ДОНСКОЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ

УПРАВЛЕНИЕ ЦИФРОВЫХ ОБРАЗОВАТЕЛЬНЫХ ТЕХНОЛОГИЙ
ПИ (филиал) ДГТУ в г. Таганроге

ЦМК «Прикладная информатика

Практикум

по дисциплине

«Разработка программных модулей (часть 1)»

Авторы
Андриян И.В.,
Андриян О.В.

Ростов-на-Дону, 2026

Аннотация

Практикум по выполнению практических работ по дисциплине «Разработка программных модулей» (часть 1)» предназначен для студентов очной формы обучения по специальности 09.02.11 «Разработка и управление программным обеспечением».

Авторы

преподаватель, председатель ЦМК «Прикладная информатика»,
Андриян И.В.

преподаватель ЦМК «Прикладная информатика»
Андриян И.В

Оглавление

Введение 4
Практическая работа № 1 «Анализ предметной области, разработка и оформление технического задания» 5
Практическая работа № 2 «Изучение работы в системе контроля версий (на примере GIT)» 11
Практическая работа № 3 «Использование инструментальных средств документирования алгоритмов программ» 20
Практическая работа № 4 «Определение сложности алгоритмов сортировки, поиска, рекурсивных и эвристических алгоритмов» 28
Практическая работа № 5 «Работа с классами и перегрузка методов» 39
Практическая работа № 6 «Определение операций в классе» 57
Практическая работа № 7 «Создание наследованных классов» 72
Практическая работа № 8 «Работа с объектами через интерфейсы» 86
Практическая работа № 9 «Использование стандартных интерфейсов» 97
Практическая работа № 10 «Работа с типом данных структура, использование коллекций» 113
Практическая работа № 11 «Операции со списками» 127
Практическая работа № 12 «Разработка приложения с использованием текстовых компонентов» 136
Практическая работа № 13 «Разработка приложения с несколькими формами» 148
Практическая работа № 14 «Разработка приложения с невизуальными компонентами» 157
Практическая работа № 15 «Разработка игрового приложения» 165
Практическая работа № 16 «Разработка приложения с функционально связанными модулями» 172
Практическая работа № 17 «Разработка приложения с информационно связанными модулями». 192
Перечень использованных информационных ресурсов 210

Введение

Практикум предназначен для оказания методической помощи обучающимся при выполнении практических работ, направленных на закрепление теоретических знаний и формирование навыков решения задач. В указаниях представлены краткие теоретические сведения по основным темам курса, примеры решения типовых задач, а также задания, подлежащие выполнению на практических занятиях.

Обучающийся должен знать: основные принципы проектирования модулей программного обеспечения; языки программирования и технологии для реализации модулей; паттерны проектирования и структуры данных для создания эффективных и масштабируемых модулей; методы анализа требований и способов определения функциональности модуля; принципы создания интерфейсов для взаимодействия с другими модулями и системами; принципы обеспечения безопасности, производительности и масштабируемости при проектировании модулей; методы анализа и оптимизации проектируемых модулей.

Обучающийся должен уметь: проектировать модули, соответствующие бизнес-задачам; создавать архитектурные диаграммы и документацию; определять структуру и интерфейсы модулей; анализировать требования к модулю и определять его функциональность; проектировать архитектуру модуля, включая выбор подходящих паттернов проектирования и структуры данных; создавать диаграммы классов, последовательностей и прочих диаграмм для визуализации проектируемого модуля; выбирать подходящие языки программирования и технологии для реализации модуля; проектировать интерфейсы программного обеспечения для взаимодействия с другими модулями и системами.

Общие положения. Практические занятия выполняются каждым обучающимся самостоятельно в полном объеме и согласно содержанию методических указаний. Перед выполнением обучающийся должен отчитаться перед преподавателем за выполнение предыдущего занятия (сдать отчет).

Обучающийся должен на уровне понимания и воспроизведения предварительно усвоить необходимую для выполнения практических занятий теоретическую и информацию. Обучающийся, получивший положительную оценку и сдавший отчет по предыдущему практическому занятию, допускается к выполнению следующему занятию. Обучающийся, пропустивший практическое занятие по уважительной, либо неуважительной причине, закрывает задолженность в процессе выполнения последующих практических занятий.

Порядок выполнения работы: выполнить задание; показать преподавателю; ответить на вопросы преподавателя; оформить отчет.

Форма отчета: титульный лист с указанием темы, ФИО, группы, варианта описание задания; введение (цель и задачи); выполнение; заключение

Практическая работа № 1 «Анализ предметной области, разработка и оформление технического задания»

Цель работы: Изучить методологию анализа предметной области, освоить структуру и правила оформления технического задания на разработку программного обеспечения в соответствии с требованиями ГОСТ 34.602-2020.

Время выполнения: 2 часа

Теоретическая часть

Анализ предметной области

Анализ предметной области является первым и одним из наиболее важных этапов разработки программного обеспечения. Предметная область — это сфера деятельности, для автоматизации которой создается программный продукт. Качественный анализ предметной области позволяет правильно сформулировать требования к системе, избежать ошибок проектирования и снизить затраты на доработку программы.

Основные задачи анализа предметной области:

– выявление основных бизнес-процессов и их участников;
– определение информационных потоков;
– выявление функциональных и нефункциональных требований;

– определение ограничений и рисков проекта;
- формирование глоссария терминов предметной области.

Техническое задание (ТЗ)

Техническое задание — это основной документ, определяющий требования и порядок создания автоматизированной системы или программного продукта. В Российской Федерации разработка ТЗ регламентируется стандартом ГОСТ 34.602-2020 «Информационные технологии. Комплекс стандартов на автоматизированные системы. Техническое задание на создание автоматизированной системы».

Структура технического задания

1. Общие сведения
   - Полное наименование системы и ее условное обозначение
   - Шифр темы или шифр (номер) договора
   - Наименование предприятий (объединений) разработчика и заказчика системы
   - Перечень документов, на основании которых создается система
   - Плановые сроки начала и окончания работы

2. Назначение и цели создания системы
   - Назначение системы (для чего создается)
   - Цели создания системы (какие проблемы должны быть решены)

3. Характеристика объектов автоматизации
   - Краткие сведения об объекте автоматизации
   - Сведения об условиях эксплуатации объекта автоматизации

4. Требования к системе

4.1. Требования к системе в целом:
- Требования к структуре и функционированию системы
- Требования к численности и квалификации персонала
- Показатели назначения (функциональные возможности)
- Требования к надежности
- Требования безопасности

4.2. Требования к функциям (задачам):
- Перечень функций системы
- Требования к организации входных и выходных данных
- Требования к математическому обеспечению

- Требования к информационному обеспечению

4.3. Требования к видам обеспечения:

- Требования к программному обеспечению
- Требования к техническому обеспечению
- Требования к метрологическому обеспечению
- Требования к организационному обеспечению

5. Состав и содержание работ по созданию системы

- Перечень стадий и этапов работ
- Сроки выполнения работ
- Исполнители работ

6. Порядок контроля и приемки системы

- Виды, состав, объем и методы испытаний
- Общие требования к приемке работы
- Статус приемочной комиссии

7. Требования к составу и содержанию работ по подготовке объекта автоматизации к вводу системы в действие

8. Требования к документированию

9. Источники разработки

Методы сбора информации при анализе предметной области

1.Интервьюирование — проведение бесед с заказчиком и будущими пользователями системы

2.Анкетирование — сбор информации с помощью опросных листов

3.Наблюдение — изучение рабочих процессов непосредственно на месте

4.Анализ документации — изучение существующих регламентов, инструкций, отчетов

5.Прототипирование — создание упрощенных моделей будущей системы для уточнения требований

Функциональные и нефункциональные требования

Функциональные требования описывают, что система должна делать:

- обработка данных
- вычисления
- формирование отчетов
- взаимодействие с пользователем

Нефункциональные требования описывают, как система

должна работать:

- производительность
- надежность
- безопасность
- удобство использования
- масштабируемость
- совместимость

Диаграммы для анализа предметной области

При анализе предметной области используют различные визуальные модели:

- Use Case диаграммы (диаграммы вариантов использования) — для описания функций системы с точки зрения пользователей
- Диаграммы потоков данных (DFD) — для моделирования информационных потоков
- ER-диаграммы (Entity-Relationship) — для моделирования структуры данных
- BPMN-диаграммы — для моделирования бизнес-процессов.

Рабочее задание. Выполните анализ предметной области в соответствии с вариантом задания и разработайте техническое задание на создание программной системы.

Вариант 1. Система учета книг в библиотеке

Разработать программную систему для автоматизации учета книжного фонда библиотеки учебного заведения. Система должна обеспечивать регистрацию книг, учет выдачи и возврата литературы студентам, формирование отчетов о наличии и движении книг.

Вариант 2. Система расписания занятий

Разработать программную систему для автоматизации составления и управления расписанием занятий в колледже. Система должна учитывать доступность аудиторий, нагрузку преподавателей, распределение групп, формировать расписание в различных форматах.

Вариант 3. Система учета успеваемости студентов

Разработать программную систему для автоматизации учета успеваемости студентов. Система должна обеспечивать ввод

оценок, расчет среднего балла, формирование ведомостей, отчетов по успеваемости групп и отдельных студентов.

Вариант 4. Система управления складом

Разработать программную систему для автоматизации учета товаров на складе небольшого предприятия. Система должна обеспечивать регистрацию поступления и отпуска товаров, контроль остатков, формирование отчетов о движении товаров.

Вариант 5. Система регистрации заявок в службе технической поддержки

Разработать программную систему для регистрации и обработки заявок пользователей в службе технической поддержки. Система должна обеспечивать создание заявок, назначение исполнителей, отслеживание статуса выполнения, формирование отчетов.

Вариант 6. Система учета посещаемости занятий

Разработать программную систему для автоматизации учета посещаемости студентов. Система должна обеспечивать ежедневную регистрацию посещаемости, расчет процента посещений, формирование отчетов по группам и дисциплинам.

Вариант 7. Система бронирования переговорных комнат

Разработать программную систему для управления бронированием переговорных комнат и конференц-залов в офисе. Система должна отображать доступность помещений, обеспечивать резервирование на определенное время, уведомлять пользователей.

Вариант 8. Система учета рабочего времени сотрудников

Разработать программную систему для учета рабочего времени сотрудников организации. Система должна регистрировать время прихода и ухода, учитывать опоздания, рассчитывать отработанные часы, формировать табели учета рабочего времени.

Вариант 9. Система управления проектами

Разработать программную систему для управления небольшими IT-проектами. Система должна обеспечивать создание проектов, постановку задач, назначение исполнителей, отслеживание прогресса, формирование отчетов о ходе выполнения проектов.

Вариант 10. Система учета оборудования

Разработать программную систему для учета компьютерной техники и оборудования в учебном заведении. Система должна вести реестр оборудования, отслеживать его размещение, регистрировать ремонты и обслуживание, формировать инвентаризационные отчеты.

Требования к выполнению рабочего задания

1.Проведите анализ предметной области:
– опишите основные бизнес-процессы
– определите участников (пользователей системы)
– выявите основные информационные объекты
– сформулируйте функциональные требования (не менее 7)
– сформулируйте нефункциональные требования (не менее 5)
2.Разработайте техническое задание, которое должно включать следующие разделы:
– Общие сведения
– Назначение и цели создания системы
– Характеристика объектов автоматизации
– Требования к системе (функциональные и нефункциональные)
– Требования к программному и техническому обеспечению
– Требования к документированию
3.Оформите отчет, содержащий: титульный лист; результаты анализа предметной области; техническое задание; выводы по работе

Контрольные вопросы

1.Что такое предметная область и каковы основные задачи ее анализа при разработке программного обеспечения?

2.Какой ГОСТ регламентирует разработку технического задания на создание автоматизированных систем в Российской Федерации? Перечислите основные разделы технического задания.

3.В чем заключается различие между функциональными и нефункциональными требованиями к программной системе? Приведите примеры.

4.Какие методы сбора информации используются при анализе предметной области? Опишите преимущества и недостатки каждого метода.

5.Почему качественное техническое задание является критически важным документом для успешной разработки программного обеспечения? К чему может привести неполное или некорректное ТЗ?

Практическая работа № 2 «Изучение работы в системе контроля версий (на примере GIT)»

Цель работы: Изучить назначение и принципы работы систем контроля версий, освоить основные команды Git, научиться создавать репозитории, выполнять коммиты, работать с ветками и удаленными репозиториями. Получить практические навыки интеграции Git с Visual Studio 2022 для управления версиями программного кода на C#.

Время выполнения: 2 ч

Теоретическая часть

Системы контроля версий

Система контроля версий (Version Control System, VCS) — это программное обеспечение, которое позволяет отслеживать изменения в файлах проекта с течением времени, сохранять историю разработки и обеспечивать совместную работу нескольких разработчиков над одним проектом.

Основные задачи систем контроля версий:

- отслеживание истории изменений кода;
- возможность возврата к предыдущим версиям файлов;
- организация совместной работы команды разработчиков;
- разрешение конфликтов при одновременном редактировании;
- ведение параллельных веток разработки;
- резервное копирование исходного кода.

Виды систем контроля версий:

1.Локальные — хранят все версии на локальном компьютере (RCS);

2.Централизованные — используют центральный сервер для хранения всех версий (SVN, CVS);

3.Распределенные — каждый разработчик имеет полную копию репозитория (Git, Mercurial).

Система контроля версий Git

Git — распределенная система контроля версий, созданная в 2005 году Линусом Торвальдсом для управления разработкой ядра Linux. На сегодняшний день Git является стандартом индустрии разработки программного обеспечения.

Преимущества Git:

- высокая скорость работы;
- распределенная архитектура;
- эффективная работа с ветками;
- целостность данных (контрольные суммы SHA-1);
- возможность работы в автономном режиме;
- бесплатность и открытый исходный код.

Основные понятия Git

Репозиторий (Repository) — хранилище файлов проекта и всей истории их изменений.

Коммит (Commit) — зафиксированное состояние проекта в определенный момент времени. Каждый коммит содержит информацию об авторе, дате, сообщении и изменениях.

Ветка (Branch) — независимая линия разработки. Позволяет работать над новыми функциями, не затрагивая основной код.

Слияние (Merge) — объединение изменений из разных веток.

Удаленный репозиторий (Remote Repository) — версия репозитория, размещенная на сервере (например, GitHub, GitLab, Bitbucket).

Клонирование (Clone) — создание локальной копии удаленного репозитория.

Индекс (Staging Area) — промежуточная область, куда добавляются файлы перед коммитом.

Три состояния файлов в Git

1.Modified (Измененный) — файл изменен, но изменения не зафиксированы

2.Staged (Подготовленный) — файл добавлен в индекс и готов к коммиту

3.Committed (Зафиксированный) — изменения сохранены в локальном репозитории

Основные команды Git

Настройка Git:

```
git config --global user.name "Имя Пользователя"
git config --global user.email "email@example.com"
git config --list
```

Создание репозитория:

```
git init                  # создать новый локальный репозиторий
git clone <url>           # клонировать удаленный репозиторий
```

Работа с изменениями:

```
git status                # проверить статус файлов
git add <file>            # добавить файл в индекс
git add .                 # добавить все измененные файлы
git commit -m "message"   # создать коммит с сообщением
git commit -am "message"  # добавить и закоммитить измененные файлы
```

Просмотр истории:

```
git log                   # просмотреть историю коммитов
git log --oneline         # краткая история
git diff                  # просмотреть изменения
```

Работа с ветками:

```
git branch                # просмотреть список веток
git branch <name>         # создать новую ветку
git checkout <branch>     # переключиться на ветку
git checkout -b <branch>  # создать и переключиться на ветку
git merge <branch>        # слить ветку с текущей
git branch -d <branch>    # удалить ветку
```

Работа с удаленными репозиториями:

```
git remote add origin <url>   # добавить удаленный репозиторий
git push origin <branch>      # отправить изменения на сервер
git pull                  # получить изменения с сервера
git fetch                 # загрузить изменения без слияния
```

Отмена изменений:

```
git reset <file>          # убрать файл из индекса
git checkout -- <file>    # отменить изменения в файле
```

git revert <commit> # создать коммит, отменяющий изменения

Файл gitignore

Файл gitignore содержит список файлов и папок, которые Git должен игнорировать. Обычно это временные файлы, файлы сборки, конфигурационные файлы с паролями и т.д.

Пример gitignore для проектов C# в Visual Studio:

```
## Visual Studio
.vs/
bin/
obj/
*.user
*.suo
*.cache
```

```
## Build results
[Dd]ebug/
[Rr]elease/
x64/
x86/
```

```
## User-specific files
*.suo
*.user
*.userosscache
*.sln.docstates
```

Работа с Git в Visual Studio 2022

Visual Studio 2022 имеет встроенную поддержку Git, которая позволяет выполнять основные операции через графический интерфейс.

Основные возможности:

- создание и клонирование репозиториев;
- выполнение коммитов через окно Git Changes;
- управление ветками через Git Repository;
- просмотр истории коммитов;
- разрешение конфликтов слияния;
- синхронизация с удаленными репозиториями;
- интеграция с GitHub, Azure DevOps;

Окна Git в Visual Studio 2022:

- Git Changes — просмотр изменений и создание коммитов;
- Git Repository — управление ветками и просмотр истории;
- Team Explorer — альтернативный интерфейс для работы с Git;

Платформы для хостинга Git-репозиториев:
GitHub — крупнейшая платформа для хостинга open-source и частных проектов;
GitLab — платформа с возможностью развертывания на собственных серверах;
Bitbucket — платформа от Atlassian с интеграцией с Jira;
Azure Repos — сервис от Microsoft, интегрированный с Azure DevOps.
Рабочий процесс (Workflow) с Git
Базовый рабочий процесс:
1.Клонирование или создание репозитория;
2.Создание новой ветки для разработки функции;
3.Внесение изменений в код;
4.Добавление файлов в индекс (git add);
5.Создание коммита с описанием изменений;
6.Отправка изменений в удаленный репозиторий;
7.Создание Pull Request для ревью кода;
8.Слияние ветки с основной веткой (main/master).
Правила хороших коммитов:

- делайте атомарные коммиты (одна логическая единица изменений)
- пишите осмысленные сообщения коммитов
- используйте настоящее время ("Add feature", а не "Added feature")
- коммитьте часто, но только рабочий код
- не коммитьте сгенерированные файлы

Стратегии ветвления
Git Flow — классическая модель с ветками master, develop, feature, release, hotfix.
GitHub Flow — упрощенная модель с основной веткой и feature-ветками.

Trunk-Based Development — разработка в основной ветке с короткоживущими feature-ветками.

Рабочее задание: Выполните практическое задание по работе с системой контроля версий Git в соответствии с вариантом.

Вариант 1. Калькулятор

Создайте консольное приложение на C# — простой калькулятор с операциями сложения, вычитания, умножения и деления. Используйте Git для контроля версий, создавая отдельную ветку для каждой операции.

Вариант 2. Конвертер величин

Создайте консольное приложение на C# для конвертации величин (температура, длина, вес, объем). Используйте Git для контроля версий, создавая отдельную ветку для каждого типа конвертации.

Вариант 3. Работа с массивами

Создайте консольное приложение на C# с функциями обработки массивов (поиск минимума/максимума, сортировка, реверс, среднее значение). Используйте Git, создавая отдельную ветку для каждой функции.

Вариант 4. Генератор паролей

Создайте консольное приложение на C# — генератор паролей с возможностью настройки длины и типов символов. Используйте Git, создавая ветки для разных типов генерации (цифры, буквы, спецсимволы).

Вариант 5. Текстовый анализатор

Создайте консольное приложение на C# для анализа текста (подсчет слов, символов, предложений, частота символов). Используйте Git, создавая отдельную ветку для каждой функции анализа.

Вариант 6. База данных студентов

Создайте консольное приложение на C# для учета студентов с использованием списков (добавление, удаление, поиск, сортировка). Используйте Git, создавая ветки для каждой операции CRUD.

Вариант 7. Игра "Угадай число"

Создайте консольное приложение на C# — игру "Угадай число" с разными уровнями сложности. Используйте Git,

создавая ветки для добавления различных функций (подсказки, счет попыток, таблица рекордов).

Вариант 8. Работа с файлами

Создайте консольное приложение на C# для работы с текстовыми файлами (чтение, запись, добавление, поиск). Используйте Git, создавая отдельную ветку для каждой операции с файлами.

Вариант 9. Менеджер задач

Создайте консольное приложение на C# — простой менеджер задач (добавление, удаление, пометка выполненных, вывод списка). Используйте Git, создавая ветки для каждой функции управления задачами.

Вариант 10. Библиотека математических функций

Создайте консольное приложение на C# с математическими функциями (факториал, числа Фибоначчи, проверка на простоту, НОД). Используйте Git, создавая отдельную ветку для каждой функции.

Требования к выполнению рабочего задания

Часть 1. Установка и настройка Git

1.Скачайте и установите Git с официального сайта https://git-scm.com/

2.Настройте имя пользователя и email:

3.git config --global user.name "Ваше Имя"

4.git config --global user.email "your.email@example.com"

5.Проверьте настройки командой git config --list

Часть 2. Создание проекта в Visual Studio 2022

1.Создайте новый консольный проект C# в Visual Studio 2022

2.Присвойте проекту имя в соответствии с вашим вариантом

3.Реализуйте базовую структуру программы (меню, основной цикл)

Часть 3. Инициализация Git-репозитория

Вариант А (через командную строку):

1.Откройте командную строку в папке проекта

2.Выполните команду git init

3.Создайте файл .gitignore для Visual Studio

4.Выполните первый коммит:

5.git add.

6.git commit -m "Initial commit: project structure"

Вариант Б (через Visual Studio):

1.Откройте меню Git → Create Git Repository
2.Выберите Local repository only
3.Убедитесь, что .gitignore создан автоматически
4.Создайте первый коммит через окно Git Changes

Часть 4. Работа с ветками

1.Создайте ветку develop для разработки:
2.git branch develop
3.git checkout develop или git checkout -b develop
4.Для каждой функции вашего приложения:
– Создайте отдельную ветку от develop (например, feature/add-operation)
– Реализуйте функцию
– Сделайте коммит с описанием изменений
– Переключитесь на ветку develop
– Выполните слияние ветки функции с develop
– Удалите ветку функции
5.Создайте минимум *4 функциональные ветки* и выполните их слияние

Часть 5. Работа с историей

1.Просмотрите историю коммитов командой git log или в Visual Studio
2.Сделайте скриншоты истории коммитов и графа веток
3.Выполните команду git log --graph --oneline --all и сохраните результат

Часть 6. Работа с удаленным репозиторием (опционально)

1.Создайте аккаунт на GitHub (если его нет)
2.Создайте новый пустой репозиторий на GitHub
3.Подключите удаленный репозиторий:
4.git remote add origin <URL вашего репозитория>
5.Отправьте все ветки на GitHub:
6.git push -u origin --all

Часть 7. Демонстрация конфликта и его разрешение

1.Создайте две ветки от develop: branch-a и branch-b
2.В обеих ветках измените одну и ту же строку кода по-разному

3.Сделайте коммиты в обеих ветках
4.Слейте branch-a с develop
5.Попытайтесь слить branch-b с develop — возникнет конфликт
6.Разрешите конфликт вручную
7.Завершите слияние коммитом
8.Сделайте скриншоты процесса разрешения конфликта

Требования к отчету

Оформите отчет, содержащий:
1.Ход выполнения работы:
– Скриншот результата команды git config --list
– Описание созданного проекта (вариант задания)
– Скриншоты основных команд Git с результатами их выполнения
– Список созданных веток с описанием их назначения
– Скриншоты окон Git Changes и Git Repository в Visual Studio
– История коммитов (скриншот git log --graph)
– Процесс разрешения конфликта (скриншоты)
– Ссылка на GitHub-репозиторий (при выполнении часть 6)
2.Листинг программного кода основных файлов проекта
3.Выводы по работе: какие команды Git вы освоили; какие сложности возникли при работе с ветками; как вы разрешили конфликт слияния; преимущества использования Git для разработки ПО

Контрольные вопросы

1.Что такое система контроля версий и для чего она используется в разработке программного обеспечения? В чем отличие распределенных систем контроля версий от централизованных?

2.Что такое коммит в Git? Какую информацию он содержит? Какие правила следует соблюдать при создании коммитов?

3.Объясните разницу между командами git add, git commit, git push и git pull. Для чего используется каждая из них?

4.Что такое ветка (branch) в Git? Зачем используются ветки при разработке? Опишите процесс создания ветки, работы в ней и слияния с основной веткой.

5.Что такое конфликт слияния (merge conflict)? Когда он возникает и как его разрешить?

6.Для чего используется файл .gitignore? Какие файлы обычно добавляют в .gitignore для проектов на C# в Visual Studio?

7.В чем преимущества интеграции Git с Visual Studio 2022 по сравнению с работой через командную строку?

8.Что такое удаленный репозиторий? Назовите популярные платформы для хостинга Git-репозиториев. В чем разница между командами git fetch и git pull?

9.Опишите типичный рабочий процесс (workflow) разработки с использованием Git в команде разработчиков.

10.Какие преимущества дает использование систем контроля версий при разработке программных модулей? Приведите примеры ситуаций, когда Git помог бы избежать потери данных или ошибок.

Практическая работа № 3 «Использование инструментальных средств документирования алгоритмов программ»

Цель работы: Изучить инструментальные средства для документирования алгоритмов программ, освоить стандарты оформления блок-схем алгоритмов в соответствии с ГОСТ 19.701-90, научиться использовать специализированные программные средства для создания и документирования алгоритмов на языке C#.

Время выполнения: 2 ч

Теоретическая часть

Документирование алгоритмов программ является неотъемлемой частью разработки программного обеспечения. Качественная документация алгоритмов позволяет:

- понять логику работы программы без анализа исходного кода;
- упростить процесс сопровождения и модификации программы;
- обеспечить передачу знаний между разработчиками;

- провести верификацию корректности алгоритма до начала кодирования;
- использовать в качестве учебного материала.

Стандарты документирования алгоритмов

В Российской Федерации документирование алгоритмов регламентируется следующими стандартами:

ГОСТ 19.701-90 «Единая система программной документации. Схемы алгоритмов, программ, данных и систем. Обозначения условные и правила выполнения» — определяет графические символы для построения блок-схем.

ГОСТ 19.402-78 «Единая система программной документации. Описание программы» — определяет состав и содержание документа с описанием программы.

ГОСТ 19.404-79 «Единая система программной документации. Пояснительная записка. Требования к содержанию и оформлению» — регламентирует оформление пояснительной записки.

Основные элементы блок-схем по ГОСТ 19.701-90

Символ	Название	Назначение
Овал	Терминатор	Начало или конец алгоритма
Прямоугольник	Процесс	Выполнение операции или группы операций
Ромб	Решение	Проверка условия, выбор направления выполнения
Параллелограмм	Ввод/вывод	Ввод или вывод данных
Прямоугольник с двойными вертикальными линиями	Предопределенный процесс	Вызов подпрограммы, функции
Шестиугольник	Подготовка	Инициализация, модификация команды

Символ	Название	Назначение
Стрелки	Линии потока	Указание последовательности выполнения операций

Инструментальные средства документирования алгоритмов

1. Microsoft Visio. Профессиональный инструмент для создания диаграмм и блок-схем. Содержит готовые шаблоны и библиотеки символов, соответствующих ГОСТ 19.701-90.

Преимущества: богатый функционал; профессиональное оформление; интеграция с Microsoft Office; экспорт в различные форматы.

Недостатки: платное программное обеспечение; требует установки на компьютер.

2. Draw.io (diagrams.net). Бесплатный онлайн-редактор диаграмм с возможностью работы в браузере или установки десктопной версии.

Преимущества: бесплатность; кроссплатформенность; не требует регистрации; сохранение в различных форматах; возможность работы офлайн.

Недостатки: требуется настройка символов под ГОСТ; менее профессиональный интерфейс по сравнению с Visio.

3. Flowgorithm. Специализированный инструмент для создания алгоритмов в виде блок-схем с возможностью их выполнения и генерации кода на различных языках программирования, включая C#.

Преимущества: бесплатность; возможность выполнения алгоритма; генерация кода на C#; образовательная направленность.

Недостатки: ограниченные возможности оформления; символы не полностью соответствуют ГОСТ.

4. yEd Graph Editor. Мощный бесплатный редактор диаграмм с автоматическим размещением элементов.

Преимущества: бесплатность; автоматическое выравнивание

элементов; экспорт в различные форматы; кроссплатформенность.

5. Встроенные средства Visual Studio. Visual Studio 2022 предоставляет инструменты для визуализации кода:

- **Code Map** — визуализация структуры и зависимостей кода;
- **Class Designer** — диаграммы классов;
- **Architecture Explorer** — анализ архитектуры решения.

Комментарии в коде C#. Документирование непосредственно в исходном коде осуществляется с помощью комментариев:

Однострочные комментарии

```
// Это однострочный комментарий
int sum = a + b; // Вычисление суммы
```

Многострочные комментарии

```
/* Это многострочный комментарий,
   который может занимать
   несколько строк */
```

XML-комментарии (документирующие комментарии)

XML-комментарии используются для автоматической генерации документации:

```
/// <summary>
/// Вычисляет сумму двух целых чисел
/// </summary>
/// <param name="a">Первое слагаемое</param>
/// <param name="b">Второе слагаемое</param>
/// <returns>Сумма чисел a и b</returns>
public int Add(int a, int b)
{
    return a + b;
}
```

Основные теги XML-комментариев:

- <summary> — краткое описание элемента;
- <param> — описание параметра метода;
- <returns> — описание возвращаемого значения;
- <remarks> — дополнительная информация;
- <example> — пример использования;

- <exception> — описание исключений;
- <see> — ссылка на другой элемент;
- <code> — пример кода.

Генерация документации из XML-комментариев

Visual Studio позволяет генерировать XML-файл документации:

1.Откройте свойства проекта (правый клик по проекту → Properties)
2.Перейдите на вкладку Build → Output
3.Установите флажок "Documentation file"
4.Укажите путь для сохранения XML-файла

Для генерации документации в формате HTML можно использовать инструменты:

- Sandcastle Help File Builder — создание файлов справки в различных форматах;
- DocFX — современный генератор документации от Microsoft;
- Doxygen — универсальный генератор документации, поддерживающий C#.

Правила оформления блок-схем

1.Направление потока**:** сверху вниз и слева направо.
2.Линии потока: должны быть параллельны линиям рамки.
3.Пересечения**:** следует избегать пересечения линий.
4.Размеры**:** символы должны быть соразмерными и читаемыми.
Подписи: внутри символов используется краткое описание операции.
5.Нумерация: символы могут быть пронумерованы для ссылок в тексте.

Уровни детализации алгоритмов

Укрупненный алгоритм — отражает основные этапы решения задачи без детализации. Детальный алгоритм — показывает все операции, включая проверки, циклы, вызовы функций. Алгоритм подпрограммы — детализирует работу отдельной функции или метода.

Рабочее задание: Выполните документирование алгоритмов программы в соответствии с вариантом задания, используя инструментальные средства и стандарты оформления.

Вариант 1. Поиск максимального элемента в массиве

Разработать программу на C# для поиска максимального элемента в одномерном массиве целых чисел. Создать блок-схему алгоритма, добавить XML-комментарии в код, сгенерировать документацию.

Вариант 2. Сортировка массива методом пузырька

Разработать программу на C# для сортировки одномерного массива целых чисел методом пузырька. Создать блок-схему алгоритма, добавить XML-комментарии в код, сгенерировать документацию.

Вариант 3. Вычисление факториала числа

Разработать программу на C# для вычисления факториала натурального числа (рекурсивный и итеративный варианты). Создать блок-схемы алгоритмов, добавить XML-комментарии в код, сгенерировать документацию.

Вариант 4. Проверка числа на простоту

Разработать программу на C# для проверки, является ли заданное число простым. Создать блок-схему алгоритма, добавить XML-комментарии в код, сгенерировать документацию.

Вариант 5. Вычисление суммы элементов массива с заданным условием

Разработать программу на C# для вычисления суммы положительных элементов массива. Создать блок-схему алгоритма, добавить XML-комментарии в код, сгенерировать документацию.

Вариант 6. Поиск элемента в массиве (линейный поиск)

Разработать программу на C# для поиска заданного элемента в массиве методом линейного поиска. Создать блок-схему алгоритма, добавить XML-комментарии в код, сгенерировать документацию.

Вариант 7. Переворот массива

Разработать программу на C# для переворота одномерного массива (изменение порядка элементов на обратный). Создать блок-схему алгоритма, добавить XML-комментарии в код, сгенерировать документацию.

Вариант 8. Вычисление чисел Фибоначчи
Разработать программу на C# для вычисления n-го числа Фибоначчи (рекурсивный и итеративный варианты). Создать блок-схемы алгоритмов, добавить XML-комментарии в код, сгенерировать документацию.
Вариант 9. Подсчет количества четных и нечетных элементов массива
Разработать программу на C# для подсчета количества четных и нечетных элементов в массиве. Создать блок-схему алгоритма, добавить XML-комментарии в код, сгенерировать документацию.
Вариант 10. Объединение двух отсортированных массивов
Разработать программу на C# для объединения двух отсортированных массивов в один отсортированный массив. Создать блок-схему алгоритма, добавить XML-комментарии в код, сгенерировать документацию.

