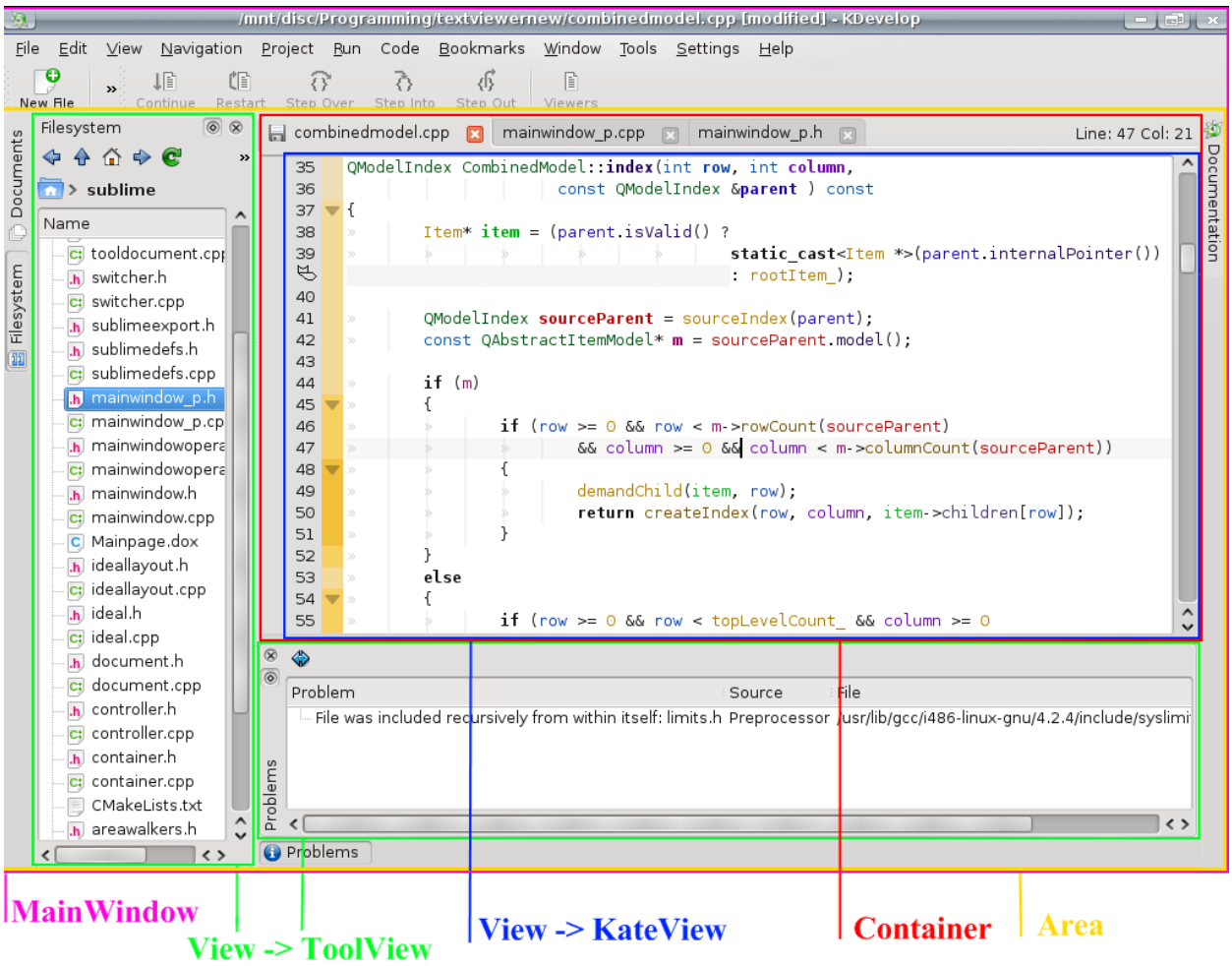


**Рисунок 1. Структура интерфейса Sublime**

- *Sublime::MainWindow* – основное окно интерфейса Sublime. Внутренняя логика реализуется классом Sublime::MainWindowPrivate.
- *Sublime::Area* – область, содержащая инструментальные окна и окно текстового редактора. Допускается несколько областей и их переключение, в каждый момент времени в главном окне отображается только одна. Каждой области соответствует своё расположение окон внутри MainWindow.
- *Sublime::View* – класс, определяющий некоторое окно (как инструментальное, так и текстовое) в одной из Sublime::Area.
- Инструментальные окна реализуются классом *Sublime::ToolView*, наследующим Sublime::View, а текстовое окно – классом KateView, также наследующим Sublime::View. При отображении документа его View ставится в однозначное соответствие QWidget, собственно отображающий редактор документа (класс-предок произвольного графического элемента в Qt) [15].
- *Sublime::Document* – класс документа Sublime. Каждому документу может соответствовать несколько View, предназначенных для его редактирования.
- *Sublime::Container* – графический элемент, отображающий KateView документов (текстовый редактор) с возможностью переключения между KateView с помощью вкладок.
- *Sublime::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.

