Министерство науки и высшего образования Российской Федерации

Федеральное государственное автономное образовательное
учреждение высшего образования
**ТОМСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
СИСТЕМ УПРАВЛЕНИЯ И РАДИОЭЛЕКТРОНИКИ (ТУСУР)**

Кафедра автоматизированных систем управления (АСУ)

СИСТЕМА УПРАВЛЕНИЯ ПЕРСОНАЛОМ «СЛУЖАЩИЕ ФИРМЫ»

Пояснительная записка к курсовому проекту
по дисциплине «Объектно-ориентированное программирование»

Обучающийся гр. 433-1:

_____ Д.Е. Пикулин
(подпись) (И.О. Фамилия)

«___» _____ 2025 г.
(дата)

Руководитель:

ассистент каф. АСУ
(должность, ученая степень, звание)

_____ _____ П.Д. Тихонов
(оценка) (подпись) (И.О. Фамилия)

«___» _____ 2025 г.
(дата)

Томск 2025

Министерство науки и высшего образования Российской Федерации

Федеральное государственное автономное образовательное учреждение высшего образования
ТОМСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ СИСТЕМ УПРАВЛЕНИЯ И РАДИОЭЛЕКТРОНИКИ (ТУСУР)

Кафедра автоматизированных систем управления (АСУ)

ЗАДАНИЕ

на курсовое проектирование по дисциплине «Объектно-ориентированное программирование»

студенту гр. 433-1 факультета систем управления

Пикулину Дмитрию Евгеньевичу
(Ф.И.О. студента)

1. Тема курсового проекта: Система управления персоналом «служащие фирмы»»

2. Срок сдачи студентом законченного проекта: « 24 » июня 2025 г.

3. Цель проекта: разработка функциональной системы управления персоналом, которая эффективно моделирует реальные бизнес-процессы и отношения в рамках организации. Это достигается путем применения ключевых концепций ООП, таких как инкапсуляция, наследование, полиморфизм и абстракция.

4. Задачи проекта: 1) провести детальный объектный анализ и декомпозицию предметной области "Служащие фирмы". 2) Обосновать выбранные архитектурные и проектные решения, включая структуру классов, использование структур данных и применение интерфейсов. 3) Построить унифицированную диаграмму классов (UML), наглядно отражающую спроектированную архитектуру системы. 4) Реализовать программную часть системы на языке C#, обеспечив ее соответствие принципам ООП.

5) Предусмотреть и реализовать механизм сохранения и загрузки данных для обеспечения персистентности информации. 6) Выполнить тестирование

разработанного программного кода и обеспечить его адекватное документирование.

5. Исходные данные: 1) Предметная область – служащие фирмы. 2) Список источников по теме работы. 3) Методические указания Романенко Владимир Васильевич Р–69 Объектно-ориентированное программирование: методические указания к лабораторным работам, практическим занятиям и курсовому проекту / В. В. Романенко. – Томск: Томск. гос. ун-т систем упр. и радиоэлектроники, 2024. – 44 с. [Электронный ресурс] - Режим доступа: https://sdo.tusur.ru/pluginfile.php/801361/mod_resource/content/3/УМП%20ООП%202024.pdf

6. Дата выдачи задания: « 12 » февраля 2025 г.

Задание выдал:

ассистент каф. АСУ		П.Д. Тихонов
(должность, ученая степень, звание)	(подпись)	(И.О. Фамилия)

Задание принял к исполнению: « 20 » февраля 2025 г.

Студент гр. 433-1		Д.Е. Пикулин
	(подпись)	(И.О. Фамилия)

Реферат

Пояснительная записка к курсовому проекту состоит из 89 страниц, содержит 5 таблиц, 9 рисунков, 4 источника.

Курсовой проект "Система управления персоналом" успешно реализован на языке C# с последовательным применением ключевых принципов объектно-ориентированного программирования. В ходе работы была разработана четкая иерархическая структура классов, которая точно отражает предметную область "Служащие фирмы", включая такие сущности, как компания, отделы, проекты, а также различные типы сотрудников (менеджеры и специалисты).

Пояснительная записка к курсовому проекту выполнена в Microsoft Word 2021.

Сокращения и обозначения

1. UML (Unified Modeling Language) – Язык графического описания объектно-ориентированных систем.
2. SRP (Single Responsibility Principle) - это фундаментальный принцип объектно-ориентированного программирования, который гласит, что каждый модуль, класс или функция должны иметь одну и только одну причину для изменения.
3. CEO (Chief Executive Officer) — это высшая управленческая должность в компании, аналогичная должности генерального директора в российской традиции.
4. Enums (Перечисления) - в C# используются для определения набора именованных целочисленных констант. Их применение значительно улучшает читаемость кода, обеспечивает типобезопасность и предотвращает использование некорректных значений.
5. XML-комментарии — это специальные блоки комментариев в коде C#, начинающиеся с ///, которые используются для описания классов, интерфейсов, методов, свойств и других элементов кода.
6. ООП (Объектно-Ориентированное Программирование) — это парадигма программирования, в которой программа строится из взаимодействующих "объектов", объединяющих данные и методы для работы с этими данными.

Оглавление

ВЕДЕНИЕ

Настоящий курсовой проект посвящен разработке системы управления персоналом, выполненной на языке C# с применением принципов объектно-ориентированного программирования (ООП). Проект призван продемонстрировать глубокое понимание теоретических основ ООП и способность их практического применения при создании программного обеспечения. Предметная область "Служащие фирмы" выбрана не случайно и является исключительно подходящей для демонстрации возможностей ООП. Она естественным образом содержит иерархические отношения, например, различные типы сотрудников, такие как менеджеры и специалисты, которые могут быть представлены с использованием наследования. Разнообразие сущностей с уникальными атрибутами и поведением, включая сотрудников, отделы и проекты, позволяет эффективно применять инкапсуляцию для сокрытия внутренней реализации классов. Сложные взаимодействия между этими сущностями, такие как назначение сотрудников в отделы или на проекты, а также расчет заработной платы, идеально подходят для реализации полиморфизма через интерфейсы. Такой выбор предметной области дает возможность наглядно показать, как объектно-ориентированный подход упрощает моделирование реального мира в программном коде, делая его более структурированным, расширяемым и поддерживаемым.

1 ЦЕЛЬ И ЗАДАЧИ КУРСОВОГО ПРОЕКТА

Основной целью данного курсового проекта является разработка функциональной системы управления персоналом, которая эффективно моделирует реальные бизнес-процессы и отношения в рамках организации. Это достигается путем применения ключевых концепций ООП, таких как инкапсуляция, наследование, полиморфизм и абстракция.

Для достижения поставленной цели были определены следующие задачи:

- Провести детальный объектный анализ и декомпозицию предметной области "Служащие фирмы".
- Обосновать выбранные архитектурные и проектные решения, включая структуру классов, использование структур данных и применение интерфейсов.
- Построить унифицированную диаграмму классов (UML), наглядно отражающую спроектированную архитектуру системы.
- Реализовать программную часть системы на языке C#, обеспечив ее соответствие принципам ООП.
- Предусмотреть и реализовать механизм сохранения и загрузки данных для обеспечения персистентности информации.
- Выполнить тестирование разработанного программного кода и обеспечить его адекватное документирование.

1.1 Обзор предметной области "Служащие фирмы"

Система предусматривает реализацию следующих основных операций:

- **Управление персоналом:** включает найм новых сотрудников (как специалистов, так и менеджеров), их увольнение, изменение должностей и обновление контактной информации.
- **Управление отделами:** позволяет создавать новые отделы, удалять существующие (при условии отсутствия сотрудников), просматривать детальную информацию об отделах и назначать руководителей.
- **Управление проектами:** предоставляет функционал для создания новых проектов, их удаления (при условии отсутствия команды), просмотра детальной информации, назначения менеджеров проектов и управления составом команд.
- **Расчет заработной платы:** Автоматизированный расчет заработной платы для различных типов сотрудников с учетом их специфических бонусов (например, за опыт, квалификацию, количество подчиненных или участие в проектах).
- **Формирование отчетов:** Генерация кратких и полных отчетов по компании, отделам, проектам и сотрудникам, обеспечивающая наглядное представление данных.
- **Сохранение и загрузка данных:** Механизм, позволяющий сохранять текущее данные в файл и восстанавливать их.

2. ОБЪЕКТНЫЙ АНАЛИЗ И ДЕКОМПОЗИЦИЯ ПРЕДМЕТНОЙ ОБЛАСТИ

Объектный анализ предметной области "Служащие фирмы" был выполнен с целью выявления ключевых сущностей, их атрибутов, поведения и взаимосвязей. Декомпозиция предметной области является фундаментальным шагом в объектно-ориентированном проектировании, позволяющим разбить сложную систему на управляемые, логически связанные компоненты.

2.1 Обоснование декомпозиции предметной области на логические компоненты

Предметная область "Служащие фирмы" была декомпозирована на основе естественных бизнес-сущностей, каждая из которых обладает четко определенным набором атрибутов и поведений. Такой подход соответствует принципу единственной ответственности (Single Responsibility Principle - SRP), который утверждает, что у класса должна быть только одна причина для изменения. Это способствует модульности, упрощает тестирование и повышает читаемость кода, поскольку каждая часть системы имеет четко определенную роль.

Основные сущности и их роли:

- **Company (Компания):** представляет собой высший уровень агрегации в системе. Этот класс объединяет все отделы и проекты, а также имеет ссылку на генерального директора (CEO). Company является центральным объектом, инкапсулирующим общие операции и отчетность для всей организации.
- **Department (Отдел):** моделирует логическую единицу внутри компании, которая объединяет сотрудников по функциональному признаку. Каждый отдел имеет своего руководителя (Manager), список сотрудников и бюджет. Класс Department отвечает за управление сотрудниками внутри отдела и его бюджетом, но не занимается управлением проектами или общекорпоративными вопросами.
- **Project (Проект):** представляет собой временное предприятие, выполняемое командой сотрудников для достижения определенной цели. Проект имеет статус, бюджет и назначенного менеджера проекта. Класс Project инкапсулирует логику управления командой проекта и его статусом.
- **Employee (Сотрудник):** является базовой абстрактной единицей персонала, обладающей общими характеристиками, такими как идентификатор, имя, даты найма и рождения, оклад и должность. Абстрактность класса Employee позволяет определить общие черты для всех сотрудников, при этом оставляя специфическую логику (например, расчет зарплаты) для производных классов.
- **Manager (Менеджер):** Специализированный тип сотрудника, который наследует общие черты от Employee и добавляет специфические функции и атрибуты, связанные с управлением подчиненными.

- **Specialist (Специалист):** другой специализированный тип сотрудника, наследующий от Employee и добавляющий специфические свойства, связанные с их специализацией и участием в проектах.

Такая декомпозиция обеспечивает четкое разделение ответственности между классами, что делает систему более модульной, понятной и легкой для расширения и поддержки.

2.2 Взаимодействие компонентов и отношения между классами

Взаимодействие между компонентами системы "Служащие фирмы" реализуется через различные типы отношений между классами, что является основой объектно-ориентированного дизайна. Широкое использование интерфейсов и четкая декомпозиция предметной области на независимые, но взаимодействующие компоненты приводит к слабой связанности между модулями. Например, класс Company взаимодействует с Department через его публичный интерфейс, не зная о его внутренней реализации. Это означает, что изменения в одном компоненте (например, изменение логики расчета зарплаты в Specialist) с меньшей вероятностью повлияют на другие компоненты, если контракт (интерфейс) остается неизменным. Такая архитектура значительно упрощает дальнейшее развитие системы, позволяя добавлять новые типы сотрудников, отделов или проектов без существенных изменений в существующем коде, что критически важно для долгосрочной поддержки и масштабирования.

Основные типы отношений, используемые в проекте:

- **Иерархические отношения (Агрегация/Композиция):**
  - Company содержит коллекции Departments и Projects. Это отношение агрегации, где Company является "целым", а Department и Project – "частями", которые могут существовать независимо от Company.
  - Department содержит коллекцию Employees. Это также отношение агрегации, где Department является "целым", а Employee – "частями".
  - Manager содержит коллекцию Subordinates (подчиненных Employee). Это отношение агрегации, поскольку подчиненные могут существовать и без конкретного менеджера.
  - Project содержит коллекцию Team (членов команды Employee). Это также агрегация.
  - Specialist содержит коллекцию ProjectAssignments. Это отношение композиции, так как ProjectAssignment логически является частью Specialist и не имеет смысла без него.
  - Employee содержит ContactInfo. Это отношение композиции, так как контактная информация является неотъемлемой частью сотрудника.
- **Ассоциации:**
  - Company имеет ссылку на CEO (тип Employee). Это ассоциация с мощностью 0..1, так как CEO может быть не назначен.

    - Department имеет ссылку на Head (тип Manager). Это ассоциация с мощностью 0..1.
    - Project имеет ссылку на ProjectManager (тип Manager). Это ассоциация с мощностью 0..1.
    - Employee имеет ссылку на Department. Это ассоциация с мощностью 0..1, так как сотрудник может принадлежать только одному отделу в данный момент.
    - ProjectAssignment имеет ссылку на Project, к которому относится назначение. Это ассоциация с мощностью 1.
- **Наследование (Генерализация):**
    - Классы Manager и Specialist наследуют от абстрактного базового класса Employee. Это позволяет им использовать общие атрибуты и методы, определенные в Employee, и при этом реализовывать свою специфическую логику.
- **Реализация интерфейсов (Реализация):**
    - Абстрактный класс Employee реализует интерфейсы IPayable и IReportable.
    - Класс Manager реализует интерфейс IManagerial.
    - Классы Project, Company и Department реализуют интерфейс IReportable.
    - Реализация интерфейсов обеспечивает полиморфное поведение, позволяя взаимодействовать с различными объектами через единый контракт, не зная их конкретного типа.
- **Зависимости:**
    - Класс DataService зависит от класса Company и других моделей, поскольку он оперируют этими объектами для сохранения, загрузки или импорта данных.