Требования к выполнению задания

1. Разработка программы на C#
– Создайте консольное приложение в Visual Studio 2022
– Реализуйте алгоритм согласно варианту задания
– Код должен быть структурированным и читаемым
– Используйте осмысленные имена переменных и методов
2. Добавление XML-комментариев
– Добавьте XML-комментарии ко всем публичным методам
– Используйте теги: <summary>, <param>, <returns>, <remarks>, <example>
– Опишите назначение метода, параметры, возвращаемое значение
– Добавьте примеры использования там, где это необходимо
3. Создание блок-схемы алгоритма
Используя один из инструментов (Draw.io, Visio, Flowgorithm, yEd):
– Создайте блок-схему основного алгоритма
– Используйте символы в соответствии с ГОСТ 19.701-90
– Обеспечьте читаемость и логичность схемы
– Соблюдайте направление потока (сверху вниз)
– Избегайте пересечения линий

– Сохраните блок-схему в формате изображения (PNG, JPEG или PDF)

4. Генерация документации

– Включите генерацию XML-файла документации в настройках проекта

– Убедитесь, что XML-файл создается при сборке проекта

– Сохраните сгенерированный XML-файл

5. Оформление отчета

Отчет должен содержать:

1.Постановку задачи: описание варианта задания; входные и выходные данные;

2.Блок-схема алгоритма: блок-схема, созданная в графическом редакторе; пояснения к основным блокам

3.Исходный код программы с XML-комментариями: полный листинг программы; выделение XML-комментариев

4.Результаты тестирования: скриншоты выполнения программы; примеры входных и выходных данных

5.Сгенерированная документация: фрагмент XML-файла документации

6.Выводы по работе: какие инструменты использовались; какие навыки приобретены; возможности применения изученных средств

Контрольные вопросы

1.Какой ГОСТ регламентирует обозначения условных графических символов для блок-схем алгоритмов? Перечислите основные символы и их назначение.

2.В чем заключается различие между однострочными, многострочными и XML-комментариями в C#? Когда следует использовать каждый тип?

3.Какие основные теги используются в XML-комментариях C#? Приведите примеры их использования.

4.Перечислите инструментальные средства для создания блок-схем алгоритмов. Сравните их преимущества и недостатки.

5.Как настроить Visual Studio 2022 для автоматической генерации XML-файла документации? Какие инструменты

можно использовать для преобразования XML-документации в HTML-формат?

6.Какие уровни детализации используются при документировании алгоритмов? В каких случаях применяется каждый уровень?

7.Почему качественное документирование алгоритмов важно для разработки и сопровождения программного обеспечения? Приведите конкретные примеры.

8.Какие правила оформления блок-схем необходимо соблюдать согласно ГОСТ 19.701-90?

Практическая работа № 4 «Определение сложности алгоритмов сортировки, поиска, рекурсивных и эвристических алгоритмов»

Цель работы: Изучить методы анализа временной и пространственной сложности алгоритмов, освоить практические навыки определения и сравнения сложности алгоритмов сортировки, поиска, рекурсивных и эвристических алгоритмов с использованием языка программирования C# в среде Visual Studio 2022.

Время выполнения: 4 ч

Теоретическая часть

Понятие сложности алгоритма

Сложность алгоритма — это количественная характеристика, определяющая объем ресурсов, необходимых для выполнения алгоритма. Основными видами сложности являются:

- Временная сложность — количество элементарных операций, выполняемых алгоритмом в зависимости от размера входных данных
- Пространственная сложность — объем памяти, необходимый для работы алгоритма

Асимптотическая нотация (О-нотация)

Для описания сложности алгоритмов используется асимптотическая нотация «О» (О-большое), которая показывает верхнюю границу роста функции при увеличении размера входных данных.

Основные классы сложности (от лучшего к худшему):

- **O(1)** — константная сложность (операция выполняется за постоянное время)
- **O(log n)** — логарифмическая сложность (бинарный поиск)
- **O(n)** — линейная сложность (простой перебор)
- **O(n log n)** — логлинейная сложность (эффективные сортировки)
- **O(n^2)** — квадратичная сложность (простые сортировки)
- **O(n^3)** — кубическая сложность
- **O(2^n)** — экспоненциальная сложность (перебор всех подмножеств)
- **O(n!)** — факториальная сложность (перебор всех перестановок)

Виды оценок сложности

- **Лучший случай** (Best Case) — минимальное время выполнения при наиболее благоприятных входных данных
- **Средний случай** (Average Case) — среднее время выполнения для типичных входных данных
- **Худший случай** (Worst Case) — максимальное время выполнения при наименее благоприятных входных данных

Сложность алгоритмов сортировки

1. Простые алгоритмы сортировки (O(n^2))

Сортировка пузырьком (Bubble Sort)

Принцип: последовательное сравнение и обмен соседних элементов.

- Временная сложность:
  –Лучший случай: O(n) — массив уже отсортирован
  –Средний случай: O(n^2)
  –Худший случай: O(n^2)
- Пространственная сложность: O(1)
- Стабильность: да

Сортировка вставками (Insertion Sort)

Принцип: построение отсортированной последовательности путем вставки элементов в нужную позицию.

- Временная сложность:
  - Лучший случай: O(n)

- Средний случай: $O(n^2)$
- Худший случай: $O(n^2)$

- Пространственная сложность: O(1)
- Стабильность: да

Сортировка выбором (Selection Sort)

Принцип: поиск минимального элемента и помещение его в начало неотсортированной части.

- Временная сложность: $O(n^2)$ во всех случаях
- Пространственная сложность: O(1)
- Стабильность: нет

2. Эффективные алгоритмы сортировки (O(n log n))

Быстрая сортировка (Quick Sort)

Принцип: разделение массива относительно опорного элемента и рекурсивная сортировка частей.

- Временная сложность:

–Лучший случай: O(n log n)

–Средний случай: O(n log n)

–Худший случай: $O(n^2)$ — при неудачном выборе опорного элемента

- Пространственная сложность: O(log n) — за счет рекурсии
- Стабильность: нет

Сортировка слиянием (Merge Sort)

Принцип: разделение массива пополам, рекурсивная сортировка частей и их слияние.

- Временная сложность: O(n log n) во всех случаях
- Пространственная сложность: O(n)
- Стабильность: да

Пирамидальная сортировка (Heap Sort)

Принцип: построение двоичной кучи и последовательное извлечение максимального элемента.

- Временная сложность: O(n log n) во всех случаях
- Пространственная сложность: O(1)
- Стабильность: нет

Сложность алгоритмов поиска

Линейный поиск. Принцип: последовательный просмотр всех элементов.

- Временная сложность:
  –Лучший случай: O(1) — элемент в начале
  –Средний случай: O(n)
  –Худший случай: O(n) — элемент в конце/ отсутствует
- Пространственная сложность: O(1)
- Требования: работает с любым массивом

Бинарный (двоичный) поиск. Принцип: деление отсортированного массива пополам и сравнение с искомым элементом.

- Временная сложность:
  –Лучший случай: O(1)
  –Средний и худший случай: O(log n)
- Пространственная сложность: O(1) для итеративной реализации, O(log n) для рекурсивной
- Требования: массив должен быть отсортирован

Интерполяционный поиск. Принцип: использование интерполяции для предсказания позиции искомого элемента.

- Временная сложность:
  –Лучший и средний случай: O(log log n) — для равномерно распределенных данных
  –Худший случай: O(n)
- Пространственная сложность: O(1)
- Требования: отсортированный массив с равномерным распределением

Рекурсивные алгоритмы

Рекурсия — способ организации алгоритма, при котором функция вызывает саму себя. *Типы рекурсии:*

1.Прямая рекурсия — функция вызывает саму себя непосредственно

2.Косвенная рекурсия — функция A вызывает функцию B, которая вызывает функцию A

3.Хвостовая рекурсия — рекурсивный вызов является последней операцией

Примеры рекурсивных алгоритмов:

– Вычисление факториала

Временная сложность: O(n)

Пространственная сложность: O(n) — глубина стека вызовов

– Числа Фибоначчи (наивная реализация)

Временная сложность: $O(2^n)$ — экспоненциальная
Пространственная сложность: O(n) — глубина рекурсии
– Числа Фибоначчи (с мемоизацией)
Временная сложность: O(n)
Пространственная сложность: O(n)
– Обход дерева
Временная сложность: O(n) — где n количество узлов
Пространственная сложность: O(h) — где h высота дерева
Эвристические алгоритмы
Эвристический алгоритм — алгоритм, который находит приближенное решение задачи за разумное время, когда точное решение требует неприемлемо больших вычислительных ресурсов. *Характеристики эвристических алгоритмов:*

- Не гарантируют нахождение оптимального решения
- Работают значительно быстрее точных алгоритмов
- Часто дают решения, близкие к оптимальным
- Используются для NP-полных задач

Примеры эвристических алгоритмов:
1. Жадные алгоритмы (Greedy Algorithms). Принцип: на каждом шаге выбирается локально оптимальное решение. Пример — задача о рюкзаке (дробный вариант):

- Временная сложность: O(n log n)
- Не гарантирует оптимум для целочисленного варианта

2. Алгоритм поиска A*. Используется для поиска кратчайшего пути в графе. Принцип: использование эвристической функции для оценки расстояния до цели.

- Временная сложность: зависит от эвристики, в худшем случае $O(b^d)$, где b — фактор ветвления, d — глубина решения
- Пространственная сложность: $O(b^d)$

3. Генетические алгоритмы. Принцип: имитация процесса естественного отбора.

- Временная сложность: $O(g \times n \times f)$, где g — количество поколений, n — размер популяции, f — сложность функции приспособленности
- Не гарантируют нахождение глобального оптимума

4. Метод отжига (Simulated Annealing). Принцип: случайный

поиск с постепенным уменьшением вероятности принятия худших решений. Сложность зависит от параметров алгоритма. Может избежать локальных минимумов

Методы определения сложности алгоритма

1. Теоретический анализ

- Подсчет количества элементарных операций
- Анализ вложенных циклов
- Решение рекуррентных соотношений (для рекурсивных алгоритмов)

Правила анализа циклов:

- Один цикл от 1 до n: O(n)
- Два вложенных цикла от 1 до n: $O(n^2)$
- Цикл с делением пополам: O(log n)
- Цикл от 1 до n с внутренним циклом от 1 до i: $O(n^2)$

2. Экспериментальный анализ

- Измерение времени выполнения на разных размерах входных данных
- Построение графиков зависимости времени от размера данных
- Использование класса Stopwatch в C#

3. Использование мастер-теоремы для рекуррентных соотношений

Для рекуррентных соотношений вида $T(n) = aT(n/b) + f(n)$:

- a — количество подзадач
- n/b — размер каждой подзадачи
- f(n) — стоимость разделения и объединения

Примеры кода на C#

Пример 1: Измерение времени выполнения алгоритма

```
using System;
using System.Diagnostics;

class Program
{
   static void Main()
   {
      int[] sizes = { 1000, 5000, 10000, 50000 };

      foreach (int size in sizes)
```

```
        {
            int[] array = GenerateRandomArray(size);

            Stopwatch sw = Stopwatch.StartNew();
            BubbleSort(array);
            sw.Stop();

            Console.WriteLine($"Размер: {size}, Время: {sw.ElapsedMilliseconds}
мс");
        }
    }

    static int[] GenerateRandomArray(int size)
    {
        Random rand = new Random();
        int[] array = new int[size];
        for (int i = 0; i < size; i++)
            array[i] = rand.Next(1, 10000);
        return array;
    }

    static void BubbleSort(int[] array)
    {
        int n = array.Length;
        for (int i = 0; i < n - 1; i++)
            for (int j = 0; j < n - i - 1; j++)
                if (array[j] > array[j + 1])
                {
                    int temp = array[j];
                    array[j] = array[j + 1];
                    array[j + 1] = temp;
                }
    }
}
```

Пример 2: Подсчет операций

```
static long BubbleSortWithCounter(int[] array)
{
    long operations = 0;
    int n = array.Length;

    for (int i = 0; i < n - 1; i++)
    {
        for (int j = 0; j < n - i - 1; j++)
        {
```

```
            operations++; // Счетчик сравнений
            if (array[j] > array[j + 1])
            {
                int temp = array[j];
                array[j] = array[j + 1];
                array[j + 1] = temp;
                operations += 3; // Три операции присваивания
            }
        }
    }

    return operations;
}
```

Рабочее задание: Разработайте программу на языке C# в среде Visual Studio 2022 для сравнительного анализа сложности алгоритмов в соответствии с вариантом задания.

Вариант 1. Сравнение алгоритмов сортировки простых типов

Реализовать и сравнить алгоритмы: сортировка пузырьком, сортировка вставками, быстрая сортировка. Провести анализ на массивах размером 100, 500, 1000, 5000, 10000 элементов для трех случаев: случайные данные, отсортированные данные, обратно отсортированные данные.

Вариант 2. Сравнение алгоритмов сортировки сложных объектов

Реализовать и сравнить алгоритмы: сортировка выбором, сортировка слиянием, пирамидальная сортировка для массива объектов класса Student (поля: ФИО, группа, средний балл). Сортировка по среднему баллу.

Вариант 3. Сравнение алгоритмов поиска

Реализовать и сравнить: линейный поиск, бинарный поиск (итеративный), бинарный поиск (рекурсивный), интерполяционный поиск. Провести анализ на отсортированных массивах размером 1000, 10000, 100000, 1000000 элементов.

Вариант 4. Анализ рекурсивных алгоритмов

Реализовать и сравнить рекурсивные алгоритмы: вычисление факториала (рекурсивное и итеративное), числа Фибоначчи (наивная рекурсия, с мемоизацией, итеративная), возведение в

степень (наивное, быстрое возведение). Измерить время и глубину рекурсии.

Вариант 5. Анализ алгоритмов на строках

Реализовать и сравнить алгоритмы поиска подстроки: наивный алгоритм, алгоритм Кнута-Морриса-Пратта, алгоритм Бойера-Мура. Провести анализ на текстах разной длины (100, 1000, 10000, 100000 символов).

Вариант 6. Жадные алгоритмы

Реализовать жадные алгоритмы для задач: размен монет (минимальное количество монет), задача о рюкзаке (дробный вариант), задача о расписании (максимальное количество непересекающихся интервалов). Сравнить с точными решениями малых размеров.

Вариант 7. Алгоритмы на графах

Реализовать и сравнить алгоритмы поиска кратчайшего пути: алгоритм Дейкстры, алгоритм Беллмана-Форда, поиск в ширину (BFS). Провести анализ на графах с различным количеством вершин (10, 50, 100, 500).

Вариант 8. Алгоритмы обхода дерева

Реализовать рекурсивные и итеративные версии обхода бинарного дерева: прямой обход (preorder), центрированный обход (inorder), обратный обход (postorder). Сравнить сложность и производительность на деревьях разной глубины.

Вариант 9. Эвристический алгоритм — поиск A*

Реализовать алгоритм A* для поиска пути на двумерной сетке с препятствиями. Сравнить с алгоритмом поиска в ширину (BFS). Использовать разные эвристические функции (манхэттенское расстояние, евклидово расстояние).

Вариант 10. Генетический алгоритм

Реализовать простой генетический алгоритм для решения задачи коммивояжера (небольшое количество городов — 5-10). Исследовать влияние параметров (размер популяции, вероятность мутации, количество поколений) на качество решения и время работы.

Требования к выполнению задания

1. Реализация алгоритмов

–Создать консольное приложение на C# в Visual Studio 2022

–Реализовать все алгоритмы согласно варианту задания
–Обеспечить корректность работы алгоритмов (проверить на тестовых данных)
–Код должен быть структурирован, прокомментирован
2. Измерение производительности. Для каждого алгоритма измерить:

- Время выполнения — использовать класс Stopwatch
- Количество операций — ввести счетчики для ключевых операций (сравнения, присваивания, обмены)
- Использование памяти — теоретически оценить пространственную сложность

3. Тестирование. Провести тестирование на различных наборах данных:

- Разные размеры входных данных (минимум 4-5 различных размеров)
- Разные типы данных (случайные, упорядоченные, обратно упорядоченные — где применимо)
- Граничные случаи (пустые массивы, массивы из одного элемента)

4. Анализ результатов. Провести сравнительный анализ–:
–Построить таблицы с результатами измерений
–Определить теоретическую сложность каждого алгоритма
–Сравнить теоретическую и практическую сложность
–Определить, при каких условиях какой алгоритм эффективнее
–Построить графики зависимости времени выполнения от размера данных (можно в Excel или с помощью библиотек визуализации)
5. Структура программы. Программа должна содержать:
Класс с реализацией алгоритмов; Класс для генерации тестовых данных; Класс для измерения производительности; Класс для вывода результатов. Главный класс Program с методом Main()

Оформление отчета

1.Отчет должен содержать описание: понятие сложности алгоритма; О-нотацию; теоретическую сложность каждого реализованного алгоритма; принцип работы каждого алгоритма (своими словами)

2. Исходный код программы. Привести полный код программы с комментариями. Использовать форматирование кода.

3. Описание реализации**.** Описать структуру программы, назначение основных классов и методов.

4. Результаты тестирования**.** Привести таблицы с результатами измерений для каждого алгоритма:

Размер данных	**Время выполнения (мс)**	**Количество операций**	**Примечания**
100			
500			
1000			
...			

5.Графики. Построить графики зависимости времени выполнения от размера входных данных для всех сравниваемых алгоритмов на одной координатной плоскости.

6.Сравнительный анализ. Провести сравнение алгоритмов: сравнить теоретическую и практическую сложность; определить, какой алгоритм быстрее на разных объемах данных; объяснить полученные результаты; указать преимущества и недостатки каждого алгоритма; дать рекомендации по применению алгоритмов

7 Скриншоты. Привести скриншоты: Окна Visual Studio с кодом программы; Результатов выполнения программы; Графиков (если построены в программе)

8. Выводы. Сформулировать выводы по работе: какие навыки были получены; какие алгоритмы были реализованы и исследованы; какие результаты были получены; какие закономерности обнаружены.

Контрольные вопросы

1.Что такое временная и пространственная сложность алгоритма? Приведите примеры алгоритмов с различной сложностью.

2.Объясните O-нотацию. Что означают обозначения O(1), O(n), O(n^2), O(log n), O(n log n)? Расположите их в порядке возрастания сложности.

3.В чем различие между лучшим, средним и худшим случаями выполнения алгоритма? Приведите пример алгоритма, для которого эти случаи существенно различаются.

4.Сравните алгоритмы сортировки пузырьком и быстрой сортировки. В каких случаях каждый из них предпочтительнее?

5.Почему бинарный поиск работает быстрее линейного? Какое условие необходимо для применения бинарного поиска?

6.Что такое рекурсия? Каковы преимущества и недостатки рекурсивных алгоритмов по сравнению с итеративными?

7.Объясните понятие «стек вызовов» в контексте рекурсивных алгоритмов. Как глубина рекурсии влияет на использование памяти?

8.Что такое эвристические алгоритмы? Приведите примеры задач, для которых целесообразно использовать эвристические подходы.

9.В чем заключается принцип работы жадных алгоритмов? Всегда ли жадный алгоритм дает оптимальное решение?

10.Как экспериментально определить сложность неизвестного алгоритма? Какие измерения необходимо провести?

11.Что показывает мастер-теорема? Для каких алгоритмов она применима?

12.Объясните, почему сортировка слиянием имеет сложность $O(n \log n)$, а сортировка пузырьком — $O(n^2)$.

13.Чем отличается стабильная сортировка от нестабильной? Приведите примеры алгоритмов обоих типов.

14.Как влияет выбор опорного элемента на производительность быстрой сортировки?

15.В каких практических задачах важно учитывать не только временную, но и пространственную сложность алгоритма?

Практическая работа № 5 «Работа с классами и перегрузка методов»

Цель работы: Изучить принципы объектно-ориентированного программирования в языке C#, освоить создание классов, работу с конструкторами, свойствами и методами, а также механизм перегрузки методов.

Время выполнения: 2 ч

Теоретическая часть

Основные понятия объектно-ориентированного программирования

Класс — это пользовательский тип данных, который объединяет данные (поля) и методы их обработки. Класс является шаблоном для создания объектов.

Объект — это экземпляр класса, конкретная реализация, созданная на основе описания класса.

Структура класса в C#

Класс в C# может содержать следующие элементы:

Поля (Fields) — переменные, хранящие состояние объекта;

Свойства (Properties) — специальные методы доступа к полям класса, обеспечивающие инкапсуляцию данных;

Конструкторы (Constructors) — специальные методы, вызываемые при создании объекта класса;

Методы (Methods) — функции, определяющие поведение объекта;

События (Events) — механизм уведомления других объектов об изменениях.

Синтаксис объявления класса

```
public class ИмяКласса
{
    // Поля
    private тип имяПоля;

    // Свойства
    public тип ИмяСвойства { get; set; }

    // Конструктор
    public ИмяКласса()
    {
        // Инициализация
    }
```

```
    // Методы
    public тип ИмяМетода(параметры)
    {
        // Реализация
    }
}
```

Модификаторы доступа

public — доступ из любой части программы.
private — доступ только внутри класса (используется по умолчанию).
protected — доступ внутри класса и производных классов.
internal — доступ внутри текущей сборки.

Конструкторы. Конструктор — это специальный метод класса, который вызывается автоматически при создании объекта. Конструктор имеет то же имя, что и класс, и не имеет типа возвращаемого значения.

Конструктор по умолчанию — не имеет параметров:

```
public class Student
{
    public Student()
    {
        // Инициализация по умолчанию
    }
}
```

Параметризованный конструктор — принимает параметры для инициализации полей:

```
public class Student
{
    private string name;
    private int age;

    public Student(string name, int age)
    {
        this.name = name;
        this.age = age;
    }
}
```

Перегрузка конструкторов

В классе может быть несколько конструкторов с разным количеством и типами параметров:

```
public class Student
```

```
{
    private string name;
    private int age;
    private string group;

    // Конструктор без параметров
    public Student()
    {
        name = "Неизвестно";
        age = 0;
        group = "Не назначена";
    }

    // Конструктор с двумя параметрами
    public Student(string name, int age)
    {
        this.name = name;
        this.age = age;
        this.group = "Не назначена";
    }

    // Конструктор с тремя параметрами
    public Student(string name, int age, string group)
    {
        this.name = name;
        this.age = age;
        this.group = group;
    }
}
```

Свойства

Свойства обеспечивают контролируемый доступ к полям класса. Свойства используют акцессоры **get** (для чтения) и **set** (для записи):

```
public class Student
{
    private string name;

    // Полное свойство с проверкой
    public string Name
    {
        get { return name; }
        set
        {
            if (!string.IsNullOrEmpty(value))
```

```
            name = value;
        }
    }

    // Автоматическое свойство
    public int Age { get; set; }

    // Свойство только для чтения
    public string Group { get; private set; }
}
```

Перегрузка методов. Перегрузка методов — это возможность создания нескольких методов с одинаковым именем, но с разными параметрами (количество, типы или порядок). Перегрузка методов позволяет: упростить интерфейс класса; сделать код более читаемым; обеспечить гибкость использования методов.

Правила перегрузки методов: 1. Методы должны иметь одинаковое имя; 2. Методы должны отличаться сигнатурой (количеством, типами или порядком параметров); 3.Тип возвращаемого значения не учитывается при перегрузке.

Примеры перегрузки методов:

```
public class Calculator
{
    // Сложение двух целых чисел
    public int Add(int a, int b)
    {
        return a + b;
    }

    // Сложение трех целых чисел
    public int Add(int a, int b, int c)
    {
        return a + b + c;
    }

    // Сложение двух вещественных чисел
    public double Add(double a, double b)
    {
        return a + b;
    }
```

```
    // Сложение массива чисел
    public int Add(params int[] numbers)
    {
        int sum = 0;
        foreach (int num in numbers)
            sum += num;
        return sum;
    }
}
```

Ключевое слово this. Ключевое слово **this** ссылается на текущий экземпляр класса и используется для:
–Разрешения конфликтов имен между параметрами и полями
–Вызова одного конструктора из другого

```
public class Student
{
    private string name;

    public Student(string name)
    {
        this.name = name; // this.name - поле, name - параметр
    }

    public Student() : this("Неизвестно") // вызов другого конструктора
    {
    }
}
```

Статические члены класса. Статические члены принадлежат классу в целом, а не отдельному объекту:

```
public class Student
{
    private static int totalStudents = 0; // статическое поле

    public static int TotalStudents // статическое свойство
    {
        get { return totalStudents; }
    }

    public Student()
    {
        totalStudents++; // увеличиваем счетчик при создании объекта
    }
```

```
    public static void ShowTotal() // статический метод
    {
        Console.WriteLine($"Всего студентов: {totalStudents}");
    }
}
```

Методы ToString() и Equals(). Рекомендуется переопределять стандартные методы класса Object:

```
public class Student
{
    public string Name { get; set; }
    public int Age { get; set; }

    // Переопределение ToString для строкового представления объекта
    public override string ToString()
    {
        return $"Студент: {Name}, возраст: {Age}";
    }

    // Переопределение Equals для сравнения объектов
    public override bool Equals(object obj)
    {
        if (obj is Student other)
            return this.Name == other.Name && this.Age == other.Age;
        return false;
    }
}
```

Рабочее задание: Выполните разработку программы в соответствии с вариантом задания. Программа должна демонстрировать работу с классами, использование конструкторов, свойств и перегрузку методов.

Вариант 1. Класс "Книга". Разработать класс **Book** для работы с информацией о книгах. Класс должен содержать:

Поля/свойства:

- Название книги
- Автор
- Год издания
- Количество страниц
- Цена

Конструкторы:

- Конструктор по умолчанию
- Конструктор с параметрами (название, автор)

- Конструктор с параметрами (название, автор, год, страницы, цена)

Методы:

- Метод **Display()** для вывода информации о книге (перегрузить для вывода полной и краткой информации)
- Метод **CalculateDiscount()** для расчета цены со скидкой (перегрузить для разных типов скидок: процент, фиксированная сумма)
- Метод **Compare()** для сравнения книг (перегрузить для сравнения по цене, году издания, количеству страниц)

Создать несколько объектов класса, продемонстрировать работу всех методов.

Вариант 2. Класс "Студент". Разработать класс **Student** для работы с информацией о студентах. Класс должен содержать:

Поля/свойства:

- ФИО
- Номер группы
- Курс
- Средний балл
- Возраст

Конструкторы:

- Конструктор по умолчанию
- Конструктор с параметрами (ФИО, группа)
- Конструктор с параметрами (ФИО, группа, курс, средний балл)

Методы:

- Метод **GetInfo()** для получения информации о студенте (перегрузить для разных форматов вывода)
- Метод **CalculateScholarship()** для расчета стипендии (перегрузить для разных критериев: только средний балл, средний балл и курс)
- Метод **IsExcellent()** для проверки успеваемости (перегрузить с разными пороговыми значениями)
- Статический метод для подсчета общего количества студентов

Создать несколько объектов класса, продемонстрировать работу всех методов.

Вариант 3. Класс "Банковский счет". Разработать класс **BankAccount** для работы с банковским счетом. Класс должен содержать:

Поля/свойства:

- Номер счета
- Владелец счета
- Баланс
- Тип счета (текущий, накопительный)
- Процентная ставка

Конструкторы:

- Конструктор по умолчанию
- Конструктор с параметрами (номер счета, владелец)
- Конструктор с параметрами (номер счета, владелец, начальный баланс)

Методы:

- Метод **Deposit()** для пополнения счета (перегрузить для разных валют или способов пополнения)
- Метод **Withdraw()** для снятия средств (перегрузить с проверкой минимального остатка и без)
- Метод **CalculateInterest()** для расчета процентов (перегрузить для разных периодов: месяц, год)
- Метод **Transfer()** для перевода средств (перегрузить с комиссией и без)

Создать несколько объектов класса, продемонстрировать работу всех методов.

Вариант 4. Класс "Прямоугольник". Разработать класс **Rectangle** для работы с прямоугольниками. Класс должен содержать:

Поля/свойства:

- Длина
- Ширина
- Цвет
- Координаты (X, Y)

Конструкторы:

- Конструктор по умолчанию (создает квадрат со стороной 1)
- Конструктор с параметрами (длина, ширина)

- Конструктор с параметрами (длина, ширина, цвет)
- Конструктор с параметрами (длина, ширина, координаты X, Y)

Методы:

- Метод **CalculateArea()** для расчета площади
- Метод **CalculatePerimeter()** для расчета периметра
- Метод **Resize()** для изменения размеров (перегрузить: изменение обеих сторон, изменение только одной стороны, изменение с сохранением пропорций)
- Метод **Compare()** для сравнения прямоугольников (перегрузить для сравнения по площади, периметру)
- Метод **IsSquare()** для проверки, является ли прямоугольник квадратом

Создать несколько объектов класса, продемонстрировать работу всех методов.

Вариант 5. Класс "Товар". Разработать класс **Product** для работы с информацией о товарах. Класс должен содержать:

Поля/свойства:

- Наименование
- Производитель
- Цена
- Количество на складе
- Категория

Конструкторы:

- Конструктор по умолчанию
- Конструктор с параметрами (наименование, цена)
- Конструктор с параметрами (наименование, производитель, цена, количество)

Методы:

- Метод **CalculateTotalCost()** для расчета общей стоимости (перегрузить: для всего количества, для указанного количества, с учетом скидки)
- Метод **UpdateQuantity()** для изменения количества (перегрузить для увеличения и уменьшения)
- Метод **Display()** для вывода информации (перегрузить для краткого и полного формата)

- Метод **ApplyDiscount()** для применения скидки (перегрузить для процентной и фиксированной скидки)

Создать несколько объектов класса, продемонстрировать работу всех методов.

Вариант 6. Класс "Сотрудник". Разработать класс **Employee** для работы с информацией о сотрудниках. Класс должен содержать:

Поля/свойства:

- ФИО
- Должность
- Оклад
- Стаж работы (лет)
- Отдел

Конструкторы:

- Конструктор по умолчанию
- Конструктор с параметрами (ФИО, должность)
- Конструктор с параметрами (ФИО, должность, оклад, стаж)

Методы:

- Метод **CalculateSalary()** для расчета зарплаты (перегрузить: базовая, с надбавкой за стаж, с премией)
- Метод **GetInfo()** для вывода информации (перегрузить для разных уровней детализации)
- Метод **GiveRaise()** для повышения оклада (перегрузить для процентного и фиксированного повышения)
- Статический метод для подсчета среднего оклада по всем сотрудникам

Создать несколько объектов класса, продемонстрировать работу всех методов.

Вариант 7. Класс "Автомобиль". Разработать класс **Car** для работы с информацией об автомобилях. Класс должен содержать:

Поля/свойства:

- Марка
- Модель
- Год выпуска
- Пробег (км)

- Цена

Конструкторы:

- Конструктор по умолчанию
- Конструктор с параметрами (марка, модель)
- Конструктор с параметрами (марка, модель, год, цена)

Методы:

- Метод **Drive()** для увеличения пробега (перегрузить для разных единиц измерения: км, мили)
- Метод **CalculateDepreciation()** для расчета износа (перегрузить: по годам, по пробегу, комбинированный)
- Метод **Display()** для вывода информации (перегрузить для краткого и подробного формата)
- Метод **Compare()** для сравнения автомобилей (перегрузить для сравнения по цене, году, пробегу)

Создать несколько объектов класса, продемонстрировать работу всех методов.

Вариант 8. Класс "Дата". Разработать класс **Date** для работы с датами. Класс должен содержать:

Поля/свойства:

- День
- Месяц
- Год

Конструкторы:

- Конструктор по умолчанию (устанавливает текущую дату)
- Конструктор с параметрами (день, месяц, год)
- Конструктор с параметром (строка в формате "дд.мм.гггг")

Методы:

- Метод **AddDays()** для добавления дней к дате (перегрузить для добавления дней, недель, месяцев)
- Метод **GetDifference()** для расчета разницы между датами (перегрузить для вывода в днях, неделях, месяцах)
- Метод **ToString()** для преобразования в строку (перегрузить для разных форматов)
- Метод **IsLeapYear()** для проверки високосного года
- Метод **Compare()** для сравнения дат

Создать несколько объектов класса, продемонстрировать работу всех методов.

Вариант 9. Класс "Время". Разработать класс **Time** для работы со временем. Класс должен содержать:

Поля/свойства:

- Часы
- Минуты
- Секунды

Конструкторы:

- Конструктор по умолчанию (устанавливает 00:00:00)
- Конструктор с параметрами (часы, минуты)
- Конструктор с параметрами (часы, минуты, секунды)
- Конструктор с параметром (общее количество секунд)

Методы:

- Метод **Add()** для добавления времени (перегрузить для добавления секунд, минут, часов, другого объекта Time)
- Метод **GetDifference()** для расчета разницы во времени (перегрузить для вывода в разных единицах)
- Метод **ToString()** для преобразования в строку (перегрузить для 12-часового и 24-часового формата)
- Метод **ToSeconds()** для преобразования в секунды
- Метод **Compare()** для сравнения времени

Создать несколько объектов класса, продемонстрировать работу всех методов.

Вариант 10. Класс "Круг". Разработать класс **Circle** для работы с окружностями. Класс должен содержать:

Поля/свойства:

- Радиус
- Координаты центра (X, Y)
- Цвет

Конструкторы:

- Конструктор по умолчанию (радиус = 1, центр в точке 0,0)
- Конструктор с параметром (радиус)
- Конструктор с параметрами (радиус, координаты X, Y)
- Конструктор с параметрами (радиус, координаты X, Y, цвет)

Методы:

- Метод **CalculateArea()** для расчета площади
- Метод **CalculateCircumference()** для расчета длины окружности
- Метод **Resize()** для изменения размера (перегрузить: по новому радиусу, по коэффициенту масштабирования)
- Метод **Contains()** для проверки принадлежности точки (перегрузить: для отдельных координат X и Y, для объекта Point)
- Метод **Compare()** для сравнения окружностей (перегрузить для сравнения по радиусу, площади)

Создать несколько объектов класса, продемонстрировать работу всех методов.