### *5.1.2 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:

- *KDevelop::Core* – ядро KDevelop. Служит для доступа и управления различными контроллерами, хранит информацию о них.
- *KDevelop::DocumentController* реализует контроллер документов, который управляет всеми документами, содержит информацию о них, испускает сигналы об открытии/закрытии.
- *KDevelop::UiController* – контроллер интерфейса. Отвечает за создание интерфейса KDevelop и обработку его изменений.
- *KDevelop::RunController* – контроллер задач. Управляет запуском отладки, запуском скомпилированных программ и т.д.
- *KDevelop::PluginController* – контроллер подгружаемых модулей. Управляет загрузкой и работой этих модулей.
- *KDevelop::SessionController* – контроллер сессий, служащий для управления запущенными процессами.
- *KDevelop::LanguageController* – управление языками кода пользователя. Также осуществляет проверку синтаксиса кода в фоновом режиме.
- *KDevelop::MainWindow* описывает главное окно интерфейса KDevelop. Содержит Sublime::MainWindow в качестве члена класса.
- *KDevelop::OpenProjectDialog, Shell::SaveDialog* – диалоги открытия проекта, сохранения файла.
- *KDevelop::ProblemReporterPlugin* – подгружаемый модуль, реализующий фоновую проверку синтаксиса и предоставляющий окно для отображения результатов этой проверки.
- *KDevelop::BreakpointWidget* – класс, реализующий инструментальное окно точек останова.
- *KDevelop::FrameStackModel* – Qt-модель [25], хранящая данные о нитях и вызовах отлаживаемого приложения.

### *5.1.3 Уровень KDevelop*

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

При реализации были использованы следующие классы KDevelop:

- *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.

- *KDevelop::GdbFrameStackModel* – наследник Shell::FrameStackModel, содержащий специфичные для GDB данные о нитях и вызовах.

- *KDevelop::MakeBuilder* – подгружаемый модуль сборки, связанный с инструментом Make. Используется в реализации для получения данных об ошибках сборки.

- *KDevelop::MakeJob* – класс, хранящий информацию об ошибках сборки инструментом Make.

## 5.2 Механизм навигации

Эта часть работы описывает реализацию механизма навигации, абстрактный вид которого был описан в подразделе 4.2.1 данной работы. Реализация данного механизма основана на архитектуре Model/View, поэтому детальному описанию этой реализации предшествует краткий обзор архитектуры Model/View.

### *5.2.1 Архитектура Model/View*

Архитектура Model/View [26] – шаблон объектно-ориентированной архитектуры в разработке программного обеспечения, заключающийся в разделении данных приложения и его интерфейса с пользователем. Достоинства архитектуры Model/View:

- изоляция данных и их представления,

- независимость модификации данных и их представления.

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 [25]. В Qt модель – 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 – 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.

### *5.2.2 Механизм навигации: навигационная полоска*

*Навигационная полоска*. Графическим отображением навигационного механизма является полоска, состоящая из кнопок со стрелками, находящаяся между окном текстового редактора и вкладками документов. 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).

Классы, составляющие навигационную полоску:

- *Sublime::GenericNavigator* – собственно класс навигатора. Наследует QWidget. Для использования навигатора достаточно вставить этот визуальный элемент в интерфейс и предоставить ему движок навигации.
- *Sublime::GenericNavigatorButton* – навигационная кнопка. Наследует QPushButton. Отличается от последней прорисовкой: рисуется не просто текст, а текст со стрелкой вправо.
- *Sublime::ExtendedDialog* – диалог, появляющийся при нажатии на одну из навигационных кнопок. Использует ExtendedDialogViewDelegate.
- *Sublime::ExtendedDialogViewDelegate* – класс, предназначенный для специальной прорисовки элементов всплывающего диалога ExtendedDialog.
- *Sublime::INavigationEngine* – класс-интерфейс движка навигации. Определяет операции задания/получения навигационной модели и обработки выбора элемента модели.

### *5.2.3 Механизм навигации: модель навигации по файловой системе*

Как сказано ранее, на уровне Shell необходимо построить модель навигации по файловой системе для её передачи уровню Sublime. Для каждого открытого документа строится модель навигации, определяющаяся положением этого документа в файловой системе.

*Структура модели*. Опишем структуру модели на примере. Пусть мы имеем проект Project0, в нём файлы part0.h и part0.cpp находятся в подпапке src0, а

part1.h, part1.cpp – в src1. Пусть во вкладках редактора открыты файлы part0.cpp и part1.cpp. На рисунке 2 показана описанная структура каталогов. На рисунке 3 изображена навигационная модель файловой системы для документа part0.cpp.

