МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ
Федеральное государственное автономное образовательное учреждение
высшего образования
**«Южно-Уральский государственный университет
(национальный исследовательский университет)»**
**Высшая школа электроники и компьютерных наук**
**Кафедра системного программирования**

ДОПУСТИТЬ К ЗАЩИТЕ

Заведующий кафедрой, д.ф.-м.н.,
профессор

_____ Л.Б. Соколинский

«___»_____ 2024 г.

# Разработка программы по выявлению больших файлов-кандидатов на удаление на основе последних обращений к файлу

ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА
ЮУрГУ – 09.03.04.2024.308-573.ВКР

Научный руководитель,
доцент кафедры СП, к.ф.-м.н.
_____ И.И. Клебанов

Автор работы,
студент группы КЭ-433
_____ М.А. Ваганов

Ученый секретарь
(нормоконтролер)
_____ И.Д. Володченко
«___»_____ 2024 г.

Челябинск, 2024 г.

МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ
Федеральное государственное автономное образовательное учреждение
высшего образования
**«Южно-Уральский государственный университет**
**(национальный исследовательский университет)»**
**Высшая школа электроники и компьютерных наук**
**Кафедра системного программирования**

УТВЕРЖДАЮ
Зав. кафедрой СП
_____ Л.Б. Соколинский
29.01.2024 г.

**ЗАДАНИЕ**
**на выполнение выпускной квалификационной работы бакалавра**
студенту группы КЭ-433
Ваганову Михаилу Александровичу,
обучающемуся по направлению
09.03.04 «Программная инженерия»

1. **Тема работы** (утверждена приказом ректора от 22.04.2024 г. № 764-13/12)
Разработка программы по выявлению больших файлов-кандидатов на удаление на основе последних обращений к файлу.
2. **Срок сдачи студентом законченной работы:** 03.06.2024 г.
3. **Исходные данные к работе**
3.1. Документация C#. [Электронный ресурс] URL: https://learn.microsoft.com/ru-ru/dotnet/csharp/ (дата обращения: 12.02.2024 г.).
3.2. Соломонов Д.А., Ионеску А., Русинович М. Внутреннее устройство Windows: Архитектура, процессы, потоки, управление памятью и 1многое другое. Часть 1. 7-е изд. М.: Вильямс, 2018. – 752 с.
4. **Перечень подлежащих разработке вопросов**
4.1. Определения критерия «больших» файлов.
4.2. Методы и алгоритмы анализа и оценки, используемые при определении кандидатов на удаление.
5. **Дата выдачи задания:** 29.01.2024 г.

**Научный руководитель,**
доцент кафедры СП, к.ф.-м.н. И.И. Клебанов

**Задание принял к исполнению** М.А. Ваганов

**ОГЛАВЛЕНИЕ**

# ВВЕДЕНИЕ

## Актуальность

В эпоху информационных технологий пользователи персональных компьютеров часто устанавливают множество программ и редко производят очистку памяти. Это приводит к накоплению большого количества файлов, которые могут занимать значительное место на жестком диске, замедляя работу системы. Часто пользователи забывают о старых и редко используемых файлах, не имеют четкого представления о том, какие из них можно безопасно удалить без ущерба для работы системы и своих данных. Эта проблема особенно актуальна для тех, кто активно использует свои устройства для работы с большими объемами данных, таких как разработчики программного обеспечения, дизайнеры, видеооператоры и другие профессионалы.

Современные методы и алгоритмы для анализа файловой системы и управления данными позволяют разработать решения, которые могут эффективно сканировать файловую структуру, анализировать их размеры. Однако существующие решения часто ограничены в возможностях настройки и гибкости использования, что создает необходимость в разработке более универсальных и настраиваемых инструментов. Выпускная квалификационная работа направлена на решение данной проблемы путем анализа файлов на компьютере и оценки их полезности [14].

## Постановка задачи

Целью выпускной квалификационной работы является разработка программы по выявлению больших файлов-кандидатов на удаление на основе последних обращений к файлу. Для достижения поставленной цели необходимо решить следующие задачи.

1. Исследовать существующие методы и инструменты для управления дисковым пространством.
2. Разработать алгоритм для автоматической классификации файлов по их размеру и дате последнего к ним обращения.

3. Провести тестирование программы на различных наборах данных для проверки ее эффективности и надежности.

**Структура и содержание работы**

Работа состоит из введения, четырех глав, заключения и списка литературы. Объем работы составляет 36 страниц, объем списка литературы – 15 источников.

В первой главе описывается анализ предметной области и обзор аналогичных проектов. Проводится исследование существующих методов и инструментов для анализа дискового пространства, рассматриваются основные принципы работы файловых систем.

Во второй главе производится проектирование системы, включая функциональные и нефункциональные требования, а также диаграмму вариантов использования. Описываются требования к программному обеспечению и иллюстрируется взаимодействие пользователя с системой.

В третьей главе описывается реализация системы, включая программные средства, реализацию сканирования файлов, сортировки и оценки файлов, а также основное тело программы. Приводятся подробные описания технологий, ключевых компонентов системы и примеры кода.

В четвертой главе проводится тестирование системы, включая функциональное тестирование и разработку unit-тестов. Описывается процесс тестирования, приводятся примеры тестовых сценариев и результаты их выполнения.

В приложениях содержатся исходные тексты программ. В приложении А приведен листинг реализации класса `FileScanner`, который отвечает за сканирование файловой структуры и извлечение информации о файлах. В приложении Б представлен основной код программы, включая метод `Main`, выполняющий ключевые функции по сканированию, фильтрации и сортировке файлов.

# 1. АНАЛИЗ ПРЕДМЕТНОЙ ОБЛАСТИ

## 1.1. Описание предметной области

Для разработки программы по выявлению крупных файлов-кандидатов на удаление на основе последних обращений к файлу понадобится использовать алгоритмы анализа файловой системы. Программа должна сканировать файловую структуру, отслеживать обращения к файлам и анализировать их размеры и даты последнего обращения.

Также важно предусмотреть возможность настройки критериев определения «крупных» файлов, так как это понятие может быть относительным в зависимости от конкретной файловой системы и контекста использования. Кроме того, при разработке программы важно учитывать безопасность и правильное управление доступом к файлам, чтобы избежать удаления важных данных.

Windows использует несколько различных файловых систем, включая NTFS (New Technology File System), которая является основной файловой системой для версий Windows с 2000 года. NTFS обеспечивает поддержку различных функций, включая управление правами доступа, журналирование и шифрование данных. Другой распространенной файловой системой для Windows является FAT32, которая имеет ограничения на размер файлов и томов по сравнению с NTFS.

Файловая структура в операционной системе Windows организована в виде древовидной иерархии, начиная от корневого каталога, который обозначается буквой диска (например, C:). В каждом каталоге могут находиться подкаталоги, файлы или ссылки на другие части файловой системы.

Файлы в Windows имеют имена и расширения (например, document.docx). Имена файлов не могут содержать определенные символы, такие как /, , *, :, ", <, >, |, и ?, а также не могут заканчиваться точкой или пробелами [12].

Кроме того, Windows поддерживает метаданные файлов, такие как дата создания, дата изменения, разрешения доступа и владелец файла. Эти

данные могут быть использованы для управления файлами и каталогами, а также для обеспечения безопасности данных.

Таким образом, для разработки программы по выявлению крупных файлов-кандидатов на удаление, основываясь на последних обращениях к файлам, необходимо использовать алгоритмы анализа файловой системы. Программа должна сканировать файловую структуру, отслеживать обращения к файлам, анализировать их размеры и даты последнего обращения.

## 1.2. Сравнительный анализ аналогов

### WinDirStat

«WinDirStat» – это бесплатная программа, которая предоставляет пользователям возможность анализировать использование дискового пространства на компьютере под управлением операционной системы Microsoft Windows. Она отображает информацию в виде древовидной карты, где каждый файл и папка представлены в виде блока, размер которого пропорционален их объему на диске. Такой подход делает процесс визуализации и анализа более интуитивно понятным для пользователя [6].

В «WinDirStat» присутствует возможность быстро и удобно выявлять крупные файлы или папки, которые занимают большое количество места на диске и могут быть потенциально удалены для освобождения места. Программа также обеспечивает цветовое кодирование, позволяя легко отличать различные типы файлов (например, видео, аудио, документы и т. д.), что делает процесс анализа более удобным и информативным.

Кроме того, «WinDirStat» предоставляет пользователю возможность проводить анализ не только локальных дисков, но и сетевых ресурсов, что делает ее полезным инструментом для системных администраторов и пользователей, работающих с сетевыми хранилищами данных.

Важным преимуществом «WinDirStat» является его бесплатность и открытый исходный код, что делает его доступным для широкого круга

пользователей без необходимости платить за лицензию или использовать проприетарное программное обеспечение (рисунок 1).