Требования к выполнению задания

1.Создайте новый проект Console Application в Visual Studio 2022

2.Разработайте класс в соответствии с вариантом задания, который должен включать:

–Не менее 4-5 полей/свойств

–Не менее 3 конструкторов

–Не менее 4 методов с перегрузкой (каждый метод должен иметь минимум 2 варианта)

–Переопределение метода ToString()

3.В методе Main() продемонстрируйте: Создание объектов с использованием разных конструкторов; Вызов всех вариантов перегруженных методов; Работу со свойствами класса; Использование статических членов (если есть)

4.Обеспечьте валидацию входных данных в конструкторах и методах

5.Добавьте комментарии к коду, поясняющие назначение полей, свойств и методов

6.Используйте осмысленные имена для переменных, методов и классов

7.Оформите вывод информации в консоль в удобном для восприятия виде

Пример выполнения задания

```
using System;
```

```
namespace PracticalWork02
{
    // Класс для представления книги
    public class Book
    {
        // Приватные поля
        private string title;
        private string author;
        private int year;
        private int pages;
        private decimal price;

        // Свойства с проверкой данных
        public string Title
        {
            get { return title; }
            set { title = string.IsNullOrEmpty(value) ? "Без названия" : value; }
        }

        public string Author
        {
            get { return author; }
            set { author = string.IsNullOrEmpty(value) ? "Неизвестен" : value; }
        }

        public int Year
        {
            get { return year; }
            set { year = (value > 0 && value <= DateTime.Now.Year) ? value :
DateTime.Now.Year; }
        }

        public int Pages
        {
            get { return pages; }
            set { pages = value > 0 ? value : 1; }
        }

        public decimal Price
        {
            get { return price; }
            set { price = value >= 0 ? value : 0; }
        }
```

```
// Конструктор по умолчанию
public Book()
{
    Title = "Без названия";
    Author = "Неизвестен";
    Year = DateTime.Now.Year;
    Pages = 0;
    Price = 0;
}

// Конструктор с двумя параметрами
public Book(string title, string author)
{
    Title = title;
    Author = author;
    Year = DateTime.Now.Year;
    Pages = 0;
    Price = 0;
}

// Конструктор с пятью параметрами
public Book(string title, string author, int year, int pages, decimal price)
{
    Title = title;
    Author = author;
    Year = year;
    Pages = pages;
    Price = price;
}

// Перегрузка метода Display - краткая информация
public void Display()
{
    Console.WriteLine($"{Title} - {Author}");
}

// Перегрузка метода Display - полная информация
public void Display(bool fullInfo)
{
    if (fullInfo)
    {
        Console.WriteLine($"Название: {Title}");
        Console.WriteLine($"Автор: {Author}");
        Console.WriteLine($"Год издания: {Year}");
```

```
            Console.WriteLine($"Страниц: {Pages}");
            Console.WriteLine($"Цена: {Price:C}");
        }
        else
        {
            Display();
        }
    }

    // Перегрузка метода CalculateDiscount - скидка в процентах
    public decimal CalculateDiscount(int percent)
    {
        if (percent < 0 || percent > 100)
            return Price;
        return Price * (100 - percent) / 100;
    }

    // Перегрузка метода CalculateDiscount - фиксированная скидка
    public decimal CalculateDiscount(decimal fixedAmount)
    {
        decimal newPrice = Price - fixedAmount;
        return newPrice >= 0 ? newPrice : 0;
    }

    // Переопределение ToString
    public override string ToString()
    {
        return $"{Title} ({Author}, {Year})";
    }
}

class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine("=== Практическая работа: Работа с классами
===\n");

        // Создание объектов с разными конструкторами
        Book book1 = new Book();
        Console.WriteLine("Книга 1 (конструктор по умолчанию):");
        book1.Display(true);

        Console.WriteLine("\n" + new string('-', 40) + "\n");
```

```
        Book book2 = new Book("Война и мир", "Л.Н. Толстой");
        book2.Year = 1869;
        book2.Pages = 1300;
        book2.Price = 850;
        Console.WriteLine("Книга 2:");
        book2.Display(true);

        Console.WriteLine("\n" + new string('-', 40) + "\n");

        Book book3 = new Book("Мастер и Маргарита", "М.А. Булгаков",
                              1967, 480, 650);
        Console.WriteLine("Книга 3:");
        book3.Display(true);

        // Демонстрация перегруженных методов
        Console.WriteLine("\n" + new string('=', 40));
        Console.WriteLine("Расчет скидок для книги 3:");
        Console.WriteLine($"Цена со скидкой 15%: {book3.CalculateDis-
count(15):C}");
        Console.WriteLine($"Цена со скидкой 100 руб: {book3.CalculateDis-
count(100m):C}");

        // Использование ToString
        Console.WriteLine("\n" + new string('=', 40));
        Console.WriteLine("Информация через ToString():");
        Console.WriteLine(book2.ToString());
        Console.WriteLine(book3.ToString());

        Console.WriteLine("\nНажмите любую клавишу для выхода...");
        Console.ReadKey();
    }
  }
}
```

Оформление отчета

Отчет должен содержать:

1.Описание разработанного класса: назначение класса; перечень полей/свойств с описанием; перечень конструкторов с описанием параметров; перечень методов с описанием назначения и вариантов перегрузки

2.Листинг программы с комментариями

3.Скриншоты результатов выполнения программы, демонстрирующие: Создание объектов с разными конструкторами; Работу всех перегруженных методов; Корректность обработки данных.

4.Выводы по работе: какие навыки были освоены; какие трудности возникли при выполнении; практическая значимость полученных знаний

Контрольные вопросы

1.Что такое класс и объект? В чем различие между этими понятиями?

2.Какие члены может содержать класс в C#? Опишите назначение каждого типа членов.

3.Что такое конструктор класса? Какие виды конструкторов существуют? Можно ли в классе определить несколько конструкторов?

4.В чем заключается различие между полями и свойствами? Какие преимущества дает использование свойств?

5.Перечислите модификаторы доступа в C#. В каких случаях применяется каждый из них?

6.Что такое перегрузка методов? По каким правилам она осуществляется? Может ли тип возвращаемого значения служить признаком перегрузки?

7.Для чего используется ключевое слово **this**? Приведите примеры использования.

8.Что такое статические члены класса? Их отличия от обычных членов? Приведите примеры использования.

9.Зачем переопределять методы **ToString()** и **Equals()**? Как это делается?

10.В чем преимущества объектно-ориентированного подхода к программированию по сравнению с процедурным?

Практическая работа № 6 «Определение операций в классе»

Цель работы: Изучить принципы перегрузки операторов в C#, освоить способы определения унарных и бинарных операций в пользовательских классах, научиться реализовывать операции

преобразования типов в среде Visual Studio 2022.

Время выполнения: 2 ч

Теоретическая часть

Перегрузка операторов в C#

Перегрузка операторов — это механизм, позволяющий определять поведение стандартных операторов (+, -, *, /, ==, != и др.) для пользовательских типов данных. Это повышает выразительность кода и позволяет работать с объектами классов так же естественно, как со встроенными типами.

Синтаксис перегрузки операторов

Перегрузка оператора определяется с помощью ключевого слова operator внутри класса или структуры:

```
public static возвращаемый_тип operator символ_оператора(параметры)
{
    // тело метода
}
```

Основные правила:

- Метод должен быть объявлен как public static
- Хотя бы один из параметров должен иметь тип содержащего класса
- Нельзя изменить приоритет, ассоциативность или количество операндов оператора

Категории перегружаемых операторов

1. Унарные операторы. Принимают один операнд:

- +, - — унарные плюс и минус
- ! — логическое отрицание
- ~ — побитовое дополнение
- ++, -- — инкремент и декремент
- true, false — логические операторы

Пример:

```
public class Counter
{
    public int Value { get; set; }

    // Перегрузка унарного оператора ++
    public static Counter operator ++(Counter c)
    {
        c.Value++;
```

```
        return c;
    }

    // Перегрузка унарного оператора -
    public static Counter operator -(Counter c)
    {
        return new Counter { Value = -c.Value };
    }
}
```

2. Бинарные арифметические операторы. Принимают два операнда:

- +, -, *, /, % — арифметические операции

Пример:

```
public class Complex
{
    public double Real { get; set; }
    public double Imaginary { get; set; }

    // Перегрузка оператора сложения
    public static Complex operator +(Complex c1, Complex c2)
    {
        return new Complex
        {
            Real = c1.Real + c2.Real,
            Imaginary = c1.Imaginary + c2.Imaginary
        };
    }

    // Перегрузка оператора умножения
    public static Complex operator *(Complex c1, Complex c2)
    {
        return new Complex
        {
            Real = c1.Real * c2.Real - c1.Imaginary * c2.Imaginary,
            Imaginary = c1.Real * c2.Imaginary + c1.Imaginary * c2.Real
        };
    }
}
```

3. Операторы сравнения. Должны перегружаться парами:

- == и != — равенство и неравенство
- < и > — меньше и больше
- <= и >= — меньше или равно, больше или равно

Пример:

```
public class Money
{
    public decimal Amount { get; set; }

    public static bool operator ==(Money m1, Money m2)
    {
        if (ReferenceEquals(m1, m2)) return true;
        if (m1 is null || m2 is null) return false;
        return m1.Amount == m2.Amount;
    }

    public static bool operator !=(Money m1, Money m2)
    {
        return !(m1 == m2);
    }

    public static bool operator >(Money m1, Money m2)
    {
        return m1.Amount > m2.Amount;
    }

    public static bool operator <(Money m1, Money m2)
    {
        return m1.Amount < m2.Amount;
    }

    // Также необходимо переопределить Equals и GetHashCode
    public override bool Equals(object obj)
    {
        if (obj is Money money)
            return this == money;
        return false;
    }

    public override int GetHashCode()
    {
        return Amount.GetHashCode();
    }
}
```

4. Операторы преобразования типов

Явное преобразование (explicit) — требует явного указания в коде:

```
public static explicit operator int(MyClass obj)
{
```

```
    return obj.Value;
}
// Использование: int x = (int)myObject;
```

Неявное преобразование (implicit) — выполняется автоматически:

```
public static implicit operator double(MyClass obj)
{
    return obj.Value;
}
// Использование: double x = myObject;
```

Пример:

```
public class Temperature
{
    public double Celsius { get; set; }

    // Неявное преобразование из double в Temperature
    public static implicit operator Temperature(double celsius)
    {
        return new Temperature { Celsius = celsius };
    }

    // Явное преобразование из Temperature в double
    public static explicit operator double(Temperature temp)
    {
        return temp.Celsius;
    }
}

// Использование:
Temperature temp = 36.6;  // неявное преобразование
double value = (double)temp;  // явное преобразование
```

Операторы, которые нельзя перегрузить. Следующие операторы **не могут быть перегружены**:

- = — присваивание
- . — доступ к членам
- ?: — тернарный оператор
- ?? — оператор объединения с null
- -> — доступ к членам через указатель
- new, typeof, sizeof, is, as
- && и || — условные логические операторы (но можно перегрузить & и |)

Рекомендации по перегрузке операторов

1.Интуитивность — поведение оператора должно быть понятным и ожидаемым

2.Согласованность — связанные операторы должны работать согласованно

3.Симметричность — если перегружается ==, то должен быть перегружен и !=

4.Переопределение Equals и GetHashCode — при перегрузке == и != необходимо переопределить методы Equals() и GetHashCode()

5.Использование по назначению — не следует перегружать операторы для действий, не соответствующих их семантике

Пример полноценного класса с перегрузкой операторов

```
public class Fraction
{
   public int Numerator { get; set; }      // Числитель
   public int Denominator { get; set; }    // Знаменатель

   public Fraction(int numerator, int denominator)
   {
      if (denominator == 0)
         throw new ArgumentException("Знаменатель не может быть равен
нулю");

      Numerator = numerator;
      Denominator = denominator;
      Simplify();
   }

   // Упрощение дроби
   private void Simplify()
   {
      int gcd = GCD(Math.Abs(Numerator), Math.Abs(Denominator));
      Numerator /= gcd;
      Denominator /= gcd;

      if (Denominator < 0)
      {
         Numerator = -Numerator;
         Denominator = -Denominator;
      }
   }
```

```
// Наибольший общий делитель
private int GCD(int a, int b)
{
    while (b != 0)
    {
        int temp = b;
        b = a % b;
        a = temp;
    }
    return a;
}

// Сложение дробей
public static Fraction operator +(Fraction f1, Fraction f2)
{
    return new Fraction(
        f1.Numerator * f2.Denominator + f2.Numerator * f1.Denominator,
        f1.Denominator * f2.Denominator
    );
}

// Вычитание дробей
public static Fraction operator -(Fraction f1, Fraction f2)
{
    return new Fraction(
        f1.Numerator * f2.Denominator - f2.Numerator * f1.Denominator,
        f1.Denominator * f2.Denominator
    );
}

// Умножение дробей
public static Fraction operator *(Fraction f1, Fraction f2)
{
    return new Fraction(
        f1.Numerator * f2.Numerator,
        f1.Denominator * f2.Denominator
    );
}

// Деление дробей
public static Fraction operator /(Fraction f1, Fraction f2)
{
    return new Fraction(
```

```
            f1.Numerator * f2.Denominator,
            f1.Denominator * f2.Numerator
        );
    }

    // Унарный минус
    public static Fraction operator -(Fraction f)
    {
        return new Fraction(-f.Numerator, f.Denominator);
    }

    // Сравнение на равенство
    public static bool operator ==(Fraction f1, Fraction f2)
    {
        if (ReferenceEquals(f1, f2)) return true;
        if (f1 is null || f2 is null) return false;
        return f1.Numerator == f2.Numerator && f1.Denominator == f2.Denomina-
tor;
    }

    public static bool operator !=(Fraction f1, Fraction f2)
    {
        return !(f1 == f2);
    }

    // Явное преобразование в double
    public static explicit operator double(Fraction f)
    {
        return (double)f.Numerator / f.Denominator;
    }

    // Неявное преобразование из int
    public static implicit operator Fraction(int value)
    {
        return new Fraction(value, 1);
    }

    public override bool Equals(object obj)
    {
        if (obj is Fraction fraction)
            return this == fraction;
        return false;
    }
```

```
    public override int GetHashCode()
    {
        return HashCode.Combine(Numerator, Denominator);
    }

    public override string ToString()
    {
        return $"{Numerator}/{Denominator}";
    }
}
```

Рабочее задание: Выполните разработку класса в соответствии с вариантом задания, реализуйте необходимые операции и продемонстрируйте их работу в консольном приложении.

Вариант 1. Класс Vector2D (двумерный вектор). Создайте класс для представления двумерного вектора с координатами X и Y. Реализуйте:

- Арифметические операции: сложение векторов, вычитание векторов, умножение вектора на число
- Унарный минус (изменение направления вектора)
- Операторы сравнения: равенство и неравенство векторов
- Метод расчета длины вектора
- Явное преобразование в строку формата "(X; Y)"

Вариант 2. Класс Matrix2x2 (матрица 2×2). Создайте класс для представления матрицы 2×2.Реализуйте:

- Сложение и вычитание матриц
- Умножение матрицы на число
- Умножение матриц
- Операторы сравнения
- Метод вычисления определителя
- Транспонирование матрицы (унарный оператор +)

Вариант 3. Класс Time (время). Создайте класс для представления времени (часы, минуты, секунды). Реализуйте:

- Сложение времени (Time + Time)
- Вычитание времени
- Операторы сравнения (<, >, ==, !=, <=, >=)
- Инкремент и декремент (добавление/вычитание одной секунды)
- Неявное преобразование из int (количество секунд)

- Явное преобразование в int (общее количество секунд)

Вариант 4. Класс Money (денежная сумма). Создайте класс для представления денежной суммы с указанием валюты. Реализуйте:

- Сложение и вычитание сумм (только одной валюты)
- Умножение и деление на число
- Операторы сравнения
- Унарный минус
- Неявное преобразование из decimal
- Явное преобразование в decimal
- Проверку совместимости валют при операциях

Вариант 5. Класс Polynomial (полином первой степени ax + b)

Создайте класс для представления линейного полинома вида ax + b. Реализуйте:

- Сложение и вычитание полиномов
- Умножение полинома на число
- Умножение полиномов
- Унарный минус
- Метод вычисления полинома при заданном x
- Операторы сравнения
- Неявное преобразование из числа (создание константного полинома)

Вариант 6. Класс Distance (расстояние в метрах и сантиметрах). Создайте класс для представления расстояния (метры и сантиметры). Реализуйте:

- Сложение и вычитание расстояний
- Умножение и деление на число
- Операторы сравнения
- Инкремент и декремент (изменение на 1 см)
- Неявное преобразование из int (количество сантиметров)
- Явное преобразование в double (общая длина в метрах)

Вариант 7. Класс ComplexNumber (комплексное число). Создайте класс для представления комплексного числа (действительная и мнимая части). Реализуйте:

- Сложение, вычитание, умножение и деление комплексных чисел
- Унарный минус
- Операторы сравнения на равенство
- Метод вычисления модуля комплексного числа
- Метод получения сопряженного числа
- Неявное преобразование из double (создание действительного числа)

Вариант 8. Класс Rational (рациональное число — дробь) Создайте класс для представления обыкновенной дроби (числитель/знаменатель). Реализуйте:

- Сложение, вычитание, умножение и деление дробей
- Унарный минус
- Операторы сравнения (<, >, ==, !=, <=, >=)
- Автоматическое сокращение дроби
- Неявное преобразование из int
- Явное преобразование в double

Вариант 9. Класс Point3D (точка в трёхмерном пространстве). Создайте класс для представления точки в пространстве (X, Y, Z). Реализуйте:

- Сложение и вычитание точек (как векторов)
- Умножение на число
- Унарный минус
- Операторы сравнения
- Метод расчета расстояния до другой точки
- Метод расчета расстояния до начала координат

Вариант 10. Класс Counter (счётчик). Создайте класс для представления счётчика с минимальным и максимальным значением. Реализуйте:

- Инкремент и декремент (с учетом границ)
- Сложение и вычитание счётчиков
- Операторы сравнения
- Унарный оператор ! (сброс счётчика в 0)
- Неявное преобразование из int

- Явное преобразование в int
- Проверку выхода за границы диапазона

Требования к выполнению задания

1.Создайте консольное приложение в Visual Studio 2022 (.NET 6.0 или выше)
2.Реализуйте класс согласно варианту, включая:
–Свойства для хранения данных
–Конструкторы (по умолчанию и с параметрами)
–Перегруженные операторы (не менее 6)
–Методы переопределения Equals () и GetHashCode() (при перегрузке == и !=)
–Переопределение ToString () для удобного отображения
3.В методе Main() продемонстрируйте работу всех перегруженных операторов**:**
–Создайте несколько объектов класса
–Выполните операции с использованием перегруженных операторов
–Выведите результаты на консоль
–Проверьте граничные случаи и обработку исключений
4.Добавьте комментарии к коду, поясняющие логику работы операторов
5.Протестируйте программу на различных наборах данных

Пример выполнения (для ознакомления)

```
using System;

namespace OperatorOverloadingDemo
{
    // Класс для представления комплексного числа
    class ComplexNumber
    {
        public double Real { get; set; }
        public double Imaginary { get; set; }

        // Конструктор
        public ComplexNumber(double real, double imaginary)
        {
            Real = real;
            Imaginary = imaginary;
        }
```

```
    // Конструктор по умолчанию
    public ComplexNumber() : this(0, 0) { }

    // Перегрузка оператора сложения
    public static ComplexNumber operator +(ComplexNumber c1, Com-
plexNumber c2)
    {
        return new ComplexNumber(c1.Real + c2.Real, c1.Imaginary + c2.Imagi-
nary);
    }

    // Перегрузка оператора вычитания
    public static ComplexNumber operator -(ComplexNumber c1, Com-
plexNumber c2)
    {
        return new ComplexNumber(c1.Real - c2.Real, c1.Imaginary - c2.Imagi-
nary);
    }

    // Перегрузка оператора умножения
    public static ComplexNumber operator *(ComplexNumber c1, Com-
plexNumber c2)
    {
        return new ComplexNumber(
            c1.Real * c2.Real - c1.Imaginary * c2.Imaginary,
            c1.Real * c2.Imaginary + c1.Imaginary * c2.Real
        );
    }

    // Унарный минус
    public static ComplexNumber operator -(ComplexNumber c)
    {
        return new ComplexNumber(-c.Real, -c.Imaginary);
    }

    // Оператор равенства
    public static bool operator ==(ComplexNumber c1, ComplexNumber c2)
    {
        if (ReferenceEquals(c1, c2)) return true;
        if (c1 is null || c2 is null) return false;
        return c1.Real == c2.Real && c1.Imaginary == c2.Imaginary;
    }

    public static bool operator !=(ComplexNumber c1, ComplexNumber c2)
```

```
    {
        return !(c1 == c2);
    }

    // Модуль комплексного числа
    public double Magnitude()
    {
        return Math.Sqrt(Real * Real + Imaginary * Imaginary);
    }

    // Переопределение Equals
    public override bool Equals(object obj)
    {
        if (obj is ComplexNumber complex)
            return this == complex;
        return false;
    }

    public override int GetHashCode()
    {
        return HashCode.Combine(Real, Imaginary);
    }

    // Переопределение ToString
    public override string ToString()
    {
        if (Imaginary >= 0)
            return $"{Real} + {Imaginary}i";
        else
            return $"{Real} - {Math.Abs(Imaginary)}i";
    }
}

class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine("=== Демонстрация перегрузки операторов
===\n");

        // Создание объектов
        ComplexNumber c1 = new ComplexNumber(3, 4);
        ComplexNumber c2 = new ComplexNumber(1, -2);
```

```
            Console.WriteLine($"c1 = {c1}");
            Console.WriteLine($"c2 = {c2}\n");

            // Сложение
            ComplexNumber sum = c1 + c2;
            Console.WriteLine($"c1 + c2 = {sum}");

            // Вычитание
            ComplexNumber diff = c1 - c2;
            Console.WriteLine($"c1 - c2 = {diff}");

            // Умножение
            ComplexNumber product = c1 * c2;
            Console.WriteLine($"c1 * c2 = {product}");

            // Унарный минус
            ComplexNumber neg = -c1;
            Console.WriteLine($"-c1 = {neg}");

            // Сравнение
            Console.WriteLine($"\nc1 == c2: {c1 == c2}");
            Console.WriteLine($"c1 != c2: {c1 != c2}");

            // Модуль
            Console.WriteLine($"\n|c1| = {c1.Magnitude():F2}");
            Console.WriteLine($"|c2| = {c2.Magnitude():F2}");

            Console.ReadKey();
        }
    }
}
```

Оформление отчёта

Отчет должен содержать:

1.Результаты выполнения работы: полный код разработанного класса с комментариями; код программы для демонстрации работы класса; скриншоты результатов работы программы

2.Анализ результатов: описание реализованных операторов; пояснение логики работы каждого оператора; особенности реализации

3.Выводы: что было изучено в ходе практической работы; какие навыки приобретены; трудности, возникшие при выполнении

Контрольные вопросы

1.Что такое перегрузка операторов и для чего она используется в объектно-ориентированном программировании?

2.Какие ключевые слова используются при объявлении перегруженного оператора? Почему метод перегрузки оператора должен быть объявлен как public static?

3.Чем отличаются унарные операторы от бинарных? Приведите примеры каждого типа.

4.Какие операторы в C# должны перегружаться парами? Почему это требование существует?

5.В чем различие между явным (explicit) и неявным (implicit) преобразованием типов? Когда следует использовать каждый из них?

6.Какие операторы в C# нельзя перегрузить? Объясните, почему существуют такие ограничения.

7.Почему при перегрузке операторов == и != необходимо также переопределять методы Equals() и GetHashCode()?

8.Какие рекомендации следует соблюдать при перегрузке операторов, чтобы код был понятным и безопасным?

9.Как обеспечить корректную обработку значений null при перегрузке операторов сравнения?

10.Приведите пример ситуации из реальной разработки, где перегрузка операторов улучшает читаемость и удобство использования кода.

Практическая работа № 7 «Создание наследованных классов»

Цель работы: Изучить принципы объектно-ориентированного программирования, освоить механизм наследования классов в языке C#, научиться создавать иерархии классов, использовать модификаторы доступа, переопределять методы базового класса и применять полиморфизм.

Время выполнения: 4 ч

Теоретическая часть

Основные понятия наследования

Наследование — один из четырех основных принципов объектно-ориентированного программирования, который позволяет создавать новые классы на основе существующих. Класс, от которого происходит наследование, называется **базовым классом** (родительским, суперклассом), а класс, который наследует, — **производным классом** (дочерним, подклассом).

Преимущества наследования: повторное использование кода; логическая организация классов; упрощение модификации и расширения программы; возможность применения полиморфизма.

Синтаксис наследования в C#

```
class BaseClass
{
    // Члены базового класса
}

class DerivedClass : BaseClass
{
    // Члены производного класса
    // + унаследованные члены базового класса
}
```

Модификаторы доступа. В C# при работе с наследованием используются следующие модификаторы доступа:

Модификатор	Описание
public	Доступ из любого места программы
private	Доступ только внутри класса (не наследуется)
protected	Доступ внутри класса и в производных классах
internal	Доступ внутри текущей сборки

Модификатор	Описание
protected internal	Доступ внутри сборки или в производных классах

Ключевое слово base. Ключевое слово **base** используется для обращения к членам базового класса из производного класса:

```
class Animal
{
    protected string name;

    public Animal(string name)
    {
        this.name = name;
    }

    public virtual void MakeSound()
    {
        Console.WriteLine("Животное издает звук");
    }
}

class Dog : Animal
{
    private string breed;

    public Dog(string name, string breed) : base(name)
    {
        this.breed = breed;
    }

    public override void MakeSound()
    {
        base.MakeSound(); // Вызов метода базового класса
        Console.WriteLine("Гав-гав!");
    }
}
```

Виртуальные методы и переопределение. Виртуальный метод (virtual) — метод базового класса, который может быть переопределен в производном классе. Переопределение (override)

— создание новой реализации виртуального метода в производном классе.

```
class Shape
{
    public virtual double GetArea()
    {
        return 0;
    }
}

class Circle : Shape
{
    private double radius;

    public Circle(double radius)
    {
        this.radius = radius;
    }

    public override double GetArea()
    {
        return Math.PI * radius * radius;
    }
}
```

Абстрактные классы и методы. Абстрактный класс (abstract) — класс, который не может быть инстанцирован и предназначен только для наследования. Абстрактный метод — метод без реализации, который обязательно должен быть переопределен в производном классе.

```
abstract class Vehicle
{
    protected string model;

    public Vehicle(string model)
    {
        this.model = model;
    }

    public abstract void Move(); // Абстрактный метод

    public void ShowInfo()
    {
        Console.WriteLine($"Транспорт: {model}");
```

```
    }
}

class Car : Vehicle
{
    public Car(string model) : base(model) { }

    public override void Move()
    {
        Console.WriteLine("Автомобиль едет по дороге");
    }
}
```

Запечатанные классы (sealed). Запечатанный класс (sealed) — класс, от которого нельзя наследоваться.

```
sealed class FinalClass
{
    // От этого класса нельзя наследоваться
}
```

Полиморфизм. Полиморфизм — возможность использовать объекты производных классов через ссылку на базовый класс.

```
Animal animal1 = new Dog("Шарик", "Дворняга");
Animal animal2 = new Cat("Мурка", "Сиамская");

animal1.MakeSound(); // Вызовется метод Dog
animal2.MakeSound(); // Вызовется метод Cat
```

Конструкторы в иерархии наследования. При создании объекта производного класса всегда вызывается конструктор базового класса. Если не указано явно, вызывается конструктор базового класса без параметров.

```
class Person
{
    protected string name;
    protected int age;

    public Person()
    {
        Console.WriteLine("Конструктор Person()");
    }

    public Person(string name, int age)
    {
        this.name = name;
        this.age = age;
```

```
        Console.WriteLine("Конструктор Person(string, int)");
    }
}

class Student : Person
{
    private string studentId;

    public Student(string name, int age, string studentId) : base(name, age)
    {
        this.studentId = studentId;
        Console.WriteLine("Конструктор Student");
    }
}
```

Множественное наследование. C# не поддерживает множественное наследование классов**,** но один класс может реализовывать несколько интерфейсов.

Рабочее задание: Выполните разработку иерархии классов в соответствии с вариантом задания. Создайте консольное приложение в Visual Studio 2022, демонстрирующее работу созданных классов.

Вариант 1. Иерархия "Сотрудники предприятия". Создайте базовый класс **Employee** (Сотрудник) с полями: имя, возраст, должность, базовая зарплата. Создайте производные классы:

- **Manager** (Менеджер) — добавить поле "количество подчиненных", метод расчета зарплаты с надбавкой за подчиненных
- **Developer** (Разработчик) — добавить поле "язык программирования", метод расчета зарплаты с надбавкой за квалификацию
- **Designer** (Дизайнер) — добавить поле "специализация", метод расчета зарплаты

Вариант 2. Иерархия "Геометрические фигуры". Создайте абстрактный базовый класс **Shape** (Фигура) с абстрактными методами расчета площади и периметра. Создайте производные классы:

- **Rectangle** (Прямоугольник) — поля: длина, ширина
- **Circle** (Круг) — поле: радиус
- **Triangle** (Треугольник) — поля: три стороны Реализуйте методы расчета площади и периметра для каждой фигуры.

Вариант 3. Иерархия "Транспортные средства". Создайте базовый класс **Transport** (Транспорт) с полями: модель, год выпуска, скорость. Создайте производные классы:

- **Car** (Автомобиль) — добавить поле "количество дверей", метод вывода информации
- **Motorcycle** (Мотоцикл) — добавить поле "тип мотоцикла", метод вывода информации
- **Truck** (Грузовик) — добавить поле "грузоподъемность", метод расчета стоимости перевозки

Вариант 4. Иерархия "Учебные заведения". Создайте базовый класс **EducationalInstitution** (Учебное заведение) с полями: название, адрес, количество учащихся. Создайте производные классы:

- **School** (Школа) — добавить поле "количество классов"
- **College** (Колледж) — добавить поле "специальности"
- **University** (Университет) — добавить поле "количество факультетов", "наличие аспирантуры"

Вариант 5. Иерархия "Банковские счета". Создайте базовый класс **BankAccount** (Банковский счет) с полями: номер счета, владелец, баланс, методами пополнения и снятия средств. Создайте производные классы:

- **SavingsAccount** (Сберегательный счет) — добавить процентную ставку, метод начисления процентов
- **CheckingAccount** (Расчетный счет) — добавить лимит овердрафта
- **DepositAccount** (Депозитный счет) — добавить срок вклада, запрет на снятие до окончания срока

Вариант 6. Иерархия "Устройства". Создайте абстрактный базовый класс **Device** (Устройство) с полями: производитель, модель, цена. Создайте производные классы:

- **Smartphone** (Смартфон) — добавить поля "диагональ экрана", "операционная система"
- **Laptop** (Ноутбук) — добавить поля "процессор", "объем оперативной памяти"
- **Tablet** (Планшет) — добавить поле "поддержка стилуса"

Вариант 7. Иерархия "Публикации". Создайте базовый класс **Publication** (Публикация) с полями: название, автор, год издания. Создайте производные классы:

- **Book** (Книга) — добавить поля "издательство", "количество страниц", "ISBN"
- **Magazine** (Журнал) — добавить поля "номер выпуска", "периодичность"
- **Article** (Статья) — добавить поле "название журнала/сборника"

Вариант 8. Иерархия "Животные". Создайте абстрактный базовый класс **Animal** (Животное) с полями: кличка, возраст, вес, абстрактным методом MakeSound(). Создайте производные классы:

- **Dog** (Собака) — добавить поле "порода"
- **Cat** (Кошка) — добавить поле "длина шерсти"
- **Bird** (Птица) — добавить поле "размах крыльев" Реализуйте метод MakeSound() для каждого класса.