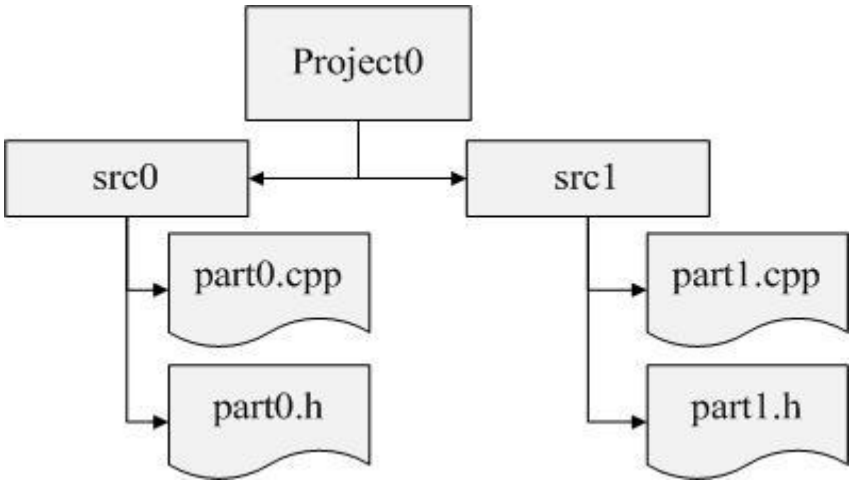


**Рисунок 2. Структура каталогов Project0**

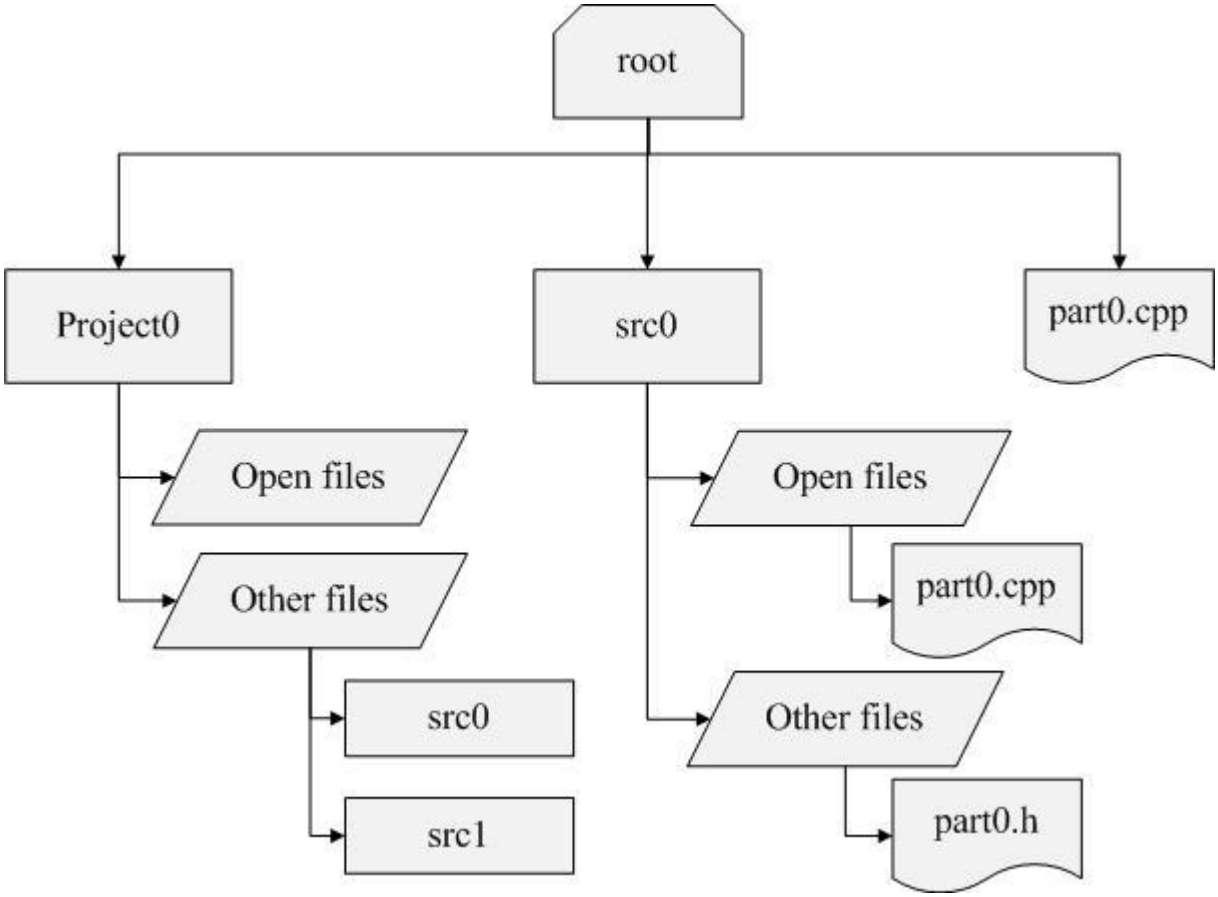


**Рисунок 3. Пример навигационной модели для part0.cpp**

Верхний уровень навигационной модели состоит из элементов, соответствующих частям пути (каталогам) до открытого документа. Рассмотрим некоторый элемент

верхнего уровня, отличный от последнего (он соответствует документу). Его потомками являются два элемента: «Open files» и «Other files». Потомки первого – файлы из каталога, определяемого элементом верхнего уровня, открытые в KDevelop. Потомки второго – остальные файлы, каталоги.

*Реализованные классы*. В ходе реализации было выяснено, что построение такой модели с нуля затруднительно, поэтому для удобства были реализованы следующие классы-модели:

- *KDevelop::HeaderModel*. Принимает на вход одну модель и одну строку. Добавляет в поданную на вход модель заголовочный элемент (он становится отцом всех элементов верхнего уровня) с заданной строкой.
- *KDevelop::CombinedModel*. Принимает на вход список моделей. Выходная модель состоит из соединения по верхнему уровню моделей-аргументов в одну.
- *KDevelop::FileFilterModel*, получая на вход модель, приводимую к модели файлов ФС (в Qt реализуется классом QDirModel), и некоторый путь, показывает на верхнем уровне лишь файлы, лежащие по полученному пути и находящиеся в списке открытых файлов KDevelop. Возможна параметризация класса для показа файлов, не находящихся в списке открытых файлов KDevelop.
- *KDevelop::ErrorDataModel* является моделью-фильтром (наследником QSortFilterProxyModel), помещаемым между навигационной моделью по файловой системе и собственно навигационной полоской. ErrorDataModel сообщает, какие каталоги и элементы следует отметить как содержащие ошибку сборки
- *KDevelop::FileNavigationEngine* – наследник INavigationEngine, содержащий модель навигации по ФС и обрабатывающий выбор элемента в этой модели.

Также как вспомогательная используется Qt-модель ФС *QDirModel*.

*Построение модели.* Определяется путь до документа. Для каждого элемента пути строится его модель: модель файловой системы служит фундаментом для двух различно параметризованных FileFilterModel, каждая из которых подается на вход HeaderModel (со строками «Open files» и «Other files» соответственно), после чего две HeaderModel объединяются с помощью CombinedModel, к которой добавляется заголовок (с помощью HeaderModel). Так, после построения модели для каждого элемента пути эти модели объединяются в одну (FileFilterModel). Полученная модель подается на вход фильтру ErrorDataModel, которая, при взаимодействии с ProblemReporter отмечает в модели элементы, соответствующие файлам с ошибками сборки. Так получается модель навигации по файловой системе, передаваемая на уровень Sublime.