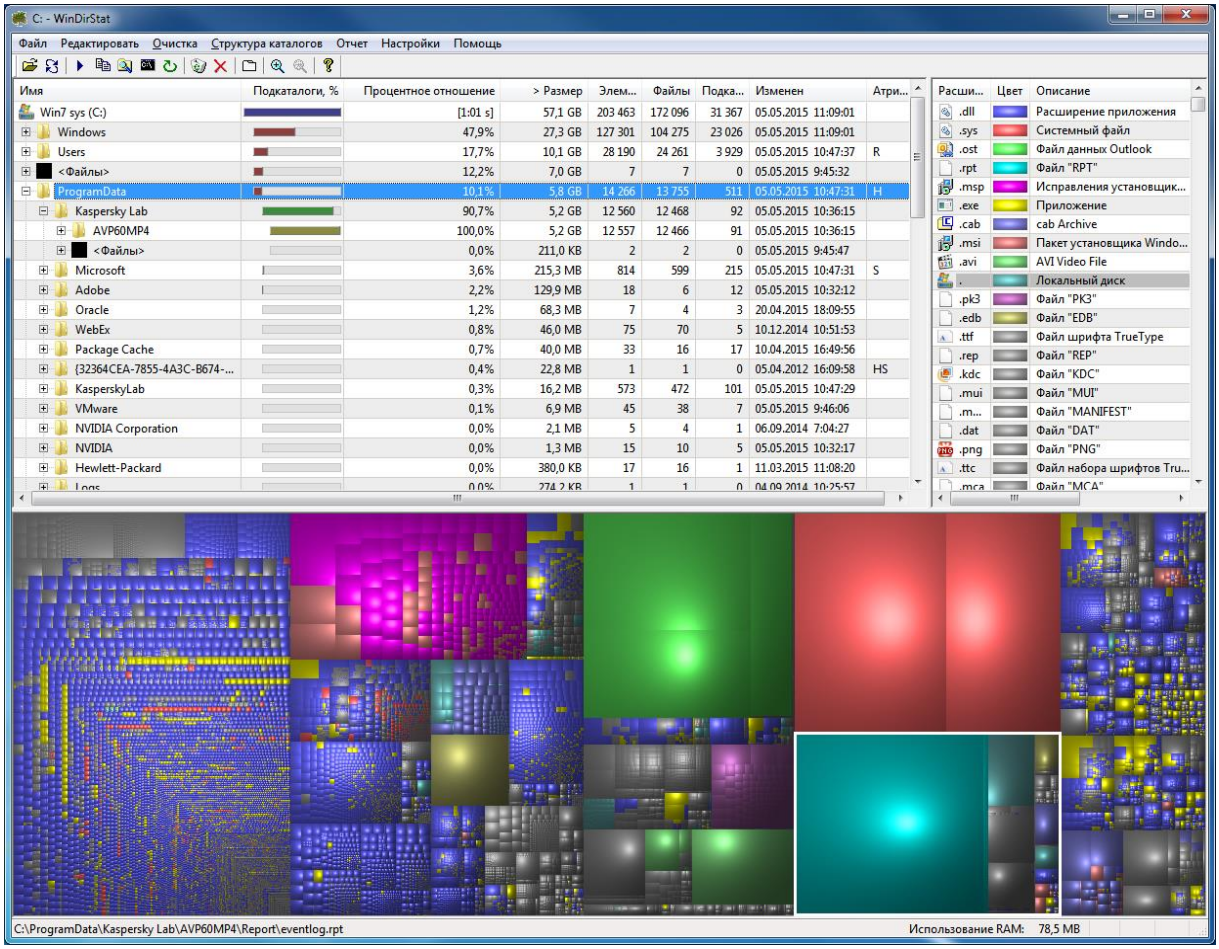


Рисунок 1 – Скриншот основного окна «WinDirStat»

Программы «WinDirStat» включают в себя следующие особенности.

1. Визуальное представление. «WinDirStat» позволяет пользователям визуально оценить использование дискового пространства с помощью древовидной карты, что облегчает быстрое выявление крупных файлов и папок.

2. Цветовое кодирование. Различные типы файлов и папок отображаются разными цветами, делая процесс их идентификации более интуитивным и удобным для пользователя.

3. Интерактивное управление. Пользователи могут взаимодействовать с древовидной картой, щелкая по различным элементам, чтобы получить более детальные сведения о своих файлах и папках.

4. Полезные отчеты. «WinDirStat» предоставляет возможность создания отчетов об использовании дискового пространства, что помогает пользователям лучше понять, какие файлы занимают больше всего места.

**TreeSize**

Это программа для управления дисками, которая позволяет пользователям анализировать использование дискового пространства и идентифицировать крупные файлы на их дисках, помогая освободить место за счет удаления ненужных файлов. Она предоставляет иерархическое представление использования дискового пространства, отображая размеры всех подпапок в выбранной директории. Это позволяет пользователям легко определить, какие папки и файлы занимают больше всего места на диске, обеспечивая эффективную очистку и оптимизацию хранения данных (рисунок 2) [7].

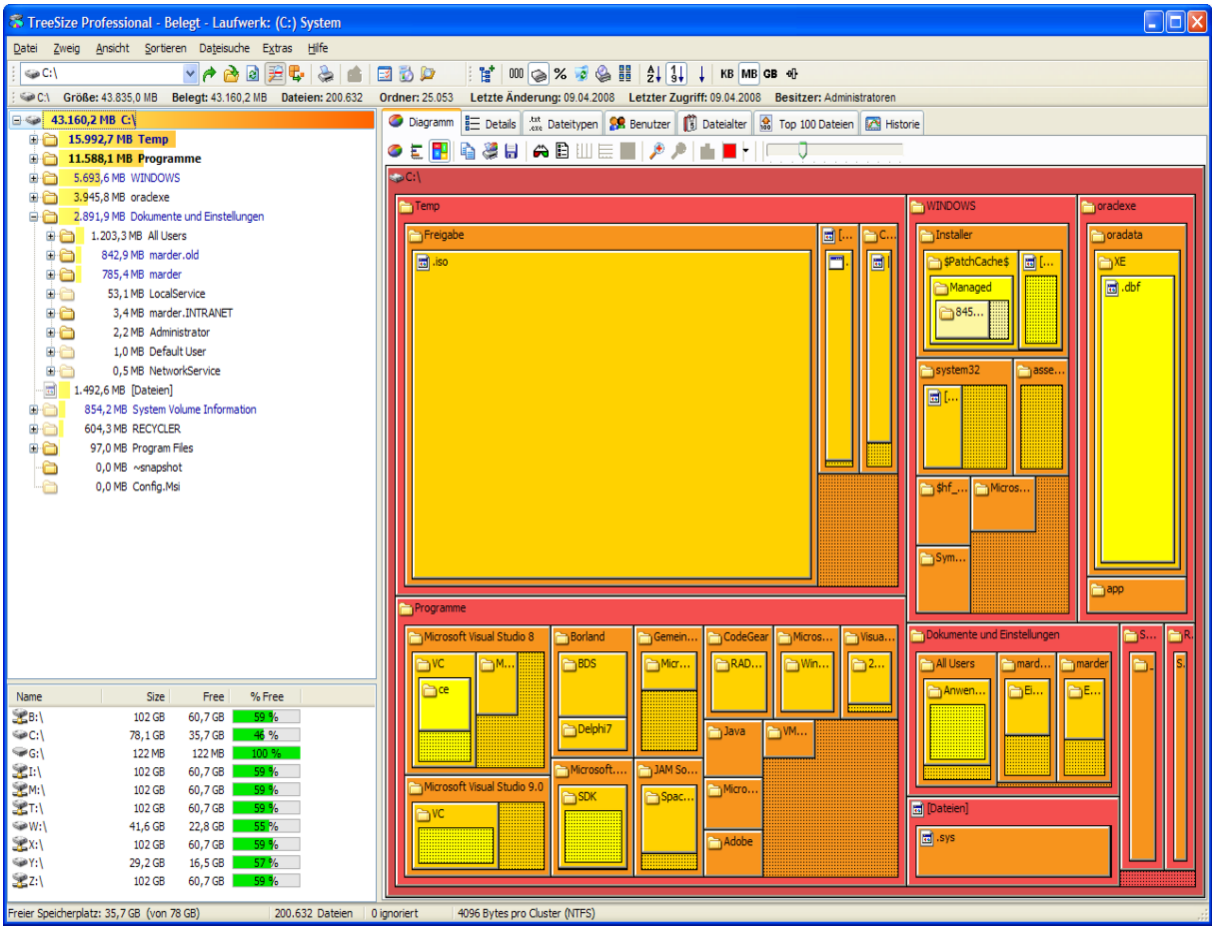


Рисунок 2 – Скриншот основного окна «TreeSize»

Программы «TreeSize» включают в себя следующие особенности.

1. Иерархический просмотр. «TreeSize» предоставляет иерархическое представление использования дискового пространства, позволяя пользователям легко увидеть, какие файлы и папки занимают больше всего места.