Вариант 9. Иерархия "Товары". Создайте базовый класс **Product** (Товар) с полями: наименование, цена, производитель. Создайте производные классы:

- **Food** (Продукт питания) — добавить поля "срок годности", "калорийность"
- **Electronics** (Электроника) — добавить поля "гарантийный срок", "энергопотребление"
- **Clothing** (Одежда) — добавить поля "размер", "материал"

Вариант 10. Иерархия "Документы". Создайте абстрактный базовый класс **Document** (Документ) с полями: номер, дата создания, автор. Создайте производные классы:

- **Report** (Отчет) — добавить поля "период", "тип отчета", метод формирования отчета
- **Contract** (Договор) — добавить поля "стороны договора", "сумма", "срок действия"
- **Invoice** (Счет) — добавить поля "получатель", "сумма к оплате", "срок оплаты"

Требования к выполнению задания

1.Создайте новый проект в Visual Studio 2022:
–Тип проекта: Console App (.NET или .NET Framework)

–Язык: C#
–Имя проекта: соответствует вашему варианту (например, EmployeeHierarchy)
2.Разработайте иерархию классов:
–создайте базовый класс с необходимыми полями и методами
–используйте модификаторы доступа protected для полей, которые должны быть доступны в производных классах
–создайте конструкторы базового класса
–создайте не менее трех производных классов
–в производных классах добавьте специфичные поля и методы
–используйте ключевое слово base для вызова конструкторов и методов базового класса
3.Реализуйте механизмы ООП:
–используйте виртуальные методы (virtual) и их переопределение (override) минимум для двух методов
–создайте минимум один абстрактный метод (если используется абстрактный базовый класс)
–продемонстрируйте полиморфизм
4.Создайте демонстрационную программу:
–в методе Main() создайте массив или список объектов базового типа
–заполните его объектами различных производных классов
–продемонстрируйте работу методов через полиморфизм
–выведите информацию о каждом объекте
5.Добавьте комментарии к коду:
–опишите назначение каждого класса
–добавьте комментарии к полям и методам
–используйте XML-комментарии для документирования публичных членов
6.Примените принципы чистого кода:
–используйте осмысленные имена переменных и методов
–следуйте соглашениям об именовании C# (PascalCase для классов и методов, camelCase для полей)
–обеспечьте читаемость и структурированность кода

Пример выполнения задания

```
using System;
using System.Collections.Generic;
```

```
namespace AnimalHierarchy
{
    /// <summary>
    /// Абстрактный базовый класс "Животное"
    /// </summary>
    abstract class Animal
    {
        protected string name;
        protected int age;
        protected double weight;

        public Animal(string name, int age, double weight)
        {
            this.name = name;
            this.age = age;
            this.weight = weight;
        }

        /// <summary>
        /// Абстрактный метод издания звука
        /// </summary>
        public abstract void MakeSound();

        /// <summary>
        /// Виртуальный метод вывода информации
        /// </summary>
        public virtual void ShowInfo()
        {
            Console.WriteLine($"Кличка: {name}");
            Console.WriteLine($"Возраст: {age} лет");
            Console.WriteLine($"Вес: {weight} кг");
        }
    }

    /// <summary>
    /// Производный класс "Собака"
    /// </summary>
    class Dog : Animal
    {
        private string breed;

        public Dog(string name, int age, double weight, string breed)
            : base(name, age, weight)
```

```
    {
        this.breed = breed;
    }

    public override void MakeSound()
    {
        Console.WriteLine("Гав-гав!");
    }

    public override void ShowInfo()
    {
        Console.WriteLine("\n=== Собака ===");
        base.ShowInfo();
        Console.WriteLine($"Порода: {breed}");
    }
}

/// <summary>
/// Производный класс "Кошка"
/// </summary>
class Cat : Animal
{
    private string furLength;

    public Cat(string name, int age, double weight, string furLength)
        : base(name, age, weight)
    {
        this.furLength = furLength;
    }

    public override void MakeSound()
    {
        Console.WriteLine("Мяу-мяу!");
    }

    public override void ShowInfo()
    {
        Console.WriteLine("\n=== Кошка ===");
        base.ShowInfo();
        Console.WriteLine($"Длина шерсти: {furLength}");
    }
}

/// <summary>
```

```
/// Производный класс "Птица"
/// </summary>
class Bird : Animal
{
    private double wingspan;

    public Bird(string name, int age, double weight, double wingspan)
        : base(name, age, weight)
    {
        this.wingspan = wingspan;
    }

    public override void MakeSound()
    {
        Console.WriteLine("Чирик-чирик!");
    }

    public override void ShowInfo()
    {
        Console.WriteLine("\n=== Птица ===");
        base.ShowInfo();
        Console.WriteLine($"Размах крыльев: {wingspan} см");
    }
}

class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine("Демонстрация иерархии классов 'Животные'\n");

        // Создание списка животных (полиморфизм)
        List<Animal> animals = new List<Animal>
        {
            new Dog("Шарик", 5, 25.5, "Лабрадор"),
            new Cat("Мурка", 3, 4.2, "Длинная"),
            new Bird("Кеша", 2, 0.3, 45.0),
            new Dog("Бобик", 7, 30.0, "Овчарка")
        };

        // Вывод информации о каждом животном
        foreach (Animal animal in animals)
        {
            animal.ShowInfo();
```

```
            Console.Write("Звук: ");
            animal.MakeSound();
        }

        Console.WriteLine("\n\nДемонстрация завершена.");
        Console.ReadKey();
    }
  }
}
```

Результат выполнения программы: Демонстрация иерархии классов 'Животные'
=== Собака ===
Кличка: Шарик
Возраст: 5 лет
Вес: 25.5 кг
Порода: Лабрадор
Звук: Гав-гав!

=== Кошка ===
Кличка: Мурка
Возраст: 3 лет
Вес: 4.2 кг
Длина шерсти: Длинная
Звук: Мяу-мяу!
=== Птица ===
Кличка: Кеша
Возраст: 2 лет
Вес: 0.3 кг
Размах крыльев: 45 см
Звук: Чирик-чирик!
=== Собака ===
Кличка: Бобик
Возраст: 7 лет
Вес: 30 кг
Порода: Овчарка
Звук: Гав-гав!
Демонстрация завершена.

Содержание отчета

Отчет по практической работе должен содержать:

1.Описание иерархии классов**:** –диаграмма классов (можно нарисовать от руки или в любом редакторе); описание базового класса: назначение, поля, методы; описание каждого производного класса: *назначение, дополнительные поля, переопределенные методы*

2.Листинг программы — полный код программы с комментариями

3.Результаты выполнения программы — скриншоты работы программы или текст вывода консоли

4.Выводы по работе — что было изучено, какие навыки приобретены, какие трудности возникли при выполнении

Контрольные вопросы

1.Что такое наследование в объектно-ориентированном программировании? Какие преимущества оно дает?

2.Какой синтаксис используется в C# для создания класса-наследника? Приведите пример.

3.Объясните назначение модификаторов доступа public, private и protected. В каких случаях следует использовать каждый из них при работе с наследованием?

4.Что означает ключевое слово **base** в C#? В каких ситуациях оно применяется?

5.В чем разница между виртуальным методом (virtual) и абстрактным методом (abstract)?

6.Что такое переопределение метода (override)? Чем оно отличается от сокрытия метода (new)?

7.Что такое абстрактный класс? Можно ли создать экземпляр абстрактного класса? Почему?

8.Объясните понятие полиморфизма. Приведите пример использования полиморфизма в C#.

9.В каком порядке вызываются конструкторы при создании объекта производного класса?

10.Что такое запечатанный класс (sealed)? Для чего используется этот модификатор?

11.Поддерживает ли C# множественное наследование классов? Как можно реализовать функциональность множественного наследования?

12.Можно ли в производном классе изменить уровень доступа переопределяемого метода по сравнению с базовым классом?

Практическая работа № 8 «Работа с объектами через интерфейсы»

Цель работы: Изучить принципы работы с интерфейсами в языке C#, освоить технологию создания и реализации интерфейсов, научиться применять полиморфизм через интерфейсы для создания гибких и расширяемых программных модулей.
Время выполнения: 2 ч
Теоретическая часть
Понятие интерфейса. Интерфейс — это программная конструкция, определяющая контракт (набор сигнатур методов, свойств, событий и индексаторов), который должны реализовать классы. Интерфейс описывает **что** должен делать класс, но не описывает **как** это должно быть реализовано.
В C# интерфейс объявляется с использованием ключевого слова interface. По соглашению имена интерфейсов начинаются с заглавной буквы **I**.
Синтаксис объявления интерфейса:

```
interface IИмяИнтерфейса
{
    // Сигнатуры методов
    void МетодБезПараметров();
    int МетодСПараметрами(string параметр);

    // Свойства
    string Свойство { get; set; }

    // События (опционально)
    // Индексаторы (опционально)
}
```

Особенности интерфейсов в C#
1.Интерфейс не содержит реализации — только сигнатуры членов (до C# 8.0; начиная с C# 8.0 возможна реализация по умолчанию).
2.Множественная реализация — класс может реализовывать несколько интерфейсов одновременно (в отличие от наследования классов).
3.Все члены интерфейса публичны — модификаторы доступа не указываются (по умолчанию public).

4.Интерфейсы не могут содержать поля — только свойства, методы, события и индексаторы.
5.Интерфейсы могут наследоваться от других интерфейсов.
Реализация интерфейсов. Синтаксис реализации интерфейса классом:

```
class ИмяКласса : IИмяИнтерфейса
{
    // Реализация всех членов интерфейса
    public void МетодБезПараметров()
    {
        // Реализация
    }

    public int МетодСПараметрами(string параметр)
    {
        // Реализация
        return 0;
    }

    public string Свойство { get; set; }
}
```

Важно: Класс, реализующий интерфейс, **обязан** реализовать все его члены, иначе возникнет ошибка компиляции.

Пример базового использования интерфейсов

```
// Объявление интерфейса
interface IMovable
{
    void Move();
    double Speed { get; set; }
}

// Реализация интерфейса классом Car
class Car : IMovable
{
    public double Speed { get; set; }

    public void Move()
    {
        Console.WriteLine($"Автомобиль движется со скоростью {Speed}
км/ч");
    }
}
```

```
// Реализация интерфейса классом Bicycle
class Bicycle : IMovable
{
    public double Speed { get; set; }

    public void Move()
    {
        Console.WriteLine($"Велосипед едет со скоростью {Speed} км/ч");
    }
}
```

Полиморфизм через интерфейсы. Ключевое преимущество интерфейсов — возможность использовать **полиморфизм**. Переменная типа интерфейса может ссылаться на объект любого класса, реализующего этот интерфейс.

```
IMovable vehicle1 = new Car() { Speed = 80 };
IMovable vehicle2 = new Bicycle() { Speed = 20 };

vehicle1.Move(); // Вызовется реализация из класса Car
vehicle2.Move(); // Вызовется реализация из класса Bicycle
```

Множественная реализация интерфейсов. Класс может реализовывать несколько интерфейсов, перечисляя их через запятую:

```
interface IDrawable
{
    void Draw();
}

interface IResizable
{
    void Resize(int width, int height);
}

class Rectangle : IDrawable, IResizable
{
    public void Draw()
    {
        Console.WriteLine("Рисуем прямоугольник");
    }

    public void Resize(int width, int height)
    {
        Console.WriteLine($"Изменяем размер: {width}x{height}");
    }
}
```

Явная реализация интерфейса. Если класс реализует несколько интерфейсов с одинаковыми сигнатурами методов, используется явная реализация:

```
interface IPrinter
{
    void Print();
}

interface IScanner
{
    void Print(); // Метод с тем же именем
}

class MultiFunctionDevice : IPrinter, IScanner
{
    // Явная реализация IPrinter
    void IPrinter.Print()
    {
        Console.WriteLine("Печать документа");
    }

    // Явная реализация IScanner
    void IScanner.Print()
    {
        Console.WriteLine("Сканирование документа");
    }
}

// Использование
MultiFunctionDevice device = new MultiFunctionDevice();
((IPrinter)device).Print(); // Печать документа
((IScanner)device).Print(); // Сканирование документа
```

Наследование интерфейсов. Интерфейсы могут наследоваться от других интерфейсов:

```
interface IShape
{
    double GetArea();
}

interface IColoredShape : IShape
{
    string Color { get; set; }
}
```

```
class Circle : IColoredShape
{
    public double Radius { get; set; }
    public string Color { get; set; }

    public double GetArea()
    {
        return Math.PI * Radius * Radius;
    }
}
```

Преимущества использования интерфейсов

1.Слабая связанность (Loose Coupling) — компоненты системы зависят от абстракций, а не от конкретных реализаций.

2.Гибкость и расширяемость — легко добавлять новые реализации без изменения существующего кода.

3.Тестируемость — упрощается создание mock-объектов для модульного тестирования.

4.Множественное поведение — один класс может реализовывать несколько интерфейсов.

5.Контракты — интерфейсы определяют четкие контракты взаимодействия между компонентами.

Применение интерфейсов в реальных задачах

Пример: Система обработки платежей

```
interface IPaymentProcessor
{
    bool ProcessPayment(decimal amount);
    string GetPaymentMethod();
}

class CreditCardPayment : IPaymentProcessor
{
    public bool ProcessPayment(decimal amount)
    {
        Console.WriteLine($"Обработка платежа {amount} руб. картой");
        return true;
    }

    public string GetPaymentMethod()
    {
        return "Кредитная карта";
    }
```

```
}

class PayPalPayment : IPaymentProcessor
{
    public bool ProcessPayment(decimal amount)
    {
        Console.WriteLine($"Обработка платежа {amount} руб. через PayPal");
        return true;
    }

    public string GetPaymentMethod()
    {
        return "PayPal";
    }
}

// Использование
class PaymentService
{
    public void MakePayment(IPaymentProcessor processor, decimal amount)
    {
        Console.WriteLine($"Метод оплаты: {processor.GetPaymentMethod()}");
        processor.ProcessPayment(amount);
    }
}
```

Интерфейсы в коллекциях. Интерфейсы позволяют создавать коллекции разнородных объектов с общим поведением:

```
List<IMovable> vehicles = new List<IMovable>
{
    new Car() { Speed = 100 },
    new Bicycle() { Speed = 25 },
    new Motorcycle() { Speed = 150 }
};

foreach (IMovable vehicle in vehicles)
{
    vehicle.Move();
}
```

Рабочее задание: Выполните задание в соответствии с вариантом. Создайте консольное приложение в Visual Studio 2022, реализующее работу с объектами через интерфейсы.

Вариант 1. Система управления устройствами

Разработать систему управления электронными устройствами.

Создать интерфейс IDevice с методами TurnOn(), TurnOff(), свойствами IsOn и DeviceName. Реализовать классы Television, Computer, Smartphone. Создать коллекцию устройств и продемонстрировать управление ими через интерфейс.

Вариант 2. Система геометрических фигур

Разработать систему для работы с геометрическими фигурами. Создать интерфейс IShape с методами CalculateArea(), CalculatePerimeter() и свойством Name. Реализовать классы Circle, Rectangle, Triangle. Создать массив фигур и вычислить суммарную площадь и периметр всех фигур.

Вариант 3. Система уведомлений

Разработать систему отправки уведомлений. Создать интерфейс INotificationSender с методом SendNotification(string message) и свойством SenderType. Реализовать классы EmailSender, SmsSender, PushNotificationSender. Создать сервис рассылки уведомлений, который может отправлять сообщения через разные каналы.

Вариант 4. Система хранения данных

Разработать систему для работы с различными хранилищами данных. Создать интерфейс IDataStorage с методами Save(string data), Load(), Delete(). Реализовать классы FileStorage, DatabaseStorage, CloudStorage. Продемонстрировать работу с данными независимо от типа хранилища.

Вариант 5. Транспортная система

Разработать систему управления транспортом. Создать интерфейсы IVehicle (методы Start(), Stop(), свойство Speed) и IFuelable (методы Refuel(double amount), свойство FuelLevel). Реализовать классы Car, Truck, ElectricCar (только IVehicle). Создать парк транспортных средств.

Вариант 6. Система воспроизведения медиафайлов

Разработать систему для воспроизведения медиа. Создать интерфейс IMediaPlayer с методами Play(), Pause(), Stop(), свойствами Duration и CurrentPosition. Реализовать классы AudioPlayer, VideoPlayer, StreamingPlayer. Создать плейлист и управлять воспроизведением.

Вариант 7. Система обработки документов

Разработать систему обработки документов. Создать

интерфейс IDocument с методами Open(), Save(), Print(), Close(), свойством FileName. Реализовать классы TextDocument, SpreadsheetDocument, PresentationDocument. Создать менеджер документов для работы с различными типами файлов.

Вариант 8. Система платежей

Разработать систему обработки платежей. Создать интерфейс IPayment с методами ProcessPayment(decimal amount), RefundPayment(decimal amount), свойствами PaymentId и Status. Реализовать классы CashPayment, CardPayment, OnlinePayment. Создать систему обработки заказов с различными способами оплаты.

Вариант 9. Система логирования

Разработать систему логирования событий. Создать интерфейс ILogger с методами LogInfo(string message), LogWarning(string message), LogError(string message). Реализовать классы FileLogger, ConsoleLogger, DatabaseLogger. Создать приложение, которое использует несколько логгеров одновременно.

Вариант 10. Система управления сотрудниками

Разработать систему учета сотрудников. Создать интерфейсы IEmployee (свойства Name, Position, метод GetInfo()) и ISalaryCalculable (метод CalculateSalary()). Реализовать классы HourlyEmployee, SalariedEmployee, ContractEmployee. Создать отдел с разными типами сотрудников и рассчитать зарплатный фонд.

Требования к выполнению задания

1.Создайте проект в Visual Studio 2022: Тип проекта: Console App (.NET 6.0 или выше); Назовите проект согласно варианту (например, DeviceManagementSystem)

2.Реализуйте требуемые интерфейсы: определите все необходимые методы и свойства; соблюдайте соглашения об именовании (интерфейсы начинаются с **I**)

3.Создайте классы, реализующие интерфейсы: не менее трех классов для каждого основного интерфейса; реализуйте все члены интерфейса; добавьте конструкторы для инициализации объектов

4.Продемонстрируйте полиморфизм: создайте коллекцию (массив или List) объектов типа интерфейса; заполните

коллекцию объектами разных классов; обработайте коллекцию в цикле, вызывая методы через интерфейс

5.Добавьте пользовательский интерфейс: реализуйте консольное меню для взаимодействия с пользователем; обеспечьте возможность создания объектов, вызова методов, просмотра результатов

6.Обработайте возможные исключения: используйте конструкции try-catch для обработки ошибок; проверяйте корректность вводимых данных.

7.Прокомментируйте код: добавьте XML-комментарии к интерфейсам и классам; объясните логику работы ключевых фрагментов

Пример выполнения задания (демонстрационный вариант)

```
using System;
using System.Collections.Generic;

namespace AnimalSystemDemo
{
    // Интерфейс для животных
    interface IAnimal
    {
        string Name { get; set; }
        void MakeSound();
        void Eat();
    }

    // Интерфейс для домашних животных
    interface IPet
    {
        string Owner { get; set; }
        void Play();
    }

    // Класс Собака
    class Dog : IAnimal, IPet
    {
        public string Name { get; set; }
        public string Owner { get; set; }

        public void MakeSound()
        {
            Console.WriteLine($"{Name} говорит: Гав-гав!");
```

```
    }

    public void Eat()
    {
        Console.WriteLine($"{Name} ест корм для собак");
    }

    public void Play()
    {
        Console.WriteLine($"{Name} играет с мячиком");
    }
}

// Класс Кошка
class Cat : IAnimal, IPet
{
    public string Name { get; set; }
    public string Owner { get; set; }

    public void MakeSound()
    {
        Console.WriteLine($"{Name} говорит: Мяу!");
    }

    public void Eat()
    {
        Console.WriteLine($"{Name} ест рыбу");
    }

    public void Play()
    {
        Console.WriteLine($"{Name} играет с клубком");
    }
}

// Класс Лев (дикое животное)
class Lion : IAnimal
{
    public string Name { get; set; }

    public void MakeSound()
    {
        Console.WriteLine($"{Name} рычит: Р-р-р!");
    }
```

```
    public void Eat()
    {
        Console.WriteLine($"{Name} охотится на добычу");
    }
}

class Program
{
    static void Main(string[] args)
    {
        // Создание коллекции животных
        List<IAnimal> animals = new List<IAnimal>
        {
            new Dog { Name = "Бобик", Owner = "Иван" },
            new Cat { Name = "Мурка", Owner = "Мария" },
            new Lion { Name = "Симба" }
        };

        // Полиморфное использование
        Console.WriteLine("=== Все животные ===");
        foreach (IAnimal animal in animals)
        {
            animal.MakeSound();
            animal.Eat();
            Console.WriteLine();
        }

        // Работа с домашними животными
        Console.WriteLine("=== Домашние животные ===");
        foreach (IAnimal animal in animals)
        {
            if (animal is IPet pet)
            {
                Console.WriteLine($"Владелец: {pet.Owner}");
                pet.Play();
                Console.WriteLine();
            }
        }

        Console.ReadKey();
    }
}
```

Содержание отчета

Отчет должен содержать:

1.Теоретическая часть**:** определение интерфейса; синтаксис объявления и реализации; преимущества использования интерфейсов

2.Практическая часть**: о**писание варианта задания; UML-диаграмма классов (опционально); полный код программы с комментариями; скриншоты результатов выполнения программы

3.Выводы по работе: что было изучено; какие навыки приобретены; какие сложности возникли

Контрольные вопросы

1.Что такое интерфейс в языке C#? Чем интерфейс отличается от абстрактного класса?

2.Какие члены может содержать интерфейс? Может ли интерфейс содержать поля или реализацию методов?

3.Как объявить интерфейс в C#? Какое соглашение об именовании используется для интерфейсов?

4.Что означает фраза «класс реализует интерфейс»? Какие обязательства накладывает на класс реализация интерфейса?

5.Может ли класс реализовывать несколько интерфейсов одновременно? Приведите пример синтаксиса.

6.Что такое полиморфизм через интерфейсы? Приведите пример использования переменной типа интерфейса.

7.Что такое явная реализация интерфейса? В каких случаях она применяется?

8.Может ли интерфейс наследоваться от другого интерфейса? Что при этом происходит?

9.Какие преимущества дает использование интерфейсов при разработке программного обеспечения?

10.Приведите примеры стандартных интерфейсов в .NET (например, IEnumerable, IComparable, IDisposable). Для чего они используются?

Практическая работа № 9 «Использование стандартных интерфейсов»

Цель работы: Изучить назначение и применение стандартных интерфейсов в C#, освоить практические навыки реализации и использования интерфейсов IComparable, IComparer,

IEnumerable, IEnumerator, ICloneable, IDisposable для создания гибких и расширяемых программных модулей.

Время выполнения: 4 ч

Теоретическая часть

Понятие интерфейса в C#. Интерфейс — это контракт, определяющий набор методов, свойств, событий и индексаторов, которые должен реализовать класс или структура. Интерфейс описывает «что должно быть сделано», но не определяет «как это сделать». Интерфейсы в C# позволяют:

- обеспечить полиморфизм без использования наследования классов
- создавать слабосвязанные системы
- реализовывать множественное наследование поведения
- следовать принципу инверсии зависимостей (SOLID)

Стандартные интерфейсы .NET. В библиотеке .NET Framework определено множество стандартных интерфейсов, которые обеспечивают унифицированное взаимодействие с объектами. Рассмотрим наиболее важные из них.

1. Интерфейс IComparable. Назначение: определяет метод сравнения текущего объекта с другим объектом того же типа. Используется для сортировки объектов. Объявление:

```
public interface IComparable
{
    int CompareTo(object obj);
}

public interface IComparable<T>
{
    int CompareTo(T other);
}
```

Возвращаемое значение метода CompareTo:
– отрицательное число — текущий объект меньше сравниваемого
– ноль — объекты равны
– положительное число — текущий объект больше сравниваемого

Пример реализации:

```
public class Student : IComparable<Student>
{
    public string Name { get; set; }
    public double AverageScore { get; set; }
```

```
    public int CompareTo(Student other)
    {
        if (other == null) return 1;
        return this.AverageScore.CompareTo(other.AverageScore);
    }
}
```

Применение:

```
List<Student> students = new List<Student>();
students.Sort(); // сортировка по AverageScore
```

2. Интерфейс IComparer. Назначение: определяет метод сравнения двух объектов. Позволяет создавать различные стратегии сравнения для одного типа. Объявление:

```
public interface IComparer
{
    int Compare(object x, object y);
}

public interface IComparer<T>
{
    int Compare(T x, T y);
}
```

Пример реализации:

```
public class StudentNameComparer : IComparer<Student>
{
    public int Compare(Student x, Student y)
    {
        if (x == null || y == null)
            return 0;
        return string.Compare(x.Name, y.Name);
    }
}
```

Применение:

```
List<Student> students = new List<Student>();
students.Sort(new StudentNameComparer()); // сортировка по имени
```

3. Интерфейсы IEnumerable и IEnumerator. Назначение: обеспечивают возможность перебора элементов коллекции с помощью цикла foreach. Объявление:

```
public interface IEnumerable
{
    IEnumerator GetEnumerator();
}

public interface IEnumerator
```

```
{
    object Current { get; }
    bool MoveNext();
    void Reset();
}
```

Обобщенные версии:

```
public interface IEnumerable<T> : IEnumerable
{
    IEnumerator<T> GetEnumerator();
}

public interface IEnumerator<T> : IEnumerator, IDisposable
{
    T Current { get; }
}
```

Пример реализации:

```
public class StudentGroup : IEnumerable<Student>
{
    private List<Student> students = new List<Student>();

    public void Add(Student student)
    {
        students.Add(student);
    }

    public IEnumerator<Student> GetEnumerator()
    {
        return students.GetEnumerator();
    }

    IEnumerator IEnumerable.GetEnumerator()
    {
        return GetEnumerator();
    }
}
```

Применение:

```
StudentGroup group = new StudentGroup();
foreach (Student student in group)
{
    Console.WriteLine(student.Name);
}
```

4. Интерфейс ICloneable. Назначение: создает копию объекта.
Объявление:

```
public interface ICloneable
{
   object Clone();
}
```

Типы копирования:

- Поверхностное копирование (Shallow Copy) — копируются значения полей; ссылочные поля указывают на те же объекты
- Глубокое копирование (Deep Copy) — создаются копии всех объектов, на которые ссылаются поля

Пример реализации (поверхностное копирование):

```
public class Student : ICloneable
{
   public string Name { get; set; }
   public double AverageScore { get; set; }

   public object Clone()
   {
      return this.MemberwiseClone();
   }
}
```

Пример реализации (глубокое копирование):

```
public class Student : ICloneable
{
   public string Name { get; set; }
   public Address HomeAddress { get; set; }

   public object Clone()
   {
      Student clone = (Student)this.MemberwiseClone();
      clone.HomeAddress = (Address)this.HomeAddress.Clone();
      return clone;
   }
}
```

5. Интерфейс IDisposable. Назначение: освобождает неуправляемые ресурсы (файлы, соединения с базой данных, сетевые подключения).
Объявление:

```
public interface IDisposable
```

```
{
    void Dispose();
}
```

Пример реализации:

```
public class FileLogger : IDisposable
{
    private StreamWriter writer;
    private bool disposed = false;

    public FileLogger(string path)
    {
        writer = new StreamWriter(path);
    }

    public void Log(string message)
    {
        if (disposed)
            throw new ObjectDisposedException("FileLogger");
        writer.WriteLine($"{DateTime.Now}: {message}");
    }

    public void Dispose()
    {
        Dispose(true);
        GC.SuppressFinalize(this);
    }

    protected virtual void Dispose(bool disposing)
    {
        if (!disposed)
        {
            if (disposing)
            {
                // освобождение управляемых ресурсов
                writer?.Close();
                writer?.Dispose();
            }
            // освобождение неуправляемых ресурсов
            disposed = true;
        }
    }

    ~FileLogger()
    {
```

```
        Dispose(false);
    }
}
```

Применение с использованием using:

```
using (FileLogger logger = new FileLogger("log.txt"))
{
    logger.Log("Начало работы");
    logger.Log("Операция выполнена");
} // автоматически вызывается Dispose()
```

6. Интерфейс IEquatable. Назначение: определяет строго типизированный метод для проверки равенства объектов. Объявление:

```
public interface IEquatable<T>
{
    bool Equals(T other);
}
```

Пример реализации:

```
public class Student : IEquatable<Student>
{
    public int Id { get; set; }
    public string Name { get; set; }

    public bool Equals(Student other)
    {
        if (other == null) return false;
        return this.Id == other.Id;
    }

    public override bool Equals(object obj)
    {
        return Equals(obj as Student);
    }

    public override int GetHashCode()
    {
        return Id.GetHashCode();
    }
}
```

Преимущества использования стандартных интерфейсов

1.Совместимость — объекты работают со стандартными методами .NET (Sort, foreach и др.)

2.Читаемость кода — разработчики знают назначение стандартных интерфейсов
3.Переиспользование — можно применять существующие алгоритмы и коллекции
4.Полиморфизм — единый способ работы с разными типами данных
5.Соответствие стандартам — код следует общепринятым практикам .NET

Рабочее задание: Выполните задание в соответствии с вариантом. Реализуйте класс и необходимые интерфейсы, продемонстрируйте их использование в консольном приложении.

Вариант 1. Система учета книг

Создайте класс **Book** со свойствами: Title (название), Author (автор), Year (год издания), ISBN (номер). Реализуйте интерфейсы:

- IComparable — сравнение по году издания
- ICloneable — создание копии книги
- IEquatable — сравнение по ISBN

Создайте классы-компараторы**:**

- BookTitleComparer — сравнение по названию
- BookAuthorComparer — сравнение по автору

Создайте класс Library, реализующий IEnumerable для хранения коллекции книг.

Продемонстрируйте: сортировку книг разными способами; поиск книги; клонирование книги; перебор книг в foreach

Вариант 2. Система учета студентов

Создайте класс **Student** со свойствами: FullName (ФИО), Group (группа), AverageScore (средний балл), StudentId (номер студенческого билета). Реализуйте интерфейсы:

- IComparable — сравнение по среднему баллу
- ICloneable — создание копии студента
- IEquatable — сравнение по StudentId

Создайте классы-компараторы:

- StudentNameComparer — сравнение по ФИО
- StudentGroupComparer — сравнение по группе

Создайте класс StudentGroup, реализующий IEnumerable.

Продемонстрируйте: сортировку студентов; поиск студента по

ID; получение списка отличников (средний балл > 4.5)

Вариант 3. Система управления задачами

Создайте класс **Task** со свойствами: Title (название), Description (описание), Priority (приоритет: 1-5), DueDate (срок выполнения), Status (статус). Реализуйте интерфейсы:

- IComparable — сравнение по приоритету
- ICloneable — создание копии задачи
- IEquatable — сравнение по Title

Создайте классы-компараторы**:**

- TaskDueDateComparer — сравнение по сроку
- TaskStatusComparer — сравнение по статусу

Создайте класс TaskManager, реализующий IEnumerable и IDisposable (запись в лог-файл).

Продемонстрируйте: добавление задач; сортировку по различным критериям; фильтрацию просроченных задач

Вариант 4. Система учета товаров на складе

Создайте класс **Product** со свойствами: Name (название), Category (категория), Price (цена), Quantity (количество), ArticleNumber (артикул). Реализуйте интерфейсы:

- IComparable — сравнение по цене
- ICloneable — создание копии товара
- IEquatable — сравнение по артикулу

Создайте классы-компараторы:

- ProductNameComparer — сравнение по названию
- ProductQuantityComparer — сравнение по количеству

Создайте класс Warehouse, реализующий IEnumerable и IDisposable.

Продемонстрируйте: добавление товаров; поиск товара по артикулу; получение списка товаров с низким остатком (Quantity < 10)

Вариант 5. Система учета сотрудников

Создайте класс **Employee** со свойствами: FullName (ФИО), Position (должность), Salary (зарплата), HireDate (дата приема), EmployeeId (табельный номер). Реализуйте интерфейсы:

- IComparable — сравнение по зарплате
- ICloneable — создание копии сотрудника
- IEquatable — сравнение по EmployeeId

Создайте классы-компараторы:

- EmployeeNameComparer — сравнение по ФИО
- EmployeeHireDateComparer — сравнение по дате приема

Создайте класс Department, реализующий IEnumerable.
Продемонстрируйте: сортировку сотрудников; расчет средней зарплаты по отделу; поиск сотрудников по должности

Вариант 6. Система учета транспортных средств

Создайте класс **Vehicle** со свойствами: Brand (марка), Model (модель), Year (год выпуска), Mileage (пробег), LicensePlate (гос. номер). Реализуйте интерфейсы:

- IComparable — сравнение по году выпуска
- ICloneable — создание копии транспорта
- IEquatable — сравнение по гос. номеру

Создайте классы-компараторы:

- VehicleMileageComparer — сравнение по пробегу
- VehicleBrandComparer — сравнение по марке

Создайте класс Fleet, реализующий IEnumerable.
Продемонстрируйте: сортировку транспорта; поиск по гос. Номеру; фильтрацию транспорта с пробегом > 100000 км

Вариант 7. Система учета курсов обучения

Создайте класс **Course** со свойствами: Title (название), Duration (длительность в часах), Price (стоимость), Instructor (преподаватель), CourseId (идентификатор). Реализуйте интерфейсы:

- IComparable — сравнение по стоимости
- ICloneable — создание копии курса
- IEquatable — сравнение по CourseId

Создайте классы-компараторы**:**

- CourseDurationComparer — сравнение по длительности
- CourseTitleComparer — сравнение по названию

Создайте класс CoursesCatalog, реализующий IEnumerable.
Продемонстрируйте: сортировку курсов; поиск курсов определенного преподавателя; фильтрацию курсов по ценовому диапазону

Вариант 8. Система учета медицинских карт

Создайте класс **MedicalCard** со свойствами: PatientName (ФИО пациента), BirthDate (дата рождения), CardNumber (номер карты), LastVisit (дата последнего визита), Diagnosis (диагноз).

Реализуйте интерфейсы:

- IComparable — сравнение по дате последнего визита
- ICloneable — создание копии карты
- IEquatable — сравнение по номеру карты

Создайте классы-компараторы:

- MedicalCardNameComparer — сравнение по ФИО
- MedicalCardBirthDateComparer — сравнение по дате рождения

Создайте класс MedicalRegistry, реализующий IEnumerable и IDisposable.

Продемонстрируйте: поиск карты по номеру; получение списка пациентов старше определенного возраста; сортировку карт

Вариант 9. Система учета банковских счетов

Создайте класс **BankAccount** со свойствами: AccountNumber (номер счета), Owner (владелец), Balance (баланс), OpenDate (дата открытия), AccountType (тип счета).

Реализуйте интерфейсы:

- IComparable — сравнение по балансу
- ICloneable — создание копии счета
- IEquatable — сравнение по номеру счета

Создайте классы-компараторы:

- BankAccountOwnerComparer — сравнение по владельцу
- BankAccountOpenDateComparer — сравнение по дате открытия

Создайте класс Bank, реализующий IEnumerable и IDisposable (логирование операций).

Продемонстрируйте: поиск счета по номеру; получение суммарного баланса всех счетов; сортировку счетов

Вариант 10. Система учета недвижимости

Создайте класс **RealEstate** со свойствами: Address (адрес), Area (площадь), Price (цена), Rooms (количество комнат), PropertyId (ID объекта). Реализуйте интерфейсы:

- IComparable — сравнение по цене
- ICloneable — создание копии объекта
- IEquatable — сравнение по PropertyId

Создайте классы-компараторы:

- RealEstateAreaComparer — сравнение по площади

- RealEstateRoomsComparer — сравнение по количеству комнат

Создайте класс RealEstateAgency**,** реализующий IEnumerable. Продемонстрируйте: сортировку объектов; поиск объектов в заданном ценовом диапазоне; расчет средней цены за квадратный метр

Требования к выполнению задания

1.Создайте консольное приложение в Visual Studio 2022 (.NET 6.0 или выше).

2.Реализуйте основной класс с необходимыми свойствами и конструктором.

3.Реализуйте все указанные интерфейсы: Правильно переопределите методы; Учитывайте проверку на null; Для ICloneable реализуйте глубокое копирование, если класс содержит ссылочные типы

4.Создайте классы-компараторы для альтернативных способов сравнения.