*Навигационный движок*. Функционирование движка навигации по файловой системе заключается в открытии нового документа или переключении на уже открытый документ при выборе пользователем элемента-документа из модели.

Такая функциональность предоставляется при помощи обращения из кода навигационного движка к классу Shell::DocumentController.

Реализация механизма навигации по файловой системе показана на рисунке 4.

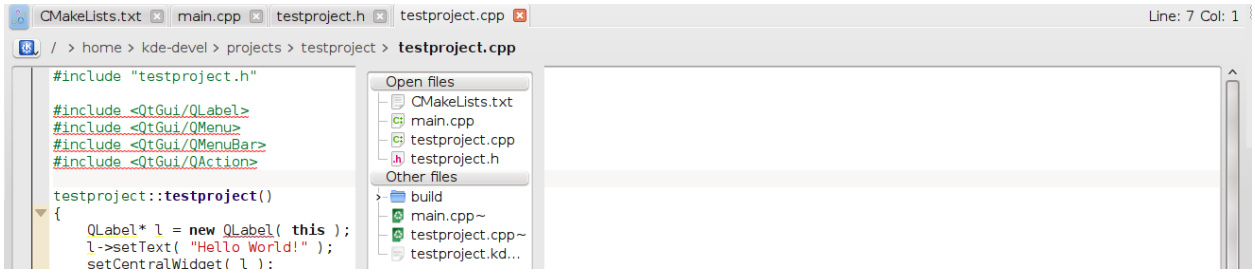


**Рисунок 4. Реализация механизма навигации по файловой системе**

### *5.2.4 Механизм навигации: модель навигации по вызовам функций*

На уровнях Shell и KDevelop также требуется построить модель навигации по вызовам функций. Как описано в главе 4, эта навигация будет использоваться при отладке для обзора состояния стека вызовов и перехода к коду различных функций.

*Структура модели.* На верхнем уровне модели будем отображать текущую нить исполнения и текущий вызов функции, а потомками этих двух элементов будет список всех нитей и всех вызовов в стеке соответственно. Для модельного набора нитей CurrentThread (текущая нить), Thread0, Thread1 и стека вызовов CurrentCall (текущая функция), Call0, Call1 модель навигации изображена на рисунке 5

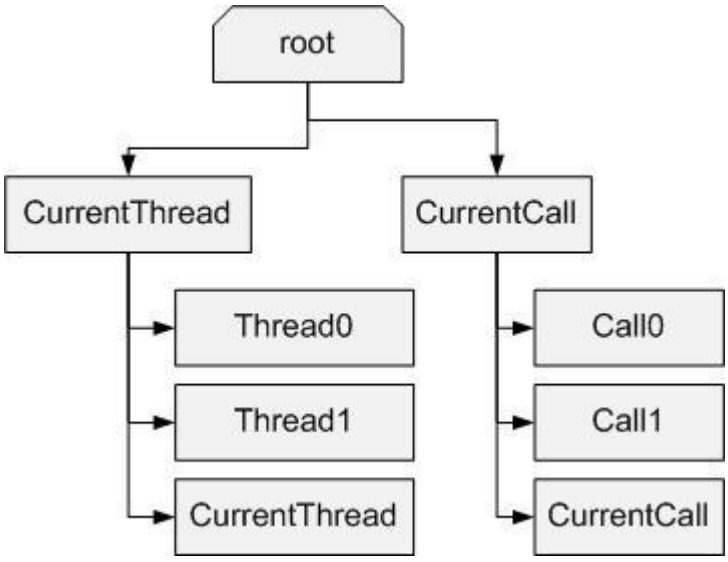


**Рисунок 5. Модель навигации по стеку вызовов и нитям отлаживаемой программы**

*Реализованные классы.* Для создания модели навигации по вызовам функций был создан класс KDevelop::FrameStackNavigationEngine, наследующий Sublime::INavigationEngine. Объект класса FrameStackNavigationEngine передается навигатору на уровень Sublime для организации навигации по вызовам функций.

*Построение модели.* Создание этой модели, в силу ее простоты относительно модели навигации по ФС, производилось при помощи наследования QStandardItemModel. Был описан класс-наследник FrameStackNavigationModel, реализующий необходимую функциональность и наполняемый элементами при каждом изменении модели в отладчике.

*Навигационный движок.* Навигационный движок FrameStackNavigationEngine для вызовов функций осуществляет переход к коду функции при выборе ее в навигаторе. Данная операция осуществляется в коде класса KDevelop::DebugController.

Реализация навигации по нитям и стеку вызовов отлаживаемой программы показана на рисунке 6.

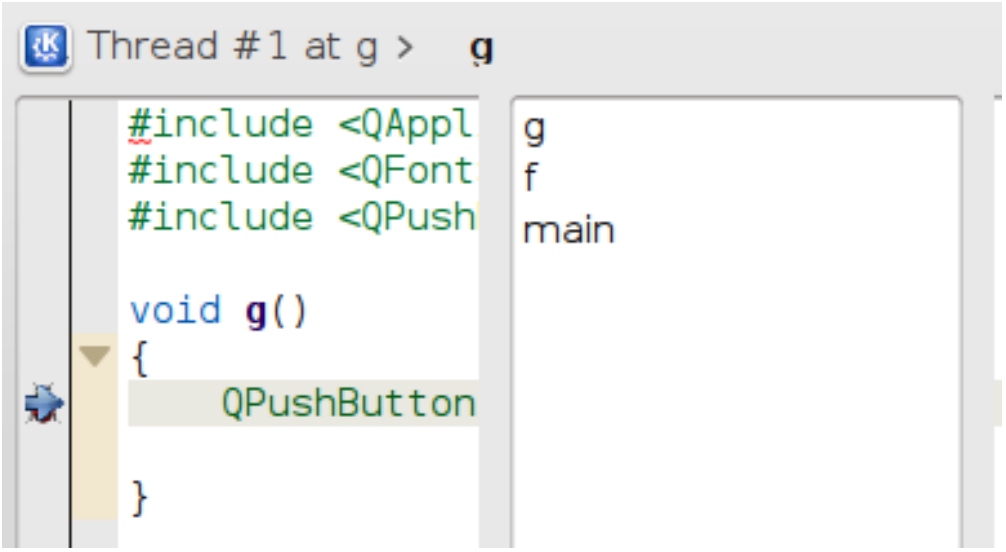


**Рисунок 6. Реализация навигации по нитям и стеку вызовов отлаживаемой программы**

## 5.3 Показ произвольных объектов в тексте программы

Ниже будет описана реализация механизма показа произвольных визуальных объектов между строк текстового редактора Kate и применение этого механизма для показа текста ошибок в тексте программы.