2.3 Отношения между сущностями:

- Работник может принадлежать только к одному отделу
- Работник занимает одну должность
- Одна должность, может быть, у нескольких сотрудников
- Менеджер управляет множеством подчиненных
- Специалист может работать над несколькими проектами
- Отдел содержит множество сотрудников (связь "один ко многим")
- Проект имеет одного руководителя (менеджера) и команду специалистов
- Фирма содержит множество отделов

Эти отношения формируют четкую и гибкую структуру системы, где каждый компонент выполняет свою роль и взаимодействует с другими компонентами предсказуемым образом.

3. ОБОСНОВАНИЕ ПРОЕКТНЫХ РЕШЕНИЙ

При проектировании системы управления персоналом были приняты решения относительно использования классов, структур, интерфейсов и перечислений, каждое из которых имеет свое обоснование, направленное на повышение модульности, гибкости, читаемости и поддерживаемости кода.

3.1 Использование классов

Классы являются основными строительными блоками в объектно-ориентированном программировании, позволяющими инкапсулировать данные и поведение. В данном проекте каждый класс моделирует ключевую сущность предметной области.

- **Employee (Абстрактный класс):**
  - **Назначение:** служит базовым классом для всех типов сотрудников в системе. Он инкапсулирует общие атрибуты, такие как идентификатор, имя, даты рождения и найма, базовая зарплата, должность, контактная информация и отдел. Также он определяет общее поведение, например, расчет стажа работы (CalculateExperience(), CalculateExperienceString()) и генерацию отчетов (GenerateReport()).
  - **Обоснование абстрактности:** Этот класс абстрактный, так как в чистом виде сотрудник не существует – он всегда принадлежит к конкретной должности. Метод CalculateSalary() объявлен как абстрактный (public abstract decimal CalculateSalary();). Это обусловлено тем, что логика расчета заработной платы существенно различается для разных типов сотрудников (например, для менеджеров и специалистов). Объявление метода абстрактным в базовом классе Employee гарантирует, что каждый производный класс *обязан* предоставить свою конкретную реализацию этого метода. Это обеспечивает полиморфизм и соблюдение контракта, определенного интерфейсом IPayable, который реализует Employee.
- **Manager:**
  - **Назначение:** представляет сотрудников, выполняющих управленческие функции. Этот класс наследует от Employee и дополнительно реализует интерфейс IManagerial.
  - **Обоснование:** Класс Manager добавляет специфические свойства, такие как список Subordinates (подчиненных), ManagementLevel (уровень управления) и ManagementBonus (бонус за управление). Он переопределяет метод CalculateSalary() для включения бонусов, зависящих от стажа и количества подчиненных. Реализация IManagerial предоставляет методы для добавления, удаления и получения списка подчиненных, что является ключевой управленческой функцией.
- **Specialist:**
  - **Назначение:** представляет сотрудников, выполняющих специализированные

задачи, таких как разработчики или аналитики. Этот класс также наследует от Employee.
    - **Обоснование:** Specialist добавляет специфические свойства, включая Specialization (специализация), QualificationLevel (уровень квалификации) и ProjectAssignments (список назначений на проекты). Метод CalculateSalary() переопределен для учета бонусов, зависящих от стажа, уровня квалификации и часов, отработанных на проектах. Класс также предоставляет методы для управления проектами, к которым привязан специалист.
- **Department:**
    - **Назначение:** моделирует структурное подразделение компании. Он инкапсулирует информацию об отделе (идентификатор, название, руководитель, бюджет) и управляет списком своих сотрудников.
    - **Обоснование:** Класс Department отвечает за логику добавления (AddEmployee()) и удаления (RemoveEmployee()) сотрудников, а также за расчет общей зарплаты отдела (CalculateTotalSalary()) и остатка бюджета (GetBudgetRemainder()). Он также позволяет назначать руководителя отдела (SetHead()), обеспечивая, чтобы все сотрудники отдела становились подчиненными нового руководителя.
- **Project:**
    - **Назначение:** представляет проект, выполняемый компанией. Он инкапсулирует информацию о проекте, такую как идентификатор, название, даты начала и окончания, бюджет, статус и назначенного менеджера проекта. Подробнее см *Диаграмма состояний для проекта*
    - **Обоснование:** Класс Project управляет списком членов команды (Team) и предоставляет методы для их назначения (AssignTeamMember()) и удаления (RemoveTeamMember()). Он также позволяет обновлять статус проекта (UpdateStatus()) и рассчитывать его продолжительность (GetDuration()).

3.2 Диаграмма состояний для проекта

Рисунок 3.1 – диаграмма состояний для проекта

- **Company:**
  - **Назначение:** представляет всю организацию, являясь корневым объектом системы. Он агрегирует коллекции Departments и Projects, а также имеет ссылку на генерального директора (CEO).
  - **Обоснование:** Company обеспечивает высокоуровневое управление и отчетность по всей организации. Он предоставляет методы для добавления/удаления отделов и проектов, получения общего количества сотрудников и генерации отчетов о состоянии всей компании.

Представленная ниже таблица систематизирует ключевую информацию о каждом классе, делая ее легкодоступной для понимания структуры проекта.

Таблица 1: Обзор классов и их назначение

Название класса	Назначение (краткое описание)	Ключевые свойства	Ключевые методы	Примечания
Employee	Абстрактный базовый класс для всех сотрудников.	Id, Name, BaseSalary, Position, ContactInfo, Department	CalculateSalary() (абстрактный), GenerateReport(), CalculateExperience()	Абстрактный, реализует IPayable, IReportable
Manager	Сотрудник с управленческими функциями.	Subordinates, ManagementLevel, ManagementBonus	CalculateSalary() (переопределен), AddSubordinate(), GetSubordinates()	Наследует от Employee, реализует IManagerial
Specialist	Сотрудник, выполняющий специализированные задачи.	Specialization, QualificationLevel, ProjectAssignments	CalculateSalary() (переопределен), AddProject(), UpdateQualification()	Наследует от Employee
Department	Структурное подразделение компании.	Id, Name, Head, Employees, Budget	SetHead(), AddEmployee(), CalculateTotalSalary(), GenerateReport()	Реализует IReportable
Project	Инициатива, выполняемая командой сотрудников.	Id, Name, StartDate, EndDate, Budget, Status, ProjectManager, Team	AssignTeamMember(), UpdateStatus(), GetDuration(), GenerateReport()	Реализует IReportable
Company	Высший уровень агрегации, представляющий всю организацию.	Name, Departments, CEO, Projects	AddDepartment(), AddProject(), GetTotalEmployees(), GenerateReport()	Реализует IReportable

3.3 Использование структур

В C# структуры (struct) являются значимыми типами, в отличие от классов, которые являются ссылочными. Выбор между классом и структурой зависит от семантики типа и его предполагаемого использования. В данном проекте структуры ContactInfo и ProjectAssignment используются для представления небольших, логически связанных групп данных, которые по своей сути являются *значениями*, а не *сущностями* с уникальной идентичностью, требующими полиморфизма или сложной иерархии.

- **ContactInfo:**
  - **Назначение:** используется для хранения контактных данных сотрудника, включая адрес электронной почты (Email), номер телефона (Phone) и физический адрес (Address).
  - **Обоснование выбора struct:** ContactInfo представляет собой небольшую, неизменяемую (по сути, хотя свойства и имеют set) группу данных. Две структуры ContactInfo с одинаковыми данными считаются эквивалентными, что характерно для значимых типов. Использование struct в этом случае может привести к более эффективному использованию памяти, так как экземпляры структур обычно размещаются на стеке (для локальных переменных) или встраиваются непосредственно в объекты, содержащие их, что уменьшает накладные расходы сборщика мусора. Это также может улучшить производительность при копировании, поскольку копируется само значение, а не ссылка. Структура помечена атрибутом **[Serializable]**, что позволяет ей быть сериализованной вместе с классом Employee.

- **ProjectAssignment:**
  - **Назначение:** описывает факт назначения сотрудника на конкретный проект, включая идентификатор проекта (ProjectId), ссылку на объект Project, идентификатор сотрудника (EmployeeId), дату назначения (AssignmentDate) и количество часов в неделю (HoursPerWeek), выделенных на проект.
  - **Обоснование выбора struct:** аналогично ContactInfo, ProjectAssignment является небольшим набором связанных данных, описывающих факт назначения. Это значение, а не сущность, и его жизненный цикл тесно связан с сотрудником, которому оно принадлежит. Использование структуры здесь также способствует оптимизации памяти и производительности. Атрибут **[Serializable]** обеспечивает возможность сохранения этой информации в рамках общей системы персистентности.

Использование структур для таких данных демонстрирует понимание тонкостей системы типов C# и оптимизации, где выбор между классом и структурой делается на основе семантики данных и требований к производительности.

3.4 Использование интерфейсов

Интерфейсы в ООП определяют контракт поведения, который могут реализовывать различные классы, обеспечивая полиморфизм и слабую связанность между компонентами системы.

- **IPayable:**
  - **Назначение:** определяет единый контракт для объектов, которым может начисляться оплата. Позволяет обрабатывать различные типы оплачиваемых сущностей единообразно. Интерфейс содержит единственный метод CalculateSalary(), который возвращает сумму к выплате.
  - **Реализация:** Реализован абстрактным классом Employee. Конкретные реализации метода CalculateSalary() предоставлены в производных классах Manager и Specialist, каждый из которых имеет свою уникальную логику расчета зарплаты с учетом специфических бонусов.
  - **Обоснование:** обеспечивает полиморфизм в расчете зарплаты. Любой объект, реализующий IPayable, может быть использован для вычисления оплаты, независимо от его конкретного типа. Это позволяет, например, легко суммировать зарплаты всех сотрудников в отделе или компании, просто итеративно вызывая CalculateSalary() для каждого объекта IPayable в коллекции, без необходимости проверки его конкретного типа.
- **IManagerial:**
  - **Назначение:** определяет контракт для объектов, обладающих управленческими функциями. Абстрагирует управленческую функциональность от конкретных классов. Интерфейс включает методы для добавления подчиненных (AddSubordinate()), удаления подчиненных (RemoveSubordinate()) и получения списка подчиненных (GetSubordinates()).
  - **Реализация:** Реализован классом Manager.
  - **Обоснование:** позволяет унифицировать операции по управлению подчиненными. Любой объект, реализующий IManagerial, может добавлять, удалять и получать список подчиненных, что способствует принципу "разделяй и властвуй". Это позволяет легко расширять систему, добавляя новые типы менеджеров или сущностей с управленческими функциями, без изменения существующего кода, работающего с этим интерфейсом.
- **IReportable:**
  - **Назначение:** определяет контракт для объектов, способных формировать отчеты. Стандартизация механизма отчетности. Интерфейс включает метод GenerateReport(string format), который возвращает строку с отчетом в указанном формате.
  - **Реализация:** Реализован множеством классов в системе: Employee (абстрактный класс), Project, Company, Department. Классы Specialist и Manager переопределяют реализацию GenerateReport() из базового класса Employee для добавления специфической информации.

- **Обоснование:** обеспечивает единый механизм генерации отчетов для различных сущностей системы. Это позволяет, например, в консольном приложении (UserInterface.cs) вызывать GenerateReport(format) для любого из этих объектов, получая информацию в нужном формате ("short" или "full"), без необходимости знать их конкретный тип. Это демонстрирует мощь полиморфизма и упрощает создание универсальных функций для отчетности и отображения информации.

Представленная ниже таблица наглядно демонстрирует, как интерфейсы используются для определения контрактов и обеспечения полиморфизма в системе.

Таблица 2: Обзор интерфейсов и их реализация

Название интерфейса	Назначение	Ключевые методы	Реализующие классы	Обоснование использования
IPayable	Определяет контракт для объектов, которым может начисляться оплата.	CalculateSalary()	Employee (абстрактный), Manager, Specialist	Обеспечивает полиморфный расчет зарплаты для разных типов сотрудников.
IManagerial	Определяет контракт для объектов с управленческими функциями.	AddSubordinate(), RemoveSubordinate(), GetSubordinates()	Manager	Унифицирует операции по управлению подчиненными, способствует расширяемости.
IReportable	Определяет контракт для объектов, способных формировать отчеты.	GenerateReport(string format)	Employee, Project, Company, Department, Manager, Specialist	Обеспечивает единый механизм генерации отчетов для различных сущностей системы.

3.5 Использование перечислений (Enums)

Перечисления (enum) в C# используются для определения набора именованных целочисленных констант. Их применение значительно улучшает читаемость кода, обеспечивает типобезопасность и предотвращает использование некорректных значений.

- **EmployeePosition:**
  - **Назначение:** определяет различные должности, которые может занимать сотрудник в компании, от JuniorSpecialist до CEO.
  - **Обоснование:** Использование enum вместо строковых или целочисленных констант для должностей обеспечивает типобезопасность и читаемость кода.

Вместо использования строковых значений, подверженных ошибкам ввода (например, "Junior Specialist" против "JuniorSpecialist"), используются предопределенные, проверяемые компилятором константы. Это устраняет риск опечаток, обеспечивает проверку типов на этапе компиляции и делает намерения разработчика более явными. Кроме того, интегрированные среды разработки (IDE) могут предлагать автодополнение для значений перечислений, что ускоряет разработку и снижает количество ошибок.

- **ProjectStatus:**
  - **Назначение:** определяет возможные статусы проекта на различных этапах его жизненного цикла, такие как Planning, InProgress, Completed и Cancelled.
  - **Обоснование:** аналогично EmployeePosition, использование enum для статусов проекта гарантирует, что проект может находиться только в одном из допустимых и предопределенных состояний. Это упрощает логику управления проектами, предотвращает присвоение некорректных или неопределенных статусов и делает код более надежным.
- **ManagementLevel:**
  - **Назначение:** определяет различные уровни управления для менеджеров: TeamLead, MiddleManager, TopManager.
  - **Обоснование:** позволяет четко классифицировать менеджеров по их иерархическому уровню. Это может быть использовано для определения прав доступа, структуры подчинения или, как в данном проекте, для расчета управленческих бонусов (ManagementBonus в классе Manager).

Использование перечислений является примером хорошей практики кодирования, направленной на создание более надежного и поддерживаемого программного продукта. Представленная ниже таблица демонстрирует, как перечисления используются для определения фиксированных наборов значений, улучшая типобезопасность и читаемость.