2. Удобное взаимодействие. Пользователи могут быстро найти крупные файлы и папки на своем диске, проводя анализ на различных уровнях иерархии файловой системы.

3. Анализ мобильных устройств. «TreeSize» также способен проводить анализ мобильных устройств и облачного хранения, обеспечивая всестороннее понимание использования дискового пространства.

4. Возможность экспорта данных. Пользователи могут создавать отчеты о использовании дискового пространства и предпринимать дальнейшие действия на основе полученных результатов анализа.

**DiskSavvy**

«DiskSavvy» – это удобная и мощная программа для анализа использования дискового пространства на вашем компьютере. Ее основная цель – предоставить пользователю подробную статистику о том, как именно используется каждый байт вашего дискового пространства [8].

Одним из ключевых преимуществ «DiskSavvy» является возможность анализировать дисковое пространство с различных точек зрения. Файлы можно классифицировать по их расширениям, размерам, датам создания и последнему доступу. Это дает полное представление о характеристиках использования диска и позволяет принимать обоснованные решения относительно управления и очистки дискового пространства.

Благодаря возможности анализировать крупные и устаревшие файлы, «DiskSavvy» помогает выявить потенциально ненужные данные, которые можно безопасно удалить, освобождая ценное место на диске. Это особенно полезно при ограниченном пространстве на жестком диске и необходимости его оптимизации.

Помимо этого, «DiskSavvy» обладает интуитивно понятным интерфейсом, что делает его доступным для широкого круга пользователей. Можно легко найти необходимую информацию о дисковом пространстве и принять меры для его оптимизации.

Таким образом, «DiskSavvy» представляет собой необходимый инструмент для тех, кто хочет эффективно управлять использованием дискового пространства на своем компьютере. Благодаря своим многофункциональным возможностям и простому интерфейсу, эта программа делает процесс анализа и управления диском быстрым, удобным и эффективным (рисунок 3).

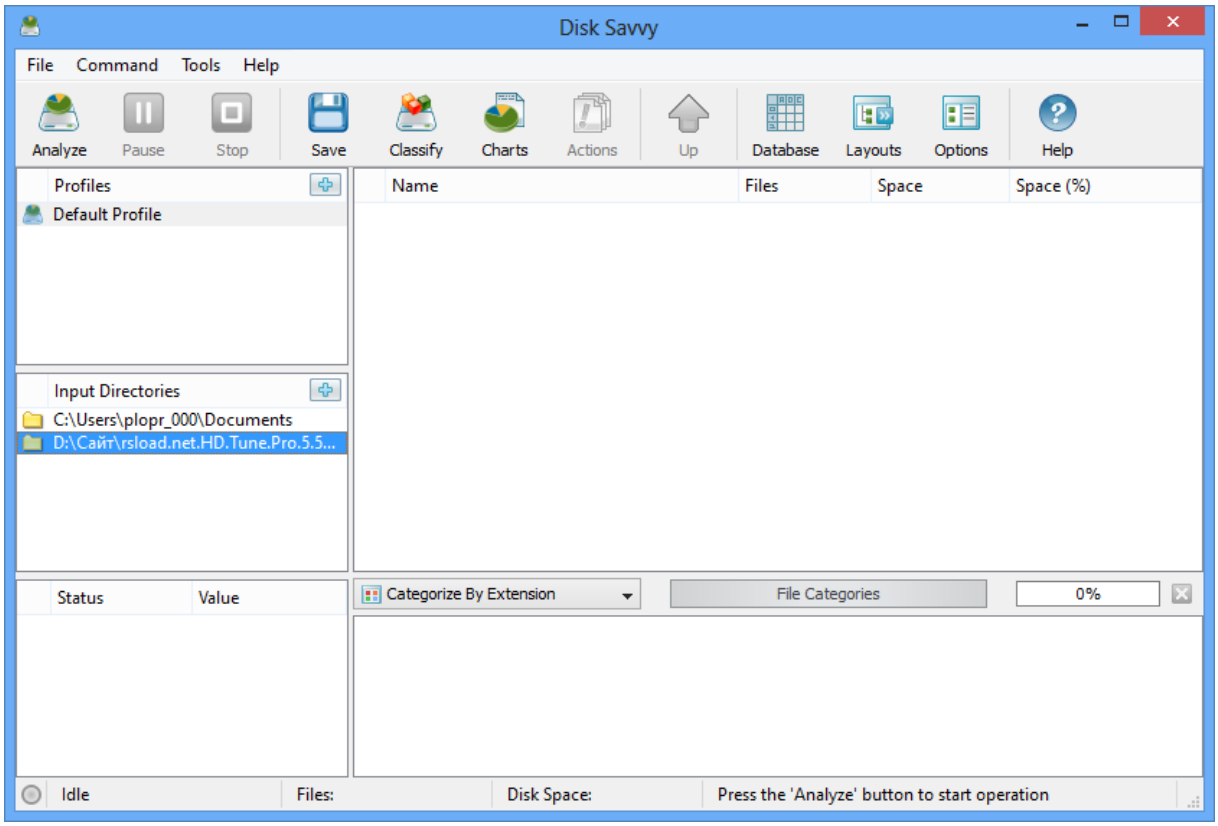


Рисунок 3 – Скриншот основного окна «DiskSavvy»

Программы «DiskSavvy» включают в себя следующие особенности.

1. Гибкие варианты анализа. «DiskSavvy» предлагает разнообразные варианты анализа, включая категоризацию файлов по расширению, размеру, дате создания и последнему доступу. Это обеспечивает пользователем гибкость при изучении образцов использования дискового пространства.

2. Наглядные отчеты. Программа предоставляет детальные отчеты и графики, которые помогают пользователям лучше понять, как именно используется их дисковое пространство (рисунок 4).

3. Автоматизация. «DiskSavvy» может быть настроен для автоматического выполнения анализа использования дискового пространства в соответствии с предварительно заданными расписаниями, что облегчает систематическое отслеживание изменений в использовании дискового пространства.

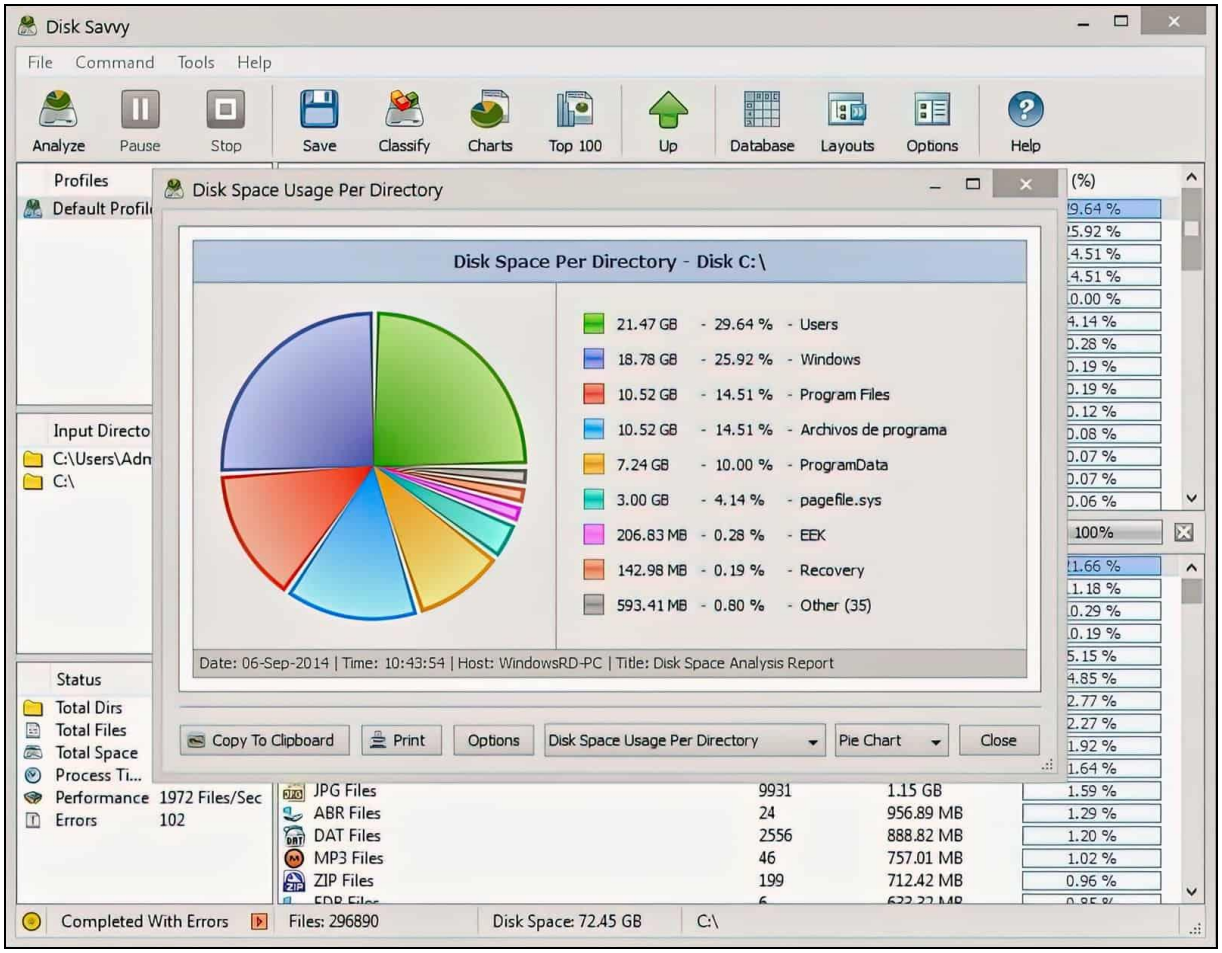


Рисунок 4 – Скриншот окна с отчетом «DiskSavvy»