### *5.3.1 Архитектура Kate*

Kate – текстовый редактор среды KDE [27]. Окно редактирования текста Kate может быть использовано в другом приложении при помощи технологии KParts [28], так что возможна модификация кода окна редактирования Kate для показа произвольных визуальных элементов в IDE KDevelop.

Отображение окна текстового редактора осуществляется классом KateView, использующего для реализации внутренней логики класс KateViewInternal. Для представления отображаемого документа используются классы KateLineLayout, KateTextLayout, созданием, хранением и обновлением которых управляет класс KateLayoutCache. KateRenderer отвечает за рисование текста на низком уровне:

он может отобразить строку текста в указанной позиции окна. Подробное описание классов Kate:

- *KateView*. Реализует окно текстового редактора (вместе с прокруткой - скроллингом). Реализует интерфейс KTextEditor::View, необходимый для окон текстовых редакторов KDE. Одному KateView соответствует только один KTextEditor::Document (интерфейс документа), отображаемый в нем. Обращается к KateViewInternal для реализации редактирования текста.
- *KateViewInternal*. Реализует редактирование текста: от отображения на экране и прокрутки до ввода символов и перемещения текстового курсор, а также расчеты, необходимые для отображения текста. Используется только KateView, не реализует интерфейсов.
- *KateLineLayout*. Класс, представляющий одну строку текста в редактируемом документе, то есть при разделении текста на строки символами конца строки.
- *KateTextLayout*. Класс, представляющий одну строку текста, отображаемую на экране. Один KateLineLayout разбивается на 1 и более KateTextLayout.
- *KateLayoutCache*. Управляет разделением редактируемого документа на KateLineLayout и KateTextLayout и хранением этих классов. Осуществляет обновление строк при изменении в отображении по требованию KateViewInternal.
- *KateRenderer*. Выполняет работу по низкоуровневому отображению текста.

### *5.3.2 Механизм показа произвольных объектов в тексте*

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

Для передачи визуального элемента из кода уровня Sublime в код Kate ,был разработан интерфейс TextMessageInterface, реализуемый KateDocument. Данный интерфейс содержит метод вставки визуального элемента с указанием номера реальной строки документа (KateLineLayout), перед которой необходимо вставить визуальный элемент. Данный метод также получает указатель на фабрику-наследника [29] TextWidgetFactory, которая необходима для создания визуального элемента внутри кода Kate. Также интерфейс TextMessageInterface имеет метод удаления визуального элемента по соответствующей ему фабрике.

Для реализации интерфейса TextMessageInterface класс KateDocument передает данные всем объектам класса KateView, связанным с ним. KateView, в свою очередь, передает их KateViewInternal. Последний запоминает необходимые данные об отображении визуальных элементов и при перерисовке текстовой области отображает визуальные элементы, используя для их создания переданные фабрики. Описанная схема показа изображена на рисунке 7.

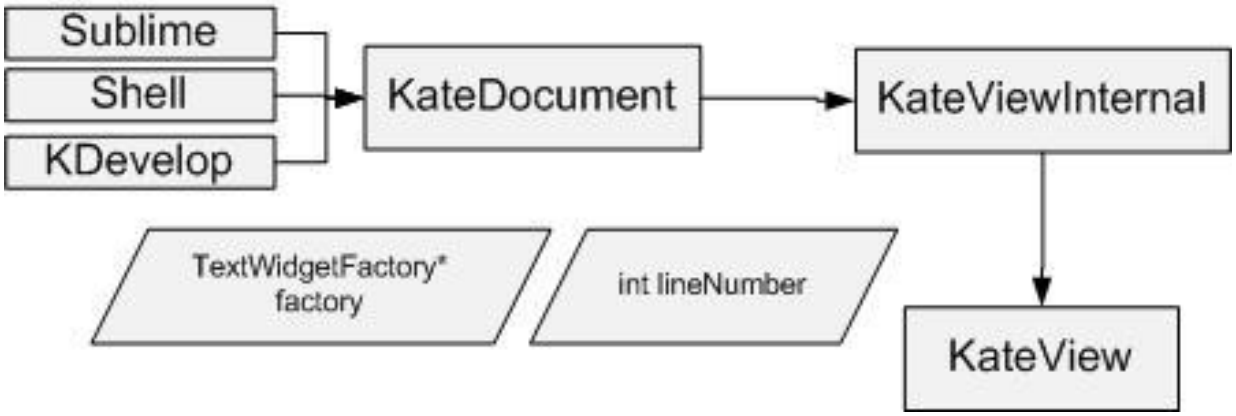


**Рисунок 7. Схема передачи произвольного объекта для показа в тексте**

### *5.3.3 Показ ошибок и предупреждений сборки в тексте программы*

Для реализации показа ошибок необходимо передать список ошибок от подключаемого модуля KDevelop MakeBuilder, расположенного на уровне KDevelop, на уровень Sublime, а затем, создав фабрику визуальных элементов с текстом ошибки, передать ее и номер строки, в которой возникла ошибка, классу KateDocument через описанный выше интерфейс TextMessageInterface. Дальнейшие решения по организации взаимодействия KDevelop и Kate во многом определены структурой этих приложений, не предназначенной для такой передачи информации: предполагается, что текстовое окно Kate вложено в KDevelop и, по замыслу разработчиков, не должно принимать информацию о внутренних структурах KDevelop [24], которой и является список ошибок сборки проекта.