Таблица 3: Обзор перечислений

Название перечисления	Назначение	Ключевые значения	Контекст использования
EmployeePosition	Должности сотрудников.	JuniorSpecialist, TeamLead, CEO и др.	Свойство Position в классе Employee.
ProjectStatus	Статусы проекта.	Planning, InProgress, Completed и др.	Свойство Status в классе Project.
ManagementLevel	Уровни управления для менеджеров.	TeamLead, MiddleManager, TopManager	Свойство ManagementLevel в классе Manager.

4. АРХИТЕКТУРА ПРОГРАММНОГО РЕШЕНИЯ

Архитектура программного решения "Система управления персоналом" организована в виде многопроектной структуры, что способствует разделению ответственности, модульности и облегчает дальнейшее развитие и поддержку системы.

4.1 Организация структуры файловой системы и проектов

Структура проекта в файловой системе:

```
EmployeeManagementSystem (решение)
├── EmployeeManagementSystem.Core (библиотека классов)
│   ├── Interfaces/
│   │   ├── IPayable.cs
│   │   ├── IManagerial.cs
│   │   └── IReportable.cs
│   ├── Enums/
│   │   ├── EmployeePosition.cs
│   │   ├── ProjectStatus.cs
│   │   └── ManagementLevel.cs
│   ├── Models/
│   │   ├── Structs/
│   │   │   ├── ContactInfo.cs
│   │   │   └── ProjectAssignment.cs
│   │   ├── Employee.cs
│   │   ├── Manager.cs
│   │   ├── Specialist.cs
│   │   ├── Department.cs
│   │   ├── Project.cs
│   │   └── Company.cs
│   └── Services/
│       └── DataService.cs
└── EmployeeManagementSystem.ConsoleApp (консольное приложение)
    ├── Program.cs
    └── UserInterface.cs
```

Решение EmployeeManagementSystem разделено на два основных проекта: EmployeeManagementSystem.Core и EmployeeManagementSystem.ConsoleApp.

- **EmployeeManagementSystem.Core (Библиотека классов):**
  - **Назначение:** Этот проект является ядром системы и содержит основную бизнес-логику, модели данных и сервисы, не зависящие от конкретного типа пользовательского интерфейса. Он разработан как библиотека классов, что позволяет легко переиспользовать его функциональность в различных приложениях (например, в будущем может быть разработан графический интерфейс или веб-приложение на основе этого ядра).

- **Структура:**
  - **Interfaces**/: содержит определения всех интерфейсов (IPayable.cs, IManagerial.cs, IReportable.cs), которые определяют контракты поведения и способствуют полиморфизму и слабой связанности между компонентами.
  - **Enums**/: содержит определения перечислений (EmployeePosition.cs, ProjectStatus.cs, ManagementLevel.cs), обеспечивающих типобезопасность и улучшающих читаемость кода.
  - **Models**/: В этой папке находятся определения основных классов и структур, представляющих сущности предметной области (Employee.cs, Manager.cs, Specialist.cs, Department.cs, Project.cs, Company.cs). Внутри Models/ есть вложенная папка Structs/, где хранятся определения структур (ContactInfo.cs, ProjectAssignment.cs).
  - **Services**/: содержит класс, предоставляющий сервисные функции, такие как сохранение и загрузка данных (DataService.cs).
- **EmployeeManagementSystem.ConsoleApp (Консольное приложение):**
  - **Назначение:** Этот проект представляет собой пользовательский интерфейс для взаимодействия с основной логикой, определенной в EmployeeManagementSystem.Core. Он является "тонким клиентом", который лишь использует функциональность Core для получения данных от пользователя, вызова соответствующих бизнес-операций и отображения результатов. ConsoleApp имеет зависимость от проекта Core.
  - **Структура:**
    - Program.cs: является точкой входа в приложение (Main метод), где происходит инициализация пользовательского интерфейса и обработка глобальных исключений.
    - UserInterface.cs: Основной класс, реализующий консольное меню, логику обработки пользовательского ввода и вывода информации. Он взаимодействует с классами из Core для выполнения всех бизнес-операций.

Разделение решения на два проекта (Core и ConsoleApp) является ключевым архитектурным решением, демонстрирующим принцип разделения ответственности. Core выступает как библиотека классов, содержащая чистую бизнес-логику, которая не зависит от конкретного типа пользовательского интерфейса. ConsoleApp, в свою очередь, является тонким клиентом, который лишь использует функциональность Core. Это позволяет в будущем легко заменить ConsoleApp на другой тип интерфейса (например, веб-приложение или настольное GUI-приложение) без изменения основной логики. Такая архитектура повышает модульность, облегчает тестирование (Core можно тестировать независимо), способствует повторному использованию кода и упрощает поддержку системы в целом.

4.2 Назначение ключевых папок и файлов

Каждая папка и файл в структуре проекта имеют четко определенное назначение, что способствует организации кода и его легкому пониманию.

- **EmployeeManagementSystem.Core/Interfaces/**: Эта папка предназначена для хранения определений интерфейсов. Интерфейсы (IPayable.cs, IManagerial.cs, IReportable.cs) служат контрактами, которые определяют поведение классов, не вдаваясь в детали реализации. Это способствует полиморфизму и слабой связанности, позволяя различным классам взаимодействовать через общий набор методов.
- **EmployeeManagementSystem.Core/Enums/**: Здесь хранятся определения перечислений (EmployeePosition.cs, ProjectStatus.cs, ManagementLevel.cs). Перечисления используются для создания наборов именованных констант, что повышает типобезопасность, читаемость кода и предотвращает использование некорректных значений.
- **EmployeeManagementSystem.Core/Models/**: Эта папка содержит определения классов и структур, которые представляют основные сущности предметной области. Здесь находятся базовые классы (Employee.cs), специализированные классы (Manager.cs, Specialist.cs), а также классы, описывающие организационную структуру (Department.cs, Project.cs, Company.cs).
    - **EmployeeManagementSystem.Core/Models/Structs/**: Вложенная папка для структур (ContactInfo.cs, ProjectAssignment.cs), которые используются для представления небольших, логически связанных групп данных.
- **EmployeeManagementSystem.Core/Services/**: содержит классы, предоставляющие сервисные функции, которые не относятся напрямую к бизнес-сущностям, но необходимы для работы системы. Примерами являются DataService.cs для сохранения/загрузки данных.
- **EmployeeManagementSystem.ConsoleApp/Program.cs**: это точка входа в консольное приложение. Здесь происходит инициализация основного объекта пользовательского интерфейса (UserInterface) и запускается главный цикл программы. Также предусмотрена базовая обработка глобальных исключений.
- **EmployeeManagementSystem.ConsoleApp/UserInterface.cs**: Этот класс реализует всю логику взаимодействия с пользователем через консоль. Он отвечает за отображение меню, обработку пользовательского ввода, вызов соответствующих методов бизнес-логики из Core и вывод результатов. Здесь также реализован метод ImportTestData() для быстрой инициализации системы тестовыми данными.

Представленная ниже таблица визуализирует иерархию проекта и кратко объясняет назначение каждого компонента, что является прямым ответом на вопрос об организации структуры программного проекта.

Таблица 4: Структура файловой системы проекта

Путь к файлу/папке	Назначение/Содержимое	Ключевые классы/интерфейсы
EmployeeManagementSystem.Core/	Основная бизнес-логика, модели данных, сервисы.	Employee, Manager, Specialist, Department, Project, Company, DataService
Interfaces/	Определения контрактов поведения.	IPayable, IManagerial, IReportable
Enums/	Определения перечислений.	EmployeePosition, ProjectStatus, ManagementLevel
Models/	Классы и структуры, представляющие сущности предметной области.	Employee, Manager, Specialist, Department, Project, Company
Structs/	Вложенная папка для структур.	ContactInfo, ProjectAssignment
Services/	Классы, предоставляющие сервисные функции.	DataService
EmployeeManagementSystem.ConsoleApp/	Пользовательский интерфейс консольного приложения.	Program, UserInterface
Program.cs	Точка входа в приложение.	Program
UserInterface.cs	Логика взаимодействия с пользователем, консольное меню.	UserInterface

5. UML ДИАГРАММА КЛАССОВ

Построение UML-диаграммы классов для проекта EmployeeManagementSystem.Core является ключевым этапом, позволяющим визуализировать архитектуру системы, ее компоненты и взаимосвязи. Диаграмма служит не просто требованием курсового проекта, а фундаментальным инструментом в разработке программного обеспечения. Она позволяет быстро оценить структуру системы, выявить потенциальные проблемы в дизайне (например, циклические зависимости или чрезмерную связанность) до начала кодирования. Это способ верифицировать, насколько реализованный код соответствует задуманному дизайну, и насколько хорошо принципы ООП (наследование, полиморфизм, инкапсуляция) были применены. UML-диаграмма является ключевым инструментом для объяснения сложных отношений между классами и демонстрации глубокого понимания объектно-ориентированного подхода.

5.1 Построение UML диаграммы для EmployeeManagementSystem.Core

1. Элементы:

- **Классы:** Company, Department, Employee (абстрактный), Manager, Specialist, Project, ContactInfo (структура), ProjectAssignment (структура), DataService.
- **Интерфейсы:** IPayable, IManagerial, IReportable.
- **Перечисления:** EmployeePosition, ProjectStatus, ManagementLevel.

2. Атрибуты и методы:

Для каждого элемента были определены его атрибуты (свойства) и методы, а также их видимость (публичные, приватные, защищенные) и типы. Например, класс Employee имеет публичные атрибуты Id, Name, BaseSalary, Position, ContactInfo, Department, приватные BirthDate, HireDate, а также абстрактный метод CalculateSalary() и виртуальный GenerateReport().

3. Отношения:

- **Наследование (Generalization):**
  - Manager наследует от Employee.
  - Specialist наследует от Employee.
  - На диаграмме это отображается сплошной линией с полым треугольником, указывающим от подкласса к суперклассу.
- **Реализация интерфейсов (Realization):**
  - Employee реализует IPayable и IReportable.
  - Manager реализует IManagerial.
  - Project реализует IReportable.
  - Company реализует IReportable.
  - Department реализует IReportable.

- **Ассоциации:**
  - **Company и Department:** Company агрегирует Departments (мн. число). Мощность: Company (1) --- Departments (0..*).
  - **Company и Project:** Company агрегирует Projects (мн. число). Мощность: Company (1) --- Projects (0..*).
  - **Company и Employee (CEO):** Company ассоциирована с CEO. Мощность: Company (1) --- CEO (0..1).
  - **Department и Employee:** Department агрегирует Employees (мн. число). Мощность: Department (1) --- Employees (0..*).
  - **Department и Manager (Head):** Department ассоциирован с Head. Мощность: Department (1) --- Head (0..1).
  - **Employee и Department:** Employee ассоциирован с Department. Мощность: Employee (0..*) --- Department (0..1).
  - **Manager и Employee (Subordinates):** Manager агрегирует Subordinates (мн. число). Мощность: Manager (1) --- Subordinates (0..*).
  - **Project и Employee (Team):** Project агрегирует Team (мн. число). Мощность: Project (1) --- Team (0..*).
  - **Project и Manager (ProjectManager):** Project ассоциирован с ProjectManager. Мощность: Project (1) --- ProjectManager (0..1).
  - **Specialist и ProjectAssignment:** Specialist композиционно связан с ProjectAssignments (мн. число). Мощность: Specialist (1) --- ProjectAssignments (0..*).
  - **ProjectAssignment и Project:** ProjectAssignment ассоциирован с Project. Мощность: ProjectAssignment (0..*) --- Project (1).
  - **Employee и ContactInfo:** Employee композиционно связан с ContactInfo. Мощность: Employee (1) --- ContactInfo (1).
- **Зависимости (Dependency):**
  - **DataService** зависит от Company (и других моделей) для операций сохранения/загрузки.
  - На диаграмме это отображается пунктирной линией с открытой стрелкой, указывающей от зависимого класса к классу, от которого он зависит.

5.2 Диаграмма классов (основная)