5.Реализуйте класс-коллекцию с IEnumerable.

6.Создайте тестовые данные (не менее 5 объектов).

7.Продемонстрируйте работу**:** Сортировку с использованием IComparable; Сортировку с использованием IComparer; Перебор элементов через foreach; Клонирование объектов и проверку независимости копий; Проверку равенства объектов; Использование IDisposable (если требуется по варианту)

8.Добавьте комментарии к коду, поясняющие реализацию интерфейсов.

9.Обработайте возможные исключения (например, при сравнении с null).

10.Оформите вывод результатов в консоль в читаемом виде.

Пример для варианта 1 (фрагмент):

```
using System;
using System.Collections;
using System.Collections.Generic;

namespace LibrarySystem
{
   // Класс Book с реализацией интерфейсов
```

```
public class Book : IComparable<Book>, ICloneable, IEquatable<Book>
{
    public string Title { get; set; }
    public string Author { get; set; }
    public int Year { get; set; }
    public string ISBN { get; set; }

    public Book(string title, string author, int year, string isbn)
    {
        Title = title;
        Author = author;
        Year = year;
        ISBN = isbn;
    }

    // Реализация IComparable<Book> - сравнение по году
    public int CompareTo(Book other)
    {
        if (other == null) return 1;
        return this.Year.CompareTo(other.Year);
    }

    // Реализация ICloneable
    public object Clone()
    {
        return new Book(this.Title, this.Author, this.Year, this.ISBN);
    }

    // Реализация IEquatable<Book> - сравнение по ISBN
    public bool Equals(Book other)
    {
        if (other == null) return false;
        return this.ISBN == other.ISBN;
    }

    public override bool Equals(object obj)
    {
        return Equals(obj as Book);
    }

    public override int GetHashCode()
    {
        return ISBN.GetHashCode();
    }
```

```
    public override string ToString()
    {
        return $"{Title} ({Author}, {Year}) - ISBN: {ISBN}";
    }
}

// Компаратор для сортировки по названию
public class BookTitleComparer : IComparer<Book>
{
    public int Compare(Book x, Book y)
    {
        if (x == null || y == null) return 0;
        return string.Compare(x.Title, y.Title);
    }
}

// Класс Library с реализацией IEnumerable<Book>
public class Library : IEnumerable<Book>
{
    private List<Book> books = new List<Book>();

    public void AddBook(Book book)
    {
        books.Add(book);
    }

    public IEnumerator<Book> GetEnumerator()
    {
        return books.GetEnumerator();
    }

    IEnumerator IEnumerable.GetEnumerator()
    {
        return GetEnumerator();
    }

    public void Sort()
    {
        books.Sort();
    }

    public void Sort(IComparer<Book> comparer)
    {
```

```
            books.Sort(comparer);
        }
    }

    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("=== Система учета книг ===\n");

            // Создание библиотеки и добавление книг
            Library library = new Library();
            library.AddBook(new Book("Война и мир", "Толстой Л.Н.", 1869,
"978-5-17-098424-4"));
            library.AddBook(new Book("Преступление и наказание", "Достоев-
ский Ф.М.", 1866, "978-5-17-082515-1"));
            library.AddBook(new Book("Мастер и Маргарита", "Булгаков М.А.",
1967, "978-5-17-107184-0"));

            // Демонстрация foreach (IEnumerable)
            Console.WriteLine("Все книги в библиотеке:");
            foreach (var book in library)
            {
                Console.WriteLine(book);
            }

            // Сортировка по году (IComparable)
            Console.WriteLine("\nКниги, отсортированные по году издания:");
            library.Sort();
            foreach (var book in library)
            {
                Console.WriteLine(book);
            }

            // Сортировка по названию (IComparer)
            Console.WriteLine("\nКниги, отсортированные по названию:");
            library.Sort(new BookTitleComparer());
            foreach (var book in library)
            {
                Console.WriteLine(book);
            }

            Console.ReadKey();
        }
```

```
    }
}
```

Содержание отчета

Отчет должен содержать:

1.Теоретическая часть – краткое описание использованных интерфейсов

2.Листинг программы с комментариями: код основного класса с реализованными интерфейсами; код классов-компараторов; код класса-коллекции; код программы для демонстрации

3.Результаты выполнения программы (скриншоты консольного вывода)

4.Выводы по работе: что освоено, какие навыки получены

Контрольные вопросы

1.Что такое интерфейс в C#? Чем интерфейс отличается от абстрактного класса?

2.Какие стандартные интерфейсы .NET вы знаете? Каково их назначение?

3.В чем разница между интерфейсами IComparable и IComparer? В каких случаях следует использовать каждый из них?

4.Что должен возвращать метод CompareTo интерфейса IComparable? Как интерпретируются различные возвращаемые значения?

5.Зачем нужен интерфейс IEnumerable? Какие методы необходимо реализовать для обеспечения перебора элементов в цикле foreach?

6.В чем разница между поверхностным и глубоким копированием при реализации интерфейса ICloneable? Приведите примеры.

7.Для чего используется интерфейс IDisposable? Что такое паттерн Dispose и как правильно его реализовать?

8.Почему рекомендуется использовать обобщенные версии интерфейсов (IComparable, IEnumerable) вместо необобщенных?

9.Можно ли в C# реализовать несколько интерфейсов в одном классе? Приведите пример.

10.Какие преимущества дает использование стандартных интерфейсов при разработке программных модулей?

Практическая работа № 10 «Работа с типом данных структура, использование коллекций»

Цель работы: Изучить принципы работы с пользовательскими типами данных (структурами) в языке C#, освоить основные методы работы с обобщенными коллекциями, научиться применять структуры и коллекции для решения практических задач обработки и хранения данных.

Время выполнения: 2 ч

Теоретическая часть

Структуры в C#. Структура (struct) — это пользовательский тип значения, который может содержать данные и функциональность. Структуры используются для представления небольших объектов, которые логически являются единым целым. Основные характеристики структур:

–Структуры являются типами значений (value type) и хранятся в стеке

–Не поддерживают наследование от других структур или классов

–Могут реализовывать интерфейсы

–Не могут иметь деструктор

–Все поля должны быть проинициализированы

–Автоматически имеют конструктор по умолчанию без параметров.

Синтаксис объявления структуры:

```
struct Название
{
    // поля
    public тип поле1;
    public тип поле2;

    // конструктор
    public Название(параметры)
    {
        // инициализация полей
    }

    // методы
    public void Метод()
```

```
    {
        // код метода
    }
}
```

Пример структуры:

```
struct Point
{
    public int X;
    public int Y;

    public Point(int x, int y)
    {
        X = x;
        Y = y;
    }

    public double GetDistance()
    {
        return Math.Sqrt(X * X + Y * Y);
    }

    public override string ToString()
    {
        return $"({X}, {Y})";
    }
}
```

Использование структуры:

```
Point p1 = new Point(3, 4);
Console.WriteLine(p1.ToString()); // (3, 4)
Console.WriteLine(p1.GetDistance()); // 5
```

Различия между структурами и классами

Характеристика	Структура (struct)	Класс (class)
Тип	Тип значения	Ссылочный тип
Где хранится	Стек	Куча
Наследование	Не поддерживает	Поддерживает
Конструктор по умолчанию	Всегда присутствует	Можно переопределить

Характеристика	Структура (struct)	Класс (class)
Производительность	Быстрее для малых объектов	Медленнее из-за работы с кучей
Присваивание	Копирует значения	Копирует ссылку

Коллекции в C#. Коллекция — это объект, предназначенный для хранения и управления группой связанных объектов. Пространство имен System.Collections.Generic содержит обобщенные коллекции, которые обеспечивают типобезопасность и лучшую производительность.

Основные обобщенные коллекции

1. List — динамический массив. List — коллекция элементов одного типа с возможностью динамического изменения размера. Основные методы и свойства:

- Add(T item) — добавить элемент в конец списка
- Insert(int index, T item) — вставить элемент на указанную позицию
- Remove(T item) — удалить первое вхождение элемента
- RemoveAt(int index) — удалить элемент по индексу
- Clear() — удалить все элементы
- Contains(T item) — проверить наличие элемента
- Count — количество элементов
- Sort() — сортировать элементы

Пример использования:

```
List<int> numbers = new List<int>();
numbers.Add(10);
numbers.Add(20);
numbers.Add(30);

numbers.Insert(1, 15); // [10, 15, 20, 30]
numbers.Remove(20); // [10, 15, 30]

foreach (int num in numbers)
{
   Console.WriteLine(num);
}
```

2. Dictionary<TKey, TValue> — словарь (ассоциативный массив). Dictionary<TKey, TValue> — коллекция пар "ключ-значение", где каждый ключ уникален. Основные методы и свойства:

- Add(TKey key, TValue value) — добавить пару ключ-значение
- Remove(TKey key) — удалить элемент по ключу
- ContainsKey(TKey key) — проверить наличие ключа
- ContainsValue(TValue value) — проверить наличие значения
- TryGetValue(TKey key, out TValue value) — безопасное получение значения
- Keys — коллекция ключей
- Values — коллекция значений
- Count — количество пар

Пример использования:

```
Dictionary<string, int> ages = new Dictionary<string, int>();
ages.Add("Иван", 20);
ages.Add("Мария", 22);
ages["Петр"] = 21; // альтернативный способ добавления

if (ages.ContainsKey("Иван"))
{
   Console.WriteLine($"Возраст Ивана: {ages["Иван"]}");
}

foreach (KeyValuePair<string, int> pair in ages)
{
   Console.WriteLine($"{pair.Key}: {pair.Value}");
}
```

3. Queue — очередь (FIFO - First In, First Out). Queue — коллекция, реализующая принцип "первым пришел — первым вышел".Основные методы:

- Enqueue(T item) — добавить элемент в конец очереди
- Dequeue() — извлечь и удалить первый элемент
- Peek() — получить первый элемент без удаления
- Clear() — очистить очередь
- Contains(T item) — проверить наличие элемента
- Count — количество элементов

Пример использования:

```
Queue<string> queue = new Queue<string>();
queue.Enqueue("Первый");
queue.Enqueue("Второй");
queue.Enqueue("Третий");

Console.WriteLine(queue.Dequeue()); // "Первый"
Console.WriteLine(queue.Peek()); // "Второй" (не удаляется)
```

4. Stack — стек (LIFO - Last In, First Out). Stack — коллекция, реализующая принцип "последним пришел — первым вышел". Основные методы**:**

- Push(T item) — добавить элемент на вершину стека
- Pop() — извлечь и удалить элемент с вершины
- Peek() — получить элемент с вершины без удаления
- Clear() — очистить стек
- Contains(T item) — проверить наличие элемента
- Count — количество элементов

Пример использования:

```
Stack<int> stack = new Stack<int>();
stack.Push(1);
stack.Push(2);
stack.Push(3);

Console.WriteLine(stack.Pop()); // 3
Console.WriteLine(stack.Peek()); // 2
```

5. HashSet — множество уникальных элементов. HashSet — коллекция уникальных элементов без определенного порядка. Основные методы:

- Add(T item) — добавить элемент (возвращает false, если уже существует)
- Remove(T item) — удалить элемент
- Contains(T item) — проверить наличие элемента
- UnionWith(IEnumerable<T>) — объединение множеств
- IntersectWith(IEnumerable<T>) — пересечение множеств
- ExceptWith(IEnumerable<T>) — разность множеств

Пример использования:

```
HashSet<int> set = new HashSet<int>();
set.Add(1);
set.Add(2);
```

```
set.Add(2); // не добавится, т.к. уже есть

HashSet<int> set2 = new HashSet<int> { 2, 3, 4 };
set.UnionWith(set2); // {1, 2, 3, 4}
```

Использование структур с коллекциями. Структуры часто используются в качестве элементов коллекций для эффективного хранения данных.

Пример:

```
struct Student
{
    public string Name;
    public int Age;
    public double GPA;

    public Student(string name, int age, double gpa)
    {
        Name = name;
        Age = age;
        GPA = gpa;
    }

    public override string ToString()
    {
        return $"{Name}, {Age} лет, средний балл: {GPA}";
    }
}

// Использование
List<Student> students = new List<Student>();
students.Add(new Student("Иван Петров", 19, 4.5));
students.Add(new Student("Мария Сидорова", 20, 4.8));

foreach (Student s in students)
{
    Console.WriteLine(s);
}
```

Рабочее задание: Выполните задание в соответствии с вариантом. Создайте консольное приложение в Visual Studio 2022, реализующее функциональность согласно заданию.

Вариант 1. Система учета товаров на складе

Создать структуру Product для хранения информации о товаре (код товара, название, цена, количество на складе, категория).

Реализовать программу, которая:

- Использует List<Product> для хранения товаров
- Позволяет добавлять новые товары
- Позволяет удалять товары по коду
- Выводит список всех товаров
- Находит товары по категории (использовать Dictionary<string, List<Product>>)
- Рассчитывает общую стоимость товаров на складе
- Находит самый дорогой и самый дешевый товар

Вариант 2. Система учета книг в библиотеке

Создать структуру Book для хранения информации о книге (ISBN, название, автор, год издания, количество страниц, статус выдачи). Реализовать программу, которая:

- Использует List<Book> для хранения книг
- Позволяет добавлять и удалять книги
- Выводит список всех книг
- Группирует книги по авторам (использовать Dictionary<string, List<Book>>)
- Находит книги по году издания
- Выводит статистику: общее количество книг, количество выданных книг, количество доступных
- Находит самую объемную книгу

Вариант 3. Система учета студентов

Создать структуру Student для хранения информации о студенте (номер зачетки, ФИО, группа, курс, средний балл, специальность). Реализовать программу, которая:

- Использует List<Student> для хранения студентов
- Позволяет добавлять и удалять студентов
- Выводит список всех студентов
- Группирует студентов по группам (использовать Dictionary<string, List<Student>>)
- Находит студентов с баллом выше заданного
- Рассчитывает средний балл по группе
- Находит лучшего студента (по среднему баллу)

Вариант 4. Система учета сотрудников

Создать структуру Employee для хранения информации о сотруднике (табельный номер, ФИО, должность, отдел, зарплата, дата приема). Реализовать программу, которая:

- Использует List<Employee> для хранения сотрудников
- Позволяет добавлять и удалять сотрудников
- Выводит список всех сотрудников
- Группирует сотрудников по отделам (использовать Dictionary<string, List<Employee>>)
- Рассчитывает фонд заработной платы по отделу
- Находит сотрудника с максимальной и минимальной зарплатой
- Выводит список сотрудников, работающих более заданного количества лет

Вариант 5. Система учета автомобилей в автопарке

Создать структуру Car для хранения информации об автомобиле (государственный номер, марка, модель, год выпуска, пробег, статус - свободен/занят). Реализовать программу, которая:

- Использует List<Car> для хранения автомобилей
- Позволяет добавлять и удалять автомобили
- Выводит список всех автомобилей
- Группирует автомобили по маркам (использовать Dictionary<string, List<Car>>)
- Находит свободные автомобили
- Находит автомобили старше заданного года выпуска
- Рассчитывает средний пробег всех автомобилей

Вариант 6. Система учета заказов в интернет-магазине

Создать структуру Order для хранения информации о заказе (номер заказа, имя клиента, дата заказа, сумма заказа, статус, способ оплаты). Реализовать программу, которая:

- Использует List<Order> для хранения заказов
- Позволяет добавлять новые заказы
- Позволяет изменять статус заказа
- Выводит список всех заказов
- Группирует заказы по статусу (использовать Dictionary<string, List<Order>>)
- Рассчитывает общую сумму заказов за период
- Находит самый крупный заказ
- Выводит статистику по способам оплаты

Вариант 7. Система учета медицинских карт пациентов

Создать структуру Patient для хранения информации о пациенте (номер карты, ФИО, дата рождения, диагноз, лечащий врач, дата последнего визита). Реализовать программу, которая:

- Использует List<Patient> для хранения пациентов
- Позволяет добавлять и удалять карты пациентов
- Выводит список всех пациентов
- Группирует пациентов по врачам (использовать Dictionary<string, List<Patient>>)
- Находит пациентов по диагнозу
- Выводит список пациентов, не посещавших врача более заданного количества дней
- Рассчитывает возраст пациента

Вариант 8. Система учета занятий в расписании

Создать структуру Lesson для хранения информации о занятии (день недели, время начала, название дисциплины, преподаватель, аудитория, группа). Реализовать программу, которая:

- Использует List<Lesson> для хранения занятий
- Позволяет добавлять и удалять занятия
- Выводит расписание на день
- Группирует занятия по группам (использовать Dictionary<string, List<Lesson>>)
- Находит все занятия преподавателя
- Проверяет занятость аудитории в заданное время
- Выводит расписание для конкретной группы

Вариант 9. Система учета счетов за коммунальные услуги

Создать структуру Bill для хранения информации о счете (номер лицевого счета, адрес, месяц, год, сумма за электричество, сумма за воду, сумма за газ, статус оплаты). Реализовать программу, которая:

- Использует List<Bill> для хранения счетов
- Позволяет добавлять новые счета
- Позволяет отмечать счета как оплаченные
- Выводит список всех счетов
- Группирует счета по адресам (использовать Dictionary<string, List<Bill>>)
- Рассчитывает общую сумму к оплате
- Находит неоплаченные счета

- Рассчитывает среднюю стоимость услуг за месяц

Вариант 10. Система учета билетов в кинотеатре

Создать структуру Ticket для хранения информации о билете (номер билета, название фильма, дата и время сеанса, номер зала, номер места, цена, категория - обычный/VIP). Реализовать программу, которая:

- Использует List<Ticket> для хранения проданных билетов
- Позволяет продавать билеты (добавлять в список)
- Выводит список всех проданных билетов
- Группирует билеты по фильмам (использовать Dictionary<string, List<Ticket>>)
- Находит свободные места на сеанс (использовать HashSet<int> для занятых мест)
- Рассчитывает выручку за день
- Находит самый популярный фильм (по количеству проданных билетов)

Требования к выполнению задания

1.Создайте структуру в соответствии с вариантом: объявите все необходимые поля с соответствующими типами данных; создайте конструктор для инициализации всех полей; переопределите метод ToString() для удобного вывода информации; при необходимости добавьте дополнительные методы

2.Реализуйте основную функциональность: создайте меню для взаимодействия с пользователем; реализуйте все операции, указанные в варианте задания; используйте соответствующие коллекции (List, Dictionary, HashSet и др.); обеспечьте корректную обработку пользовательского ввода.

3.Организуйте код**: и**спользуйте осмысленные имена переменных и методов; добавьте комментарии к ключевым участкам кода; разделите функциональность на отдельные методы; следуйте принципам читаемости кода.

4.Протестируйте программу**: п**роверьте работу всех функций меню; убедитесь в корректности расчетов; проверьте обработку граничных случаев

5.Оформите отчет, содержащий: листинг программы с комментариями; скриншоты работы программы (демонстрация всех функций); выводы по работе

Пример выполнения (демонстрационный вариант)

Структура "Точка на плоскости"

```
using System;
using System.Collections.Generic;

namespace StructuresAndCollections
{
    struct Point
    {
        public double X;
        public double Y;

        public Point(double x, double y)
        {
            X = x;
            Y = y;
        }

        // Расчет расстояния от начала координат
        public double GetDistance()
        {
            return Math.Sqrt(X * X + Y * Y);
        }

        public override string ToString()
        {
            return $"Точка ({X:F2}, {Y:F2}), расстояние от начала: {GetDis-
tance():F2}";
        }
    }

    class Program
    {
        static List<Point> points = new List<Point>();

        static void Main(string[] args)
        {
            bool exit = false;

            while (!exit)
```

```
    {
        Console.WriteLine("\n=== МЕНЮ ===");
        Console.WriteLine("1. Добавить точку");
        Console.WriteLine("2. Показать все точки");
        Console.WriteLine("3. Найти самую удаленную точку");
        Console.WriteLine("4. Очистить список");
        Console.WriteLine("0. Выход");
        Console.Write("Выберите действие: ");

        string choice = Console.ReadLine();

        switch (choice)
        {
            case "1":
                AddPoint();
                break;
            case "2":
                ShowAllPoints();
                break;
            case "3":
                FindFarthestPoint();
                break;
            case "4":
                points.Clear();
                Console.WriteLine("Список очищен.");
                break;
            case "0":
                exit = true;
                break;
            default:
                Console.WriteLine("Неверный выбор!");
                break;
        }
    }
}

static void AddPoint()
{
    try
    {
        Console.Write("Введите X: ");
        double x = double.Parse(Console.ReadLine());
        Console.Write("Введите Y: ");
        double y = double.Parse(Console.ReadLine());
```

```
            points.Add(new Point(x, y));
            Console.WriteLine("Точка добавлена!");
        }
        catch (Exception ex)
        {
            Console.WriteLine($"Ошибка ввода: {ex.Message}");
        }
    }

    static void ShowAllPoints()
    {
        if (points.Count == 0)
        {
            Console.WriteLine("Список пуст.");
            return;
        }

        Console.WriteLine("\nВсе точки:");
        for (int i = 0; i < points.Count; i++)
        {
            Console.WriteLine($"{i + 1}. {points[i]}");
        }
    }

    static void FindFarthestPoint()
    {
        if (points.Count == 0)
        {
            Console.WriteLine("Список пуст.");
            return;
        }

        Point farthest = points[0];
        double maxDistance = farthest.GetDistance();

        foreach (Point p in points)
        {
            double distance = p.GetDistance();
            if (distance > maxDistance)
            {
                maxDistance = distance;
                farthest = p;
            }
```

```
        }

        Console.WriteLine($"\nСамая удаленная точка: {farthest}");
    }
  }
}
```

Контрольные вопросы

1.Что такое структура в C#? Чем структура отличается от класса?

2.Где хранятся структуры и классы в памяти? Как это влияет на производительность?

3.Какие ограничения существуют для структур в C#?

4.Что произойдет при присваивании одной структурной переменной другой? Как это отличается от присваивания объектов класса?

5.Что такое обобщенные коллекции? Какие преимущества они дают по сравнению с необобщенными коллекциями?

6.Опишите назначение и основные методы коллекции List<T>. В каких случаях ее использование предпочтительнее массива?

7.Что такое Dictionary<TKey, TValue>? Какие требования предъявляются к ключам словаря?

8.В чем разница между Queue<T> и Stack<T>? Приведите примеры практического применения каждой из этих коллекций.

9.Для чего используется HashSet<T>? Какое главное свойство этой коллекции?

10.Можно ли использовать структуры в качестве элементов коллекций? Приведите пример.

11.Как перебрать все элементы коллекции List<T>? Какие способы существуют?

12.Что возвращает метод Add() коллекции HashSet<T> и почему это может быть полезно?

13.В чем разница между методами Remove() и RemoveAt() коллекции List<T>?

14.Почему метод TryGetValue() словаря безопаснее обращения по индексу через []?

15.Можно ли изменить элемент структуры, находящийся в коллекции List<T>, напрямую? Как правильно изменить такой элемент?

Практическая работа № 11 «Операции со списками»

Цель работы: Изучить принципы работы со списками в языке C#, освоить основные методы и операции для управления коллекциями типа List<T>, научиться применять списки для решения практических задач.

Время выполнения: 2 ч

Теоретическая часть

Понятие списка в C#. Список (List) — это универсальная коллекция, представляющая динамический массив объектов одного типа. В отличие от обычных массивов, списки могут динамически изменять свой размер в процессе выполнения программы. Список является частью пространства имен System.Collections.Generic и реализует интерфейсы IList<T>, ICollection<T> и IEnumerable<T>.

Создание списка. Для работы со списками необходимо подключить пространство имен:

```
using System.Collections.Generic;
```

Способы создания списка:

```
// Создание пустого списка
List<int> numbers = new List<int>();

// Создание списка с начальной емкостью
List<string> names = new List<string>(10);

// Создание списка с инициализацией элементов
List<double> prices = new List<double> { 10.5, 20.3, 15.7 };

// Создание списка из массива
int[] array = { 1, 2, 3, 4, 5 };
List<int> listFromArray = new List<int>(array);
```

Основные свойства списка. Count — возвращает количество элементов в списке:

```
List<int> numbers = new List<int> { 1, 2, 3, 4, 5 };
int count = numbers.Count; // count = 5
```

Capacity — возвращает или устанавливает общее количество элементов, которое может содержать список без изменения

размера:

```
List<int> numbers = new List<int>(20);
int capacity = numbers.Capacity; // capacity = 20
```

Индексатор — позволяет обращаться к элементам списка по индексу:

```
List<string> fruits = new List<string> { "Apple", "Banana", "Orange" };
string first = fruits[0]; // "Apple"
fruits[1] = "Grape"; // изменение элемента
```

Основные методы списка

Добавление элементов. **Add(T item)** — добавляет элемент в конец списка:

```
List<int> numbers = new List<int>();
numbers.Add(10);
numbers.Add(20);
// numbers: [10, 20]
```

AddRange(IEnumerable collection) — добавляет коллекцию элементов в конец списка:

```
List<int> numbers = new List<int> { 1, 2, 3 };
numbers.AddRange(new int[] { 4, 5, 6 });
// numbers: [1, 2, 3, 4, 5, 6]
```

Insert(int index, T item) — вставляет элемент в указанную позицию:

```
List<string> colors = new List<string> { "Red", "Blue" };
colors.Insert(1, "Green");
// colors: ["Red", "Green", "Blue"]
```

InsertRange(int index, IEnumerable collection) — вставляет коллекцию элементов начиная с указанной позиции:

```
List<int> numbers = new List<int> { 1, 5 };
numbers.InsertRange(1, new int[] { 2, 3, 4 });
// numbers: [1, 2, 3, 4, 5]
```

Удаление элементов

Remove(T item) — удаляет первое вхождение указанного элемента:

```
List<int> numbers = new List<int> { 1, 2, 3, 2, 4 };
numbers.Remove(2);
// numbers: [1, 3, 2, 4]
```

RemoveAt(int index) — удаляет элемент по указанному индексу:

```
List<string> names = new List<string> { "Anna", "Bob", "Charlie" };
names.RemoveAt(1);
// names: ["Anna", "Charlie"]
```

RemoveRange(int index, int count) — удаляет диапазон элементов:

```
List<int> numbers = new List<int> { 1, 2, 3, 4, 5, 6 };
numbers.RemoveRange(1, 3);
// numbers: [1, 5, 6]
```

RemoveAll(Predicate match) — удаляет все элементы, удовлетворяющие условию:

```
List<int> numbers = new List<int> { 1, 2, 3, 4, 5, 6 };
numbers.RemoveAll(x => x % 2 == 0);
// numbers: [1, 3, 5]
```

Clear() — удаляет все элементы из списка:

```
List<int> numbers = new List<int> { 1, 2, 3 };
numbers.Clear();
// numbers: []
```

Поиск элементов

Contains(T item) — определяет, содержится ли элемент в списке:

```
List<string> fruits = new List<string> { "Apple", "Banana", "Orange" };
bool hasApple = fruits.Contains("Apple"); // true
```

IndexOf(T item) — возвращает индекс первого вхождения элемента:

```
List<int> numbers = new List<int> { 10, 20, 30, 20, 40 };
int index = numbers.IndexOf(20); // index = 1
```

LastIndexOf(T item) — возвращает индекс последнего вхождения элемента:

```
List<int> numbers = new List<int> { 10, 20, 30, 20, 40 };
int lastIndex = numbers.LastIndexOf(20); // lastIndex = 3
```

Find(Predicate match) — находит первый элемент, удовлетворяющий условию:

```
List<int> numbers = new List<int> { 1, 2, 3, 4, 5 };
int result = numbers.Find(x => x > 3); // result = 4
```

FindAll(Predicate match) — находит все элементы, удовлетворяющие условию:

```
List<int> numbers = new List<int> { 1, 2, 3, 4, 5, 6 };
List<int> evenNumbers = numbers.FindAll(x => x % 2 == 0);
// evenNumbers: [2, 4, 6]
```

Exists(Predicate match) — проверяет, существует ли элемент, удовлетворяющий условию:

```
List<int> numbers = new List<int> { 1, 2, 3, 4, 5 };
bool hasEven = numbers.Exists(x => x % 2 == 0); // true
```

Сортировка и изменение порядка

Sort() — сортирует элементы списка:

```
List<int> numbers = new List<int> { 5, 2, 8, 1, 9 };
numbers.Sort();
// numbers: [1, 2, 5, 8, 9]
```

Sort(Comparison comparison) — сортирует элементы с использованием делегата сравнения:

```
List<string> names = new List<string> { "Charlie", "Anna", "Bob" };
names.Sort((x, y) => x.Length.CompareTo(y.Length));
// Сортировка по длине строки
```

Reverse() — изменяет порядок элементов на обратный:

```
List<int> numbers = new List<int> { 1, 2, 3, 4, 5 };
numbers.Reverse();
// numbers: [5, 4, 3, 2, 1]
```

Преобразование и копирование

ToArray() — копирует элементы списка в новый массив:

```
List<int> numbers = new List<int> { 1, 2, 3, 4, 5 };
int[] array = numbers.ToArray();
```

GetRange(int index, int count) — создает поверхностную копию диапазона элементов:

```
List<int> numbers = new List<int> { 1, 2, 3, 4, 5, 6 };
List<int> subList = numbers.GetRange(1, 3);
// subList: [2, 3, 4]
```

ForEach(Action action) — выполняет указанное действие для каждого элемента:

```
List<int> numbers = new List<int> { 1, 2, 3, 4, 5 };
numbers.ForEach(x => Console.WriteLine(x * 2));
```

Обход списка

Цикл foreach:

```
List<string> names = new List<string> { "Anna", "Bob", "Charlie" };
foreach (string name in names)
{
    Console.WriteLine(name);
}
```

Цикл for:

```
List<int> numbers = new List<int> { 1, 2, 3, 4, 5 };
for (int i = 0; i < numbers.Count; i++)
{
    Console.WriteLine(numbers[i]);
}
```

Цикл while:

```
List<int> numbers = new List<int> { 1, 2, 3, 4, 5 };
int index = 0;
while (index < numbers.Count)
{
    Console.WriteLine(numbers[index]);
    index++;
}
```

Списки пользовательских типов

Списки могут содержать объекты пользовательских классов:

```
public class Student
{
    public string Name { get; set; }
    public int Age { get; set; }
    public double GPA { get; set; }
}

List<Student> students = new List<Student>();
students.Add(new Student { Name = "Иван", Age = 20, GPA = 4.5 });
students.Add(new Student { Name = "Мария", Age = 19, GPA = 4.8 });

// Поиск студента
Student found = students.Find(s => s.Name == "Иван");

// Сортировка по среднему баллу
students.Sort((s1, s2) => s2.GPA.CompareTo(s1.GPA));
```

Производительность операций

Операция	Сложность
Добавление в конец (Add)	O(1) амортизированная
Вставка в начало/середину (Insert)	O(n)
Удаление по индексу (RemoveAt)	O(n)
Доступ по индексу	O(1)
Поиск элемента (Contains, IndexOf)	O(n)
Очистка (Clear)	O(n)

Рабочее задание: Выполните программирование в соответствии с вариантом задания. Программа должна быть реализована в Visual Studio 2022 с использованием Windows Forms или Console Application.

Вариант 1. Управление списком товаров. Разработать приложение для управления списком товаров в магазине. Реализовать следующие функции:
– добавление товара (название, цена, количество);
– удаление товара по названию;
– поиск товаров по диапазону цен;
– сортировка товаров по цене (по возрастанию и убыванию);
– вывод всех товаров со стоимостью ниже заданной;
– подсчет общей стоимости всех товаров на складе;
– вывод товаров, количество которых меньше заданного значения.

Вариант 2. Список студентов группы. Разработать приложение для работы со списком студентов. Реализовать следующие функции:
– добавление студента (ФИО, номер группы, средний балл);
– удаление студента по ФИО;
– поиск студентов с баллом выше заданного значения;
– сортировка студентов по среднему баллу;
– вывод студентов определенной группы;
– подсчет среднего балла по всей группе;
– определение студента с максимальным и минимальным баллом.

Вариант 3. Телефонный справочник. Разработать приложение «Телефонный справочник». Реализовать следующие функции:
– добавление контакта (имя, номер телефона, email);
– удаление контакта по имени;
– поиск контакта по имени или номеру телефона;
– сортировка контактов по имени (алфавитный порядок);
– вывод всех контактов;
– редактирование данных контакта;
– подсчет общего количества контактов.

Вариант 4. Учет книг в домашней библиотеке. Разработать приложение для учета книг в домашней библиотеке. Реализовать следующие функции:
– добавление книги (название, автор, год издания, жанр);
– удаление книги по названию;
– поиск книг по автору;
– поиск книг по жанру;
– сортировка книг по году издания;
– вывод книг, изданных после заданного года;
– подсчет количества книг каждого жанра.
Вариант 5. Управление задачами (To-Do List). Разработать приложение для управления списком задач. Реализовать следующие функции:
– добавление задачи (название, описание, приоритет, срок выполнения);
– удаление задачи;
– отметка задачи как выполненной;
– сортировка задач по приоритету;
– сортировка задач по сроку выполнения;
– вывод только невыполненных задач;
– поиск задач по ключевому слову в названии или описании.
Вариант 6. Учет сотрудников организации. Разработать приложение для учета сотрудников. Реализовать следующие функции:
– добавление сотрудника (ФИО, должность, зарплата, стаж);
– удаление сотрудника по ФИО;
– поиск сотрудников по должности;
– сортировка сотрудников по зарплате;
– вывод сотрудников со стажем больше заданного значения;
– подсчет средней зарплаты по организации;
– определение сотрудника с максимальной и минимальной зарплатой.
Вариант 7. Список фильмов. Разработать приложение для управления коллекцией фильмов. Реализовать следующие функции:
– добавление фильма (название, режиссер, год выпуска, жанр, рейтинг);

– удаление фильма по названию;
– поиск фильмов по режиссеру;
– поиск фильмов по жанру;
– сортировка фильмов по рейтингу;
– вывод фильмов с рейтингом выше заданного значения;
– подсчет среднего рейтинга всех фильмов.