Так, мы приходим к выводу, что некоторый класс из уровней Shell или Sublime должен управлять получением списка ошибок от MakeBuilder и формировать фабрику визуальных элементов для KateDocument. В качестве такого класса выберем класс ProblemReporterPlugin, расположенный в Shell. Этот класс реализует хранение списка ошибок синтаксической проверки правильности кода, производимой в фоновом режиме, и отображение результатов в инструментальном окне Problems. Это решение основывается, во-первых, на том, что ProblemReporterPlugin уже работает с ошибками и логично именно с помощью него организовать отображение ошибок в тексте, и, во-вторых, на уровне shell доступны необходимые для реализации интерфейсы.

Для ProblemReporterPlugin был описан интерфейс ProblemReporterInterface, допускающий метод добавления ошибки, через который MakeBuilder передает информацию об ошибках сборки, и метод удаления всех ошибок. Такой вид интерфейса предпочтительней, исходя из текущей структуры MakeBuilder, и необходимости показать первую ошибку после её появления, а не после завершения сборки, которое может занимать достаточно долгое время. ProblemReporterPlugin, реализуя указанный интерфейс, при получении первой ошибки создает визуальный элемент с текстом ошибки и, получая информацию о строке ошибки, передает визуальный элемент и номер строки KateView через

интерфейс TextMessageInterface. На рисунке 8 показана описанная схема работы механизма показа ошибок с помощью ProblemReporterPlugin.

Наконец, в данной работе был созданы классы TextErrorWidgetFactory TextErrorWidget, реализующие фабрику и собственно визуальный элемент с текстом ошибки и кнопками навигации «Следующая ошибка»/«Предыдущая ошибка». Переключением отображаемой ошибки также управляет ProblemReporterPlugin: по запросу, например, следующей ошибки он удаляет отображение старой ошибки, создает новую фабрику для следующей ошибки и передает ее KateDocument.

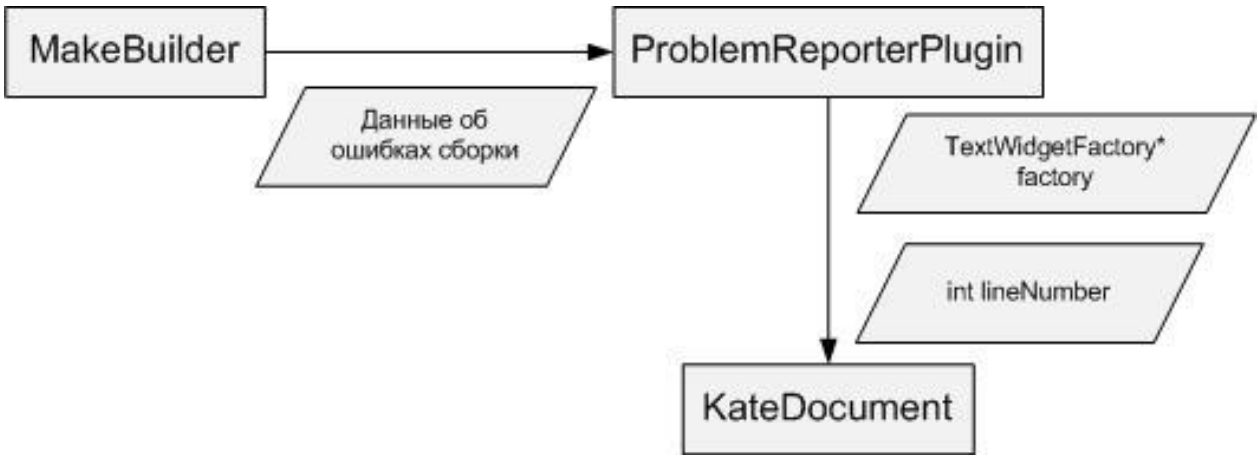


**Рисунок 8. Схема передачи данных об ошибках сборки**

## 5.4 Строка состояния

Строка состояния, необходимая для построения однооконного интерфейса, уже была описана в секции 4. Отдельно опишем архитектуру, реализацию отображения, а также реализацию динамической и статической частей строки состояния.

### *5.4.1 Строка состояния: архитектура*

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), а 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 визуальные элементы, отображающие

- число ошибок фоновой проверки синтаксиса,
- текущее состояние отладчика,

- число файлов, измененных в рабочей копии репозитория Subversion.

Для динамической части реализуем отображение сообщения о завершении фоновой проверки синтаксиса.

Данные для строки состояния будем передавать через класс Sublime::IUiController. Для этого создадим два класса-интерфейса:

- *Sublime::IStatusBarElemFactory* – интерфейс фабрики [29] графических элементов статической части. После передачи указателя на этот интерфейс строке состояния она сможет в любой момент создать соответствующий графический элемент.
- *Sublime::IStatusBarMessage* – интерфейс сообщения для динамической части. Данный интерфейс позволяет получить текст сообщения и выполнить связанное с ним действие при выборе пользователем этого сообщения.

В Sublime::IUiController 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.

### *5.4.2 Строка состояния: визуальный элемент*

Код, описывающий собственно графический элемент строки состояния, находится на уровне Sublime. Строка состояния реализуется классом Sublime::EnhancedStatusBar и добавляется в главное окно IDE KDevelop в коде класса Sublime::MainWindow.