UML ДИАГРАММА КЛАССОВ

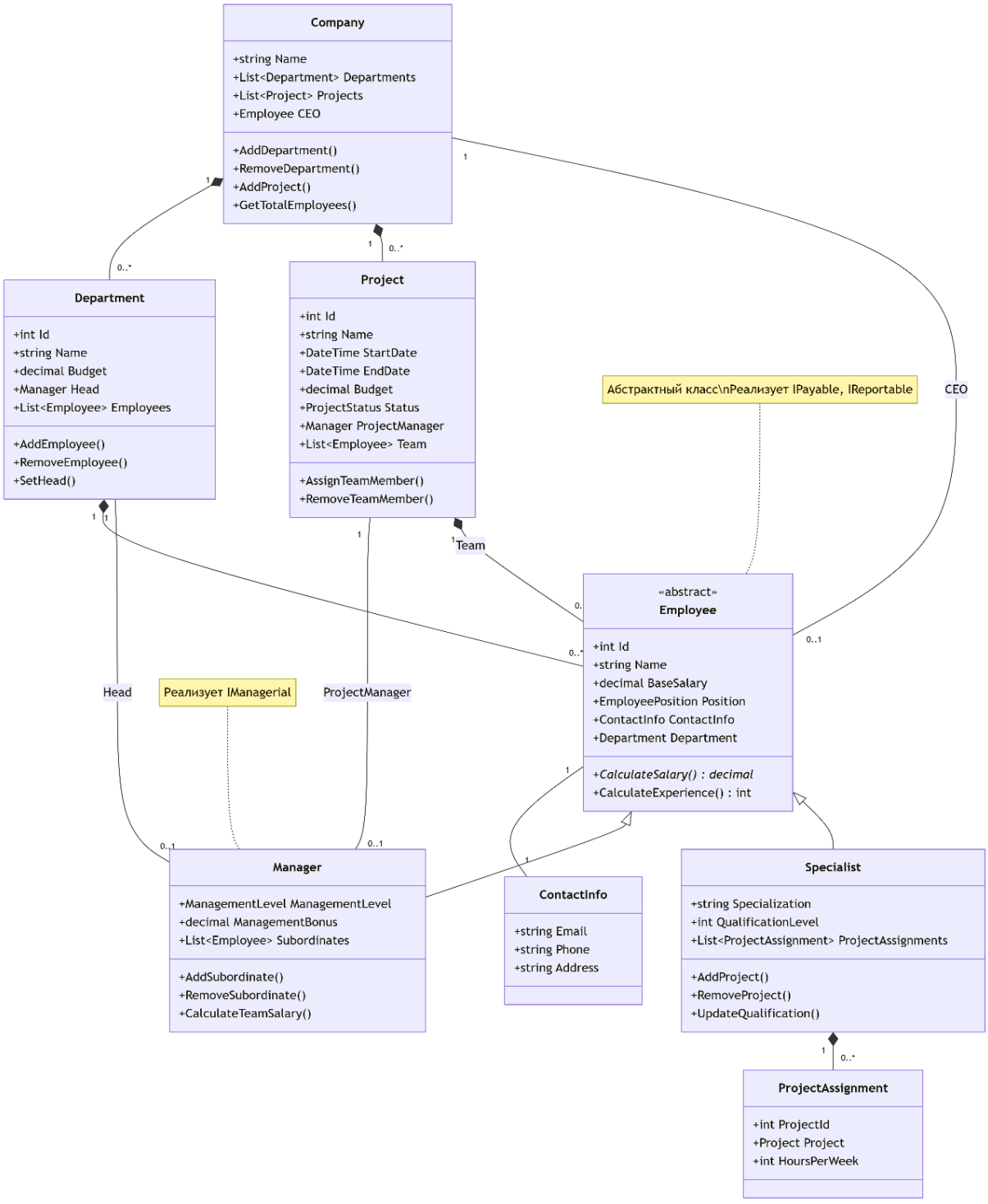

Рисунок 5.1 – UML диаграмма классов

5.3 Диаграмма вариантов использования

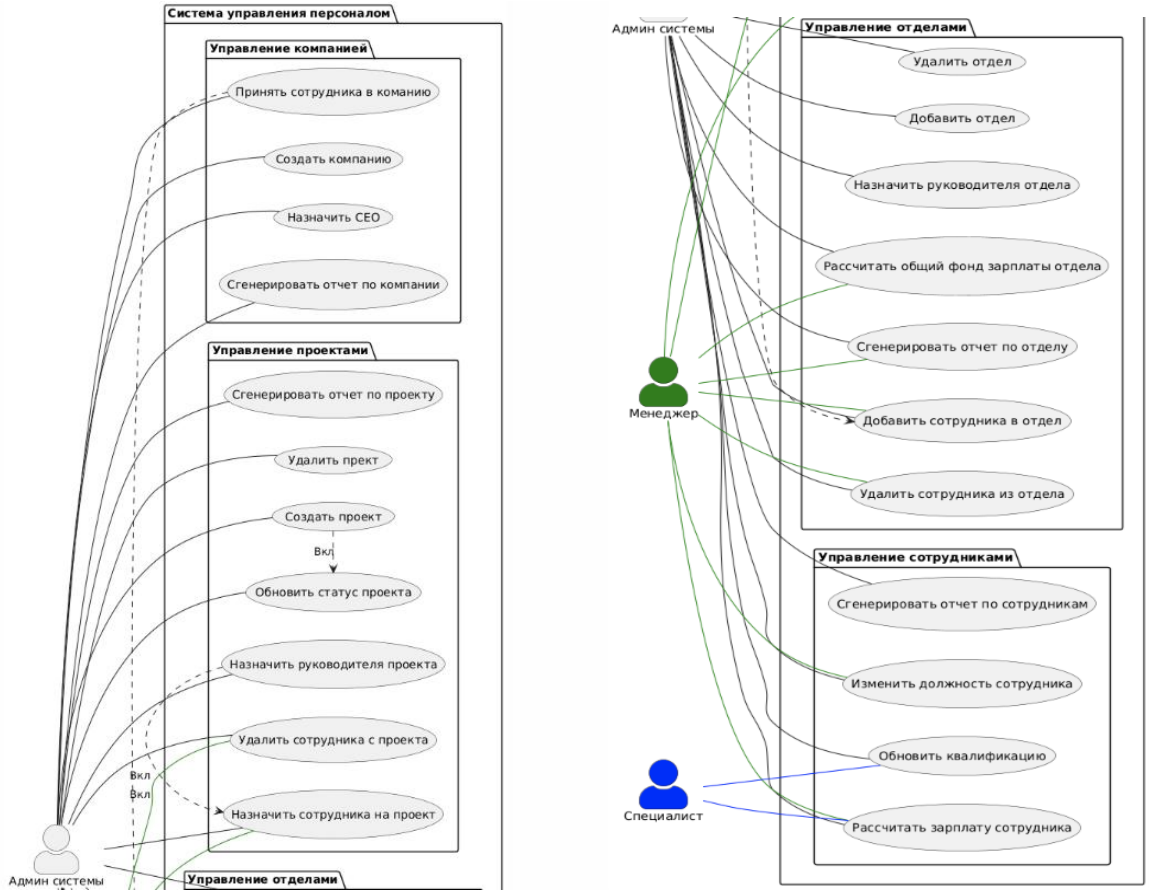

Рисунок 5.2 – диаграмма вариантов использования

6. РЕАЛИЗАЦИЯ ФУНКЦИОНАЛЬНОСТИ

Программная реализация системы управления персоналом охватывает широкий спектр функциональных возможностей, позволяющих эффективно управлять данными о сотрудниках, отделах и проектах.

6.1 Обзор ключевых функциональностей

Система предоставляет следующие ключевые функциональности, реализованные преимущественно в проекте EmployeeManagementSystem.Core и доступные через консольный интерфейс EmployeeManagementSystem.ConsoleApp :

- **Главное меню:**
  Точка входа в «систему управления персоналом»

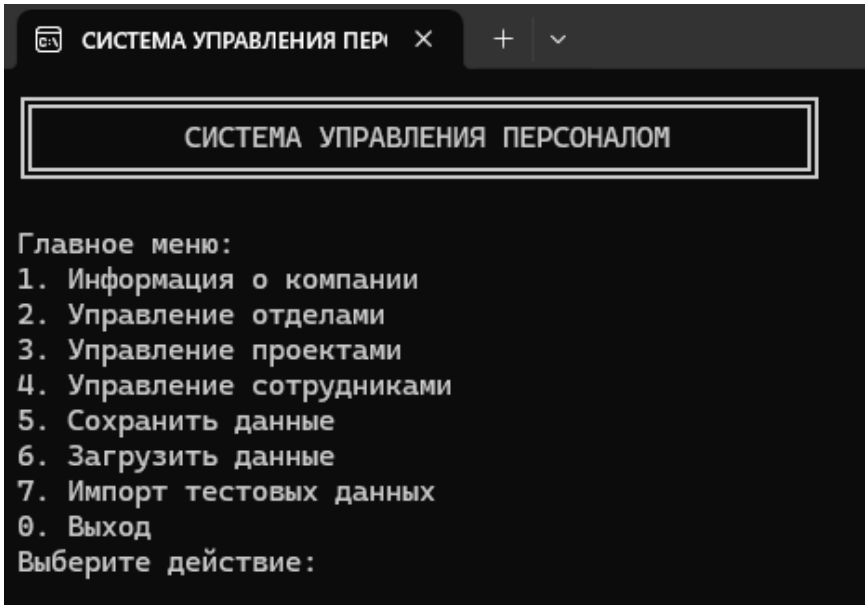

Рисунок 6.1 – главное меню

- **Управление компанией:**
  - **Изменение названия компании:** Пользователь может изменить название компании через меню.
  - **Назначение CEO:** Предусмотрена возможность назначения генерального директора из числа существующих менеджеров. При этом, если CEO уже был назначен, старый CEO может быть "уволен" из своей должности.
  - **Генерация отчетов о компании:** Система может формировать краткие и полные отчеты, предоставляющие общую информацию о компании, количестве отделов, проектов и сотрудников, а также детальные списки отделов и проектов.

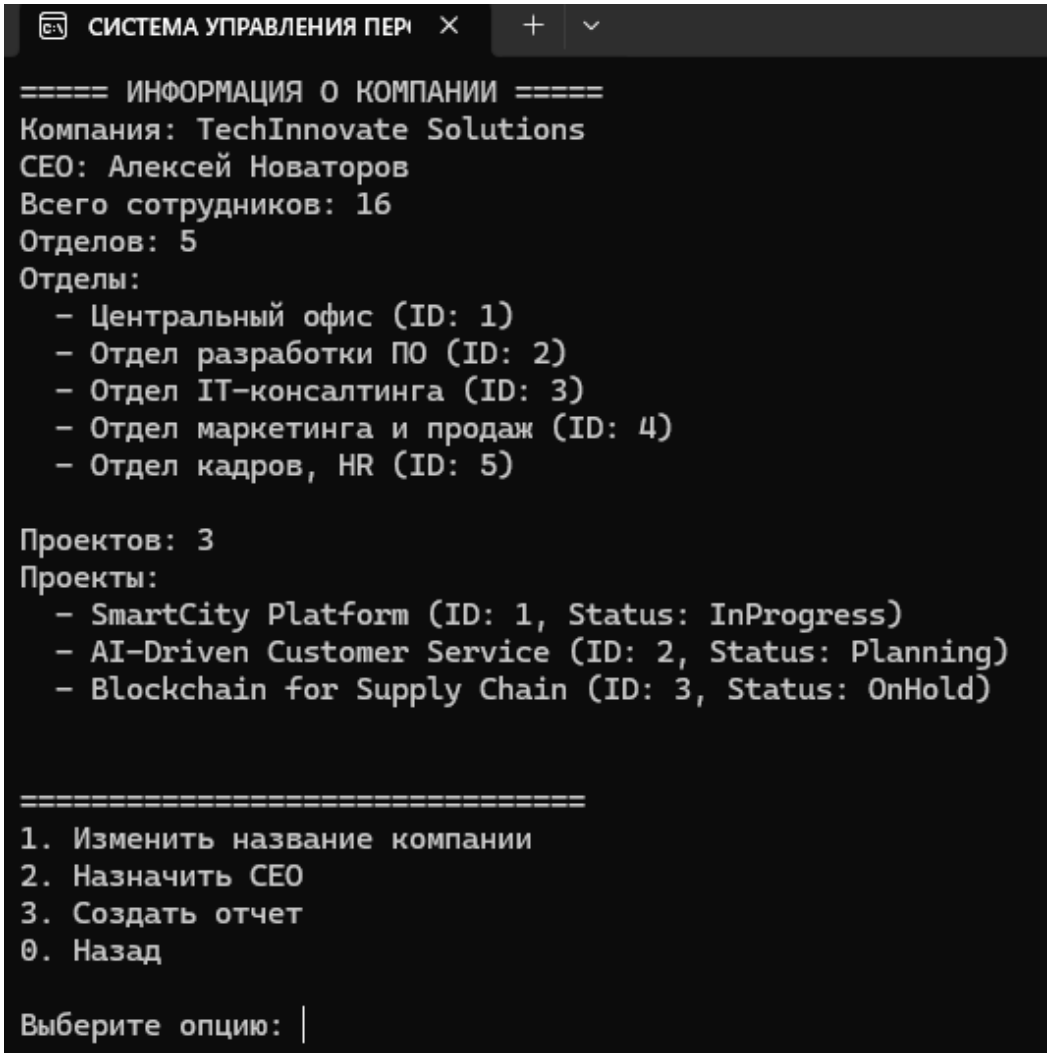

Рисунок 6.2 – информация о компании

- **Управление отделами:**
  - **Просмотр списка отделов:** отображается список всех зарегистрированных отделов с их основными параметрами.
  - **Добавление нового отдела:** Пользователь может создать новый отдел, указав его ID, название и бюджет. Система проверяет уникальность ID отдела.
  - **Удаление отдела:** Отдел может быть удален, однако существует важное ограничение: отдел не может быть удален, если в нем есть сотрудники. Это предотвращает потерю данных и обеспечивает целостность структуры компании.
  - **Просмотр детальной информации об отделе:** для выбранного отдела можно получить полный отчет, включающий его руководителя, бюджет, общую зарплату сотрудников и список всех сотрудников отдела с их должностями и зарплатами.
  - **Назначение руководителя отдела:** Существующий менеджер может быть назначен руководителем отдела. При этом все сотрудники отдела автоматически становятся подчиненными нового руководителя.

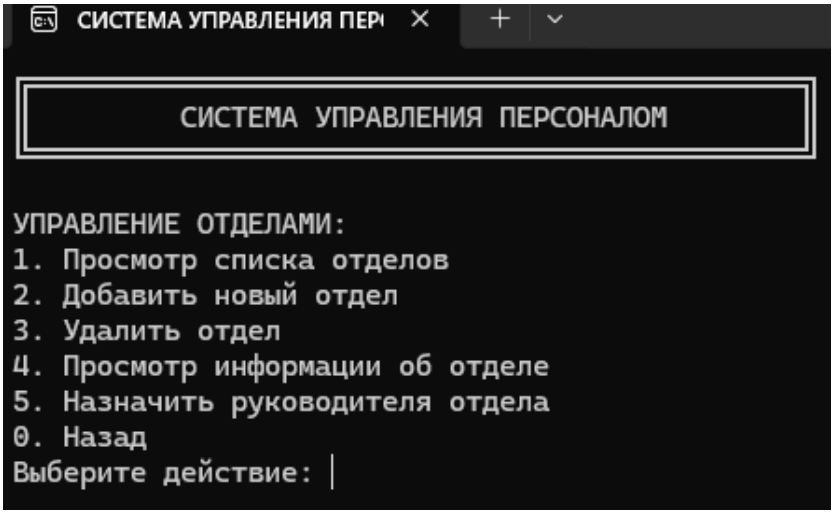

Рисунок 6.3 – управление отделами

- **Управление проектами:**
  - **Просмотр списка проектов:** отображается список всех проектов компании с их статусами, бюджетами и количеством участников.
  - **Добавление нового проекта:** Пользователь может создать новый проект, указав его ID, название, даты начала и окончания, бюджет и статус.
  - **Удаление проекта:** Проект может быть удален. Важное ограничение: проект не может быть удален, если в нем есть члены команды. При удалении проекта из системы также удаляются все связанные с ним назначения у специалистов.
  - **Просмотр детальной информации о проекте:** для выбранного проекта доступен полный отчет, включающий его продолжительность, менеджера проекта и полный список членов команды.
  - **Назначение менеджера проекта:** Существующий менеджер может быть назначен руководителем проекта.
  - **Добавление/удаление сотрудников из проекта:** Сотрудники могут быть назначены на проект с указанием количества часов в неделю. При удалении сотрудника из проекта, соответствующее назначение также удаляется из его списка проектов.