**SpaceSniffer**

Это инструмент для анализа использования дискового пространства, который предоставляет визуальное представление использования дискового пространства в виде интерактивной древовидной карты. Он позволяет поль-

зователям быстро понять распределение файлов и папок на их дисках и выявить крупные файлы и папки, которые могут быть кандидатами на удаление. Визуальное представление помогает эффективно ориентироваться в использовании дискового пространства и выявлять области, где возможно восстановление места (рисунок 5) [9].

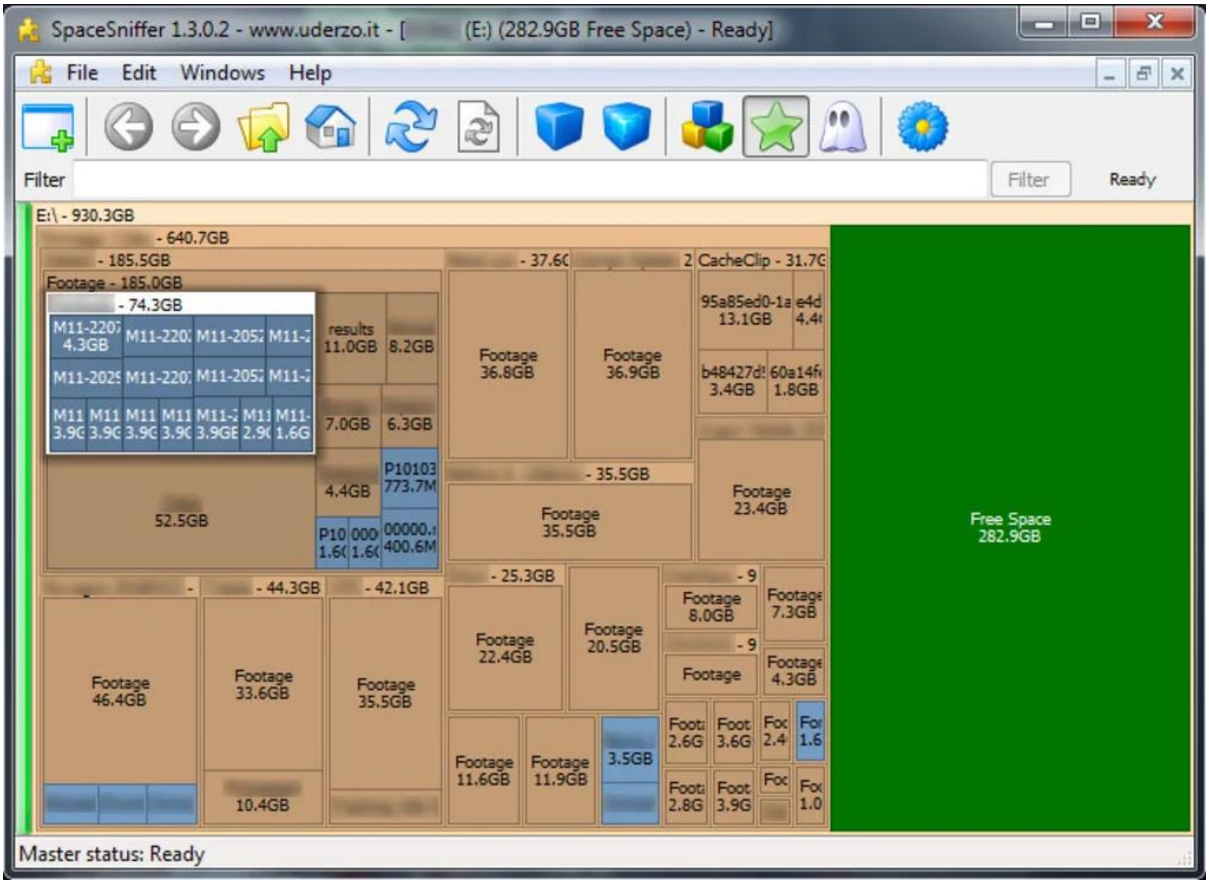


Рисунок 5 – Скриншот основного окна «SpaceSniffer»

Программы «SpaceSniffer» включают в себя следующие особенности.

1. Интерактивная древовидная карта. «SpaceSniffer» предоставляет визуальное представление занятого места на диске в виде интерактивной древовидной карты, позволяя пользователям быстро выявлять крупные файлы и папки для удаления.

2. Эффективная визуализация. Программа предлагает простой и интуитивно понятный способ визуализации использования дискового пространства, что облегчает пользователям анализ и принятие необходимых мер для оптимизации использования дискового пространства.

3. Многоуровневая навигация. «SpaceSniffer» позволяет пользователям исследовать использование дискового пространства на нескольких уровнях, упрощая процесс локализации и удаления крупных и редко используемых файлов.

Таким образом, для разработки программы необходимо учесть визуальное представление данных, иерархический анализ, возможность создания отчетов, поддержку различных файловых систем Windows, автоматизацию анализа, а также обеспечить безопасность данных.

**Вывод по первой главе**

В этой главе рассмотрена предметная область, связанная с анализом использования дискового пространства, и проведен обзор существующих аналогов. Результаты сравнительного анализа позволили выделить несколько программных решений, заслуживающих внимания.

«WinDirStat», «TreeSize», «DiskSavvy» и «SpaceSniffer» – каждая из этих программ предлагает свои уникальные возможности для визуализации использования дискового пространства и анализа размеров файлов. Некоторые из них специализируются на определенных аспектах анализа, таких как категоризация файлов по типу или дате создания, в то время как другие обеспечивают более общий и всесторонний подход к анализу.

Одна из главных целей этого сравнительного анализа была выявление программ, которые обеспечивают удобное взаимодействие с пользователем и позволяют эффективно управлять использованием дискового пространства. В конечном итоге выбор программы зависит от индивидуальных потребностей пользователя и требований конкретной ситуации.

# 2. ПРОЕКТИРОВАНИЕ

## 2.1. Функциональные и нефункциональные требования

Для разработки программы по выявлению больших файлов-кандидатов на удаление на основе последних обращений к файлу, необходимо определить функциональные и нефункциональные требования.

**Функциональные требования**

В соответствии с целями и задачами данного проекта определены следующие функциональные требования.

1. Анализ файлов. Программа должна иметь возможность анализировать размер файлов и определять, являются ли они кандидатами на удаление.

2. Отслеживание активности файлов. Необходима функция отслеживания последних обращений к файлам для определения их актуальности.

3. Интерфейс для просмотра результата. Пользовательский интерфейс или отчет должен показывать найденные кандидаты на удаление с указанием их размера и последней активности.

4. Гибкое конфигурирование. Пользователь должен иметь возможность настраивать критерии для определения «больших» файлов и период последних обращений для учета.

5. Автоматическое удаление. Программа должна предоставлять возможность автоматического удаления выявленных кандидатов на удаление, с возможностью подтверждения операции пользователем.

6. Интеграция с системой отчетности. Возможность создания отчетов о найденных больших файлах и результатов их анализа для последующего аудита или отчетности.

**Нефункциональные требования**

В соответствии с целями и задачами данного проекта определены следующие нефункциональные требования.

1. Производительность. Программа должна быть способна обрабатывать большие объемы данных быстро и эффективно.

2. Безопасность. Должны быть предусмотрены меры безопасности для защиты конфиденциальных данных, а также возможность работы программы с правами доступа к файлам.

Эти требования представляют основные пункты, которые следует учесть при разработке программы по выявлению больших файлов-кандидатов на удаление на основе последних обращений к файлу.

## 2.2. Диаграмма вариантов использования

Была построена модель взаимодействия внешнего актера с приложением в виде диаграммы вариантов использования. В ходе анализа требований к разрабатываемой системе были выявлены варианты использования (рисунок 6) [10–11].