Вариант 8. Учет расходов. Разработать приложение для учета личных расходов. Реализовать следующие функции:
– добавление записи о расходе (дата, категория, сумма, описание);
– удаление записи;
– вывод расходов по определенной категории;
– сортировка расходов по дате;
– сортировка расходов по сумме;
– подсчет общей суммы расходов;
– подсчет суммы расходов по каждой категории;
– вывод расходов за определенный период времени.

Вариант 9. Каталог музыкальных композиций. Разработать приложение для управления каталогом музыки. Реализовать следующие функции:
– добавление композиции (название, исполнитель, альбом, жанр, длительность);
– удаление композиции;
– поиск композиций по исполнителю;
– поиск композиций по жанру;
– сортировка композиций по длительности;
– вывод композиций из определенного альбома;
– подсчет общей длительности всех композиций.

Вариант 10. Учет автомобилей на стоянке. Разработать приложение для учета автомобилей на парковке. Реализовать следующие функции:
– добавление автомобиля (марка, модель, номер, цвет, год выпуска);
– удаление автомобиля по номеру;
– поиск автомобилей по марке;
– поиск автомобилей определенного цвета;
– сортировка автомобилей по году выпуска;

– вывод автомобилей, выпущенных после заданного года;
– подсчет количества автомобилей каждой марки.

Требования к выполнению задания

1.Создайте класс для хранения данных в соответствии с вариантом (например, класс Product, Student, Contact и т.д.) со всеми необходимыми свойствами.

2.Реализуйте все указанные функции с использованием методов класса List<T>.

3.Обеспечьте пользовательский интерфейс**:** *для Console Application — текстовое меню с выбором операций; *для Windows Forms — графический интерфейс с кнопками, текстовыми полями и списками.

4.Обработайте возможные исключения: *проверка на пустой список при операциях поиска и удаления; *корректность введенных данных; *обработка выхода индекса за границы списка.

5.Добавьте валидацию данных**:** *проверка на пустые строки; *проверка корректности числовых значений; *проверка формата email или номера телефона (для соответствующих вариантов).

6.Используйте комментарии в коде для пояснения логики работы сложных участков.

7.Протестируйте программу на различных наборах данных.

Оформление отчета

Отчет должен содержать:

1.Описание разработанного приложения: структура классов; описание реализованных функций; UML-диаграмма класса (опционально).

2.Исходный код программы с комментариями: код класса для хранения данных; код основной формы или программы; фрагменты кода ключевых функций.

3.Скриншоты работы программы: главное окно приложения; примеры выполнения каждой функции; обработка ошибочных ситуаций.

4.Результаты тестирования: тестовые данные; результаты выполнения операций; проверка корректности работы всех функций.

5.Выводы по работе: что было изучено; какие навыки приобретены; какие трудности возникли при выполнении работы и как они были преодолены.

Контрольные вопросы

1.Что такое список (List<T>) в C#? Чем список отличается от массива?

2.В каком пространстве имен находится класс List<T>? Какие интерфейсы он реализует?

3.Какие существуют способы создания и инициализации списка в C#?

4.В чем разница между свойствами Count и Capacity списка?

5.Какие методы используются для добавления элементов в список? В чем разница между Add() и Insert()?

6.Какие методы используются для удаления элементов из списка? В чем разница между Remove(), RemoveAt() и RemoveAll()?

7.Как найти элемент в списке? Какие методы поиска существуют? В чем разница между Find() и FindAll()?

8.Как отсортировать список? Какие варианты сортировки доступны?

9.Какова вычислительная сложность основных операций со списком (добавление, удаление, поиск)?

10.Как преобразовать список в массив и наоборот?

11.Что такое предикат (Predicate<T>)? В каких методах списка он используется?

12.Как правильно обойти список и удалить из него элементы, удовлетворяющие определенному условию?

13.Можно ли хранить в списке объекты разных типов? Если да, то как это реализовать?

14.Что произойдет при попытке обратиться к элементу списка по несуществующему индексу?

15.В каких случаях следует использовать список, а в каких — массив или другую коллекцию?

Практическая работа № 12 «Разработка приложения с использованием текстовых компонентов»

Цель работ: Изучить возможности текстовых компонентов в Windows Forms приложениях на языке C#, освоить методы работы с элементами управления TextBox, Label, RichTextBox, MaskedTextBox в среде Visual Studio 2022, научиться обрабатывать события и выполнять валидацию пользовательского ввода.
Время выполнения: 4 ч
Теоретическая часть
Текстовые компоненты Windows Forms. Текстовые компоненты являются базовыми элементами пользовательского интерфейса, обеспечивающими взаимодействие с пользователем через ввод и отображение текстовой информации. В Windows Forms приложениях на C# существует несколько типов текстовых компонентов, каждый из которых предназначен для решения определенных задач.
Основные текстовые компоненты
1. Label (Метка). Label — компонент для отображения статического текста, который обычно не редактируется пользователем.
Основные свойства:

- Text — текст метки
- Font — шрифт текста
- ForeColor — цвет текста
- AutoSize — автоматическая настройка размера под текст
- TextAlign — выравнивание текста

Пример использования:

```
Label lblTitle = new Label();
lblTitle.Text = "Введите ваше имя:";
lblTitle.Font = new Font("Arial", 12, FontStyle.Bold);
lblTitle.AutoSize = true;
```

2. TextBox (Текстовое поле). TextBox — основной компонент для ввода однострочного или многострочного текста.
Основные свойства:

- Text — содержимое текстового поля
- Multiline — режим многострочного ввода
- PasswordChar — символ для скрытия пароля
- MaxLength — максимальная длина текста
- ReadOnly — режим "только для чтения"

- ScrollBars — полосы прокрутки (для многострочного режима)
- WordWrap — перенос слов

Основные события:

- TextChanged — изменение текста
- KeyPress — нажатие клавиши
- KeyDown — клавиша нажата
- Leave — потеря фокуса
- Enter — получение фокуса

Пример использования:

```
TextBox txtName = new TextBox();
txtName.MaxLength = 50;
txtName.Width = 200;
txtName.TextChanged += TxtName_TextChanged;

private void TxtName_TextChanged(object sender, EventArgs e)
{
   lblOutput.Text = "Вы ввели: " + txtName.Text;
}
```

3. RichTextBox (Форматированное текстовое поле). RichTextBox — расширенный компонент для работы с форматированным текстом (RTF).

Основные свойства:

- Rtf — текст в формате RTF
- SelectionFont — шрифт выделенного текста
- SelectionColor — цвет выделенного текста
- SelectionBackColor — цвет фона выделенного текста
- SelectionAlignment — выравнивание выделенного текста
- SelectionBullet — маркированный список

Основные методы:

- LoadFile() — загрузка файла
- SaveFile() — сохранение файла
- Find() — поиск текста
- AppendText() — добавление текста в конец

Пример использования:

```
RichTextBox rtbEditor = new RichTextBox();
rtbEditor.SelectionFont = new Font("Times New Roman", 14);
rtbEditor.SelectionColor = Color.Blue;
rtbEditor.AppendText("Форматированный текст");
```

4. MaskedTextBox (Текстовое поле с маской ввода).

MaskedTextBox — компонент для ввода данных в определенном формате с использованием маски.

Основные свойства:

- Mask — маска ввода
- PromptChar — символ-подсказка для незаполненных позиций
- TextMaskFormat — формат возвращаемого текста

Символы маски:

- 0 — обязательная цифра (0-9)
- 9 — необязательная цифра или пробел
- # — необязательная цифра, пробел, знак + или -
- L — обязательная буква (a-z, A-Z)
- ? — необязательная буква
- A — обязательная буква или цифра
- a — необязательная буква или цифра
- . — десятичный разделитель
- , — разделитель тысяч
- / — разделитель даты
- : — разделитель времени
- \ — экранирование символа маски

Примеры масок:

```
// Телефон: +7 (999) 000-00-00
mtbPhone.Mask = "+7 (999) 000-00-00";

// Дата: 00.00.0000
mtbDate.Mask = "00.00.0000";

// ИНН: 000000000000
mtbINN.Mask = "000000000000";

// Время: 00:00
mtbTime.Mask = "00:00";
```

Валидация пользовательского ввода

Валидация — это проверка корректности введенных пользователем данных. Существует несколько способов реализации валидации:

1. Валидация в событии KeyPress. Позволяет ограничить ввод определенными символами:

```
private void txtAge_KeyPress(object sender, KeyPressEventArgs e)
{
    // Разрешаем только цифры и клавишу Backspace
    if (!char.IsDigit(e.KeyChar) && e.KeyChar != (char)Keys.Back)
    {
        e.Handled = true; // Блокируем ввод
    }
}
```

2. Валидация в событии Validating. Выполняется при потере фокуса компонентом:

```
private void txtEmail_Validating(object sender, CancelEventArgs e)
{
    string email = txtEmail.Text;
    if (!email.Contains("@") || !email.Contains("."))
    {
        MessageBox.Show("Введите корректный email!", "Ошибка",
            MessageBoxButtons.OK, MessageBoxIcon.Warning);
        e.Cancel = true; // Не позволяем покинуть поле
    }
}
```

3. Использование регулярных выражений. Для сложной валидации используется класс Regex:

```
using System.Text.RegularExpressions;
private bool ValidateEmail(string email)
{
    string pattern = @"^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}$";
    return Regex.IsMatch(email, pattern);
}
```

4. Компонент ErrorProvider. Визуальное отображение ошибок валидации:

```
ErrorProvider errorProvider = new ErrorProvider();
private void btnSave_Click(object sender, EventArgs e)
{
    errorProvider.Clear();

    if (string.IsNullOrWhiteSpace(txtName.Text))
    {
        errorProvider.SetError(txtName, "Поле обязательно для заполнения");
        return;
    }
```

```
    if (txtAge.Text.Length == 0 || int.Parse(txtAge.Text) < 18)
    {
        errorProvider.SetError(txtAge, "Возраст должен быть не менее 18 лет");
        return;
    }
}
```

Обработка событий клавиатуры. При работе с текстовыми компонентами часто используются следующие события:

KeyPress. Возникает при нажатии символьной клавиши:

```
private void txt_KeyPress(object sender, KeyPressEventArgs e)
{
    if (e.KeyChar == (char)Keys.Enter)
    {
        // Обработка Enter
    }
}
```

KeyDown. Возникает при нажатии любой клавиши:

```
private void txt_KeyDown(object sender, KeyEventArgs e)
{
    if (e.Control && e.KeyCode == Keys.A)
    {
        // Ctrl+A - выделить все
        ((TextBox)sender).SelectAll();
    }
}
```

KeyUp. Возникает при отпускании клавиши.

Работа с буфером обмена

```
// Копирование
if (txtSource.SelectionLength > 0)
{
    Clipboard.SetText(txtSource.SelectedText);
}

// Вставка
if (Clipboard.ContainsText())
{
    txtDestination.Text = Clipboard.GetText();
}

// Вырезание
if (txtSource.SelectionLength > 0)
{
```

```
    Clipboard.SetText(txtSource.SelectedText);
    txtSource.SelectedText = "";
}
```

Форматирование текста в RichTextBox

```
// Изменение шрифта выделенного текста
if (rtb.SelectionLength > 0)
{
    rtb.SelectionFont = new Font("Arial", 14, FontStyle.Bold);
    rtb.SelectionColor = Color.Red;
}

// Выравнивание
rtb.SelectionAlignment = HorizontalAlignment.Center;

// Маркированный список
rtb.SelectionBullet = true;

// Отступ
rtb.SelectionIndent = 10;
```

Практические рекомендации

1.Используйте осмысленные имена компонентов: вместо textBox1 используйте txtName, txtEmail
2.Всегда валидируйте пользовательский ввод перед обработкой данных
3.Предоставляйте понятную обратную связь при ошибках ввода
4.Используйте MaskedTextBox для полей с фиксированным форматом (телефон, дата)
5.Устанавливайте MaxLength для ограничения длины ввода
6.Используйте PasswordChar для полей с паролями
7.Обрабатывайте исключения при преобразовании типов данных

Рабочее задание: Разработайте Windows Forms приложение в соответствии с вариантом задания, используя различные текстовые компоненты для ввода и отображения информации.

Вариант 1. Регистрационная форма пользователя. Разработать приложение для регистрации нового пользователя со следующими полями:

- Фамилия, имя, отчество (TextBox с валидацией на ввод только букв);

- Дата рождения (MaskedTextBox с маской даты);
- Телефон (MaskedTextBox с маской телефона);
- Email (TextBox с валидацией email);
- Пароль и подтверждение пароля (TextBox с PasswordChar);
- Комментарий (RichTextBox с возможностью форматирования).

Требования: *реализовать валидацию всех полей; проверить совпадение паролей; вычислить возраст по дате рождения; отобразить введенные данные в отформатированном виде*.

Вариант 2. Калькулятор строк. Разработать приложение для работы со строками:

- Поле ввода исходной строки (TextBox);
- Поле для отображения результата (TextBox ReadOnly);
- Панель с кнопками операций: длина строки, количество слов, перевод в верхний/нижний регистр, инвертирование, удаление пробелов;
- Поле для поиска подстроки (TextBox);
- Поле для замены (TextBox);
- История операций (RichTextBox).

Требования: *реализовать все операции со строками; вести историю выполненных операций; добавить счетчик символов и слов в реальном времени.*

Вариант 3. Текстовый редактор. Разработать простой текстовый редактор:

- Главное поле редактирования (RichTextBox);
- Панель форматирования: выбор шрифта, размера, начертания (жирный, курсив, подчеркнутый);
- Выбор цвета текста и фона;
- Выравнивание текста;
- Меню: создать, открыть, сохранить файл;
- Поиск и замена текста;
- Счетчик символов и слов.

Требования: *реализовать сохранение и загрузку файлов в формате RTF и TXT; реализовать функции форматирования; добавить горячие клавиши (Ctrl+S, Ctrl+O и т.д.).*

Вариант 4. Анкета соискателя на работу. Разработать приложение для заполнения анкеты соискателя:

- ФИО (три отдельных TextBox);
- Дата рождения (MaskedTextBox);
- Телефон, дополнительный телефон (MaskedTextBox);
- Email (TextBox с валидацией);
- ИНН, СНИЛС (MaskedTextBox);
- Желаемая должность (TextBox);
- Желаемая зарплата (TextBox с валидацией на числа);
- Опыт работы и образование (RichTextBox).

Требования: *реализовать валидацию всех полей; проверить корректность ИНН (12 цифр) и СНИЛС (11 цифр); сформировать резюме в форматированном виде; сохранить резюме в файл.*

Вариант 5. Конвертер величин. Разработать приложение для конвертации различных величин:

- Поле ввода значения (TextBox с валидацией на числа);
- Выбор типа конвертации: длина, вес, температура, валюта;
- Выбор исходной единицы измерения;
- Выбор целевой единицы измерения;
- Поле результата (TextBox ReadOnly);
- История конвертаций (RichTextBox).

Требования: *реализовать конвертацию минимум 4 типов величин; валидация числового ввода (включая дробные числа); сохранять историю всех конвертаций; возможность очистки истории.*

Вариант 6. Генератор паролей. Разработать приложение для генерации надежных паролей:

- Длина пароля (TextBox с валидацией 8-32);
- Флажки: использовать прописные буквы, строчные буквы, цифры, специальные символы;
- Кнопка генерации;
- Поле вывода пароля (TextBox ReadOnly);
- Индикатор надежности пароля;
- Список сгенерированных паролей (RichTextBox);
- Кнопка копирования в буфер обмена.

Требования: *реализовать генерацию случайного пароля по заданным критериям; оценить надежность пароля (слабый, средний, сильный); сохранять историю сгенерированных паролей; возможность сохранения паролей в файл.*

Вариант 7. Кодировщик/декодировщик текста. Разработать приложение для кодирования и декодирования текста:

- Поле ввода исходного текста (RichTextBox);
- Выбор метода кодирования: Base64, шифр Цезаря, шифр замены;
- Параметры кодирования (например, сдвиг для шифра Цезаря);
- Кнопки: зашифровать, расшифровать;
- Поле вывода результата (RichTextBox);
- Возможность сохранения результата.

Требования: *реализовать минимум 3 метода шифрования; обеспечить возможность расшифровки; обработка ошибок при некорректных данных; статистика: количество символов до и после кодирования.*

Вариант 8. Валидатор данных. Разработать приложение для проверки корректности различных данных:

- Вкладки для разных типов данных: email, телефон, ИНН, СНИЛС, номер карты, URL;
- Поле ввода данных (TextBox или MaskedTextBox);
- Кнопка проверки;
- Индикатор результата (Label с цветом: зеленый - корректно, красный - ошибка);
- Описание ошибки (Label);
- Журнал проверок (RichTextBox);

Требования: *реализовать валидацию минимум 5 типов данных; использовать регулярные выражения; подробные сообщения об ошибках; статистика: количество успешных/неуспешных проверок.*

Вариант 9. Заметки с тегами. Разработать приложение для создания текстовых заметок:

- Заголовок заметки (TextBox);
- Тело заметки (RichTextBox с форматированием);
- Теги (TextBox, несколько тегов через запятую);

- Дата создания (MaskedTextBox, автозаполнение);
- Список сохраненных заметок;
- Поиск по тегам;
- Кнопки: создать, сохранить, удалить, очистить.

Требования: *сохранение заметок в файл; загрузка заметок из файла; поиск заметок по тегам; отображение списка всех заметок с возможностью выбора.*

Вариант 10. Анализатор текста. Разработать приложение для анализа текста:

- Поле ввода/загрузки текста (RichTextBox);
- Панель результатов: количество символов (с пробелами и без); количество слов; количество предложений; количество абзацев; самое длинное/короткое слово; частота встречаемости слов (топ-10)
- Поиск слова (TextBox) с подсветкой в тексте.

Требования: *реализовать все виды анализа; подсветка найденных слов в тексте; экспорт результатов анализа в файл; обработка текстов на русском и английском языках*.

Требования к выполнению задания

1.Создайте новый проект Windows Forms: откройте Visual Studio 2022; создайте новый проект типа "Приложение Windows Forms (.NET Framework)" на C#; задайте осмысленное имя проекту

2.Разработайте пользовательский интерфейс: разместите необходимые текстовые компоненты на форме; используйте осмысленные имена для всех компонентов; примените группировку элементов с помощью GroupBox или Panel; обеспечьте интуитивно понятное расположение элементов

3.Реализуйте функциональность: напишите обработчики всех необходимых событий; реализуйте валидацию пользовательского ввода; обработайте возможные исключения; добавьте информативные сообщения об ошибках

4.Оформление кода: используйте комментарии для пояснения логики; соблюдайте единый стиль кодирования; разделяйте код на логические блоки; используйте осмысленные имена переменных и методов.

5.Тестирование: протестируйте все функции приложения; проверьте работу валидации; проверьте обработку некорректного ввода; убедитесь в корректной работе всех кнопок и элементов управления.

6.Оформите отчет, содержащий**: о**писание разработанного приложения; снимки экрана интерфейса приложения; листинг основного кода программы с комментариями; примеры работы приложения (скриншоты); выводы по работе

Контрольные вопросы

1.Какие основные текстовые компоненты существуют в Windows Forms? Опишите назначение каждого из них.

2.В чем различие между компонентами TextBox и RichTextBox? В каких случаях следует использовать каждый из них?

3.Что такое MaskedTextBox и для чего он используется? Приведите примеры масок ввода для различных типов данных (телефон, дата, ИНН).

4.Какие события возникают при работе с текстовыми компонентами? Опишите разницу между событиями KeyPress, KeyDown и KeyUp.

5.Что такое валидация пользовательского ввода? Какие способы валидации существуют в Windows Forms приложениях?

6.Как реализовать ограничение ввода только цифр в текстовое поле? Приведите пример кода.

7.Для чего используется компонент ErrorProvider? Как его применить для визуализации ошибок валидации?

8.Какие свойства компонента TextBox отвечают за многострочный режим и отображение полос прокрутки?

9.Как программно выделить весь текст в TextBox? Как получить только выделенный текст?

10.Опишите работу с буфером обмена в C#. Как скопировать текст в буфер обмена и вставить из него?

11.Какие методы и свойства RichTextBox позволяют работать с форматированием текста? Как изменить цвет и шрифт выделенного фрагмента?

12.Что такое регулярные выражения и как их использовать для валидации email-адреса в C#?

13.Какое свойство TextBox следует использовать для создания поля ввода пароля? Как реализовать проверку совпадения двух паролей?

14.Как сохранить содержимое RichTextBox в файл формата RTF и как загрузить его обратно?

15.Какие меры безопасности следует предпринять при работе с пользовательским вводом для предотвращения ошибок и уязвимостей приложения?

Практическая работа № 13 «Разработка приложения с несколькими формами»

Цель работы: Освоить принципы разработки многооконных приложений на языке C# в среде Visual Studio 2022, изучить способы создания, отображения и взаимодействия между формами, научиться передавать данные между формами Windows Forms приложения.

Время выполнения: 4 ч

Теоретическая часть

Многооконные приложения Windows Forms. Многооконные приложения — это программные продукты, в которых пользовательский интерфейс разделен на несколько независимых или взаимосвязанных окон (форм). Такой подход позволяет логически разделить функциональность приложения, улучшить пользовательский опыт и структурировать код.

Преимущества использования нескольких форм:

- логическое разделение функциональности приложения;
- возможность повторного использования форм;
- улучшение читаемости и поддерживаемости кода;
- гибкость в организации пользовательского интерфейса;
- возможность работы с несколькими окнами одновременно.

Типы форм в приложении

Главная форма (Main Form) — стартовая форма приложения, которая создается и отображается при запуске программы. Определяется в файле Program.cs.

Дочерние формы (Child Forms) — формы, которые создаются и вызываются из главной или других форм приложения.

Модальные формы (Modal Forms) — формы, которые блокируют работу с другими окнами приложения до их закрытия. Открываются методом ShowDialog().
Немодальные формы (Modeless Forms) — формы, которые позволяют работать с другими окнами приложения одновременно. Открываются методом Show().
Создание новой формы в Visual Studio 2022. Для добавления новой формы в проект необходимо:

1.В обозревателе решений щелкнуть правой кнопкой мыши на имени проекта;
2.Выбрать «Добавить» → «Форма (Windows Forms)»;
3.Ввести имя формы (например, SecondForm.cs);
4.Нажать «Добавить».

В результате будут созданы три файла:

- **SecondForm.cs** — файл с кодом класса формы
- **SecondForm.Designer.cs** — файл с автоматически генерируемым кодом дизайнера
- **SecondForm.resx** — файл ресурсов формы

Способы открытия форм

1. Открытие модальной формы (ShowDialog())

```
// Создание экземпляра формы
SecondForm secondForm = new SecondForm();
// Открытие формы в модальном режиме
secondForm.ShowDialog();
```

При использовании метода ShowDialog():

- работа с родительской формой блокируется;
- метод возвращает значение типа DialogResult;
- форма должна быть закрыта пользователем для продолжения работы;
- используется для диалоговых окон, требующих обязательного ответа.

2. Открытие немодальной формы (Show())

```
// Создание экземпляра формы
SecondForm secondForm = new SecondForm();
// Открытие формы в немодальном режиме
secondForm.Show();
```

При использовании метода Show():

- пользователь может переключаться между окнами;
- можно работать с несколькими формами одновременно;
- форма не блокирует работу с другими окнами.

3. Открытие формы с заменой текущей

SecondForm secondForm = new SecondForm();
secondForm.Show();
this.Hide(); // Скрытие текущей формы

Передача данных между формами

Способ 1. Через конструктор формы

```
// Класс второй формы
public partial class SecondForm : Form
{
    public SecondForm(string message)
    {
        InitializeComponent();
        label1.Text = message;
    }
}

// Вызов из первой формы
SecondForm secondForm = new SecondForm("Привет из первой формы!");
secondForm.ShowDialog();
```

Способ 2. Через публичные свойства

```
// Класс второй формы
public partial class SecondForm : Form
{
    public string UserName { get; set; }

    public SecondForm()
    {
        InitializeComponent();
    }

    private void SecondForm_Load(object sender, EventArgs e)
    {
        label1.Text = $"Добро пожаловать, {UserName}!";
    }
}

// Вызов из первой формы
SecondForm secondForm = new SecondForm();
secondForm.UserName = textBox1.Text;
```

```
secondForm.ShowDialog();
```

Способ 3. Возврат данных через DialogResult

```
// Класс второй формы
public partial class SecondForm : Form
{
    public string Result { get; private set; }

    private void buttonOK_Click(object sender, EventArgs e)
    {
        Result = textBox1.Text;
        this.DialogResult = DialogResult.OK;
        this.Close();
    }
}

// Вызов из первой формы
SecondForm secondForm = new SecondForm();
if (secondForm.ShowDialog() == DialogResult.OK)
{
    label1.Text = secondForm.Result;
}
```

Способ 4. Передача ссылки на родительскую форму

```
// Класс второй формы
public partial class SecondForm : Form
{
    private Form1 parentForm;

    public SecondForm(Form1 parent)
    {
        InitializeComponent();
        parentForm = parent;
    }

    private void button1_Click(object sender, EventArgs e)
    {
        parentForm.UpdateData(textBox1.Text);
    }
}
```

Управление жизненным циклом форм

Закрытие формы:

```
this.Close(); // Закрытие текущей формы
```

Скрытие формы:

```
this.Hide(); // Скрытие формы без выгрузки из памяти
```

Отображение скрытой формы:

```
this.Show(); // Отображение ранее скрытой формы
```

Проверка существования экземпляра формы:

```
if (secondForm == null || secondForm.IsDisposed)
{
    secondForm = new SecondForm();
}
secondForm.Show();
```

Работа с главной формой в Program.cs

Файл Program.cs содержит точку входа в приложение и определяет, какая форма будет запущена первой:

```
static class Program
{
    [STAThread]
    static void Main()
    {
        ApplicationConfiguration.Initialize();
        Application.Run(new Form1()); // Form1 - главная форма
    }
}
```

События форм. Основные события форм, используемые при работе с несколькими окнами:

- **Load** — происходит при загрузке формы;
- **Shown** — происходит при первом отображении формы;
- **FormClosing** — происходит перед закрытием формы (можно отменить закрытие);
- **FormClosed** — происходит после закрытия формы;
- **Activated** — происходит при активации формы;
- **Deactivate** — происходит при деактивации формы.

Пример обработки события FormClosing:

```
private void Form1_FormClosing(object sender, FormClosingEventArgs e)
{
    DialogResult result = MessageBox.Show(
        "Вы уверены, что хотите выйти?",
        "Подтверждение",
        MessageBoxButtons.YesNo,
        MessageBoxIcon.Question);

    if (result == DialogResult.No)
    {
        e.Cancel = true; // Отмена закрытия формы
    }
```

```
}
```

MDI-приложения (Multiple Document Interface). MDI-приложения позволяют работать с несколькими дочерними окнами внутри одного родительского окна.

Создание MDI-родительской формы:

```
public Form1()
{
    InitializeComponent();
    this.IsMdiContainer = true; // Установка формы как MDI-контейнера
}
```

Создание MDI-дочерней формы:

```
SecondForm childForm = new SecondForm();
childForm.MdiParent = this; // Установка родительской формы
childForm.Show();
```

Рекомендации по проектированию многооконных приложений

1.Единая ответственность — каждая форма должна отвечать за определенную функциональность

2.Минимизация связанности — формы должны быть слабо связаны друг с другом

3.Согласованность интерфейса — все формы должны иметь единый стиль оформления

4.Управление ресурсами — своевременное освобождение ресурсов закрытых форм

5.Обработка ошибок — корректная обработка исключений при работе с формами

6.Валидация данных — проверка данных перед передачей между формами

Рабочее задание: Разработайте многооконное приложение в соответствии с вариантом задания. Приложение должно содержать не менее трех форм с реализацией передачи данных между ними.

Вариант 1. Система авторизации и профиля пользователя.

Разработать приложение с формами:

- Форма авторизации (логин, пароль, кнопки "Войти" и "Регистрация")
- Форма регистрации нового пользователя (ФИО, логин, пароль, email)

- Главная форма с профилем пользователя (отображение данных вошедшего пользователя, кнопка "Выход")

Вариант 2. Калькулятор с историей вычислений. Разработать приложение с формами:

- Главная форма с калькулятором (кнопки цифр и операций, дисплей)
- Форма истории вычислений (список всех выполненных операций)
- Форма настроек (количество знаков после запятой, тема оформления)

Вариант 3. Записная книжка. Разработать приложение с формами:

- Главная форма со списком контактов (таблица, кнопки добавления, редактирования, удаления)
- Форма добавления/редактирования контакта (имя, телефон, email, адрес)
- Форма поиска контактов (поле поиска, результаты)

Вариант 4. Тестирование знаний. Разработать приложение с формами:

- Стартовая форма (выбор темы теста, кнопка "Начать")
- Форма прохождения теста (вопрос, варианты ответов, навигация)
- Форма результатов (количество правильных ответов, процент, оценка)

Вариант 5. Конвертер величин. Разработать приложение с формами:

- Главная форма-меню (кнопки выбора типа конвертации: длина, вес, температура, валюта)
- Формы конвертации для каждого типа (поля ввода, выбор единиц измерения, результат)
- Форма "О программе" (информация о разработчике, версия)

Вариант 6. Учет личных финансов. Разработать приложение с формами:

- Главная форма (баланс, список последних операций, кнопки добавления дохода/расхода)

- Форма добавления операции (тип, категория, сумма, дата, описание)
- Форма отчетов (статистика по категориям, графическое представление)

Вариант 7. Планировщик задач. Разработать приложение с формами:

- Главная форма со списком задач (таблица задач, фильтры по статусу)
- Форма создания/редактирования задачи (название, описание, срок, приоритет)
- Форма просмотра задачи (подробная информация, кнопки "Выполнено", "Редактировать", "Удалить")

Вариант 8. Библиотека рецептов. Разработать приложение с формами:

- Главная форма с каталогом рецептов (список, категории, поиск)
- Форма добавления рецепта (название, категория, ингредиенты, способ приготовления)
- Форма просмотра рецепта (полное описание, возможность редактирования)

Вариант 9. Учет студентов группы. Разработать приложение с формами:

- Главная форма со списком студентов (таблица, кнопки управления)
- Форма добавления/редактирования студента (ФИО, номер группы, дата рождения, контакты)
- Форма просмотра успеваемости (список дисциплин и оценок выбранного студента)

Вариант 10. Генератор паролей. Разработать приложение с формами:

- Главная форма генератора (параметры: длина, использование символов, кнопка генерации)
- Форма истории сгенерированных паролей (список с датой и временем создания)
- Форма настроек (параметры генерации по умолчанию, шаблоны паролей)

Требования к выполнению задания

1.Функциональные требования: приложение должно содержать не менее трех форм; должна быть реализована передача данных между формами (минимум в двух направлениях); использовать как модальные, так и немодальные формы (где это уместно); реализовать корректное открытие и закрытие форм; обеспечить валидацию вводимых данных; реализовать обработку основных событий форм

2.Технические требования: использовать осмысленные имена форм и элементов управления; применить единый стиль оформления для всех форм; добавить комментарии к ключевым участкам кода; обеспечить обработку возможных исключений; реализовать как минимум два способа передачи данных между формами

3.Элементы управления (использовать по необходимости):

- Button (кнопки)
- TextBox (текстовые поля)
- Label (метки)
- ListBox или DataGridView (списки/таблицы)
- ComboBox (выпадающие списки)
- RadioButton, CheckBox (переключатели)
- MenuStrip (меню)
- GroupBox или Panel (группировка элементов)

Содержание отчета

Отчет должен содержать:

1.Описание разработанного приложения: назначение приложения; структура приложения (перечень форм и их назначение); описание реализованных способов передачи данных между формами

2.Скриншоты**:** главная форма приложения; все дополнительные формы; примеры работы приложения (передача данных)

3.Листинг программного кода (основные фрагменты): код создания и открытия форм; код передачи данных между формами; обработчики ключевых событий

4.Выводы по работе**:** что было изучено; какие навыки получены; с какими трудностями столкнулись

Контрольные вопросы

1.В чем заключается различие между модальными и немодальными формами? Приведите примеры использования каждого типа.

2.Перечислите и опишите не менее четырех способов передачи данных между формами. Какой способ является предпочтительным с точки зрения архитектуры приложения?

3.Что такое DialogResult и для чего он используется при работе с модальными формами?

4.Опишите жизненный цикл формы. Какие основные события формы вы знаете и в каком порядке они происходят?

5.Как правильно организовать закрытие приложения при работе с несколькими формами? Что произойдет при закрытии главной формы?

6.Что такое MDI-приложение? В каких случаях целесообразно использовать MDI-интерфейс?

7.Как проверить, существует ли экземпляр формы, прежде чем создавать новый? Зачем это может быть необходимо?

8.Объясните разницу между методами Close(), Hide() и Dispose() для форм. Когда следует использовать каждый из них?

9.Какие рекомендации следует соблюдать при проектировании архитектуры многооконных приложений для обеспечения слабой связанности компонентов?

Практическая работа № 14 «Разработка приложения с невизуальными компонентами»

Цель работы: Изучить принципы работы с невизуальными компонентами в среде Visual Studio 2022, освоить технологию разработки приложений на языке C# с использованием компонентов для работы с файловой системой, таймерами, фоновыми процессами и сериализацией данных.

Время выполнения: 2 ч

Теоретическая часть

Понятие невизуальных компонентов. Невизуальные компоненты (non-visual components) — это программные

компоненты, которые не имеют графического представления во время выполнения приложения, но обеспечивают важную функциональность программы. В отличие от визуальных элементов управления (кнопки, текстовые поля, метки), невизуальные компоненты работают "в фоне", предоставляя сервисные функции.