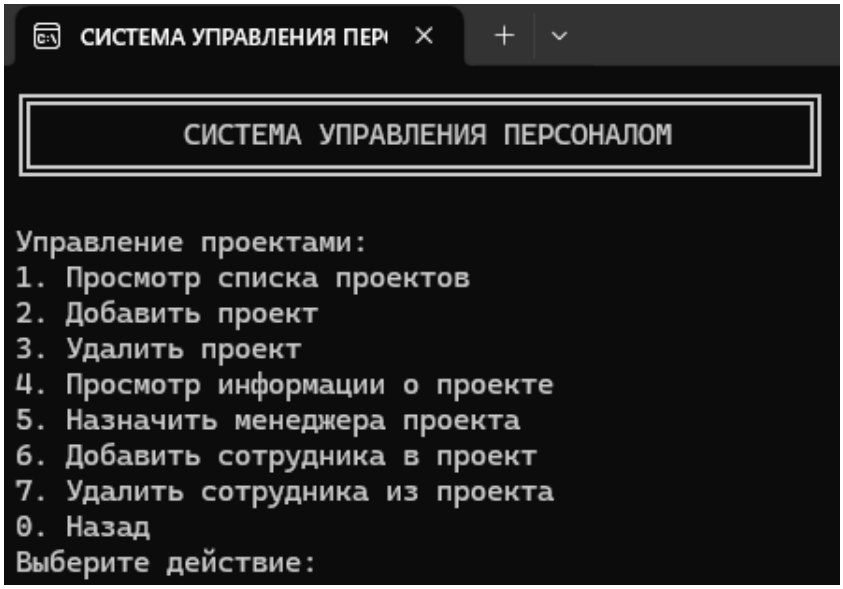

Рисунок 6.4 – управление проектами

- **Управление сотрудниками:**
  - **Просмотр списка всех сотрудников:** отображается полный список всех сотрудников компании, включая их ID, имя, тип, отдел, стаж и текущую зарплату.
  - **Добавление сотрудника:** Пользователь может добавить нового сотрудника, выбрав его тип (специалист или менеджер). Для каждого типа запрашиваются специфические данные (специализация и уровень квалификации для специалиста; уровень управления и бонус для менеджера). Система проверяет уникальность ID сотрудника.
  - **Удаление сотрудника:** Сотрудник может быть удален из системы. При этом он также удаляется из своего отдела, из списков подчиненных у менеджеров и из всех проектов, к которым он был привязан. Если удаляется CEO, его должность обнуляется.
  - **Просмотр детальной информации о сотруднике:** для выбранного сотрудника можно получить полный отчет, включающий его контактные данные, стаж, должность, отдел, базовый оклад и рассчитанную зарплату, а также специфическую информацию (специализацию/уровень квалификации для специалиста, уровень управления/список подчиненных для менеджера).
  - **Назначение сотрудника в отдел:** Сотрудник может быть переведен из одного отдела в другой.
  - **Назначение сотрудника на проект:** Сотрудник может быть назначен на один или несколько проектов.

6.2 Диаграмма последовательностей (назначение сотрудника на проект)

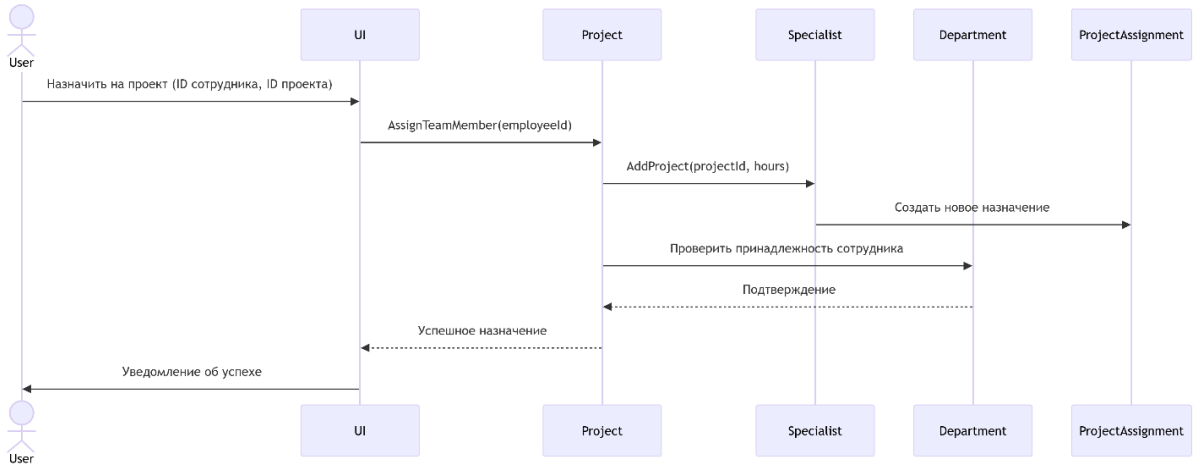

Рисунок 6.5 – диаграмма последовательностей

- **Редактирование информации о сотруднике:** Предусмотрена возможность изменения различных атрибутов сотрудника, таких как имя, базовая зарплата, должность, контактная информация, отдел, а также специфические параметры для менеджеров (уровень управления, бонус) и специалистов (специализация, уровень квалификации).

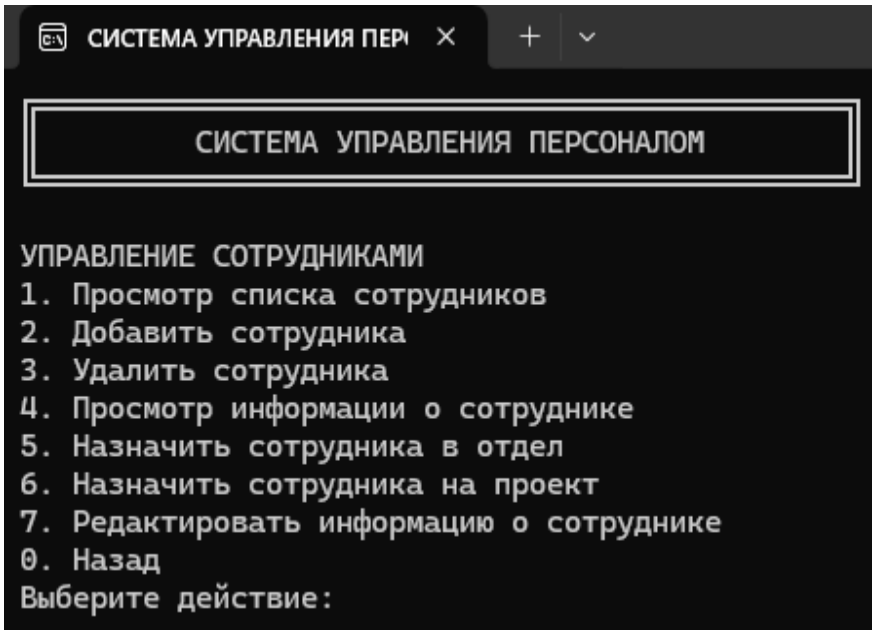

Рисунок 6.6 – управление сотрудниками

6.3 Механизм сохранения и загрузки данных

Для обеспечения персистентности данных в системе управления персоналом используется механизм сохранения и загрузки, основанный на бинарной сериализации. Этот функционал реализован в классе DataService.

- **Класс DataService:**
  - **Назначение:** Предоставляет два основных метода: SaveData() для сохранения текущего состояния объекта Company (который агрегирует все остальные данные системы) и LoadData() для его восстановления из файла.
  - **Механизм сохранения (SaveData()):** Метод принимает объект Company и опциональный путь к файлу. Если путь не указан, используется имя файла по умолчанию – "company_data.bin". Процесс сохранения включает создание FileStream для записи в файл и использование BinaryFormatter для преобразования объекта Company и всего связанного с ним графа объектов (отделов, сотрудников, проектов и т.д.) в бинарный формат. Затем эти бинарные данные записываются в файл.
  - **Механизм загрузки (LoadData()):** Метод пытается загрузить данные из файла. Перед загрузкой проверяется существование файла. Если файл найден, FileStream используется для чтения бинарных данных, а BinaryFormatter выполняет десериализацию, восстанавливая полный граф объектов Company в памяти.
  - **Обработка ошибок:** Оба метода включают try-catch блоки для обработки потенциальных исключений, таких как ошибки ввода/вывода или проблемы сериализации/десериализации, выводя соответствующие сообщения в консоль.
- **Атрибут [Serializable]:**
  - **Обоснование:** для корректной работы бинарной сериализации с BinaryFormatter все классы и структуры, экземпляры которых должны быть сохранены или загружены, должны быть помечены атрибутом **[Serializable]**. В данном проекте этим атрибутом помечены Company, Department, Employee (и его производные Manager, Specialist), Project, а также структуры ContactInfo и ProjectAssignment. Этот атрибут сигнализирует.NET Runtime, что экземпляры этих типов могут быть преобразованы в поток байтов и, при необходимости, восстановлены.

Выбор бинарной сериализации является прагматичным решением. Для небольших и средних объемов данных, она предлагает простоту реализации, позволяя легко сохранять и восстанавливать сложные графы объектов без необходимости ручного преобразования в промежуточный формат (например, XML или JSON). Это значительно сокращает объем кода, необходимый для персистентности данных. Кроме того, BinaryFormatter корректно обрабатывает ссылки на объекты, сохраняя структуру графа объектов, что критически важно для связей между сотрудниками, отделами и проектами. Однако стоит отметить, что бинарная сериализация чувствительна к изменениям в структуре классов (проблемы версионности) и создает нечитаемые для человека файлы, что затрудняет отладку вручную. Для проекта, где структура данных относительно стабильна, эти ограничения не являются

критичными. Альтернативы, такие как XML или JSON сериализация, обеспечили бы лучшую версионность и читаемость, но потребовали бы большего объема кода для ручной обработки связей между объектами или использования более сложных библиотек. Таким образом, бинарная сериализация является адекватным и эффективным выбором для данной задачи.

6.4 Импорт тестовых данных

Для удобства демонстрации и тестирования системы предусмотрен механизм импорта тестовых данных.

- **Метод ImportTestData() в UserInterface.cs:** В классе UserInterface консольного приложения реализован метод ImportTestData(). Этот метод программно создает набор предопределенных тестовых данных: компанию с CEO, несколько отделов с назначенными руководителями, различных менеджеров и специалистов, а также несколько проектов с назначенными командами. Это позволяет быстро инициализировать систему для демонстрации без необходимости ручного ввода большого объема информации, что значительно упрощает процесс проверки функциональности.

7. СТАНДАРТЫ КОДИРОВАНИЯ И ДОКУМЕНТИРОВАНИЕ

Качество программного кода определяется не только его функциональностью, но и соответствием стандартам кодирования, а также наличием адекватной документации. Соблюдение этих аспектов критически важно для читаемости, поддерживаемости и расширяемости проекта.

При кодировании старался следовать общепринятым стандартам кодирования, что значительно улучшает его читаемость и предсказуемость.

- **Соглашения по именованию (Naming Conventions):**
  - **PascalCase:** используется для именования классов (Company, Department), интерфейсов (IPayable, IReportable), перечислений (EmployeePosition, ProjectStatus), а также публичных свойств (Id, Name, Budget) и публичных методов (CalculateSalary, GenerateReport, AddEmployee). Это соответствует рекомендациям Microsoft.NET Framework Design Guidelines, что облегчает понимание кода другими разработчиками.
  - **camelCase:** применяется для именования приватных полей (_company, _dataService) и локальных переменных (choice, newName, id). Это также является стандартной практикой в C#.
  - **Осмысленные имена:** Имена переменных, методов и классов четко отражают их назначение и функциональность (например, CalculateSalary для расчета зарплаты, AddSubordinate для добавления подчиненного, GetTotalEmployees для получения общего числа сотрудников). Это делает код самодокументируемым и снижает необходимость в избыточных комментариях.
- **Читаемость кода (Readability):**
  - **Последовательная индентация:** Код имеет единообразную индентацию, что делает структуру кода легко воспринимаемой и улучшает визуальное восприятие иерархии блоков.
  - **Пустые строки:** эффективно используются для разделения логических блоков кода, таких как определения свойств, конструкторов, методов и регионов. Это повышает читаемость, разбивая код на более мелкие, управляемые секции.
  - **Регионы (#region и #endregion):** активно применяются для организации связанных членов класса (например, Свойства, Конструктор, Методы для управления командой и проектом, Реализация интерфейса IReportable). Это позволяет сворачивать и разворачивать блоки кода в интегрированной среде разработки (IDE), значительно улучшая навигацию по большим файлам и общую структуру.
  - **Четкие сигнатуры методов:** Сигнатуры методов хорошо определены, с соответствующими параметрами и возвращаемыми типами, что способствует ясности их использования.
  - **Использование StringBuilder:** для построения форматированных строк отчетов (например, в Company.GetProjectsList, Department.GenerateReport) используется

StringBuilder. Это является хорошей практикой для производительности и читаемости по сравнению с многократной конкатенацией строк.

- **Обработка ошибок и обратная связь с пользователем:** Класс UserInterface предоставляет четкие сообщения пользователю о ходе выполнения операций, ошибках и запросах ввода, что критически важно для удобства использования консольного приложения.

Соблюдение стандартов кодирования выходит за рамки простого эстетического оформления; оно напрямую влияет на качество программного обеспечения и его долгосрочную поддерживаемость. Единообразные соглашения по именованию и форматированию уменьшают когнитивную нагрузку на разработчиков, позволяя им быстрее ориентироваться в незнакомом коде. Использование регионов и осмысленных имен делает код самодокументируемым. Это критически важно для командной работы и для будущего развития проекта, так как снижает вероятность ошибок и ускоряет процесс внесения изменений. Это демонстрирует понимание важности профессиональных практик разработки.