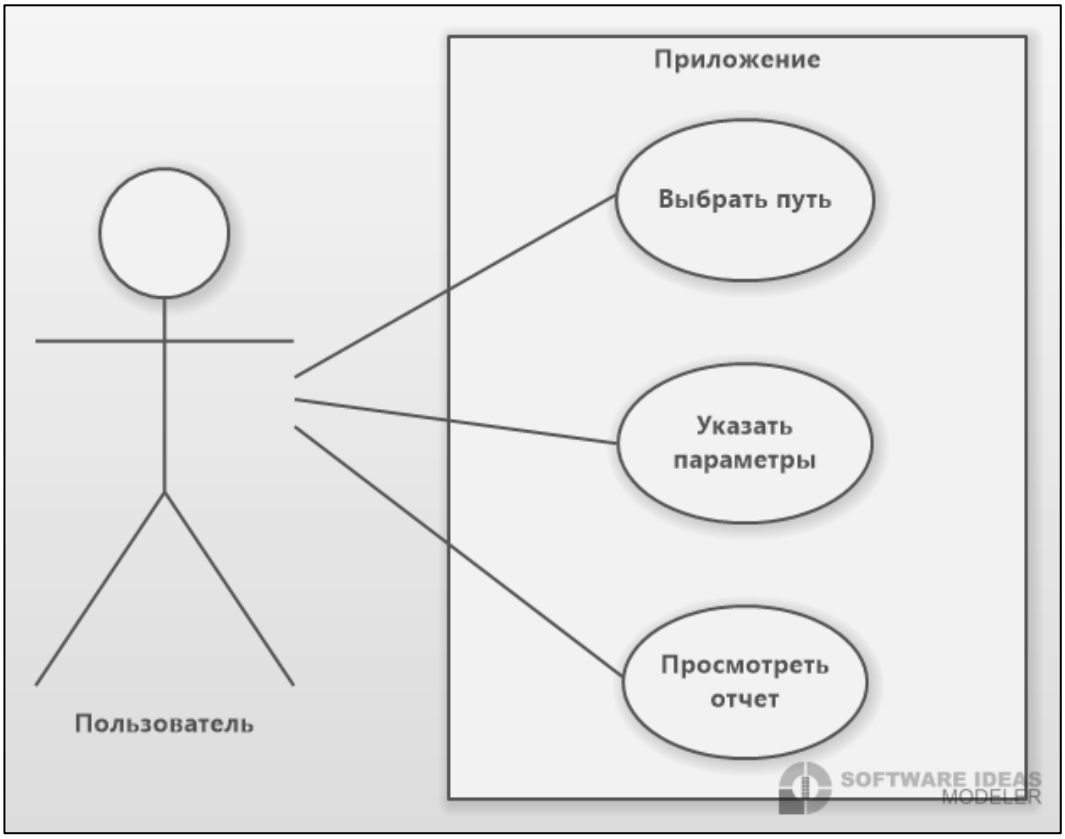


Рисунок 6 – Диаграмма вариантов использования

## 2.3. Архитектура программы

Архитектура разрабатываемого приложения предназначена для выявления крупных файлов-кандидатов на удаление на основе последних обращений к файлам. Приложение состоит из нескольких основных компонентов, каждый из которых выполняет определенные функции и взаимодействует с другими модулями для достижения общей цели (рисунок 7).

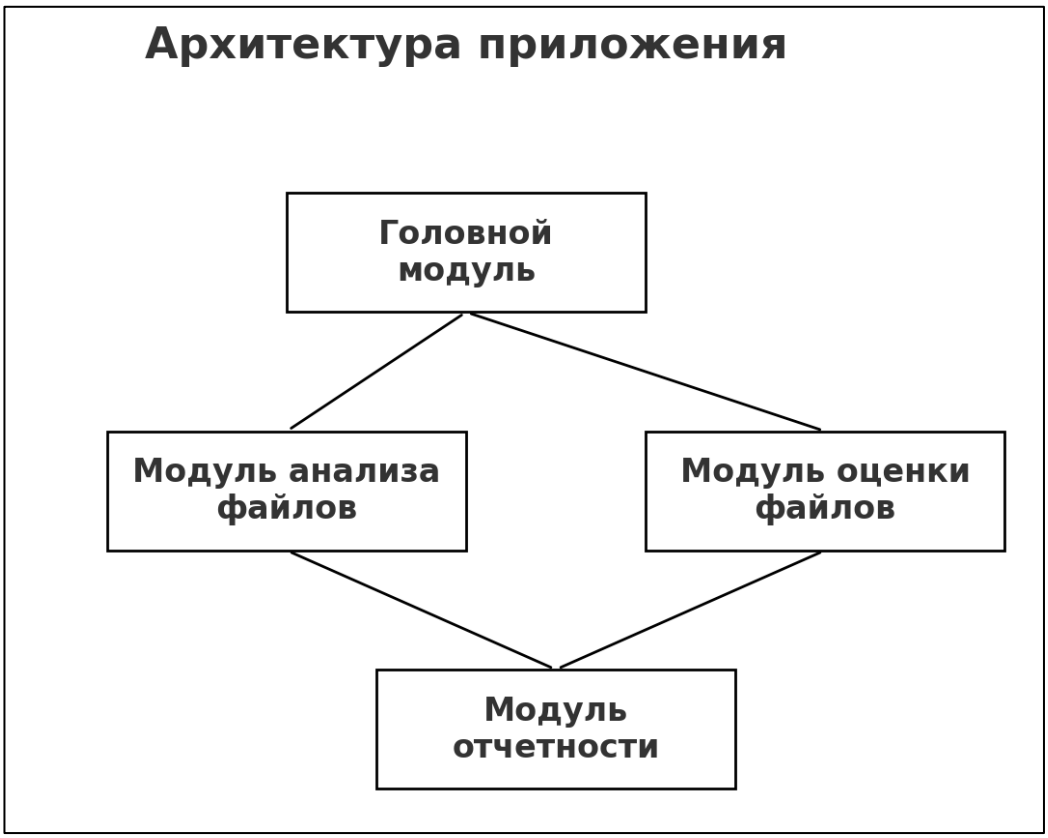


Рисунок 7 – Архитектура программы

Основой приложения является головной модуль, который отвечает за отображение окон пользовательского интерфейса и за запуск подчиненных модулей. Головной модуль инициализирует интерфейс, предоставляя пользователю доступ к функциональным возможностям приложения, таким как сканирование файловой структуры, анализ файлов и визуализация результатов. Он также управляет взаимодействием между другими модулями, вызывая их по мере необходимости.

Модуль анализа файлов играет ключевую роль в процессе сбора и обработки информации о файлах. Этот модуль сканирует файловую структуру, извлекает данные о файлах, определяет размер и дату последнего обращения к файлам, а также фильтрует и сортирует файлы по заданным критериям. Результаты анализа передаются следующему модулю для дальнейшей обработки.

Модуль оценки файлов принимает данные от модуля анализа файлов и оценивает файлы на основе их размера и времени последнего доступа. Этот модуль вычисляет оценку для каждого файла, группирует файлы по папкам и создает список кандидатов на удаление. Модуль оценки файлов тесно взаимодействует с модулем анализа, используя его результаты для выполнения своей задачи.

Модуль отчетности завершает цепочку обработки данных, формируя и отображая результаты анализа. Он создает отчеты с информацией о крупных файлах, которые могут быть удалены, и визуализирует данные в удобном для пользователя формате. Головной модуль вызывает модуль отчетности для предоставления пользователю итоговой информации.

Головной модуль инициализирует работу всех других модулей и управляет их последовательным выполнением. Модуль анализа файлов собирает и обрабатывает информацию о файлах, необходимые для анализа, и передает ее модулю оценки файлов. Модуль оценки файлов оценивает файлы и готовит данные для отчета. Наконец, модуль отчетности формирует итоговые отчеты и визуализирует результаты для пользователя.

**Вывод по второй главе**

Во второй главе данной работы было проведено проектирование системы для выявления крупных файлов-кандидатов на удаление на основе последних обращений к файлам.

В рамках проектирования были определены функциональные и нефункциональные требования к системе. Функциональные требования вклю-

чают возможности анализа файлов, отслеживания их активности, предоставления интерфейса для просмотра результатов, гибкого конфигурирования, автоматического удаления и интеграции с системой отчетности. Нефункциональные требования касаются производительности, безопасности и защиты конфиденциальных данных.

Была создана диаграмма вариантов использования, иллюстрирующая взаимодействие пользователя с системой. Диаграмма помогла визуализировать основные сценарии использования системы и определить ключевые модули и их функции.

Также в главе описана архитектура программы, включающая головной модуль, модуль анализа файлов, модуль оценки файлов и модуль отчетности. Архитектура обеспечивает эффективное взаимодействие между компонентами и позволяет достичь высокой производительности и гибкости системы.

Таким образом, результаты проектирования заложили основу для дальнейшей реализации и тестирования системы, обеспечив четкое понимание функциональных возможностей и требований к программе.

# 3. РЕАЛИЗАЦИЯ

Для реализации программной части системы был выбран язык программирования C# [1–3]. Разработка велась в среде программирования Visual Studio [4]. Для реализации были использованы следующие библиотеки языка:

1) C# System;
2) System.Collections.Generic;
3) System.IO;
4) System.Runtime.InteropServices.