Основные типы невизуальных компонентов в C#

1. Timer (Таймер). Компонент для выполнения операций через определенные интервалы времени.

Основные свойства:

- Interval — интервал срабатывания в миллисекундах
- Enabled — включение/выключение таймера

Основные события:

- Tick — возникает при каждом срабатывании таймера

Пример использования:

```
Timer timer1 = new Timer();
timer1.Interval = 1000; // 1 секунда
timer1.Tick += Timer1_Tick;
timer1.Start();

private void Timer1_Tick(object sender, EventArgs e)
{
   // Код, выполняемый каждую секунду
   label1.Text = DateTime.Now.ToString();
}
```

2. FileSystemWatcher (Наблюдатель файловой системы)

Компонент для отслеживания изменений в файловой системе.

Основные свойства:

- Path — путь к отслеживаемой директории
- Filter — фильтр файлов (например, "*.txt")
- NotifyFilter — типы отслеживаемых изменений
- EnableRaisingEvents — включение отслеживания

Основные события:

- Created — файл создан
- Deleted — файл удален
- Changed — файл изменен
- Renamed — файл переименован

Пример использования:

```
FileSystemWatcher watcher = new FileSystemWatcher();
watcher.Path = @"C:\Temp";
```

```
watcher.Filter = "*.txt";
watcher.NotifyFilter = NotifyFilters.FileName | NotifyFilters.Size;
watcher.Created += Watcher_Created;
watcher.EnableRaisingEvents = true;

private void Watcher_Created(object sender, FileSystemEventArgs e)
{
   MessageBox.Show($"Создан файл: {e.Name}");
}
```

3. BackgroundWorker (Фоновый рабочий процесс). Компонент для выполнения длительных операций в фоновом потоке без блокировки интерфейса.

Основные свойства:

- WorkerReportsProgress — разрешение отчетов о прогрессе
- WorkerSupportsCancellation — поддержка отмены операции

Основные события:

- DoWork — выполнение фоновой операции
- ProgressChanged — изменение прогресса
- RunWorkerCompleted — завершение работы

Пример использования:

```
BackgroundWorker worker = new BackgroundWorker();
worker.WorkerReportsProgress = true;
worker.DoWork += Worker_DoWork;
worker.ProgressChanged += Worker_ProgressChanged;
worker.RunWorkerCompleted += Worker_RunWorkerCompleted;

private void Worker_DoWork(object sender, DoWorkEventArgs e)
{
   BackgroundWorker worker = sender as BackgroundWorker;
   for (int i = 1; i <= 100; i++)
   {
      Thread.Sleep(50);
      worker.ReportProgress(i);
   }
}

private void Worker_ProgressChanged(object sender, ProgressChangedEven-
tArgs e)
{
   progressBar1.Value = e.ProgressPercentage;
}
```

4. SerialPort (Последовательный порт). Компонент для работы с COM-портами.

Основные свойства:

- PortName — имя порта (COM1, COM2 и т.д.)
- BaudRate — скорость передачи данных
- IsOpen — состояние порта

5. ErrorProvider (Провайдер ошибок). Компонент для отображения ошибок валидации рядом с элементами управления.

Основные методы:

- SetError(Control, string) — установка сообщения об ошибке

Пример использования:

```
ErrorProvider errorProvider1 = new ErrorProvider();

private void textBox1_Validating(object sender, CancelEventArgs e)
{
    if (string.IsNullOrEmpty(textBox1.Text))
    {
        errorProvider1.SetError(textBox1, "Поле не может быть пустым");
        e.Cancel = true;
    }
    else
    {
        errorProvider1.SetError(textBox1, "");
    }
}
```

Работа с файлами и сериализацией

Классы для работы с файлами

- File — статические методы для работы с файлами
- Directory — статические методы для работы с каталогами
- FileInfo — информация о файле
- DirectoryInfo — информация о каталоге
- StreamReader — чтение текстовых файлов
- StreamWriter — запись текстовых файлов

Сериализация данных. Сериализация — процесс преобразования объекта в поток байтов для сохранения или передачи.

Типы сериализации в C#:

1. **Binary** — двоичная сериализация

2. **XML** — сериализация в XML-формат
3. **JSON** — сериализация в JSON-формат (наиболее популярный)

Пример JSON-сериализации:

```
using System.Text.Json;

// Класс для сериализации
public class Person
{
    public string Name { get; set; }
    public int Age { get; set; }
}

// Сериализация (сохранение)
Person person = new Person { Name = "Иван", Age = 25 };
string json = JsonSerializer.Serialize(person);
File.WriteAllText("person.json", json);

// Десериализация (загрузка)
string jsonFromFile = File.ReadAllText("person.json");
Person loadedPerson = JsonSerializer.Deserialize<Person>(jsonFromFile);
```

Компоненты OpenFileDialog и SaveFileDialog

OpenFileDialog — диалоговое окно для выбора файла для открытия.

SaveFileDialog — диалоговое окно для выбора места сохранения файла.

Основные свойства:

- Filter — фильтр типов файлов
- FileName — выбранный файл
- InitialDirectory — начальная директория

Пример использования:

```
OpenFileDialog openFileDialog = new OpenFileDialog();
openFileDialog.Filter = "Текстовые файлы (*.txt)|*.txt|Все файлы (*.*)|*.*";
openFileDialog.InitialDirectory = Environment.GetFolderPath(Environment.SpecialFolder.MyDocuments);

if (openFileDialog.ShowDialog() == DialogResult.OK)
{
    string content = File.ReadAllText(openFileDialog.FileName);
    textBox1.Text = content;
}
```

Преимущества использования невизуальных компонентов
1.Инкапсуляция функциональности — сложная логика скрыта внутри компонента
2.Повторное использование — компоненты можно использовать в разных проектах
3.Упрощение кода — событийно-ориентированная модель программирования
4.Разделение ответственности — бизнес-логика отделена от представления
5.Удобство отладки — компоненты можно тестировать независимо

Рабочее задание: Выполните разработку приложения с невизуальными компонентами в соответствии с вариантом задания.

Вариант 1. Система мониторинга папки с автоматическим резервным копированием

Разработать приложение, которое отслеживает изменения в указанной папке и автоматически создает резервные копии измененных файлов. Использовать компоненты FileSystemWatcher и Timer. Приложение должно вести лог событий с сохранением в текстовый файл.

Вариант 2. Приложение для учета рабочего времени

Разработать приложение-секундомер для учета времени, затраченного на выполнение задач. Использовать компонент Timer. Приложение должно сохранять историю задач в JSON-формате и позволять загружать ее при следующем запуске. Реализовать возможность экспорта отчета в текстовый файл.

Вариант 3. Система напоминаний

Разработать приложение для создания напоминаний с возможностью установки даты и времени. Использовать компонент Timer для проверки наступления времени напоминания. Реализовать сохранение списка напоминаний в JSON-файл и загрузку при запуске приложения.

Вариант 4. Конвертер файлов с отслеживанием прогресса

Разработать приложение для массового переименования файлов в указанной папке по заданному шаблону. Использовать BackgroundWorker для выполнения операции без блокировки интерфейса. Отображать прогресс выполнения с помощью

ProgressBar.

Вариант 5. Система логирования событий приложения

Разработать приложение с системой логирования, которое записывает все действия пользователя (нажатия кнопок, ввод текста) в файл с указанием времени. Использовать Timer для периодического сохранения логов. Реализовать просмотр истории логов.

Вариант 6. Автоматический анализатор размера файлов

Разработать приложение, которое сканирует указанную папку и создает отчет о размере файлов различных типов. Использовать BackgroundWorker для сканирования. Результаты сохранять в JSON-формат. Реализовать визуализацию результатов в виде списка.

Вариант 7. Система архивации старых файлов

Разработать приложение, которое автоматически находит файлы старше указанного количества дней и перемещает их в архивную папку. Использовать BackgroundWorker для поиска файлов. Вести журнал перемещенных файлов в текстовом формате.

Вариант 8. Приложение для контроля версий документов

Разработать приложение, которое отслеживает изменения в текстовых документах и сохраняет предыдущие версии. Использовать FileSystemWatcher для отслеживания изменений. Реализовать просмотр списка версий и возможность восстановления любой версии.

Вариант 9. Планировщик задач с автозапуском

Разработать приложение-планировщик, которое позволяет создавать задачи с указанием времени выполнения. Использовать Timer для проверки времени. Задачи должны сохраняться в JSON-файл. Реализовать уведомления о наступлении времени выполнения задачи.

Вариант 10. Система мониторинга дискового пространства

Разработать приложение, которое отслеживает свободное место на дисках компьютера и выдает предупреждения при достижении критического уровня. Использовать Timer для периодической проверки. Вести лог изменений в файле. Реализовать

отображение статистики использования дисков.

Требования к выполнению задания

1. Анализ задачи: определите основные функции приложения; выберите необходимые невизуальные компоненты; спроектируйте структуру данных для хранения информации; определите формат хранения данных (JSON, TXT, XML)

2. Разработка интерфейса: создайте форму приложения с необходимыми визуальными элементами управления; добавьте на форму невизуальные компоненты; настройте свойства компонентов через окно Properties; обеспечьте удобство и интуитивность интерфейса

3. Реализация функциональности. Обязательные элементы: использование минимум **двух** невизуальных компонентов из изученных; реализация сохранения данных в файл; реализация загрузки данных из файла; обработка исключительных ситуаций (try-catch блоки); валидация вводимых данных с использованием ErrorProvider (при необходимости); использование компонентов OpenFileDialog и/или SaveFileDialog

4. Требования к коду: код должен быть структурирован и читаем; использовать осмысленные имена переменных и методов; добавить комментарии к ключевым участкам кода; следовать принципам ООП; обрабатывать возможные ошибки (файл не найден, нет прав доступа и т.д.).

5. Тестирование. Проверьте работу приложения: корректность сохранения и загрузки данных; работу невизуальных компонентов; обработку ошибочных ситуаций; работу при отсутствии необходимых файлов; корректность работы всех функций

6. Оформление отчета. Отчет должен содержать: описание используемых компонентов — какие невизуальные компоненты использованы и для чего; структура данных — описание классов и структур для хранения информации; скриншоты интерфейса приложения в режиме разработки (с отображением невизуальных компонентов в области компонентов); скриншоты работающего приложения — демонстрация основных функций; листинг кода — ключевые фрагменты кода с комментариями; результаты тестирования — примеры работы с

различными данными; выводы — что было изучено, какие навыки получены

Контрольные вопросы

1.Что такое невизуальные компоненты? Чем они отличаются от визуальных элементов управления?

2.Для каких целей используется компонент Timer? Какие основные свойства и события он имеет?

3.Опишите назначение компонента FileSystemWatcher. Какие события файловой системы он может отслеживать?

4.Зачем нужен компонент BackgroundWorker? Какую проблему он решает при разработке приложений с графическим интерфейсом?

5.Что такое сериализация данных? Какие форматы сериализации вы знаете? В чем преимущества JSON-формата?

6.Как обеспечить сохранение настроек приложения между запусками? Какие технологии для этого можно использовать?

7.В чем разница между компонентами OpenFileDialog и SaveFileDialog? Какие основные свойства они имеют?

8.Что произойдет, если попытаться прочитать несуществующий файл? Как обработать такую ситуацию?

9.Какие преимущества дает использование невизуальных компонентов при разработке приложений?

10.Приведите примеры реальных приложений, в которых активно используются невизуальные компоненты.

Практическая работа № 15 «Разработка игрового приложения»

Цель работы: Освоить принципы разработки игровых приложений на платформе .NET с использованием языка C# и среды Visual Studio 2022, изучить основные компоненты игровой логики, реализовать игровое приложение с графическим интерфейсом.

Время выполнения: 4 ч

Теоретическая часть

Основы разработки игровых приложений

Игровое приложение — это программный продукт,

предназначенный для развлечения пользователей, решения головоломок или обучения через интерактивное взаимодействие. Разработка игр требует знания не только основ программирования, но и понимания игровой механики, работы с графикой, обработки пользовательского ввода и управления игровым циклом.

Основные компоненты игрового приложения

1. Игровой цикл (Game Loop). Игровой цикл — это основа любой игры, непрерывно выполняющийся процесс, который включает:

- Обработку ввода пользователя
- Обновление состояния игры (логика)
- Отрисовку графики

```
while (gameRunning)
{
   ProcessInput();
   UpdateGameState();
   Render();
}
```

2. Игровые объекты. Игровые объекты — это сущности игрового мира (персонажи, препятствия, бонусы). Каждый объект обладает свойствами (координаты, размер, скорость) и методами (движение, столкновение).

3. Обработка событий. Игры активно работают с событиями:

- События клавиатуры (нажатие клавиш)
- События мыши (клики, перемещение)
- События таймера (обновление кадров)

4. Система коллизий. Система обнаружения столкновений определяет взаимодействие объектов в игре. Основные методы:

- Проверка пересечения прямоугольников (Bounding Box)
- Проверка расстояния между объектами
- Попиксельное определение столкновений

5. Управление состояниями игры. Игра может находиться в различных состояниях: Главное меню; Игровой процесс; Пауза; Конец игры (победа/поражение)

Использование Windows Forms для разработки игр

Windows Forms предоставляет набор инструментов для создания простых 2D-игр:

PictureBox — для отображения графики и игровых объектов
Timer — для создания игрового цикла
Panel — для игрового поля
Label — для отображения счета, жизней и другой информации
Графика в Windows Forms
Использование GDI+ для рисования**:**

```
private void GamePanel_Paint(object sender, PaintEventArgs e)
{
    Graphics g = e.Graphics;
    g.FillRectangle(Brushes.Blue, playerX, playerY, 50, 50);
}
```

Двойная буферизация для устранения мерцания:

```
this.DoubleBuffered = true;
```

Обработка пользовательского ввода. Обработка клавиатуры:

```
private void Form1_KeyDown(object sender, KeyEventArgs e)
{
    if (e.KeyCode == Keys.Left)
        playerX -= speed;
    else if (e.KeyCode == Keys.Right)
        playerX += speed;
}
```

Использование коллекции нажатых клавиш для плавного движения:

```
HashSet<Keys> pressedKeys = new HashSet<Keys>();

private void Form1_KeyDown(object sender, KeyEventArgs e)
{
    pressedKeys.Add(e.KeyCode);
}

private void Form1_KeyUp(object sender, KeyEventArgs e)
{
    pressedKeys.Remove(e.KeyCode);
}
```

Игровая физика. Базовые принципы физики в играх:

- **Гравитация** — постоянное ускорение вниз
- **Скорость** — изменение позиции объекта
- **Ускорение** — изменение скорости

velocityY += gravity; // применение гравитации
playerY += velocityY; // обновление позиции

Архитектура игрового приложения. Рекомендуемая структура классов:

- **GameObject** — базовый класс для всех игровых объектов
- **Player** — класс игрока
- **Enemy** — класс противника
- **GameEngine** — класс управления игрой
- **CollisionDetector** — класс проверки столкновений

Примеры популярных жанров для начинающих

1.Аркады — простая механика, быстрый геймплей (Арканоид, Snake)

2.Платформеры — движение по уровню с препятствиями

3.Головоломки — логические задачи (Тетрис, 2048)

4.Шутеры — стрельба по целям или врагам

Оптимизация игровых приложений

- Использование спрайтов вместо постоянной перерисовки
- Ограничение частоты обновления (60 FPS оптимально)
- Удаление неиспользуемых объектов из памяти
- Использование пула объектов для часто создаваемых элементов

Рабочее задание: Выполните разработку игрового приложения в соответствии с вариантом задания.

Вариант 1. Игра "Арканоид". Разработать классическую аркадную игру, где игрок управляет платформой, отбивая мяч для разбивания блоков.

Требования: Платформа управляется мышью или клавиатурой; Мяч отскакивает от стен, платформы и блоков; Блоки разрушаются при касании мяча; Система подсчета очков; Несколько уровней сложности; Обработка проигрыша (мяч упал) и победы (все блоки разбиты)

Вариант 2. Игра "Змейка" (Snake). Разработать игру, где игрок управляет змейкой, собирая еду и избегая столкновений.

Требования: Управление направлением движения змейки (клавиши-стрелки); Змейка растет при поедании еды; Генерация еды в случайных местах; Обнаружение столкновений со

стенами и собственным телом; Система подсчета очков; Увеличение скорости с ростом змейки; Отображение рекорда

Вариант 3. Игра "Flappy Bird". Разработать игру, где игрок управляет птицей, пролетающей между препятствиями.

Требования: Управление взмахом (пробел или клик мыши); Физика гравитации и взмаха; Генерация труб с разными промежутками; Обнаружение столкновений; Система подсчета очков за пройденные трубы; Плавная анимация движения; Экран game over с возможностью перезапуска

Вариант 4. Игра "Пинг-понг". Разработать игру для двух игроков в настольный теннис.

Требования: Две ракетки (управление клавишами W/S и стрелками); Мяч с физикой отскока; Изменение угла отскока в зависимости от места удара; Счет игры до определенного количества очков; Увеличение скорости мяча при долгом розыгрыше; Режим игры с компьютером (AI); Звуковые эффекты при ударе

Вариант 5. Игра "Космический шутер". Разработать вертикальный шутер, где игрок управляет космическим кораблем.

Требования: Управление кораблем (клавиши-стрелки); Стрельба (пробел); Генерация врагов с разной скоростью; Система здоровья игрока; Подсчет уничтоженных врагов; Бонусы (улучшение оружия, здоровье); Несколько типов врагов; Система уровней

Вариант 6. Игра "2048". Разработать головоломку с объединением чисел на сетке 4×4.

Требования: Сетка 4×4 для игровых плиток; Управление направлением сдвига (стрелки); Объединение одинаковых чисел; Генерация новых плиток (2 или 4); Проверка возможности хода; Подсчет очков; Отображение лучшего результата; Анимация движения плиток

Вариант 7. Игра "Виселица" (Hangman). Разработать словесную игру по угадыванию слова.

Требования: База слов различной категории; Отображение скрытого слова (прочерки); Ввод букв (клавиатура или экранная); Отображение использованных букв; Графическое

представление виселицы (поэтапное); Счетчик оставшихся попыток; Выбор категории слов; Система подсказок

Вариант 8. Игра "Сапёр" (Minesweeper). Разработать логическую игру по поиску мин на поле.

Требования: Сетка игрового поля (настраиваемый размер); Генерация мин в случайных позициях; Открытие клеток левым кликом; Установка флажков правым кликом; Подсчет мин вокруг клетки; Автоматическое открытие пустых областей; Таймер игры; Выбор уровня сложности

Вариант 9. Игра "Платформер". Разработать простой платформер с персонажем и препятствиями.

Требования: Управление персонажем (влево, вправо, прыжок); Физика прыжка и гравитации; Платформы разной высоты; Препятствия (враги, ямы); Система жизней; Сбор монет или бонусов; Несколько уровней; Анимация персонажа

Вариант 10. Игра "Тетрис". Разработать классическую головоломку с падающими фигурами.

Требования: Семь типов тетромино (фигур); Управление (влево, вправо, поворот, ускорение падения); Заполнение и удаление линий; Система подсчета очков; Увеличение скорости с прогрессом; Отображение следующей фигуры; Проверка окончания игры; Сохранение рекорда.

Требования к выполнению задания

1. Проектирование игры: Перед началом разработки подготовьте: Описание игровой механики; Перечень игровых объектов и их свойств; Диаграмму классов; Описание игрового цикла; Перечень обрабатываемых событий

2.Разработка приложения: Создайте проект Windows Forms Application в Visual Studio 2022, который должен содержать:

– *Обязательные элементы:* Главное меню с кнопкой "Начать игру"; Игровое поле; Отображение счета/очков; Обработка пользовательского ввода; Игровой цикл (Timer); Система обнаружения столкновений; Условия победы/поражения; Возможность перезапуска игры.

– *Классы (минимум 3):* Класс игрового объекта; Класс управления игрой; Вспомогательные классы (по необходимости)

– *Функциональные требования:* Корректная обработка всех игровых событий; Плавность анимации; Адекватная игровая физика (если применимо); Корректная работа системы подсчета очков; Обработка граничных случаев

– *Нефункциональные требования:* Производительность: стабильные 30-60 FPS; Удобство управления; Понятный интерфейс; Отсутствие критических ошибок; Читаемость кода (комментарии, именование)

3. Оформление кода: Используйте осмысленные имена переменных и методов; Добавьте комментарии к ключевым участкам кода; Соблюдайте принципы ООП; Обрабатывайте возможные исключения; Используйте константы для настроек игры

4. Оформление отчета. Отчет должен содержать:

– Описание игры: Название и жанр; Правила игры; Управление

– Проектная часть: Диаграмма классов; Описание основных классов и их методов; Описание игрового цикла; Алгоритм обнаружения столкновений;

– Программная реализация: истинг основных классов с комментариями; Снимки экрана игры (главное меню, игровой процесс, окончание игры); Описание возникших трудностей и их решений;

–Тестирование: Описание проведенных тестов: Выявленные проблемы и способы их устранения.

– Выводы: что было реализовано; какие навыки получены; возможные улучшения игры

Контрольные вопросы

1.Что такое игровой цикл и из каких основных этапов он состоит? Почему он является критически важным элементом любой игры?

2.Какие компоненты Windows Forms используются для разработки игровых приложений? Опишите назначение Timer и PictureBox в контексте игровой разработки.

3.Что такое двойная буферизация и для чего она применяется в графических приложениях? Как включить двойную буферизацию в Windows Forms?

4.Опишите основные методы обнаружения столкновений в 2D-играх. В каких случаях применяется метод Bounding Box?

5.Какие события клавиатуры и мыши используются в играх? В чем разница между событиями KeyDown и KeyPress?

6.Что такое FPS (frames per second) и как его можно контролировать с помощью Timer в Windows Forms?

7.Объясните принципы игровой физики: что такое скорость, ускорение, гравитация? Приведите пример их применения в коде.

8.Какие принципы объектно-ориентированного программирования применяются при разработке игр? Приведите примеры использования наследования и инкапсуляции.

9.Как реализовать систему состояний игры (меню, игра, пауза, game over)? Какие паттерны проектирования можно использовать?

10.Какие методы оптимизации производительности применяются в игровых приложениях для обеспечения плавной работы?

Практическая работа № 16 «Разработка приложения с функционально связанными модулями»

Цель работы: Освоить принципы модульной архитектуры программного обеспечения, научиться разрабатывать приложения на языке C# с функционально связанными модулями, реализовать механизмы взаимодействия между модулями с использованием различных способов связывания.

Время выполнения: 4 ч

Теоретическая часть

Модульная архитектура программного обеспечения. Модуль — это функционально законченная часть программы, которая представляет собой совокупность программных средств (процедур, функций, классов), способных выполнять определенную задачу и взаимодействовать с другими модулями через четко

определенный интерфейс. Модульное программирование — это организация программы как совокупности небольших независимых блоков (модулей), структура и поведение которых подчиняются определенным правилам.

Преимущества модульной архитектуры:

1.Упрощение разработки — возможность параллельной работы нескольких разработчиков над разными модулями;

2.Повторное использование кода — модули могут применяться в разных проектах;

3.Упрощение тестирования — каждый модуль можно тестировать независимо;

4.Упрощение сопровождения — изменения в одном модуле минимально влияют на другие;

5.Масштабируемость — возможность добавления новых модулей без изменения существующих;

6.Читаемость кода — структурированная организация упрощает понимание программы.

Принципы модульной разработки

1. Принцип единственной ответственности (SRP). Каждый модуль должен отвечать только за одну функциональную область и иметь только одну причину для изменения.

2. Принцип слабой связанности (Low Coupling). Модули должны быть минимально зависимы друг от друга. Изменение одного модуля не должно требовать изменений в других модулях.

3. Принцип сильной связности (High Cohesion). Элементы внутри модуля должны быть логически связаны и работать для достижения единой цели.

4. Принцип инкапсуляции. Внутренняя реализация модуля должна быть скрыта от других модулей. Взаимодействие происходит только через публичный интерфейс.

5. Принцип открытости/закрытости. Модули должны быть открыты для расширения, но закрыты для изменения.

Типы связей между модулями

1. По данным (Data Coupling) — модули обмениваются только простыми данными через параметры методов. Наиболее предпочтительный тип связи.

2. По образцу (Stamp Coupling) — модули обмениваются составными структурами данных, при этом используется только часть данных.

3. По управлению (Control Coupling) — один модуль управляет логикой работы другого, передавая управляющие флаги.

4. По внешним ссылкам (External Coupling) — модули используют общие глобальные данные.

5. По общей области (Common Coupling) — модули используют одни и те же глобальные переменные.

6. По содержимому (Content Coupling) — один модуль напрямую обращается к внутренним данным другого модуля. Наихудший тип связи.

Способы организации модулей в C#

1. Классы Базовый способ модульной организации в объектно-ориентированном программировании.

```
public class DataProcessor
{
   public string ProcessData(string input)
   {
      // обработка данных
      return result;
   }
}
```

2. Пространства имен (Namespaces) Логическая группировка связанных классов.

```
namespace MyApp.DataLayer
{
   public class DatabaseManager { }
}

namespace MyApp.BusinessLogic
{
   public class Calculator { }
}
```

3. Библиотеки классов (Class Libraries) Отдельные сборки (.dll), содержащие группы связанных классов.

4. Интерфейсы Определяют контракт взаимодействия между модулями без привязки к конкретной реализации.

```
public interface IDataProvider
{
```

```
    List<string> GetData();
}

public class DatabaseProvider : IDataProvider
{
    public List<string> GetData()
    {
        // реализация
    }
}
```

Паттерны взаимодействия модулей

1. Dependency Injection (Внедрение зависимостей) Передача зависимостей модулю извне, а не создание их внутри модуля.

```
public class ReportGenerator
{
    private readonly IDataProvider _dataProvider;

    public ReportGenerator(IDataProvider dataProvider)
    {
        _dataProvider = dataProvider;
    }
}
```

2. Factory (Фабрика) Создание объектов без указания точного класса создаваемого объекта.

3. Observer (Наблюдатель) Определение зависимости типа «один ко многим» между объектами таким образом, что при изменении состояния одного объекта все зависящие от него оповещаются об этом.

4. Facade (Фасад) Предоставление унифицированного интерфейса к группе интерфейсов подсистемы.

Структура многомодульного приложения в C#

Типичная структура проекта:

```
Solution (Решение)
├── MyApp.UI (проект презентационного слоя)
│   └── Форма взаимодействия с пользователем
│
├── MyApp.BusinessLogic (проект бизнес-логики)
│   └── Классы обработки данных, вычислений
│
```

```
├── MyApp.DataAccess (проект доступа к данным)
│   └── Классы работы с БД, файлами
│
└── MyApp.Common (проект общих компонентов)
    └── Общие классы, интерфейсы, утилиты
```

Механизмы передачи данных между модулями

1. Параметры методов

```
public decimal CalculateTotal(List<Product> products)
```

2. Возвращаемые значения

```
public Customer GetCustomer(int id)
```

3. Свойства объектов

```
public class Order
{
   public int OrderId { get; set; }
   public List<OrderItem> Items { get; set; }
}
```

4. События (Events)

```
public event EventHandler DataChanged;
```

5. Делегаты (Delegates)

```
public delegate void ProcessDataDelegate(string data);
```

Создание библиотеки классов в Visual Studio 2022

Шаг 1. Создание нового проекта типа "Библиотека классов" (Class Library)

- File → New → Project
- Выбрать "Class Library (.NET Framework)" или "Class Library"
- Указать имя и расположение

Шаг 2. Добавление классов в библиотеку

Шаг 3. Компиляция библиотеки (Build)

- Результат: файл .dll в папке bin\Debug или bin\Release

Шаг 4. Подключение библиотеки к основному проекту

- В основном проекте: Add → Reference
- Выбрать вкладку "Projects" и отметить нужную библиотеку

Шаг 5. Использование классов из библиотеки

```
using MyLibrary;
```

Рабочее задание: Разработайте приложение Windows Forms с функционально связанными модулями в соответствии с вариантом задания. Приложение должно состоять минимум из трех модулей с четко определенными зонами ответственности.

Вариант 1. Калькулятор с историей вычислений. Разработать калькулятор с возможностью сохранения истории вычислений и экспорта результатов.

Модули:

- **CalculationModule** — модуль математических операций (сложение, вычитание, умножение, деление, степень, корень)
- **HistoryModule** — модуль управления историей вычислений (добавление, удаление, поиск)
- **ExportModule** — модуль экспорта данных (сохранение истории в текстовый файл, CSV)
- **UI (главная форма)** — пользовательский интерфейс

Функциональные связи:

- UI вызывает методы CalculationModule для выполнения вычислений
- CalculationModule передает результаты в HistoryModule
- UI запрашивает у HistoryModule данные для отображения
- ExportModule получает данные от HistoryModule для сохранения

Вариант 2. Система учета личных финансов. Разработать приложение для учета доходов и расходов с категоризацией и формированием отчетов.

Модули:

- **TransactionModule** — модуль управления транзакциями (добавление доходов/расходов, редактирование, удаление)
- **CategoryModule** — модуль управления категориями (создание, редактирование категорий доходов и расходов)
- **ReportModule** — модуль формирования отчетов (расчет баланса, анализ по категориям, статистика за период)

- **UI** — пользовательский интерфейс

Вариант 3. Конвертер единиц измерения. Разработать универсальный конвертер величин с возможностью добавления пользовательских единиц.

Модули:

- **ConversionModule** — модуль конвертации (длина, вес, температура, объем, скорость)
- **UnitsModule** — модуль управления единицами измерения (добавление, редактирование, удаление единиц)
- **ValidationModule** — модуль валидации вводимых данных
- **UI** — пользовательский интерфейс

Вариант 4. Менеджер паролей. Разработать приложение для хранения и управления паролями с функцией генерации безопасных паролей.

Модули:

- **StorageModule** — модуль хранения данных (добавление, удаление, поиск записей)
- **EncryptionModule** — модуль шифрования/дешифрования паролей (использовать простое шифрование, например, XOR или Base64)
- **GeneratorModule** — модуль генерации паролей (с настройкой длины и набора символов)
- **UI** — пользовательский интерфейс

Вариант 5. Планировщик задач. Разработать приложение для управления списком задач с приоритетами и сроками выполнения.

Модули:

- **TaskModule** — модуль управления задачами (добавление, редактирование, удаление, изменение статуса)
- **PriorityModule** — модуль управления приоритетами (определение приоритета, сортировка задач)
- **NotificationModule** — модуль уведомлений (проверка просроченных задач, напоминания)
- **UI** — пользовательский интерфейс

Вариант 6. Записная книжка контактов. Разработать приложение для управления контактами с возможностью поиска и группировки.

Модули:

- **ContactModule** — модуль управления контактами (добавление, редактирование, удаление)
- **SearchModule** — модуль поиска и фильтрации (по имени, телефону, email, группе)
- **GroupModule** — модуль управления группами контактов
- **ExportModule** — модуль экспорта (сохранение в файл)
- **UI** — пользовательский интерфейс

Вариант 7. Учет книг домашней библиотеки. Разработать приложение для каталогизации личной библиотеки с возможностью поиска и статистики.

Модули:

- **BookModule** — модуль управления книгами (добавление, редактирование, удаление)
- **SearchModule** — модуль поиска (по автору, названию, жанру, году издания)
- **StatisticsModule** — модуль статистики (количество книг, распределение по жанрам, по авторам)
- **UI** — пользовательский интерфейс

Вариант 8. Калькулятор оценок студента. Разработать приложение для расчета среднего балла и прогнозирования итоговой оценки.

Модули:

- **GradeModule** — модуль управления оценками (добавление оценок по дисциплинам)
- **CalculationModule** — модуль вычислений (расчет среднего балла, GPA, взвешенного среднего)
- **PredictionModule** — модуль прогнозирования (какую оценку нужно получить для достижения целевого балла)
- **ReportModule** — модуль отчетов (формирование сводки по успеваемости)
- **UI** — пользовательский интерфейс

Вариант 9. Система учета расхода топлива. Разработать приложение для учета заправок автомобиля и расчета расхода топлива.

Moduli:

- **RefuelModule** — модуль учета заправок (дата, литры, стоимость, пробег)
- **CalculationModule** — модуль расчетов (расход на 100 км, средняя стоимость, общие затраты)
- **StatisticsModule** — модуль статистики (динамика расхода, затрат по месяцам)
- **UI** — пользовательский интерфейс

Вариант 10. Органайзер заметок. Разработать приложение для создания и организации текстовых заметок с тегами.