7.1 Использование XML-комментариев для генерации документации

XML-комментарии — это специальные блоки комментариев в коде C#, начинающиеся с ///, которые используются для описания классов, интерфейсов, методов, свойств и других элементов кода.

- **Назначение XML-комментариев:** Предоставление структурированной информации о программных элементах. Они используют специальные теги, такие как
  - /// <summary> для краткого описания,
  - /// <param name="parameterName"> для описания параметров,
  - /// <returns> для описания возвращаемого значения и
  - /// <exception cref="ExceptionType"> для документирования возможных исключений.
- **Применение в коде:** Код активно использует XML-комментарии для всех публичных и защищенных членов классов и интерфейсов, а также для перечислений и структур. Это обеспечивает высокий уровень внутренней документации, делая код более понятным для других разработчиков и для самого автора в будущем.
- **Процесс генерации документации:** при компиляции проекта C# с включенной опцией /doc (что обычно является настройкой по умолчанию в проектах Visual Studio), компилятор извлекает эти XML-комментарии и генерирует отдельный XML-файл документации (например, EmployeeManagementSystem.Core.xml). Этот XML-файл содержит структурированную информацию об API кода.
- **Использование сгенерированной документации:**

- **IDE (IntelliSense):** Одним из наиболее непосредственных и широко используемых преимуществ является интеграция с интегрированными средами разработки (IDE), такими как Visual Studio. Когда разработчик пишет код, использующий элементы, задокументированные XML-комментариями, функция IntelliSense в IDE автоматически отображает информацию из тегов summary, param и returns в виде всплывающих подсказок. Это предоставляет мгновенную, контекстно-зависимую помощь, упрощая понимание того, как использовать классы и методы, без необходимости обращения к внешней документации или исходному коду.
- **Внешние инструменты:** Сгенерированный XML-файл может быть использован различными инструментами, такими как DocFX, для создания полноценной, навигационной HTML-документации, PDF-документации или документации в формате Markdown. Эти инструменты могут создавать комплексные веб-сайты, включающие иерархии классов, подробные описания членов, примеры кода (если они включены в комментарии), функциональность поиска и перекрестные ссылки.

Использование XML-комментариев и автоматизированной генерации документации является показателем зрелости процесса разработки. Вместо создания и поддержания отдельной, часто устаревающей, документации, комментарии пишутся непосредственно рядом с кодом. Это гарантирует, что документация всегда актуальна и соответствует текущей реализации. Автоматическая генерация экономит время и ресурсы, а также обеспечивает единообразие в формате документации. Это демонстрирует понимание важности не только написания функционального кода, но и создания хорошо документированного и поддерживаемого продукта, что является ключевым навыком в профессиональной разработке.

8. СТРАТЕГИЯ ТЕСТИРОВАНИЯ

Для обеспечения надежности, корректности и стабильности системы управления персоналом была разработана комплексная стратегия тестирования, включающая модульное, интеграционное и функциональное тестирование. Такой многоуровневый подход обеспечивает всестороннюю проверку, минимизирует количество дефектов в готовом продукте и повышает уверенность в его корректной работе.

8.1 Предложение стратегии тестирования

Предлагаемая стратегия тестирования охватывает различные уровни абстракции системы, начиная от отдельных компонентов и заканчивая взаимодействием со всем приложением:

- **Модульное тестирование (Unit Testing):** фокусируется на проверке отдельных классов и методов в изоляции.
- **Интеграционное тестирование (Integration Testing):** проверяет взаимодействие между различными модулями и компонентами системы.
- **Функциональное тестирование (Functional Testing):** оценивает, насколько система соответствует функциональным требованиям с точки зрения конечного пользователя.

8.2 Ключевые сценарии тестирования

Ниже представлены ключевые сценарии тестирования для каждого уровня, разработанные на основе анализа функциональности системы.

Таблица 5: Ключевые сценарии тестирования

Тип тестирования	Цель	Описание сценария	Ожидаемый результат
Модульное тестирование	Проверка отдельных компонентов (классов, методов) в изоляции.	**Employee.CalculateSalary():** Проверка расчета зарплаты для Manager и Specialist с разными входными данными (оклад, стаж, квалификация, количество подчиненных/часов по проектам). Тестирование	Корректный расчет зарплаты для всех сценариев.

		граничных условий (нулевой стаж, максимальная/минимальная квалификация).	
		Department.AddEmployee(): Проверка добавления сотрудника в отдел, установки ссылки на отдел у сотрудника, добавления в подчиненные руководителю. Тестирование на добавление null или сотрудника с существующим ID.	Сотрудник успешно добавлен, связи установлены. Исключения или false для некорректных случаев.
		Project.RemoveTeamMember(): Проверка удаления сотрудника из команды проекта, удаления назначения на проект у Specialist, обнуления ProjectManager при удалении менеджера.	Сотрудник успешно удален, связанные данные очищены.

		DataService.SaveData() / LoadData(): Проверка сохранения и загрузки объекта Company со всем его графом объектов. Тестирование обработки ошибок при отсутствии файла или повреждении данных.	Данные успешно сохраняются и восстанавливаются. Ошибки корректно обрабатываются.
Интеграционное тестирование	Проверка взаимодействия между различными модулями и компонентами системы.	**Жизненный цикл сотрудника:** Добавление отдела, назначение руководителя, добавление сотрудника в отдел, проверка подчинения, затем удаление сотрудника, проверка удаления из подчиненных.	Все связи и состояния объектов корректно обновляются на каждом шаге.
		Поток назначения на проект: Создание проекта, назначение менеджера, назначение специалиста, проверка отражения в назначениях специалиста, затем удаление специалиста из проекта, проверка удаления назначения.	Назначения и связи между проектами и сотрудниками корректно устанавливаются и удаляются.

		Полный цикл сохранения/загрузки: Создание сложной структуры компании (отделы, сотрудники, проекты), сохранение данных, перезапуск приложения, загрузка данных, проверка целостности всех связей и атрибутов.	Загруженная система полностью соответствует сохраненной.
Функциональное тестирование	Проверка соответствия системы функциональным требованиям с точки зрения конечного пользователя через консольный интерфейс.	**Полный цикл операций через UserInterface.cs:** Создание компании, добавление отделов, сотрудников (разных типов), проектов. Назначение руководителей и менеджеров. Изменение данных. Удаление сущностей. Проверка всех отчетов на корректность отображения данных и форматирования.	Все операции выполняются корректно, отчеты формируются правильно, система ведет себя предсказуемо.
		Тестирование обработки некорректного ввода: Ввод букв вместо чисел, неверных дат, попытки работы с несуществующими сущностями.	Система выдает адекватные сообщения об ошибках и остается стабильной.

		Проверка опций сохранения и загрузки данных через меню приложения.	Данные корректно сохраняются и загружаются через пользовательский интерфейс.
		Проверка опции импорта тестовых данных и последующая верификация состояния системы.	Тестовые данные успешно загружаются, и система корректно отражает их состояние.

Применение модульного, интеграционного и функционального тестирования является критически важным для создания надежного программного обеспечения.

Модульные тесты позволяют быстро выявлять ошибки в отдельных компонентах, изолируя их. Для обеспечения корректности работы ключевых компонентов системы управления персоналом был реализован набор автоматизированных модульных тестов с использованием фреймворка **xUnit**. Тестирование охватывало основные классы бизнес-логики, такие как Employee, Specialist, Manager, Company, Project, а также их методы, ответственные за критическую функциональность (например, расчет зарплаты, управление проектами и генерацию отчетов).

Преимущества **xUnit**:

- Повторяемость: Тесты можно выполнять многократно, что упрощает проверку изменений.
- Изоляция: Каждый тест фокусируется на конкретном методе или классе, исключая влияние внешних факторов.
- Отчетность: xUnit предоставляет подробные отчеты о результатах тестирования, включая трассировку стека в случае ошибок.

Интеграционные тесты проверяют, как эти компоненты работают вместе, выявляя проблемы взаимодействия. Функциональные тесты подтверждают, что вся система в целом соответствует требованиям пользователя. Такой многоуровневый подход обеспечивает всестороннюю проверку, минимизирует количество дефектов в готовом продукте и повышает уверенность в его корректной работе. Это демонстрирует понимание полного жизненного цикла разработки программного обеспечения, а не только написания кода.

ЗАКЛЮЧЕНИЕ

Выводы по выполненному проекту:

Курсовой проект "Система управления персоналом" успешно реализован на языке C# с последовательным применением ключевых принципов объектно-ориентированного программирования. В ходе работы была разработана четкая иерархическая структура классов, которая точно отражает предметную область "Служащие фирмы", включая такие сущности, как компания, отделы, проекты, а также различные типы сотрудников (менеджеры и специалисты).

Использование абстрактного класса Employee и реализация интерфейсов IPayable, IManagerial, IReportable сыграли ключевую роль в обеспечении гибкости, расширяемости и полиморфного поведения системы. Это позволило унифицировать операции по расчету зарплаты, управлению подчиненными и формированию отчетов для различных типов объектов. Применение структур (ContactInfo, ProjectAssignment) для небольших, логически связанных данных продемонстрировало понимание семантики типов в C# и стремление к оптимизации использования памяти. Перечисления (EmployeePosition, ProjectStatus, ManagementLevel) значительно повысили типобезопасность и читаемость кода.

В проектировании данной системы я применил следующие принципы объектно-ориентированного программирования:

1. **Инкапсуляция**: скрыл внутреннюю реализацию классов и предоставил доступ к функциональности через публичные методы и свойства.

2. **Наследование**: создал иерархию классов, где Manager и Specialist наследуют от базового класса Employee.

3. **Полиморфизм**: Использовал переопределение методов (например, CalculateSalary(), GenerateReport()) для изменения поведения в производных классах.

4. **Абстракция**: выделил общее поведение в абстрактный класс Employee и определил интерфейсы для различных аспектов функциональности.

5. **Принцип единственной ответственности**: Каждый класс отвечает только за одну функциональность (например, Department управляет сотрудниками отдела, Project - проектной деятельностью).

6. **Принцип открытости/закрытости**: Классы открыты для расширения, но закрыты для модификации (например, можно добавить новый тип сотрудника, наследуя от Employee).

Реализован надежный механизм сохранения и загрузки данных с использованием бинарной сериализации, что обеспечивает персистентность состояния системы между сессиями.

Код проекта соответствует базовым стандартам кодирования, включая соглашения по именованию и форматированию, и хорошо документирован с помощью XML-комментариев. Это способствует его читаемости, поддерживаемости и возможности автоматической генерации документации. Предложена комплексная стратегия тестирования, включающая модульное, интеграционное и функциональное тестирование, что подчеркивает систематический подход к обеспечению качества программного продукта.

Возможные улучшения и дальнейшее развитие:

Способность не только реализовать проект, но и критически оценить его, выявить ограничения и предложить пути развития, является ключевым показателем глубокого понимания предметной области и принципов разработки программного обеспечения. Проект может быть значительно улучшен и расширен в следующих направлениях:

- **Расширение функциональности:** Текущая система может быть дополнена функционалом для управления отпусками, больничными листами, системами поощрений и штрафов, а также механизмами учета рабочего времени.
- **Более сложная отчетность:** Реализация более сложных и настраиваемых отчетов с возможностью фильтрации, сортировки данных и экспорта в различные форматы (например, CSV, PDF, XLSX). Это может потребовать более продвинутых библиотек для генерации отчетов.
- **Улучшение пользовательского интерфейса:** Переход от текущего консольного приложения к графическому интерфейсу пользователя (GUI) с использованием технологии WinForms, или даже к веб-приложению, значительно повысит удобство использования и доступность системы.
- **Изменение механизма персистентности:** Текущая бинарная сериализация, хотя и проста в реализации, имеет ограничения, такие как чувствительность к изменениям в структуре классов (проблемы версионности) и отсутствие читаемости данных. Целесообразно рассмотреть переход к более надежным и гибким методам хранения данных, таким как база данных. Это обеспечит лучшую масштабируемость, безопасность данных и устойчивость к изменениям в модели.
- **Валидация данных:** Добавление более строгой валидации входных данных на уровне бизнес-логики, а не только на уровне пользовательского интерфейса. Это включает проверку уникальности идентификаторов, корректности форматов дат, диапазонов числовых значений и других бизнес-правил.
- **Логирование:** Внедрение системы логирования для отслеживания операций, ошибок и важных событий в приложении. Это значительно упростит отладку и мониторинг работы системы.