## 3.1. Реализация прохода по файлам

Реализован класс `FileScanner`, представленный в приложении А, который совершает проход по файлом в указанной директории, а также получает атрибуты файла такие как размер и дата последнего обращения, без влияния на них [13].

После запуска сканера, он начинает проходиться по файлам в указанной категории. Сканер тщательно проверяет каждый файл в выбранной директории и всех ее поддиректориях. Для каждого найденного файла он извлекает важные атрибуты, такие как размер файла и дата последнего обращения к нему. Эта информация фиксируется без изменения исходных атрибутов файлов, чтобы избежать непреднамеренного воздействия на данные. Полученные атрибуты систематически сохраняются в специально организованный список. Этот список формируется для последующих операций и дальнейших манипуляций с файлами, обеспечивая эффективное управление и обработку данных в будущем. Таким образом, процесс сканирования создает подробную и структурированную базу данных обо всех файлах в заданной категории, что позволяет проводить более детальный анализ и принимать обоснованные решения на основе собранной информации.

### 3.2. Реализация сортировки

Реализованный класс `FileSorted` фильтрует файлы, передаваемые сканером, по заданному пользователю размеру, а также производит группировку оцененных файлов по папкам представлен на листинге 1.

Листинг 1 – Реализация `FileSorter`

```
public void SortAndFilterFiles(List<File> files)
{
    // Сортируем файлы по размеру
    files.Sort((a, b) => b.Size.CompareTo(a.Size));

    // Фильтруем файлы по размеру и группируем их по папкам
    foreach (var file in files)
    {
        if (file.Size > sizeThreshold)
        {
            largeFiles.Add(file);
            if (!filesByFolder.ContainsKey(file.Path))
            {
                filesByFolder[file.Path] = new List<File>();
            }
            filesByFolder[file.Path].Add(file);
        }
    }
}

public void EvaluateFiles(List<File> files, long sizeThreshold, TimeSpan
dormantSet)
{
    // Группируем оценки по папкам
    foreach (var file in files)
    {
        Evaluation evaluation = new Evaluation(file);
        evaluation.Estimate(sizeThreshold, dormantSet);

        if (!evaluationsByFolder.ContainsKey(file.Path))
        {
            evaluationsByFolder[file.Path] = new List<Evaluation>();
        }
        evaluationsByFolder[file.Path].Add(evaluation);
    }
}
```

После запуска сортировщика отсеивает файлы, размер которых меньше указанного пользователем значения, а остальные помещает в следующий список. После того как оценщик выполнит свою работу группирует файлы по директориям.

### 3.3. Реализация оценки файлов

Реализованный класс `Evaluation` проводит оценку по заданным пользователем параметрам и вычисляет среднюю оценку на основе размера и даты последнего обращения представлен на листинге 2.

Листинг 2 – Реализация `Evaluation`

```
// Метод для оценки файла на основе размера и времени последнего доступа
public double Estimate(long setSize, TimeSpan dormantSet)
{
    // Получаем дату последнего доступа к файлу
    DateTime accessDate = currFile.LastAccessTime;
    // Вычисляем длительность времени простоя (время с момента последнего
доступа до текущего момента)
    TimeSpan dormantDuration = DateTime.Now - accessDate;
    // Получаем текущий размер файла
    long currSize = currFile.Size;

    // Расчет оценки по размеру файла
    // Максимальная оценка - 100 баллов, даже если размер файла превышает
порог
    double sizeScore = Math.Min(100, ((double)currSize / setSize) * 100);

    // Расчет оценки по времени последнего доступа
    // Максимальная оценка - 100 баллов, даже если время простоя превышает
порог
    double dormantScore = Math.Min(100, (dormantDuration.TotalDays /
dormantSet.TotalDays) * 100);

    // Средняя оценка, основанная на двух критериях
    Score = (sizeScore + dormantScore) / 2;
    return Score;
}
```

После использования вычисляет баллы для веса и длительности простоя как отношение полученных атрибутов к пороговым значениям указанным пользователем, после чего возвращает средней балл файла. Чем выше балл, тем более желательно удалить файл.

### 3.4. Основное тело программы

В приложении Б представлен класс Program, который содержит статический метод `Main`. Этот метод выполняет ключевую функцию в программе, так как именно с него начинается выполнение кода. В методе `Main` осуществляется вызов других классов и методов, обеспечивая тем самым за-

пуск и координацию всех операций и процессов в приложении. Таким образом, класс Program и его статический метод `Main` играют центральную роль в управлении потоком выполнения программы, объединяя различные компоненты и функциональные модули в единое целое.

После выполнения кода, представленного в листинге 4, программа выполняет несколько ключевых операций. Сначала она запрашивает у пользователя путь к папке для сканирования и проверяет его существование. Если путь существует, создается объект корневой папки, и сканер начинает проход по всем файлам и поддиректориям, извлекая и сохраняя их атрибуты. Далее программа запрашивает у пользователя размерный порог, фильтрует файлы по этому порогу и группирует их по папкам.

Затем, программа запрашивает временной порог в днях, создает объект `TimeSpan` для этого порога и оценивает файлы на основе их размера и времени последнего доступа. Оценка файлов помогает определить, какие файлы являются кандидатами на удаление. Наконец, программа выводит оцененные файлы, сгруппированные по папкам, что позволяет пользователю легко просмотреть результаты и принять дальнейшие действия. Таким образом, данный процесс обеспечивает эффективное управление файлами, поддерживая их актуальность и оптимизацию использования дискового пространства.

**Вывод по третьей главе**

В данной главе была разработана программа для выявления файлов-кандидатов на удаление. Программа включает в себя несколько ключевых компонентов: `FileScanner` для обхода файловой системы и сбора информации о файлах, `FileSorter` для фильтрации и сортировки файлов, и `Evaluation` для оценки файлов на основе их размера и времени последнего доступа. Класс Program управляет процессом выполнения программы, взаимодействуя с пользователем и обеспечивая координацию всех операций.

# 4. ТЕСТИРОВАНИЕ

Для обеспечения качества и надежности разработанной программы по выявлению больших файлов-кандидатов на удаление на основе последних обращений к файлам необходимо провести всестороннее тестирование. Тестирование включает в себя различные подходы для проверки как функциональных, так и нефункциональных аспектов системы [5].

## 4.1. Функциональное тестирование

Функциональное тестирование направлено на проверку соответствия программы заявленным функциональным требованиям. Оно охватывает ключевые сценарии использования системы и взаимодействие между ее компонентами. В таблице 1 приведены основные тесты, проведенные в рамках функционального тестирования.

Таблица 1 – Функциональное тестирование

| № | Название теста | Шаги | Ожидаемый результат | Тест пройден? |
|---|---|---|---|---|
| 1 | Сканирование папки | 1. Указать путь к корневой папке.<br>2. Вызвать метод `ScanFolder`. | Папка просканирована, файлы и подпапки добавлены в структуру данных. | Да |
| 2 | Фильтрация файлов по размеру | 1. Задать порог размера.<br>2. Вызвать метод `SortAndFilterFiles`. | Файлы, превышающие порог, добавлены в список больших файлов. | Да |
| 3 | Оценка файлов | 1. Задать порог размера и временной порог.<br>2. Вызвать метод `EvaluateFiles`. | Файлы оценены по размеру и времени последнего доступа. | Да |
| 4 | Вывод больших файлов | Вызвать метод `PrintLargeFiles`. | Список больших файлов выведен на экран. | Да |
| 5 | Вывод файлов по папкам | Вызвать метод `PrintFilesByFolder`. | Файлы сгруппированы и выведены по папкам. | Да |
| 6 | Вывод оцененных файлов | Вызвать метод `PrintEvaluationsByFolder`. | Оцененные файлы выведены и сгруппированы по папкам. | Да |

Окончание таблицы 1

| № | Название теста | Шаги | Ожидаемый результат | Тест пройден? |
|---|---|---|---|---|
| 7 | Парсинг размерного порога | 1. Ввести значение размера (например, 10KB, 100MB).<br>2. Вызвать метод `ParseSizeInput`. | Значение корректно распознано и переведено в байты. | Да |
| 8 | Обработка некорректного пути | 1. Указать несуществующий путь.<br>2. Вызвать метод `ScanFolder`. | Отображено сообщение об ошибке пути. | Да |
| 9 | Обработка исключений доступа | 1. Указать путь к защищенной папке.<br>2. Вызвать метод `ScanFolder`. | Отображено сообщение об отсутствии доступа. | Да |

## 4.2. Unit-тестирование

Unit-тестирование – это методика тестирования программного обеспечения, при которой отдельные модули исходного кода (например, функции, методы или классы) проверяются на корректность их работы [15]. Основная цель unit-тестирования заключается в том, чтобы изолировать каждый элемент программы и убедиться, что он работает правильно. Оно позволяет выявить ошибки на ранних стадиях разработки, облегчает их исправление и повышает общее качество кода. Кроме того, unit-тесты служат дополнительной документацией, показывающей, как должны работать отдельные компоненты системы. Они также обеспечивают уверенность при рефакторинге кода, так как помогают быстро определить, где именно произошла ошибка, если тесты начинают проваливаться. В результате, unit-тестирование способствует созданию более надежного и поддерживаемого программного обеспечения.