Строка состояния состоит из (слева) места для статических элементов и (справа) области динамических сообщений. Есть возможность просмотра списка пришедших сообщений, реализуемая классом MessageDialog – наследником класса QDialog. Также по нажатию правой кнопки мыши на строке состояния пользователь может задавать, какие статические элементы отображать и в каком порядке. Эта функциональность реализуется классом OrderSelectorWidget, также наследующим QDialog. Реализованная строка состояния, визуальный элемент выбора динамического сообщения и настройки строки состояния изображены на рисунке 9.

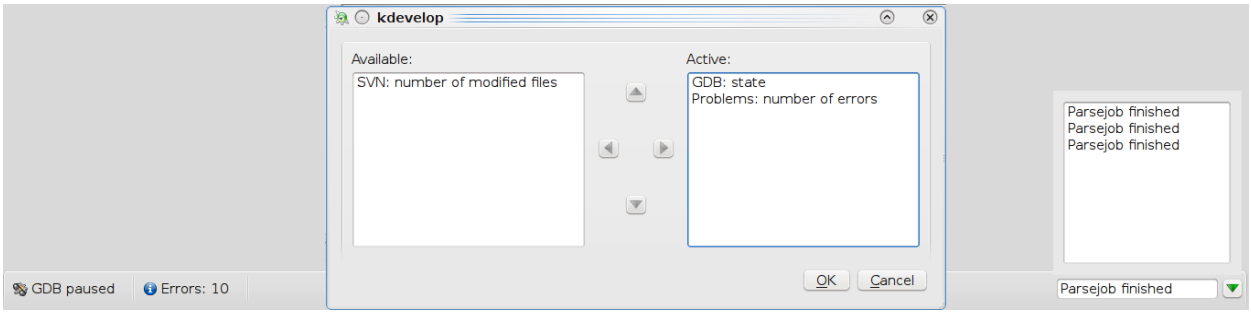


**Рисунок 9. Реализация строки состояния**

### *5.4.3 Строка состояния: динамическая часть*

Динамическая часть строки состояния отображает текст сообщений, переданных строке состояния. Программный интерфейс со строкой состояния состоит в передаче ей указателя на объект класса-наследника IStatusBarMessage.

Для реализации сообщения о завершении фоновой проверки синтаксиса в подключаемом модуле, ответственном за эту проверку, KDevelop::ProblemReporterPlugin описан класс ProblemStatusBarMessage, наследующий IStatusBarMessage и определяющий действие по активации этого события пользователем – открытие инструментального окна фоновой проверки синтаксиса. Схема передачи данных при реализации этого сообщения показана на рисунке 10.

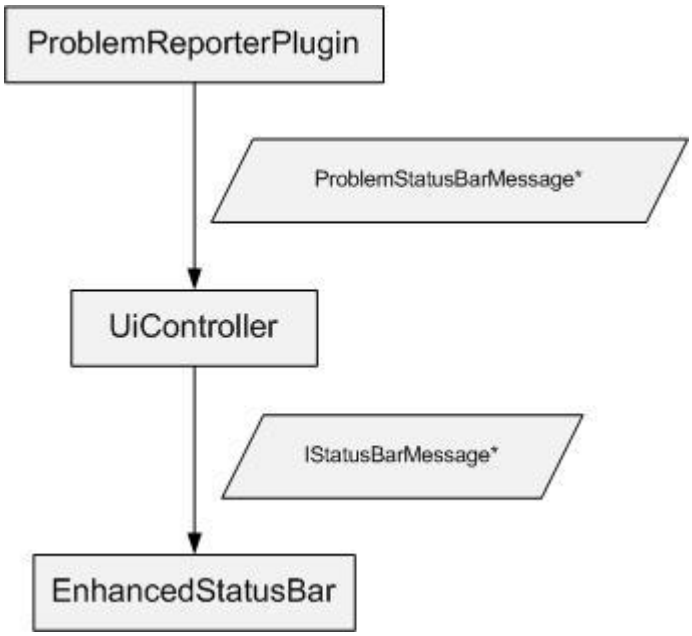


**Рисунок 10. Схема передачи динамических сообщений строки состояния**

### *5.4.4 Строка состояния: статическая часть*

Статическая часть отображает выбранные пользователем статические визуальные элементы и позволяет взаимодействовать с ними. Для регистрации статического элемента в строке состояния требуется передать строке состояния указатель на QWidget, читабельное имя статического элемента и его уникальный строковый идентификатор.

Однако в момент, когда возникает необходимость передать статический элемент от подключаемого модуля, уровень Sublime еще не до конца инициализирован, и строка состояния не может принять графический элемент. Чтобы избежать этой проблемы, UiController предлагает сервис по получению указателей на фабрики статических элементов IStatusBarElemFactory и обязуется передать их строке состояния, когда она будет инициализирована.

*Общая схема реализации статических элементов*

Пусть мы имеем некоторый класс SomeInformationClass (как правило, подключаемый модуль, реализующий некоторое инструментальное окно), располагающий данными, которые необходимо отобразить в качестве статического элемента строки состояния. С учетом описанной архитектуры нам необходимо создать класс-фабрику SomeInformationFactory, наследующую IStatusBarElemFactory, отвечающую за конструирование объекта класса статического элемента, также созданного нами. Далее в конструкторе SomeInformationClass происходит создание фабрики SomeInformationFactory, которая передается UiController. Общая схема передачи данных показана на рисунке 11.