Эти предложения по улучшению показывают видение продукта в более широком контексте его использования и развития, подчеркивая, что проект является не просто выполненным заданием, а отправной точкой для дальнейшего обучения и профессионального роста.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

- Романенко, В. В. Объектно-ориентированное программирование: Учебное пособие [Электронный ресурс] / В. В. Романенко. — Томск: ТУСУР, 2020. — 477 с.
- Документация по C# https://learn.microsoft.com/ru-ru/dotnet/csharp/tour-of-csharp/
- Полное руководство по языку программирования C# 13 и платформе .NET 9 https://metanit.com/sharp/tutorial/
- Документация по UML --- Стандарт описания объектно-ориентированных систем. [Электронный ресурс]. URL: https://www.uml.org/

Приложение А
(обязательное)

Программная реализация классов, интерфейсов, тестов

__Company.cs__

```
using EmployeeManagementSystem.Core.Interfaces;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace EmployeeManagementSystem.Core.Models
{
    /// <summary>
    /// Класс компании
    /// </summary>
    [Serializable]
    public class Company : IReportable
    {

        #region Свойства
        // Свойства
        public string Name { get; set; }
        public List<Department> Departments { get; private set; }
        public Employee CEO { get; set; }
        public List<Project> Projects { get; private set; }

        #endregion

        #region Конструктор
        // Конструктор

        /// <summary>
        /// Конструктор
        /// </summary>
        /// <param name="name"></param>
        /// <param name="ceo"></param>
        public Company(string name, Employee ceo)
        {
            Name = name;
            CEO = ceo;
            Departments = new List<Department>();
            Projects = new List<Project>();
        }

        #endregion

        #region Методы для управления отделами
        // Методы для управления отделами

        /// <summary>
        /// Добавление отдела
        /// </summary>
```

```
/// <param name="department"></param>
public void AddDepartment(Department department)
{
    Departments.Add(department);
}


/// <summary>
/// Удаление отдела
/// </summary>
/// <param name="department"></param>
/// <returns></returns>
public bool RemoveDepartment(Department department)
{
    // нельзя удалить отдел в котором есть сотрудники
    if (department.Employees.Count > 0)
    {
        return false;
    }
    return Departments.Remove(department);
}

#endregion

#region Методы для работы с проектами

// Методы для работы с проектами

/// <summary>
/// Добавление проекта
/// </summary>
/// <param name="project"></param>
public void AddProject(Project project)
{
    Projects.Add(project);
}


/// <summary>
/// Удаление проекта
/// </summary>
/// <param name="projectId"></param>
/// <returns></returns>
public bool RemoveProject(int projectId)
{
    for (int i = 0; i < Projects.Count; i++)
    {
        if (Projects[i].Id == projectId)
        {
            Projects.RemoveAt(i);
            return true;
        }
    }
    return false;
}

public bool RemoveProject(Project project)
```

```
        {
            // TODO RemoveProject Assignments
            return Projects.Remove(project);
        }

        public string GetProjectsList(bool showOrderNumber = false)
        {
            StringBuilder sb = new StringBuilder();

            string header = showOrderNumber
               ? $"\n{"№",-5}{"Название",-40}{"ID",-7}{"Статус",-14}{"Бюджет",-
16}{"Членов команды",-18}{"Менеджер",-24}"
                : $"\n{"Название",-40}{"ID",-7}{"Статус",-14}{"Бюджет",-16}{"Членов
команды",-18}{"Менеджер",-24}";

            sb.AppendLine(header);
            sb.AppendLine(new string('-', showOrderNumber ? 117 : 112));


            //string header = showOrderNumber
            //   ? $"\n{"№",-5}{"Название",-40}{"ID",-7}{"Статус",-20}{"Бюджет",-
20}{"Членов команды",-20}\n" + new string('-', showOrderNumber ? 107 : 102) + "\n"
            //   : $"\n{"Название",-40}{"ID",-7}{"Статус",-20}{"Бюджет",-20}{"Членов
команды",-20}\n" + new string('-', 102) + "\n";

            for (int i = 0; i < Projects.Count; i++)
            {
                string projectsReport = $"{Projects[i].Name,-40}{Projects[i].Id,-
7}{Projects[i].Status,-14}{Projects[i].Budget,-20:c2}{Projects[i].Team.Count,-
14}{Projects[i].ProjectManager?.Name,-24}";

                if (showOrderNumber)
                {
                    projectsReport = $"{i + 1,-5}" + projectsReport;
                }
                sb.AppendLine(projectsReport);

            }
            return sb.ToString();
        }

        #endregion

        #region Методы для работы с сотрудниками

        /// <summary>
        /// Получение общего количества сотрудников
        /// </summary>
        /// <returns>Число сотрудников</returns>
        public int GetTotalEmployees()
        {
            int totalEmployees = 0;

            for (int i = 0; i < Departments.Count; i++)
            {
                totalEmployees += Departments[i].Employees.Count;
            }
```

```
            // Учитываем CEO отдельно, так как он может не принадлежать ни к одному
департаменту
            // Добавляем CEO, если он не включен в какой-либо отдел

            if (CEO != null && CEO.Department == null)
            //if (CEO != null && CEO.DepartmentId == 0)
            {
                totalEmployees++;
            }

            return totalEmployees;
        }


        private List<Employee> GetAllEmployeesList()
        {
            List<Employee> allEmployees = new List<Employee>();

            // Добавляем CEO, если он существует (он может быть не сотрудником
компании)
            if (CEO != null)
            {
                allEmployees.Add(CEO);
            }

            // Добавляем сотрудников из всех отделов
            for (int i = 0; i < Departments.Count; i++)
            {
                for (int j = 0; j < Departments[i].Employees.Count; j++)
                {
                    // Проверяем, что сотрудник не является CEO
                    if (Departments[i].Employees[j].Id != CEO?.Id)
                    {
                        allEmployees.Add(Departments[i].Employees[j]);
                    }
                }
            }

            return allEmployees;
        }


        /// <summary>
        /// Получение сотрудника по имени
        /// </summary>
        /// <param name="name"></param>
        /// <returns>Сотрудник</returns>
        public Employee GetEmployeeByName(string name)
        {
            var employees = GetAllEmployeesList();
            foreach (var employee in employees)
            {
                if (employee.Name.ToLower() == name.ToLower())
                {
                    return employee;
                }
```

```
            }
            return null;
        }



        public IEnumerable<Employee> GetAllEmployees()
        {
            return GetAllEmployeesList();
        }

        #endregion

        #region Реализация интерфейса IReportable

        // Реализация интерфейса IReportable

        /// <summary>
        /// Формирование отчета
        /// </summary>
        /// <param name="format"></param>
        /// <returns></returns>
        public string GenerateReport(string format)
        {
            if (format.ToLower() == "short")
            {
                return $"Компания: {Name}, Отделов: {Departments.Count}, Проектов:
{Projects.Count}, Сотрудников: {GetTotalEmployees()}";
            }

            string departmentsReport = "Отделы:\n";
            for (int i = 0; i < Departments.Count; i++)
            {
                departmentsReport += $"  - {Departments[i].Name} (ID:
{Departments[i].Id})\n";
            }

            string projectsReport = "Проекты:\n";
            for (int i = 0; i < Projects.Count; i++)
            {
                projectsReport += $"  - {Projects[i].Name} (ID: {Projects[i].Id},
Status: {Projects[i].Status})\n";
            }

            return $"Компания: {Name}\nCEO: {(CEO != null ? CEO.Name : "Не
назначен")}\n" +
                   $"Всего сотрудников: {GetTotalEmployees()}\n" +
                   $"Отделов: {Departments.Count}\n{departmentsReport}\n" +
                   $"Проектов: {Projects.Count}\n{projectsReport}";
        }

        #endregion

        #region Получение списка отделов

        /// <summary>
        /// Получение списка отделов
```

```
        /// </summary>
        /// <param name="showOrderNumber"></param>
        /// <returns>Строка содержащая список отделов компании</returns>
        public string GetDepartmentsList(bool showOrderNumber = false)
        {
            StringBuilder sb = new StringBuilder();

            sb.AppendLine("СПИСОК ОТДЕЛОВ:\n----------------\n");

            string header = showOrderNumber
                ? $"{"№",-3}{"ID",-7}{"Название",-30}{"Сотрудников",-
22}{"Руководитель",-30}{"Бюджет",-15}{"Остаток бюджета",-16}"
                : $"{"ID",-7}{"Название",-30}{"Сотрудников",-22}{"Руководитель",-
30}{"Бюджет",-15}{"Остаток бюджета",-16}";

            sb.AppendLine(header);
            sb.AppendLine(new string('-', showOrderNumber ? 123 : 120));

            for (int i = 0; i < Departments.Count; i++)
            {
                string departmentInfo = Departments[i].GenerateReport("short");
                if (showOrderNumber)
                {
                    departmentInfo = $"{i + 1,-3}" + departmentInfo;
                }
                sb.AppendLine(departmentInfo);
            }

            return sb.ToString();
        }

        #endregion

        #region Получение списка всех сотрудников
        /// <summary>
        /// Получение списка всех сотрудников
        /// </summary>
        /// <param name="showOrderNumber"></param>
        /// <returns>Строка содержащая список всех сотрудников компании</returns>
        public string GetAllEmployeesListString(bool showOrderNumber = false)
        {
            StringBuilder sb = new StringBuilder();

            //sb.AppendLine("СПИСОК ВСЕХ СОТРУДНИКОВ\n");

            string header = showOrderNumber
                ? $"{"№",-5}{"ID",-5}{"Имя",-30}{"Тип сотрудника",-20}{"Отдел",-
30}{"Стаж",-10}{"Ставка",-10}"
                : $"{"ID",-5}{"Имя",-30}{"Тип сотрудника",-20}{"Отдел",-30}{"Стаж",-
10}{"Ставка",-10}";

            sb.AppendLine(header);
            sb.AppendLine(new string('-', showOrderNumber ? 110 : 105));

            var employees = GetAllEmployeesList();
            for (int i = 0; i < employees.Count; i++)
            {
```

```
                var employee = employees[i];
                string departmentName = employee.Department?.Name ?? "Нет отдела";
                string employeeInfo = $"{employee.Id,-5}{employee.Name,-
30}{employee.GetType().Name,-20}{departmentName,-
30}{employee.CalculateExperienceString(),-10}{employee.CalculateSalary(),-10:C2}";

                if (showOrderNumber)
                {
                    employeeInfo = $"{i + 1,-5}" + employeeInfo;
                }

                sb.AppendLine(employeeInfo);
            }

            return sb.ToString();
        }
        #endregion

    }

}
```

Departmen.cs

```
using EmployeeManagementSystem.Core.Interfaces;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace EmployeeManagementSystem.Core.Models
{
    /// <summary>
    /// Класс отдел
    /// </summary>
    [Serializable]
    public class Department : IReportable
    {
        #region Свойства
        // Свойства
        public int Id { get; set; }
        public string Name { get; set; }
        public Manager Head { get; private set; }
        public List<Employee> Employees { get; private set; }
        public decimal Budget { get; set; }

        #endregion

        #region Конструктор
        // Конструктор

        /// <summary>
        /// Конструктор
        /// </summary>
        /// <param name="id"></param>
        /// <param name="name"></param>
        /// <param name="head"></param>
        /// <param name="budget"></param>
        public Department(int id, string name, Manager head, decimal budget)
        {
            Id = id;
            Name = name;
            Head = head;
            Budget = budget;
            Employees = new List<Employee>();

            if (head != null)
            {
                SetHead(head);
                //Employees.Add(head);
            }
        }

        #endregion

        #region Метод для назначения главы отдела
        // Метод для назначения главы отдела
```

```
        /// <summary>
        /// Метод для назначения главы отдела
        /// </summary>
        /// <param name="newHead"></param>
        /// <exception cref="ArgumentNullException"></exception>
        public void SetHead(Manager newHead)
        {
            if (newHead == null)
                throw new ArgumentNullException(nameof(newHead));

            // Удаляем текущего главу из списка сотрудников (если он есть)
            if (Head != null)
            {
                Employees.Remove(Head);
            }

            // Назначаем нового главу
            Head = newHead;
            if (!Employees.Contains(newHead))
            {
                Employees.Add(newHead);
                newHead.Department = this;  // ссылка на этот отдел
            }

            // Все сотрудники отдела становятся подчинёнными нового главы
            foreach (var employee in Employees)
            {
                if (employee.Id != newHead.Id) // Проверяем, чтобы глава не был
подчинённым самому себе
                {
                    newHead.AddSubordinate(employee);
                }
            }
        }

        #endregion

        #region Методы для управления сотрудниками
        // Методы для управления сотрудниками

        /// <summary>
        /// Добавление сотрудника
        /// </summary>
        /// <param name="employee"></param>
        /// <exception cref="ArgumentNullException"></exception>
        public void AddEmployee(Employee employee)
        {
            if (employee == null)
                throw new ArgumentNullException(nameof(employee));

            //if (!Employees.Exists(e => e.Id == employee.Id))
            //Проверяем, чтобы сотрудник с таким ID не был в отделе
            foreach (var e in Employees)
            {
                if (e.Id == employee.Id)
                {
```

```
                    throw new ArgumentException("Сотрудник с таким ID уже существует
в отделе.");
                }
            }

            if (!Employees.Contains(employee))
            {
                Employees.Add(employee);
            }

            employee.Department = this;  // ссылка на этот отдел

            // Если у отдела есть глава, добавляем сотрудника в подчинённые главы

            if (Head != null && employee.Id != Head.Id) // Проверяем, чтобы глава не
был подчинённым самому себе
            {
                Head.AddSubordinate(employee);
            }
        }


        /// <summary>
        /// Удаление сотрудника
        /// </summary>
        /// <param name="employee"></param>
        /// <returns>Успешно?</returns>
        public bool RemoveEmployee(Employee employee)
        {
            // Если у отдела есть глава, удаляем сотрудника из подчинённых главы
            Head?.RemoveSubordinate(employee);

            return Employees.Remove(employee);
        }

        #endregion

        #region Методы для работы с бюджетом

        // Методы для работы с бюджетом

        /// <summary>
        /// Расчет общей зарплаты отдела
        /// </summary>
        /// <returns>Размер общей зарплаты</returns>
        private decimal CalculateTotalSalary()
        {
            decimal totalSalary = 0;

            for (int i = 0; i < Employees.Count; i++)
            {
                totalSalary += Employees[i].CalculateSalary();
            }

            return totalSalary;
        }
```

```
        /// <summary>
        /// Расчет остатка бюджета
        /// </summary>
        /// <returns>Остаток бюджета</returns>
        private decimal GetBudgetRemainder()
        {
            return Budget - CalculateTotalSalary();
        }

        #endregion

        #region Реализация интерфейса IReportable
        // Реализация интерфейса IReportable

        /// <summary>
        /// Генерация отчета
        /// </summary>
        /// <param name="format"></param>
        /// <returns>Строка отчета</returns>
        public string GenerateReport(string format)
        {
            if (format.ToLower() == "short")
            {
                return $"{Id,-5}{Name,-40}{Employees.Count,-12}{Head?.Name,-
30}{Budget,-18:C2}{GetBudgetRemainder(),-18:C2}";
                //return  $"{Id,-5}{Name,-40}{Employees.Count,-12}{Head?.Name,-
30}{Budget, -20:C2}{GetBudgetRemainder(),-20:C2}";
                //return $"Department: {Name} (ID: {Id}), Employees:
{Employees.Count}, Budget: {Budget:C2}";
            }

            // Если формат не "short", генерируем полный отчет
            string employeesReport = "Сотрудники:\n";
            for (int i = 0; i < Employees.Count; i++)
            {
                employeesReport += $"  - {Employees[i].Name+";",-30}" +
                    $"Должность: {Employees[i].GetType().Name+ "
("+Employees[i].Position + ");",-40}" +
                    $"Зарплата: {Employees[i].CalculateSalary(),-20:C2}\n";
            }


            return $"Отдел ID: {Id}\nНазвание: {Name}\n" +
                   $"Руководитель: {(Head != null ? Head.Name : "Не назначен")}\n" +
                   $"Бюджет: {Budget:C2}\nСумма зарплат:
{CalculateTotalSalary():C2}\n" +
                   $"Остаток бюджета: {GetBudgetRemainder():C2}\n" +
                   $"Количество сотрудников: {Employees.Count}\n{employeesReport}";
        }

        // Метод получения списка специалистов по специализации

        /// <summary>
        /// Получение списка специалистов по специализации
        /// </summary>
        /// <param name="specialization"></param>
        /// <returns>Список специалистов</returns>
```

```
        public IEnumerable<Specialist> GetSpecialistsBySpecialization(string
specialization)
        {
            List<Specialist> specialists = new List<Specialist>();

            for (int i = 0; i < Employees.Count; i++)
            {
                if (Employees[i] is Specialist specialist &&
                    specialist.Specialization.Equals(specialization,
StringComparison.OrdinalIgnoreCase))
                {
                    specialists.Add(specialist);
                }
            }

            return specialists;
        }
        /// <summary>
        /// Получение списка всех сотрудников
        /// </summary>
        /// <param name="showOrderNumber"></param>
        /// <returns>Строка содержащая список сотрудников</returns>
        public string GetAllEmployeesList(bool showOrderNumber = false)
        {
            StringBuilder sb = new StringBuilder();

            sb.AppendLine("СПИСОК ВСЕХ СОТРУДНИКОВ\n");

            string header = showOrderNumber
                ? $"{"№",-3}{"ID",-5}{"Имя",-30}{"Тип сотрудника",-20}{"Отдел",-
30}{"Стаж",-10}{"Ставка",-10}"
                : $"{"ID",-5}{"Имя",-30}{"Тип сотрудника",-20}{"Отдел",-30}{"Стаж",-
10}{"Ставка",-10}";

            sb.AppendLine(header);
            sb.AppendLine(new string('-', showOrderNumber ? 108 : 105));

            for (int i = 0; i < Employees.Count; i++)
            {
                var employee = Employees[i];
                string departmentName = employee.Department?.Name ?? "Нет отдела";
                string employeeInfo = $"{employee.Id,-5}{employee.Name,-
30}{employee.GetType().Name,-20}{departmentName,-
30}{employee.CalculateExperienceString(),-10}{employee.CalculateSalary(),-10}";

                if (showOrderNumber)
                {
                    employeeInfo = $"{i + 1,-3}" + employeeInfo;
                }

                sb.AppendLine(employeeInfo);
            }
            return sb.ToString();
        }
        #endregion
    }
```