Модули:

- **NoteModule** — модуль управления заметками (создание, редактирование, удаление)
- **TagModule** — модуль управления тегами (добавление тегов к заметкам, фильтрация по тегам)
- **SearchModule** — модуль поиска (по содержимому, по дате, по тегам)
- **StorageModule** — модуль сохранения/загрузки заметок в файлы
- **UI** — пользовательский интерфейс

Требования к выполнению задания

1. *Архитектурные требования*: приложение должно состоять минимум из **трех отдельных модулей** (классов), каждый из которых отвечает за отдельную функциональную область; модули должны быть организованы в виде **отдельных классов** в разных файлах; каждый модуль должен иметь **четко определенный публичный интерфейс** (набор публичных методов); внутренняя реализация модуля должна быть **инкапсулирована** (использовать модификаторы доступа private, protected); связь между модулями должна быть **слабой** (модули взаимодействуют через методы, а не напрямую обращаются к данным друг друга)

2. *Технические требования*: платформа: **Windows Forms**, Visual Studio 2022; Язык программирования: **C#;**

каждый модуль должен быть реализован в **отдельном файле класса** (.cs); использовать **пространства имен** для логической группировки классов; применять **стандарты именования** C# (PascalCase для классов и методов, camelCase для параметров); код должен содержать **комментарии**, описывающие назначение модулей и ключевых методов

3. *Функциональные требования*: реализовать **все модули**, указанные в варианте задания; каждый модуль должен содержать минимум **3-4 публичных метода;** главная форма (UI) должна **взаимодействовать со всеми модулями;** приложение должно корректно обрабатывать **некорректный ввод данных** (валидация); реализовать **передачу данных между модулями** минимум двумя способами (параметры методов, возвращаемые значения, свойства объектов)

4. *Требования к пользовательскому интерфейсу*: интерфейс должен быть **интуитивно понятным**; все основные функции должны быть **доступны через элементы управления** формы; предусмотреть **информирование пользователя** о результатах операций (MessageBox, Label); интерфейс должен быть **логически организован** (группировка элементов управления)

5. Пример структуры проекта

MyApplication (проект Windows Forms)
- Forms
  - MainForm.cs (главная форма)
- Modules
  - Module1.cs (первый функциональный модуль)
  - Module2.cs (второй функциональный модуль)
  - Module3.cs (третий функциональный модуль)
- Models (если требуется)
  - DataModel.cs (классы данных)

6. Документирование

В коде:

```
/// <summary>
/// Модуль выполнения математических вычислений
/// </summary>
```

```
public class CalculationModule
{
    /// <summary>
    /// Выполняет сложение двух чисел
    /// </summary>
    /// <param name="a">Первое число</param>
    /// <param name="b">Второе число</param>
    /// <returns>Результат сложения</returns>
    public double Add(double a, double b)
    {
        return a + b;
    }
}
```

Пример для варианта 1: Калькулятор с историей

Структура проекта:

1. CalculationModule.cs

```
using System;

namespace CalculatorApp.Modules
{
    /// <summary>
    /// Модуль выполнения математических операций
    /// </summary>
    public class CalculationModule
    {
        /// <summary>
        /// Выполняет сложение
        /// </summary>
        public double Add(double a, double b)
        {
            return a + b;
        }

        /// <summary>
        /// Выполняет вычитание
        /// </summary>
        public double Subtract(double a, double b)
```

```
        {
            return a - b;
        }

        /// <summary>
        /// Выполняет умножение
        /// </summary>
        public double Multiply(double a, double b)
        {
            return a * b;
        }

        /// <summary>
        /// Выполняет деление
        /// </summary>
        public double Divide(double a, double b)
        {
            if (b == 0)
                throw new DivideByZeroException("Деление на ноль
невозможно");
            return a / b;
        }

        /// <summary>
        /// Возводит число в степень
        /// </summary>
        public double Power(double baseNum, double exponent)
        {
            return Math.Pow(baseNum, exponent);
        }
    }
}
```

2. HistoryModule.cs

```
using System;
using System.Collections.Generic;

namespace CalculatorApp.Modules
```

```
{
    /// <summary>
    /// Модуль управления историей вычислений
    /// </summary>
    public class HistoryModule
    {
        private List<string> history;

        public HistoryModule()
        {
            history = new List<string>();
        }

        /// <summary>
        /// Добавляет запись в историю
        /// </summary>
        public void AddRecord(string record)
        {
            string timestamp = DateTime.Now.ToString("dd.MM.yyyy
HH:mm:ss");
            history.Add($"[{timestamp}] {record}");
        }

        /// <summary>
        /// Получает всю историю
        /// </summary>
        public List<string> GetHistory()
        {
            return new List<string>(history);
        }

        /// <summary>
        /// Очищает историю
        /// </summary>
        public void ClearHistory()
        {
            history.Clear();
```

```
        }

        /// <summary>
        /// Получает количество записей в истории
        /// </summary>
        public int GetRecordCount()
        {
            return history.Count;
        }
    }
}
```

3. ExportModule.cs

```
using System.Collections.Generic;
using System.IO;

namespace CalculatorApp.Modules
{
    /// <summary>
    /// Модуль экспорта данных
    /// </summary>
    public class ExportModule
    {
        /// <summary>
        /// Сохраняет историю в текстовый файл
        /// </summary>
        public bool SaveToTextFile(List<string> data, string filePath)
        {
            try
            {
                File.WriteAllLines(filePath, data);
                return true;
            }
            catch
            {
                return false;
            }
        }
```

```
        /// <summary>
        /// Сохраняет историю в CSV файл
        /// </summary>
        public bool SaveToCsv(List<string> data, string filePath)
        {
            try
            {
                List<string> csvLines = new List<string>();
                csvLines.Add("Дата и время,Операция");

                foreach (string record in data)
                {
                    csvLines.Add(record.Replace("[", "").Replace("] ",
","));
                }

                File.WriteAllLines(filePath, csvLines);
                return true;
            }
            catch
            {
                return false;
            }
        }
    }
}
```

4. MainForm.cs (фрагмент)

```
using System;
using System.Windows.Forms;
using CalculatorApp.Modules;

namespace CalculatorApp
{
    public partial class MainForm : Form
    {
        // Создание экземпляров модулей
```

```
private CalculationModule calcModule;
private HistoryModule historyModule;
private ExportModule exportModule;

public MainForm()
{
    InitializeComponent();

    // Инициализация модулей
    calcModule = new CalculationModule();
    historyModule = new HistoryModule();
    exportModule = new ExportModule();
}

private void btnAdd_Click(object sender, EventArgs e)
{
    try
    {
        double a = double.Parse(txtNumber1.Text);
        double b = double.Parse(txtNumber2.Text);

        // Вызов метода модуля вычислений
        double result = calcModule.Add(a, b);

        txtResult.Text = result.ToString();

        // Добавление записи в историю
        string record = $"{a} + {b} = {result}";
        historyModule.AddRecord(record);

        UpdateHistoryDisplay();
    }
    catch (Exception ex)
    {
        MessageBox.Show("Ошибка: " + ex.Message);
    }
}
```

```
        private void UpdateHistoryDisplay()
        {
            listBoxHistory.Items.Clear();
            foreach (string record in historyModule.GetHistory())
            {
                listBoxHistory.Items.Add(record);
            }
        }

        private void btnExport_Click(object sender, EventArgs e)
        {
            SaveFileDialog sfd = new SaveFileDialog();
            sfd.Filter = "Text Files|*.txt|CSV Files|*.csv";

            if (sfd.ShowDialog() == DialogResult.OK)
            {
                var history = historyModule.GetHistory();
                bool success = false;

                if (sfd.FileName.EndsWith(".csv"))
                    success = exportModule.SaveToCsv(history, sfd.File-
Name);
                else
                    success = exportModule.SaveToTextFile(history,
sfd.FileName);

                if (success)
                    MessageBox.Show("История успешно сохранена");
                else
                    MessageBox.Show("Ошибка сохранения");
            }
        }
    }
}
```

Диаграмма взаимодействия модулей:
MainForm (UI)

↓ вызывает методы
├→ CalculationModule (выполнение вычислений)
│ ↓ возвращает результат
├→ HistoryModule (сохранение результата)
│ ↓ возвращает список истории
└→ ExportModule (получает данные от HistoryModule)
↓ сохраняет в файл

Этап 1. Проектирование модульной структуры (30 минут)

1.Внимательно изучите описание вашего варианта
2.Определите функциональные области приложения
3.Распределите функции по модулям согласно принципу единственной ответственности
4.Для каждого модуля определите: назначение модуля; публичные методы (интерфейс модуля); внутренние данные (поля класса); связи с другими модулями
5.Нарисуйте схему взаимодействия модулей

Этап 2. Создание проекта и структуры файлов (15 минут)

1.Создайте новый проект Windows Forms Application
2.Создайте папку "Modules" в проекте (правая кнопка на проекте → Add → New Folder)
3.Для каждого модуля создайте отдельный файл класса: правая кнопка на папке Modules → Add → Class; укажите имя файла (например, CalculationModule.cs)
4.Добавьте пространства имен в каждый файл

Этап 3. Реализация модулей (1.5 часа)

1.Начните с модулей, которые не зависят от других (например, модуль вычислений)
2.Для каждого модуля: Объявите поля класса (private); Реализуйте конструктор (если требуется инициализация); Реализуйте публичные методы; Добавьте комментарии XML-документации (///); Добавьте обработку исключений там, где это необходимо
3.Соблюдайте принципы инкапсуляции (используйте private для внутренних данных)

Этап 4. Разработка пользовательского интерфейса (45 минут)

1.Спроектируйте главную форму приложения

2.Разместите необходимые элементы управления: Поля ввода (TextBox); Кнопки (Button); Метки (Label); Списки (ListBox, ComboBox) и другие по необходимости
3.Задайте понятные имена элементам управления (например, txtNumber1, btnCalculate)
4.Настройте свойства элементов (Text, Font, Size и т.д.)

Этап 5. Связывание модулей с интерфейсом (45 минут)

В классе главной формы объявите поля для экземпляров модулей:

```
private CalculationModule calcModule;
private HistoryModule historyModule;
```

В конструкторе формы создайте экземпляры модулей:

```
public MainForm()
{
    InitializeComponent();
    calcModule = new CalculationModule();
    historyModule = new HistoryModule();
}
```

В обработчиках событий кнопок вызывайте методы модулей:

```
private void btnAction_Click(object sender, EventArgs e)
{
    // Получение данных из интерфейса
    // Вызов методов модулей
    // Отображение результата
}
```

Реализуйте передачу данных между модулями

Этап 6. Тестирование и отладка (30 минут)

1.Протестируйте каждую функцию приложения
2.Проверьте корректность обработки некорректного ввода
3.Убедитесь, что модули правильно взаимодействуют друг с другом
4.Проверьте отсутствие исключений при различных сценариях использования
5.Исправьте найденные ошибки

Требования к оформлению отчета

Отчет должен содержать:

1.Описание проектного решения: краткое описание разработанного приложения; схема модульной структуры приложения

(диаграмма взаимодействия модулей); описание назначения каждого модуля; описание способов связи между модулями

2. Листинг кода**.** Для каждого модуля**:** название модуля; полный код класса с комментариями. Для главной формы: Код основных обработчиков событий

3. Скриншоты работы приложения: главное окно приложения; демонстрация работы основных функций (минимум 3-4 скриншота); демонстрация взаимодействия модулей (если возможно визуализировать)

4. Анализ реализованной модульной структуры. Заполните таблицу:

Модуль	**Назначение**	**Публичные методы**	**С какими модулями взаимодействует**	**Тип связи**
Module1	...	Method1(), Method2()	Module2, UI	По данным
...	...	...	...	...

5.Выводы по работе: какие принципы модульного программирования были применены; какие преимущества дала модульная архитектура; какие сложности возникли при разработке; возможности дальнейшего развития приложения

Контрольные вопросы

1.Что такое модуль в программировании? Какие основные характеристики должен иметь хорошо спроектированный модуль?

2.Перечислите и объясните основные принципы модульного программирования. Как эти принципы были реализованы в вашем приложении?

3.В чем заключается разница между сильной связностью (high cohesion) и слабой связанностью (low coupling)? Почему желательна сильная связность внутри модуля и слабая связанность между модулями?

4.Какие типы связей между модулями вы знаете? Расположите их в порядке от наиболее предпочтительного к наименее

предпочтительному. Какие типы связей реализованы в вашем приложении?

5.Что такое инкапсуляция? Как модификаторы доступа (public, private, protected) помогают реализовать инкапсуляцию в C#?

6.Объясните принцип единственной ответственности (Single Responsibility Principle). Приведите примеры из вашего приложения, где этот принцип соблюден или нарушен.

7.Какие способы передачи данных между модулями в C# вы знаете? Какие из них вы использовали в своем приложении и почему?

8.В чем преимущества использования интерфейсов при проектировании модулей? Приведите пример ситуации, когда использование интерфейса было бы полезно в вашем приложении.

9.Как модульная архитектура упрощает тестирование и сопровождение программного обеспечения?

10.Предложите, как можно расширить функциональность вашего приложения путем добавления новых модулей, не изменяя существующие. Опишите назначение и интерфейс такого модуля.

11.Что такое библиотека классов (Class Library) в .NET? В каких случаях целесообразно выносить модули в отдельные библиотеки?

12.Объясните разницу между пространством имен (namespace) и библиотекой классов (assembly/DLL).

Практическая работа № 17 «Разработка приложения с информационно связанными модулями».

Цель работы: Освоить принципы проектирования и разработки многомодульных приложений на языке C#, изучить механизмы взаимодействия между модулями, научиться организовывать информационные связи между компонентами программной системы в среде Visual Studio 2022.

Время выполнения: 4 ч

Теоретическая часть

Понятие модульной архитектуры. Модуль — это логически завершенная, функционально самостоятельная часть программы, которая выполняет определенную задачу и может взаимодействовать с другими модулями через четко определенные интерфейсы. Модульное программирование — это организация программы как совокупности небольших независимых блоков (модулей), структура и поведение которых подчиняются определенным правилам.

Преимущества модульной архитектуры

1.Упрощение разработки — возможность разделения задач между разработчиками

2.Повторное использование кода — модули можно использовать в различных проектах

3.Упрощение тестирования — каждый модуль можно тестировать отдельно

4.Упрощение сопровождения — изменения в одном модуле минимально влияют на другие

5.Масштабируемость — возможность добавления новых модулей без изменения существующих

Принципы проектирования модулей

1. Принцип единственной ответственности (Single Responsibility Principle). Каждый модуль должен отвечать только за одну функциональную область и иметь только одну причину для изменения.

2. Принцип слабой связанности (Low Coupling). Модули должны быть слабо связаны друг с другом, т.е. изменения в одном модуле не должны требовать изменений в других.

3. Принцип сильной связности (High Cohesion). Элементы внутри модуля должны быть тесно связаны и работать для достижения единой цели.

4. Принцип инкапсуляции. Внутренняя реализация модуля должна быть скрыта от других модулей, доступ осуществляется только через определенный интерфейс.

Типы модулей в C#

1. Модули бизнес-логики (Business Logic Layer). Содержат основную логику приложения, реализуют бизнес-правила и

алгоритмы обработки данных.
2. Модули доступа к данным (Data Access Layer). Отвечают за взаимодействие с источниками данных (базы данных, файлы, внешние сервисы).
3. Модули представления (Presentation Layer). Реализуют пользовательский интерфейс и взаимодействие с пользователем.
4. Модули моделей данных (Data Models). Описывают структуру данных, используемых в приложении.
5. Утилитарные модули (Utility Modules). Содержат вспомогательные функции общего назначения.

Способы организации модулей в C#

1. Использование пространств имен (Namespaces)

```
namespace MyApp.DataAccess
{
   public class DatabaseManager { }
}

namespace MyApp.BusinessLogic
{
   public class OrderProcessor { }
}
```

2. Использование отдельных файлов классов. Каждый класс размещается в отдельном файле с соответствующим именем.
3. Использование отдельных проектов (Class Library). Модули выделяются в отдельные библиотеки классов (.dll), которые подключаются к основному проекту.

Информационные связи между модулями. Информационная связь — это механизм передачи данных между модулями приложения. Типы информационных связей:

1. Прямая связь через параметры методов

```
public class ModuleA
{
   public int Calculate(int value)
   {
      return value * 2;
   }
}

public class ModuleB
{
```

```
    private ModuleA moduleA = new ModuleA();

    public void Process()
    {
        int result = moduleA.Calculate(10);
    }
}
```

2. Связь через свойства объектов

```
public class DataModel
{
    public string Name { get; set; }
    public int Value { get; set; }
}

public class ModuleA
{
    public DataModel GetData()
    {
        return new DataModel { Name = "Test", Value = 100 };
    }
}

public class ModuleB
{
    public void UseData(DataModel data)
    {
        Console.WriteLine($"{data.Name}: {data.Value}");
    }
}
```

3. Связь через события (Events)

```
public class DataProvider
{
    public event EventHandler<string> DataReceived;

    public void FetchData()
    {
        string data = "New Data";
        DataReceived?.Invoke(this, data);
    }
}

public class DataConsumer
{
    public void Subscribe(DataProvider provider)
```

```
    {
        provider.DataReceived += OnDataReceived;
    }

    private void OnDataReceived(object sender, string data)
    {
        Console.WriteLine($"Received: {data}");
    }
}
```

4. Связь через делегаты и callback-функции

```
public class Processor
{
    public void Process(Func<int, int> calculator)
    {
        int result = calculator(5);
        Console.WriteLine(result);
    }
}
```

5. Связь через общие данные (Singleton, Static классы)

```
public class DataStorage
{
    private static DataStorage instance;
    private List<string> data = new List<string>();

    public static DataStorage Instance
    {
        get
        {
            if (instance == null)
                instance = new DataStorage();
            return instance;
        }
    }

    public void AddData(string item) => data.Add(item);
    public List<string> GetData() => data;
}
```

Паттерны взаимодействия модулей

1. Паттерн Repository (Репозиторий). Инкапсулирует логику доступа к данным, предоставляя единый интерфейс для работы с хранилищем.
2. Паттерн Service Layer (Слой сервисов). Определяет границы приложения и набор доступных операций, координирует

работу различных модулей.
3. Паттерн Dependency Injection (Внедрение зависимостей). Позволяет передавать зависимости извне, снижая связанность модулей.
4. Паттерн Observer (Наблюдатель). Реализует механизм подписки на события и уведомления заинтересованных объектов об изменениях.

Организация проекта в Visual Studio 2022

Структура решения (Solution):

```
MyApplication (Solution)
│
├── MyApplication.UI (Windows Forms App / Console App)
│   ├── Forms/
│   ├── Program.cs
│   └── References → MyApplication.BusinessLogic, MyApplication.DataAccess
│
├── MyApplication.BusinessLogic (Class Library)
│   ├── Services/
│   ├── Validators/
│   └── References → MyApplication.DataModels
│
├── MyApplication.DataAccess (Class Library)
│   ├── Repositories/
│   ├── DatabaseContext.cs
│   └── References → MyApplication.DataModels
│
└── MyApplication.DataModels (Class Library)
    └── Models/
```

Создание библиотеки классов в Visual Studio 2022

1.Щелкните правой кнопкой мыши на решении → **Add → New Project**
2.Выберите **Class Library (.NET или .NET Framework)**
3.Задайте имя проекта (например, MyApp.DataAccess)
4.Добавьте ссылку на библиотеку в основном проекте: правый клик на **Dependencies → Add Project Reference**; отметьте нужную библиотеку

Передача данных между модулями. Использование классов моделей данных:

```
// MyApp.DataModels/Student.cs
```

```
namespace MyApp.DataModels
{
    public class Student
    {
        public int Id { get; set; }
        public string FullName { get; set; }
        public string Group { get; set; }
        public double AverageScore { get; set; }
    }
}

// MyApp.DataAccess/StudentRepository.cs
using MyApp.DataModels;

namespace MyApp.DataAccess
{
    public class StudentRepository
    {
        private List<Student> students = new List<Student>();

        public void Add(Student student)
        {
            students.Add(student);
        }

        public List<Student> GetAll()
        {
            return students;
        }

        public Student GetById(int id)
        {
            return students.FirstOrDefault(s => s.Id == id);
        }
    }
}

// MyApp.BusinessLogic/StudentService.cs
using MyApp.DataAccess;
using MyApp.DataModels;

namespace MyApp.BusinessLogic
{
    public class StudentService
```

```
    {
        private StudentRepository repository;

        public StudentService()
        {
            repository = new StudentRepository();
        }

        public bool AddStudent(string name, string group, double score)
        {
            if (string.IsNullOrWhiteSpace(name) || score < 0 || score > 5)
                return false;

            var student = new Student
            {
                Id = GetNextId(),
                FullName = name,
                Group = group,
                AverageScore = score
            };

            repository.Add(student);
            return true;
        }

        public List<Student> GetAllStudents()
        {
            return repository.GetAll();
        }

        private int GetNextId()
        {
            var students = repository.GetAll();
            return students.Count > 0 ? students.Max(s => s.Id) + 1 : 1;
        }
    }
}
```

Обработка ошибок при межмодульном взаимодействии. При взаимодействии модулей необходимо предусматривать обработку исключительных ситуаций:

```
public class DataService
{
    public bool SaveData(DataModel data)
    {
```

```
        try
        {
            // Валидация данных
            if (data == null)
                throw new ArgumentNullException(nameof(data));

            // Сохранение данных
            repository.Save(data);
            return true;
        }
        catch (ArgumentNullException ex)
        {
            Console.WriteLine($"Ошибка: {ex.Message}");
            return false;
        }
        catch (Exception ex)
        {
            Console.WriteLine($"Непредвиденная ошибка: {ex.Message}");
            return false;
        }
    }
}
```

Рабочее задание: Разработайте многомодульное приложение в соответствии с вариантом задания. Приложение должно состоять из нескольких информационно связанных модулей.

Вариант 1. Система учета студентов. Разработать приложение для учета студентов с модульной архитектурой:

- **Модуль моделей данных**: классы Student, Group
- **Модуль доступа к данным**: класс StudentRepository для работы с коллекцией студентов
- **Модуль бизнес-логики**: класс StudentService для обработки данных студентов (добавление, удаление, поиск, расчет среднего балла по группе)
- **Модуль UI**: форма для ввода данных и отображения списка студентов

Вариант 2. Система учета книг. Разработать приложение для учета книг в библиотеке:

- **Модуль моделей данных**: классы Book, Author
- **Модуль доступа к данным**: класс BookRepository

- **Модуль бизнес-логики**: класс LibraryService (добавление книг, поиск по автору/названию, учет выдачи/возврата)
- **Модуль UI**: интерфейс для работы с библиотекой

Вариант 3. Система учета товаров. Разработать приложение для учета товаров на складе:

- **Модуль моделей данных**: классы Product, Category
- **Модуль доступа к данным**: класс ProductRepository
- **Модуль бизнес-логики**: класс WarehouseService (приход/расход товаров, расчет остатков, поиск товаров)
- **Модуль UI**: форма управления складом

Вариант 4. Система учета сотрудников. Разработать приложение для учета сотрудников организации:

- **Модуль моделей данных**: классы Employee, Department
- **Модуль доступа к данным**: класс EmployeeRepository
- **Модуль бизнес-логики**: класс HRService (прием/увольнение, расчет стажа, поиск сотрудников)
- **Модуль UI**: интерфейс для работы с персоналом

Вариант 5. Система учета заказов. Разработать приложение для учета заказов в интернет-магазине:

- **Модуль моделей данных**: классы Order, OrderItem, Customer
- **Модуль доступа к данным**: класс OrderRepository
- **Модуль бизнес-логики**: класс OrderService (создание заказа, расчет суммы, изменение статуса)
- **Модуль UI**: форма управления заказами

Вариант 6. Система учета успеваемости. Разработать приложение для учета успеваемости студентов:

- **Модуль моделей данных**: классы Student, Subject, Grade
- **Модуль доступа к данных**: класс GradeRepository
- **Модуль бизнес-логики**: класс AcademicService (ввод оценок, расчет среднего балла, формирование рейтинга)
- **Модуль UI**: интерфейс для работы с оценками

Вариант 7. Система учета транспорта. Разработать приложение для учета транспортных средств автопарка:

- **Модуль моделей данных**: классы Vehicle, Driver, Route

- **Модуль доступа к данным**: класс VehicleRepository
- **Модуль бизнес-логики**: класс FleetService (регистрация ТС, назначение водителей, учет пробега)
- **Модуль UI**: форма управления автопарком

Вариант 8. Система учета пациентов. Разработать приложение для учета пациентов в медицинском учреждении:

- **Модуль моделей данных**: классы Patient, Doctor, Appointment
- **Модуль доступа к данным**: класс PatientRepository
- **Модуль бизнес-логики**: класс MedicalService (регистрация пациентов, запись на прием, история посещений)
- **Модуль UI**: интерфейс для работы с пациентами

Вариант 9. Система учета проектов. Разработать приложение для учета проектов и задач:

- **Модуль моделей данных**: классы Project, Task, Employee
- **Модуль доступа к данным**: класс ProjectRepository
- **Модуль бизнес-логики**: класс ProjectService (создание проектов, назначение задач, отслеживание прогресса)
- **Модуль UI**: форма управления проектами

Вариант 10. Система учета курсов. Разработать приложение для учета учебных курсов и слушателей:

- **Модуль моделей данных**: классы Course, Student, Enrollment
- **Модуль доступа к данным**: класс CourseRepository
- **Модуль бизнес-логики**: класс EducationService (регистрация на курсы, отслеживание прогресса, выдача сертификатов)
- **Модуль UI**: интерфейс для работы с курсами

Требования к выполнению задания

1. Проектирование архитектуры: определите структуру модулей приложения; опишите назначение каждого модуля; определите информационные связи между модулями; составьте диаграмму взаимодействия модулей

2. Реализация модулей

Модуль моделей данных: Создайте библиотеку классов (Class Library); Определите классы моделей с необходимыми

свойствами; Используйте автосвойства (auto-properties); Добавьте конструкторы при необходимости
Модуль доступа к данным: Создайте библиотеку классов; Реализуйте класс Repository для работы с коллекциями данных; Реализуйте методы: добавление, удаление, получение всех записей, поиск по критериям; Используйте List или Dictionary<TKey, TValue> для хранения данных
Модуль бизнес-логики: Создайте библиотеку классов; Реализуйте класс Service для обработки бизнес-правил; Добавьте валидацию входных данных; Реализуйте методы для выполнения бизнес-операций; Обеспечьте взаимодействие с модулем доступа к данным
Модуль пользовательского интерфейса: Создайте Windows Forms приложение или Console приложение; Реализуйте формы для ввода и отображения данных; Обеспечьте обработку пользовательского ввода; Реализуйте вызовы методов модуля бизнес-логики; Добавьте обработку ошибок и информирование пользователя

3. Обеспечение информационных связей: Настройте ссылки между проектами (Project References); Используйте подходящие механизмы передачи данных (параметры методов, возвращаемые значения, свойства объектов); Обеспечьте слабую связанность модулей; Документируйте интерфейсы взаимодействия

4. Функциональные требования. Приложение должно обеспечивать: Добавление новых записей; Просмотр всех записей; Поиск записей по критериям; Удаление записей; Редактирование записей; Выполнение специфичных для предметной области операций (расчеты, формирование отчетов и т.д.)

5. Технические требования: Язык программирования: C#; Среда разработки: Visual Studio 2022; Версия .NET: .NET Framework 4.7.2 или .NET 6/7/8; Используйте стандарты именования C# (PascalCase для классов и методов, camelCase для

переменных); Добавьте комментарии к классам и методам; Обработайте возможные исключения

Пример выполнения задания

Структура решения "StudentManagementSystem"

StudentManagementSystem (Solution)
├── StudentManagementSystem.Models (Class Library)
├── StudentManagementSystem.DataAccess (Class Library)
├── StudentManagementSystem.BusinessLogic (Class Library)
└── StudentManagementSystem.UI (Windows Forms App)

Код модулей

StudentManagementSystem.Models/Student.cs

```
namespace StudentManagementSystem.Models
{
    /// <summary>
    /// Модель данных студента
    /// </summary>
    public class Student
    {
        public int Id { get; set; }
        public string FullName { get; set; }
        public string Group { get; set; }
        public double AverageScore { get; set; }
        public DateTime EnrollmentDate { get; set; }

        public Student()
        {
            EnrollmentDate = DateTime.Now;
        }

        public override string ToString()
        {
            return $"{FullName} ({Group}) - {AverageScore:F2}";
        }
    }
}
```

StudentManagementSystem.DataAccess/StudentRepository.cs

```
using StudentManagementSystem.Models;

namespace StudentManagementSystem.DataAccess
{
    /// <summary>
    /// Репозиторий для работы с данными студентов
```

```
/// </summary>
public class StudentRepository
{
    private List<Student> students;

    public StudentRepository()
    {
        students = new List<Student>();
    }

    /// <summary>
    /// Добавить студента
    /// </summary>
    public void Add(Student student)
    {
        students.Add(student);
    }

    /// <summary>
    /// Получить всех студентов
    /// </summary>
    public List<Student> GetAll()
    {
        return new List<Student>(students);
    }

    /// <summary>
    /// Найти студента по ID
    /// </summary>
    public Student GetById(int id)
    {
        return students.FirstOrDefault(s => s.Id == id);
    }

    /// <summary>
    /// Найти студентов по группе
    /// </summary>
    public List<Student> GetByGroup(string group)
    {
        return students.Where(s => s.Group == group).ToList();
    }

    /// <summary>
    /// Удалить студента
```

```
        /// </summary>
        public bool Remove(int id)
        {
            var student = GetById(id);
            if (student != null)
            {
                students.Remove(student);
                return true;
            }
            return false;
        }
    }
}
```

StudentManagementSystem.BusinessLogic/StudentService.cs

```
using StudentManagementSystem.Models;
using StudentManagementSystem.DataAccess;

namespace StudentManagementSystem.BusinessLogic
{
    /// <summary>
    /// Сервис бизнес-логики для работы со студентами
    /// </summary>
    public class StudentService
    {
        private StudentRepository repository;

        public StudentService()
        {
            repository = new StudentRepository();
        }

        /// <summary>
        /// Добавить нового студента с валидацией
        /// </summary>
        public bool AddStudent(string fullName, string group, double averageScore)
        {
            // Валидация
            if (string.IsNullOrWhiteSpace(fullName))
                throw new ArgumentException("Имя студента не может быть пу-
стым");

            if (string.IsNullOrWhiteSpace(group))
                throw new ArgumentException("Группа не может быть пустой");
```

```
        if (averageScore < 2.0 || averageScore > 5.0)
            throw new ArgumentException("Средний балл должен быть от 2.0
до 5.0");

        // Создание студента
        var student = new Student
        {
            Id = GetNextId(),
            FullName = fullName,
            Group = group,
            AverageScore = averageScore
        };

        repository.Add(student);
        return true;
    }

    /// <summary>
    /// Получить всех студентов
    /// </summary>
    public List<Student> GetAllStudents()
    {
        return repository.GetAll();
    }

    /// <summary>
    /// Рассчитать средний балл по группе
    /// </summary>
    public double CalculateGroupAverage(string group)
    {
        var studentsInGroup = repository.GetByGroup(group);
        if (studentsInGroup.Count == 0)
            return 0;

        return studentsInGroup.Average(s => s.AverageScore);
    }

    /// <summary>
    /// Удалить студента
    /// </summary>
    public bool DeleteStudent(int id)
    {
        return repository.Remove(id);
    }
```

```
        private int GetNextId()
        {
            var students = repository.GetAll();
            return students.Count > 0 ? students.Max(s => s.Id) + 1 : 1;
        }
    }
}
```

StudentManagementSystem.UI/MainForm.cs (фрагмент)

```
using StudentManagementSystem.BusinessLogic;
using StudentManagementSystem.Models;

namespace StudentManagementSystem.UI
{
    public partial class MainForm : Form
    {
        private StudentService studentService;

        public MainForm()
        {
            InitializeComponent();
            studentService = new StudentService();
        }

        private void btnAdd_Click(object sender, EventArgs e)
        {
            try
            {
                string fullName = txtFullName.Text;
                string group = txtGroup.Text;
                double score = double.Parse(txtScore.Text);

                studentService.AddStudent(fullName, group, score);
                MessageBox.Show("Студент добавлен успешно!");
                RefreshStudentList();
            }
            catch (Exception ex)
            {
                MessageBox.Show($"Ошибка: {ex.Message}");
            }
        }

        private void RefreshStudentList()
        {
```

```
            var students = studentService.GetAllStudents();
            listBoxStudents.DataSource = null;
            listBoxStudents.DataSource = students;
        }
    }
}
```

Содержание отчета

Отчет должен включать следующие разделы:

1. Проектирование архитектуры приложения: Описание назначения приложения; Список модулей и их назначение; Диаграмма взаимодействия модулей; Описание информационных связей между модулями
2. Описание реализации:
2.1. Модуль моделей данных: перечень классов; код классов моделей; описание свойств
2.2. Модуль доступа к данным: описание класса Repository; код методов работы с данными; используемые коллекции
2.3. Модуль бизнес-логики: описание класса Service; код методов бизнес-логики; реализованная валидация
2.4. Модуль пользовательского интерфейса: скриншоты форм приложения; описание элементов интерфейса; код обработчиков событий
3. Тестирование: сценарии тестирования; скриншоты результатов выполнения операций; тестирование обработки ошибок
4. Выводы: опишите, какие навыки были получены; какие сложности возникли при выполнении; преимущества использованной модульной архитектуры

Контрольные вопросы

1.Что такое модуль в программировании? Какие преимущества дает модульная архитектура приложения?

2.Перечислите основные принципы проектирования модулей. Объясните принципы слабой связанности и сильной связности.

3.Какие типы модулей используются в многоуровневой архитектуре приложений? Опишите назначение каждого уровня.

4.Какие способы организации модулей в C# вы знаете? В чем разница между использованием пространств имен и отдельных библиотек классов?

5.Что такое информационная связь между модулями? Перечислите основные типы информационных связей.

6.Как создать библиотеку классов (Class Library) в Visual Studio 2022? Как подключить ее к основному проекту?

7.Для чего используется паттерн Repository? Какие методы обычно реализуются в классе Repository?

8.Что такое слой бизнес-логики (Business Logic Layer)? Какие задачи решаются в этом слое?

9.Почему важна валидация данных в модуле бизнес-логики? Приведите примеры валидации.

10.Как обеспечить передачу данных между модулями? Какие механизмы C# для этого используются?

11.В чем преимущества использования моделей данных (Data Models) в многомодульном приложении?

12.Объясните принцип единственной ответственности (Single Responsibility Principle). Приведите пример его применения при проектировании модулей.

ПЕРЕЧЕНЬ ИСПОЛЬЗОВАННЫХ ИНФОРМАЦИОННЫХ РЕСУРСОВ

1.Мейер Б. Объектно-ориентированное программирование и программная инженерия/ Москва: Интернет-Университет Информационных Технологий (ИНТУИТ), Ай Пи Эр Медиа, 2018- 285 с Режим доступа http://www.iprbookshop.ru/79706.htm

2. Федорова Г.Н. Разработка модулей программного обеспечения для компьютерных систем. Учебное пособие для СПО. Академия, 2018. 384 с.

3.Фуфаев Д.Э. Разработка и эксплуатация автоматизированных информационных систем учеб. пособие для студ. СПО. Академия, 2018, 304 с.

4. Никлаус Вирт. Алгоритмы и структуры данных. Саратов: Профобразование, 2017, 272 с.

http://www.iprbookshop.ru/63821.html