Преимущества unit-тестирования.

1. Раннее выявление ошибок. Unit-тестирование позволяет обнаружить ошибки на ранних стадиях разработки. Это упрощает их исправление, поскольку ошибки выявляются до того, как они могут повлиять на другие части системы.

2. Документация. Unit-тесты служат дополнительной формой документации для кода. Они показывают, как должны работать отдельные компоненты системы, и предоставляют примеры использования различных функций.

3. Уверенность при рефакторинге. Наличие обширного набора unit-тестов позволяет уверенно изменять и улучшать код, зная, что тесты выявят возможные регрессии. Это позволяет разработчикам вносить изменения в кодовую базу без страха сломать существующий функционал, так как тесты обеспечивают постоянную проверку корректности работы системы.

4. Повышение качества кода. Регулярное проведение unit-тестирования способствует улучшению общего качества кода. Тесты помогают поддерживать код в чистоте и порядке, выявляя и устраняя дефекты, недочеты и потенциальные проблемы. Это приводит к более стабильной и надежной системе, которая легче поддерживается и развивается.

В рамках тестирования было разработано 3 unit-теста. Unit-тесты представлены на листингах 3–5.

Листинг 3 – Тест на сканирование файлов

```
          [Test]
          public void TestScanFolder()
          {
              // Создаем временную директорию для теста
              string testDir = Path.Combine(Path.GetTempPath(),
Path.GetRandomFileName());
              Directory.CreateDirectory(testDir);

              try
              {
                  // Создаем файлы и папки в тестовой директории
                  string testFile1 = Path.Combine(testDir, "test1.txt");
                  string testFile2 = Path.Combine(testDir, "test2.txt");
                  string subDir = Path.Combine(testDir, "subDir");
                  Directory.CreateDirectory(subDir);
                  string testFile3 = Path.Combine(subDir, "test3.txt");

                  System.IO.File.WriteAllText(testFile1, "Test content 1");
                  System.IO.File.WriteAllText(testFile2, "Test content 2");
                  System.IO.File.WriteAllText(testFile3, "Test content 3");

                  Folder rootFolder = new Folder { Name = "Root", Path =
testDir };
                  FileScanner scanner = new FileScanner();
                  scanner.ScanFolder(testDir, rootFolder);

                  // Проверяем результаты сканирования
```

```
                List<File> allFiles = scanner.GetAllFiles();
                Assert.AreEqual(3, allFiles.Count, "Количество файлов
должно быть 3");
                Assert.IsTrue(allFiles.Exists(f => f.Name ==
"test1.txt"), "Файл test1.txt должен существовать");
                Assert.IsTrue(allFiles.Exists(f => f.Name ==
"test2.txt"), "Файл test2.txt должен существовать");
                Assert.IsTrue(allFiles.Exists(f => f.Name ==
"test3.txt"), "Файл test3.txt должен существовать");
            }
            finally
            {
                // Удаляем временную директорию после теста
                Directory.Delete(testDir, true);
            }
        }
    }
```

Представленный тест проверяет функциональность сканирования директорий и выявления файлов внутри них. В тесте создается временная директория, в которой размещаются несколько тестовых файлов и папок. После этого запускается метод `ScanFolder` для сканирования созданной структуры директорий. В конце теста проверяется, что все созданные файлы были обнаружены и корректно добавлены в результирующий список. По завершении теста временная директория удаляется.

Листинг 4 – Тест на сортировку по размеру

```
    [Test]
    public void TestSortAndFilterFiles()
    {
        // Создаем список файлов для теста
        List<File> files = new List<File>
    {
        new File { Name = "smallFile.txt", Size = 500, Path = "/path" },
        new File { Name = "mediumFile.txt", Size = 1500, Path =
"/path" },
        new File { Name = "largeFile.txt", Size = 2500, Path = "/path" }
    };

        long sizeThreshold = 1000;
        FileSorter sorter = new FileSorter(sizeThreshold);
        sorter.SortAndFilterFiles(files);

        // Проверяем результаты сортировки и фильтрации
        List<File> largeFiles = sorter.GetLargeFiles();

        Assert.AreEqual(2, largeFiles.Count, "Количество больших файлов
должно быть 2");
        Assert.IsTrue(largeFiles.Exists(f => f.Name == "mediumFile.txt"),
"Файл mediumFile.txt должен существовать");
        Assert.IsTrue(largeFiles.Exists(f => f.Name == "largeFile.txt"),
"Файл largeFile.txt должен существовать");
        Dictionary<string, List<File>> filesByFolder = sorter.Get-
FilesByFolder();
```

```
          Assert.IsTrue(filesByFolder.ContainsKey("/path"), "Путь /path
должен существовать в словаре");
          Assert.AreEqual(2, filesByFolder["/path"].Count, "Количество фай-
лов в /path должно быть 2");
      }
```

Представленный тест проверяет сортировку и фильтрацию файлов по размеру. Создается список файлов с различными размерами, запускается метод `SortAndFilterFiles`, который сортирует и фильтрует файлы по заданному порогу. В конце теста проверяется, что количество больших файлов и структура данных по папкам соответствуют ожидаемым значениям.

Листинг 5 – Тест оценки

```
          [Test]
          public void TestEvaluateFiles()
          {
              // Создаем список файлов для теста
              List<File> files = new List<File>
          {
              new File { Name = "recentFile.txt", Size = 1500, Path =
"/path", LastAccessTime = DateTime.Now.AddDays(-5) },
              new File { Name = "oldFile.txt", Size = 2500, Path = "/path",
LastAccessTime = DateTime.Now.AddDays(-100) }
          };

              long sizeThreshold = 1000;
              TimeSpan dormantSet = TimeSpan.FromDays(30);
              FileSorter sorter = new FileSorter(sizeThreshold);
              sorter.EvaluateFiles(files, sizeThreshold, dormantSet);

              // Проверяем результаты оценки
              Dictionary<string, List<Evaluation>> evaluationsByFolder =
sorter.GetEvaluationsByFolder();
              Assert.IsTrue(evaluationsByFolder.ContainsKey("/path"), "Путь
/path должен существовать в словаре");
              Assert.AreEqual(2, evaluationsByFolder["/path"].Count,
"Количество оценок в /path должно быть 2");

              Evaluation recentFileEvaluation = evalua-
tionsByFolder["/path"].Find(e => e.ToString().Contains("recentFile.txt"));
              Assert.IsNotNull(recentFileEvaluation, "Оценка для recent-
File.txt должна существовать");
              Assert.LessOrEqual(recentFileEvaluation.Score, 100, "Оценка
recentFile.txt должна быть меньше или равна 100");

              Evaluation oldFileEvaluation = evalua-
tionsByFolder["/path"].Find(e => e.ToString().Contains("oldFile.txt"));
              Assert.IsNotNull(oldFileEvaluation, "Оценка для oldFile.txt
должна существовать");
              Assert.LessOrEqual(oldFileEvaluation.Score, 100, "Оценка old-
File.txt должна быть меньше или равна 100");
          }
      }
```

Представленный тест проверяет оценку файлов на основе их размера и времени последнего доступа. В тесте создается список файлов с различным временем доступа, после чего запускается метод `EvaluateFiles`. В конце проверяется, что количество оценок и их значения соответствуют ожидаемым результатам.

**Вывод по четвертой главе**

В четвертой главе проведены функциональное и unit-тестирование системы для проверки ее корректной работы.

Функциональное тестирование подтвердило, что система выполняет ключевые задачи, такие как сканирование папок, фильтрация и оценка файлов, а также вывод результатов. Все тесты прошли успешно, что подтверждает правильность работы основных функций.

Unit-тестирование проверило корректность работы отдельных модулей. Тесты для сканирования, сортировки и оценки файлов помогли выявить и устранить ошибки на ранних стадиях, повысив качество и надежность кода.

Тестирование показало, что система соответствует требованиям, обладает высокой производительностью и безопасностью данных. Комплексный подход к тестированию подтвердил готовность системы к использованию. Система демонстрирует устойчивую работу и корректно выполняет все заявленные функции.

Таким образом, тестирование завершило этап разработки, подтвердив соответствие системы поставленным целям и задачам. В дальнейшем можно продолжить доработку и улучшение системы, включая расширение функционала и улучшение интерфейса.

## ЗАКЛЮЧЕНИЕ

В рамках данной выпускной квалификационной работы была разработана программа, которая осуществляет анализ последних обращений к файлам для выявления крупных файлов-кандидатов на удаление. В процессе работы были решены следующие задачи.