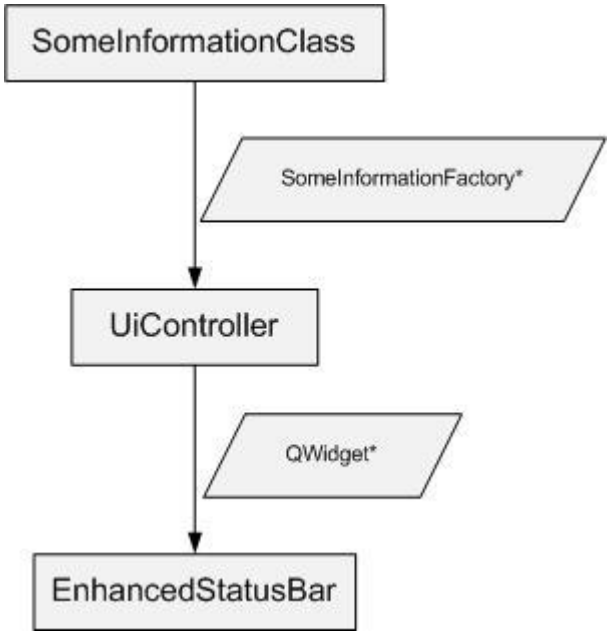


**Рисунок 11. Схема передачи статического визуального элемента в строку состояния**

*Статический элемент: число ошибок фоновой проверки синтаксиса*

Для реализации статического элемента, отображающего число ошибок фоновой проверки синтаксиса, в подключаемом модуле ProblemReporterPlugin был создан класс-фабрика ProblemStatusBarElemFactory, наследующий IStatusBarElemFactory. Этот класс инстанцируется[1] и передается UiController при подключении ProblemReporterPlugin. В этой фабрике заложена функциональность создания объекта класса ProblemStatusWidget, собственно реализующего статический элемент, отображающий число ошибок фоновой проверки синтаксиса.

*Статический элемент: состояние отладчика*

Для реализации этого статического элемента в коде подключаемого модуля DebuggerPlugin описываются классы DebugStatusBarElemFactory и DebugStatusWidget – соответственно фабрика и статический элемент. Согласно общей схеме реализации статических элементов они используются реализации отображения состояния отладчика.

[1] *Инстанцирование класса* – создание объекта этого класса [29]

*Статический элемент: число измененных файлов в рабочей копии SVN*

Данный статический элемент реализуется аналогично предыдущим статическим элементам с использованием классов SVNStatusBarElemFactory (фабрика статических элементов) и SVNStatusWidget (статический элемент).

# 6. Заключение

В данной работе был проведен обзор существующих интегрированных сред разработки программ; на основе этого обзора построена модель инструментальных окон сред разработки программ, включающая список обобщенных окон и их типов.

Был разработан проект однооконного интерфейса, в котором функциональность каждого обобщенного инструментального окна из полученной в обзоре модели предоставляется пользователю через окно текстового редактора и дополнительные визуальные элементы.

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,
- строка состояния, показывающая сообщения от удаленных инструментальных окон и статическую информацию о состоянии IDE.

Итак, проделанная работа соответствует поставленной задаче и полностью решает ее.

Дальнейшее развитие данной работы заключается в следующем:

- реализация переноса функциональности оставшихся инструментальных окон KDevelop;
- расширение функциональности строки состояния;
- разработка критериев оценки графических интерфейсов IDE;
- применение этих критериев для сравнения полученного интерфейса и существующих.

# 7. Список литературы


*1.* Dana Nourie Getting Started with an Integrated Development Environment*,* 2005 [HTML] (http://java.sun.com/developer/technicalArticles/tools/intro.html)

2. 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. 63. pp. 607-627.

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

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

5. Jacob Nielsen Usability Engineering, Academic Press, 1993, 352 p.

6. Dennis G. Jerz Usability Testing: What is it? 2000 [HTML] (http://jerz.setonhill.edu/design/usability/intro.htm)

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

8. M. Stephens 10 Things NetBeans Must Do to Survive, 2003 [HTML] (http://www.softwarereality.com/soapbox/netbeans.jsp)

9. Alan Cooper Inmates Are Running the Asylum. Sams Publishing, 2004, 288 p.

10. Donald A. Norman The Design of Everyday Things. Doubleday, 1989, 261 p.

11. Da Costa, Newton, Steven French Science and Partial Truth: A Unitary Approach to Models and Scientific Reasoning. Oxford: Oxford University Press, 2003, 272 p.

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

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

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

15. NetBeans Community NetBeans Official Website [HTML] (http://www.netbeans.org/)
16. Eclipse Foundation Eclipse Website [HTML] (http://www.eclipse.org/)
17. Code::Blocks Website [HTML] (http://www.codeblocks.org/)
18. Mono Project MonoDevelop Website [HTML] (http://monodevelop.com/)
19. KDevelop team KDevelop Website [HTML] (http://www.kdevelop.org/)
20. IntelliJ IDEA Website [HTML] (http://www.jetbrains.com/idea/)
21. IBM C++ Builder Website [HTML] (http://www.embarcadero.com/products/cbuilder)
22. Jacob Nielsen Breadcrumb navigation increasingly useful, Jakob Nielsen's Alertbox, 2007 [HTML] (http://www.useit.com/alertbox/breadcrumbs.html),
23. Morgan Kauffman GUI Bloopers 2.0 Common User Interface Design Don’ts and Dos, Morgan Kauffman Publishers, 2007, 433 p.
24. KDevelop team Sublime UI Description [HTML] (http://www.kdevelop.org/mediawiki/index.php/Sublime_UI)
25. Бланшет Ж, Саммерфилд М. QT 4: программирование GUI на C++. М.: Кудиц-Пресс, 2007, 628 стр.
26. John Deacon Model-View-Controller Architecture [PDF] // Software Development Training Courses in the UK and Europe
27. The Kate Team Kate Editor Official Website [HTML] (http://kate-editor.org/)
28. Philippe Fremy KDE Technology: KParts Components [HTML] (http://phil.freehackers.org/kde/kpart-techno/kpart-techno.html)
29. Grady Booch. Object-Oriented Analysis and Design with Applications, Addison-Wesley, 2007, 608 p.