1. Исследовать существующие методы и инструменты для управления дисковым пространством.
2. Разработать алгоритм для автоматической классификации файлов по их размеру и дате последнего к ним обращения.
3. Провести тестирование программы на различных наборах данных для проверки ее эффективности и надежности.

Программа, разработанная в ходе исследования, позволяет пользователям легко и эффективно управлять дисковым пространством, выявляя и удаляя ненужные файлы на основе их размера и времени последнего доступа. Это способствует оптимизации использования дискового пространства и повышает производительность системы.

Основные результаты включают в себя следующее.

1. Анализ и описание предметной области, включая сравнение существующих аналогов.
2. Разработка функциональных и нефункциональных требований.
3. Проектирование и реализация ключевых компонентов программы, таких как сканер файловой системы, сортировщик файлов и оценщик.
4. Проведение тестирования, включая функциональное и unit-тестирование, что подтвердило корректность работы системы и ее соответствие заданным требованиям.

В настоящий момент продолжается доработка приложения. В будущем планируется улучшение функционала и пользовательского интерфейса, а также добавление новых возможностей для анализа и управления файлами.

**ЛИТЕРАТУРА**

1. Документация C#. [Электронный ресурс] URL: https://learn.microsoft.com/ru-ru/dotnet/csharp/ (дата обращения: 12.02.2024 г.).

2. Марк П. C# 9 and .NET 5 разработка и оптимизация, СПб: Питер, 2022. – 832 с.

3. Вершинин В. Введение в C#. СПб: Питер, 2023. – 350 с.

4. Петров А. Visual Studio 2022: Руководство пользователя. М.: Альпина Паблишер, 2023. – 480 с.

5. Иванов Д. Тестирование программного обеспечения: Принципы, практики и паттерны. М.: ДМК Пресс, 2021. – 400 с.

6. Официальный сайт WinDirStat. [Электронный ресурс] URL: https://windirstat.net (дата обращения: 24.05.2024 г.).

7. Официальный сайт TreeSize. [Электронный ресурс] URL: https://treesize.en.softonic.com (дата обращения: 24.05.2024 г.).

8. Официальный сайт DiskSavvy. [Электронный ресурс] URL: https://disksavvy.com (дата обращения: 24.05.2024 г.).

9. Официальный сайт SpaceSniffer. [Электронный ресурс] URL: https://spacesniffer.ru.uptodown.com/windows (дата обращения: 24.05.2024 г.).

10. Фаулер М. UML: Основы. Краткое руководство по стандартному языку моделирования объектов. М.: Вильямс, 2004. – 208 с.

11. Обучающая информация Software Ideas Modeler. [Электронный ресурс] URL: https://www.softwareideas.net/en/help/tutorials (дата обращения: 24.05.2024 г.).

12. Соломонов Д.А., Ионеску А., Русинович М. Внутреннее устройство Windows: Архитектура, процессы, потоки, управление памятью и 1многое другое. Часть 1. 7-е изд. М.: Вильямс, 2018. – 752 с.

13. Каррьер Б. Анализ файловых систем. М.: Вильямс, 2006. – 896 с.

14. Смирнов А. Архитектура вычислительных систем. М.: Бином, 2023. – 480 с.

15. Кузнецов М. Современные методы тестирования программного обеспечения. М.: Эксмо, 2022. – 320 с.

# ПРИЛОЖЕНИЯ

## Приложение А. Исходный код класса FileScanner

В листинге 1 представлена реализация класса `FileScanner`.

Листинг 1 – Реализация `FileScanner`

```
public class FileScanner
{
    [DllImport("kernel32.dll", SetLastError = true, CharSet = Char-
Set.Auto)]
    [return: MarshalAs(UnmanagedType.Bool)]
    private static extern bool GetFileAttributesEx(string lpFileName,
GET_FILEEX_INFO_LEVELS fInfoLevelId, out WIN32_FILE_ATTRIBUTE_DATA
fileData);
    [StructLayout(LayoutKind.Sequential)]
    private struct WIN32_FILE_ATTRIBUTE_DATA
    {
        public uint FileAttributes;
        public FILETIME CreationTime;
        public FILETIME LastAccessTime;
        public FILETIME LastWriteTime;
        public uint FileSizeHigh;
        public uint FileSizeLow;
    }
    [StructLayout(LayoutKind.Sequential)]
    private struct FILETIME
    {
        public uint DateTimeLow;
        public uint DateTimeHigh;

        // Преобразование FILETIME в DateTime
        public DateTime ToDateTime()
        {
            long highBits = DateTimeHigh;
            highBits = highBits << 32;
            return DateTime.FromFileTimeUtc(highBits + DateTimeLow);
        }
    }
    public void ScanFolder(string rootPath, Folder rootFolder)
    {
        var foldersToProcess = new Stack<(string Path, Folder Folder)>();
        foldersToProcess.Push((rootPath, rootFolder));

        while (foldersToProcess.Count > 0) // Пока есть папки для обработки
        {
            var (currentPath, currentFolder) = foldersToProcess.Pop();
            try
            {
                var directoryInfo = new DirectoryInfo(currentPath);
                foreach (var fileInfo in directoryInfo.GetFiles())
                {
                    if (GetFileAttributesEx(fileInfo.FullName,
GET_FILEEX_INFO_LEVELS.GetFileExInfoStandard, out WIN32_FILE_ATTRIBUTE_DATA
fileData))
                    {
                        long fileSize = ((long)fileData.FileSizeHigh << 32)
+ fileData.FileSizeLow;
                        DateTime lastAccessTime = fileData.LastAccess-
Time.ToDateTime();

                        var file = new File
```

```
                    {
                        Name = fileInfo.Name,
                        Size = fileSize,
                        Path = currentPath,
                        LastAccessTime = lastAccessTime
                    };
                    currentFolder.Files.Add(file);
                    allFiles.Add(file);
                }
            }
            foreach (var dirInfo in directoryInfo.GetDirectories())
            {
                var subfolder = new Folder
                {
                    Name = dirInfo.Name,
                    Path = dirInfo.FullName
                };
                currentFolder.Subfolders.Add(subfolder);
                foldersToProcess.Push((dirInfo.FullName, subfolder));
            }
        }
        catch (UnauthorizedAccessException)
        {
            Console.WriteLine($"Нет доступа к {currentPath}");
        }
    }
}
```

## Приложение Б. Исходный код класса Program

В листинге 2 представлена реализация класса `Programm`.

Листинг 2 – Реализация класса `Programm`

```
public class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine(«Введите путь к папке для сканирования:»);
        string rootPath = Console.ReadLine();
        if (!Directory.Exists(rootPath))
        {
            Console.WriteLine(«Указанный путь не существует.»);
            return;
        }
        Folder rootFolder = new Folder { Name = “Root”, Path = rootPath };
        FileScanner scanner = new FileScanner();
        scanner.ScanFolder(rootPath, rootFolder);
        Console.WriteLine(«Введите размерный порог (например, 10KB, 100MB,
2GB, 1TB):»);
        string sizeInput = Console.ReadLine();
        long sizeThresholdBytes = ParseSizeInput(sizeInput);
        if (sizeThresholdBytes < 0)
        {
            Console.WriteLine(“Некорректный размерный порог.”);
            return;
        }
        Console.WriteLine(«Введите временной порог в днях:»);
        if (!int.TryParse(Console.ReadLine(), out int dormantDays))
        {
            Console.WriteLine(«Некорректный временной порог.»);
            return;
        }
        FileSorter sorter = new FileSorter(sizeThresholdBytes);
        sorter.SortAndFilterFiles(scanner.GetAllFiles());
        TimeSpan dormantSet = TimeSpan.FromDays(dormantDays);
        sorter.EvaluateFiles(scanner.GetAllFiles(), sizeThresholdBytes,
dormantSet);
        sorter.PrintEvaluationsByFolder();
    }
    private static long ParseSizeInput(string sizeInput)
    {
        sizeInput = sizeInput.Trim().ToUpper();
        long multiplier;
        if (sizeInput.EndsWith("KB"))
        {
            multiplier = 1024;
            sizeInput = sizeInput.Substring(0, sizeInput.Length - 2);
        }
        else if (sizeInput.EndsWith("MB"))
        {
            multiplier = 1024 * 1024;
            sizeInput = sizeInput.Substring(0, sizeInput.Length - 2);
        }
        else if (sizeInput.EndsWith("GB"))
        {
            multiplier = 1024 * 1024 * 1024;
            sizeInput = sizeInput.Substring(0, sizeInput.Length - 2);
        }
        else if (sizeInput.EndsWith("TB"))
        {
```

```
            multiplier = 1024L * 1024L * 1024L * 1024L;
            sizeInput = sizeInput.Substring(0, sizeInput.Length - 2);
        }
        else
        {
            return -1;
        }

        if (long.TryParse(sizeInput, out long size))
        {
            return size * multiplier;
        }
        else
        {
            return -1;
        }
    }
}
```