Employee.cs

```
using EmployeeManagementSystem.Core.Enums;
using EmployeeManagementSystem.Core.Interfaces;
using EmployeeManagementSystem.Core.Models.Structs;
using System;
using System.Runtime.InteropServices.ComTypes;

namespace EmployeeManagementSystem.Core.Models
{
    /// <summary>
    /// Базовый абстрактный класс сотрудника
    /// </summary>
    [Serializable]
    public abstract class Employee : IPayable, IReportable
    {
        #region Свойства
        public int Id { get; set; }
        public string Name { get; set; }
        private DateTime BirthDate { get; set; }
        private DateTime HireDate { get; set; }
        public decimal BaseSalary { get; set; }
        public EmployeePosition Position { get; private set; }
        public ContactInfo ContactInfo { private get; set; }
        public Department Department { get; set; }

        #endregion

        #region Конструктор
        /// <summary>
        /// Конструктор
        /// </summary>
        /// <param name="id"></param>
        /// <param name="name"></param>
        /// <param name="birthDate"></param>
        /// <param name="hireDate"></param>
        /// <param name="baseSalary"></param>
        /// <param name="position"></param>
        /// <param name="contactInfo"></param>
        /// <param name="department"></param>
        protected Employee(int id, string name, DateTime birthDate, DateTime
hireDate, decimal baseSalary,
                           EmployeePosition position, ContactInfo contactInfo,
Department department)
        {
            Id = id;
            Name = name;
            BirthDate = birthDate;
            HireDate = hireDate;
            BaseSalary = baseSalary;
            Position = position;
            ContactInfo = contactInfo;
            Department = department;
            Department?.AddEmployee(this);
        }
```

```
        #endregion

        #region Методы интерфейса IPayable
        // Методы интерфейса IPayable

        /// <summary>
        /// Вычисление зарплаты
        /// </summary>
        /// <returns></returns>
        public abstract decimal CalculateSalary();

        #endregion

        #region Методы интерфейса IReportable

        // Методы интерфейса IReportable
        public virtual string GenerateReport(string format)
        {
            return format.ToLower() == "short"
                ? $"ID: {Id}, Имя: {Name}, Должность: {Position}"
                : $"ID: {Id}\nИмя: {Name}\nДата рождения:
{BirthDate.ToShortDateString()}\n" +
                  $"Дата найма: {HireDate.ToShortDateString()} \nСтаж работы:
{CalculateExperienceString()}" +
                  $"\nДолжность: {Position}\nОтдел: {Department?.Name}" +
                  $"\nКонтактная информация: {ContactInfo}\nОклад: {BaseSalary:C2}\n"
+
                  $"Вознаграждение: {CalculateSalary():C2}";
        }

        #endregion

        #region Дополнительные методы
        // Дополнительные методы

        /// <summary>
        /// Вычисление опыта работы
        /// </summary>
        /// <returns>опыт работы в месяцах</returns>
        public int CalculateExperience()
        {
            //return (int)((DateTime.Now - HireDate).TotalDays / 365.25);
//            return (DateTime.Now.Year - HireDate.Year) * 12 + (DateTime.Now.Month -
HireDate.Month);
            int months = (DateTime.Now.Year - HireDate.Year) * 12;
            months += DateTime.Now.Month - HireDate.Month;
            if (DateTime.Now.Day < HireDate.Day)
            {
                months--;
            }
            return months;
        }


        // Вычислить опыт работы в годах и месяцах

        /// <summary>
```

```
        /// Вычисление опыта работы в годах и месяцах
        /// </summary>
        /// <returns>опыт работы в годах и месяцах</returns>
        public string CalculateExperienceString()
        {
            int years = DateTime.Now.Year - HireDate.Year;
            int months = DateTime.Now.Month - HireDate.Month;
            if (months < 0)
            {
                years--;
                months += 12;
            }
            if (DateTime.Now.Day < HireDate.Day)
            {
                months--;
                if (months < 0)
                {
                    years--;
                    months += 12;
                }
            }
            return $"{years}{GetYearWord(years)},{months}м";
        }


        /// <summary>
        /// Назначение на должность
        /// </summary>
        /// <param name="newPosition"></param>
        public void ChangePosition(EmployeePosition newPosition)
        {
            Position = newPosition;
        }

        public void DisplayInfo()
        {
            Console.WriteLine(GenerateReport("full"));
        }


        /// <summary>
        /// Возвращает слово "л" или "г" в зависимости от количества лет (правильно
склоняет слово "год" в зависимости от числа лет:
        /// </summary>
        /// <param name="years"></param>
        /// <returns>Строка "л" или "г"</returns>
        private string GetYearWord(int years)
        {
            if (years % 100 >= 11 && years % 100 <= 19)
                return "л";

            switch (years % 10)
            {
                case 1:
                    return "г";
                case 2:
                case 3:
```

```
                case 4:
                    return "г";
                default:
                    return "л";
            }
        }

        // TO-DO set Employ eeName


        #endregion
    }
}
```

Manager.cs

```
using EmployeeManagementSystem.Core.Enums;
using EmployeeManagementSystem.Core.Interfaces;
using EmployeeManagementSystem.Core.Models.Structs;
using System;
using System.Collections.Generic;

namespace EmployeeManagementSystem.Core.Models
{
    /// <summary>
    /// Класс менеджер наследует от класса Employee
    /// </summary>
    [Serializable]
    public class Manager : Employee, IManagerial
    {
        #region Дополнительные свойства
        // Дополнительные свойства
        public List<Employee> Subordinates { get; private set; }
        public ManagementLevel ManagementLevel { get; set; }
        public decimal ManagementBonus { get; set; }
        #endregion

        #region Конструктор
        // Конструктор

        /// <summary>
        /// Инициализация менеджера
        /// </summary>
        /// <param name="id"></param>
        /// <param name="name"></param>
        /// <param name="birthDate"></param>
        /// <param name="hireDate"></param>
        /// <param name="baseSalary"></param>
        /// <param name="position"></param>
        /// <param name="contactInfo"></param>
        /// <param name="department"></param>
        /// <param name="managementLevel"></param>
        /// <param name="managementBonus"></param>
        public Manager(int id, string name, DateTime birthDate, DateTime hireDate,
decimal baseSalary,
                       EmployeePosition position, ContactInfo contactInfo, Department
department,
                       ManagementLevel managementLevel, decimal managementBonus)
            : base(id, name, birthDate, hireDate, baseSalary, position, contactInfo,
department)
        {
            ManagementLevel = managementLevel;
            ManagementBonus = managementBonus;
            Subordinates = new List<Employee>();
        }

        #endregion

        #region Переопределение метода из IPayable
        // Переопределение метода из IPayable
```

```
        /// <summary>
        /// Вычисление зарплаты
        /// </summary>
        /// <returns>Зарплата с учётом бонусов</returns>
        public override decimal CalculateSalary()
        {
            decimal experienceBonus = BaseSalary * (CalculateExperience() / 100m);
            decimal subordinatesBonus = ManagementBonus * Subordinates.Count;

            return BaseSalary + experienceBonus + subordinatesBonus;
        }

        #endregion

        #region Реализация методов интерфейса IManagerial
        // Реализация методов интерфейса IManagerial

        /// <summary>
        /// Добавление подчиненного
        /// </summary>
        /// <param name="employee"></param>
        /// <returns>Статус добавления</returns>
        public bool AddSubordinate(Employee employee)
        {
            if (employee == null)
                throw new ArgumentNullException(nameof(employee));

            if (Id == employee.Id) // "Менеджер не может быть подчинённым самому себе
            {
                return false;
            }

            Subordinates.Add(employee);
            return true;
        }


        /// <summary>
        /// Удаление подчиненного
        /// </summary>
        /// <param name="employeeId"></param>
        /// <returns>Статус удаления</returns>
        public bool RemoveSubordinate(Employee employee)
        {
            for (int i = 0; i < Subordinates.Count; i++)
            {
                if (Subordinates[i] == employee)
                {
//                    Subordinates.RemoveAt(i);
                    Subordinates.Remove(employee);
                    return true;
                }
            }
            return false;
        }
```

```
        /// <summary>
        /// Получение подчиненных
        /// </summary>
        /// <returns></returns>
        public IEnumerable<Employee> GetSubordinates()
        {
            return Subordinates;
        }

        #endregion

        #region Переопределение методов из Employee
        // Переопределение методов из Employee

        /// <summary>
        /// Генерация отчета
        /// </summary>
        /// <param name="format"></param>
        /// <returns>Строка отчета</returns>
        public override string GenerateReport(string format)
        {
            string baseReport = base.GenerateReport(format);

            if (format.ToLower() == "short")
            {
                return $"{baseReport}, Уровень менеджмента: {ManagementLevel},
Подчиненных: {Subordinates.Count}";
            }

            string subordinatesReport = "Подчиненные:\n";
            for (int i = 0; i < Subordinates.Count; i++)
            {
                subordinatesReport += $"  - {Subordinates[i].Name} (ID:
{Subordinates[i].Id})\n";
            }

            return $"{baseReport}\nУровень менеджмента:
{ManagementLevel}\n{subordinatesReport}";
        }

        #endregion

        #region Дополнительные методы
        // Дополнительные методы

        /// <summary>
        /// Вычисление зарплаты команды
        /// </summary>
        /// <returns>Зарплата команды</returns>
        public decimal CalculateTeamSalary()
        {
            decimal totalSalary = CalculateSalary();

            for (int i = 0; i < Subordinates.Count; i++)
            {
                totalSalary += Subordinates[i].CalculateSalary();
```

```
            }

            return totalSalary;
        }
        #endregion

    }
}
```

Project.cs

```
using EmployeeManagementSystem.Core.Enums;
using EmployeeManagementSystem.Core.Interfaces;
using System;
using System.Collections.Generic;

namespace EmployeeManagementSystem.Core.Models
{
    [Serializable]
    public class Project : IReportable
    {
        #region Свойства
        // Свойства
        public int Id { get; set; }
        public string Name { get; set; }
        public DateTime StartDate { get; set; }
        public DateTime EndDate { get; set; }
        public decimal Budget { get; set; }
        public ProjectStatus? Status { get; set; }
        public Manager ProjectManager { get; set; }
        public List<Employee> Team { get; private set; }

        #endregion

        #region Конструктор
        // Конструктор
        public Project(int id, string name, DateTime startDate, DateTime endDate,
decimal budget,
                       ProjectStatus? status, Manager projectManager)
        {
            Id = id;
            Name = name;
            StartDate = startDate;
            EndDate = endDate;
            Budget = budget;
            Status = status;
            ProjectManager = projectManager;
            Team = new List<Employee>();

            // Добавляем менеджера проекта в команду
            if (projectManager != null)
            {
                Team.Add(projectManager);
            }
        }

        #endregion

        #region Методы для управления командой и проектом
        // Методы для управления командой
        public void AssignTeamMember(Employee employee, decimal hoursPerWeek)
        {
            Team.Add(employee);
```

```
    // Если сотрудник - специалист, добавляем проект в его список
    if (employee is Specialist specialist)
    {
        specialist.AddProject(Id, this, hoursPerWeek);
    }
}

public bool RemoveTeamMember(int employeeId)
{
    for (int i = 0; i < Team.Count; i++)
    {
        if (Team[i].Id == employeeId)
        {
            // Если удаляем специалиста, то удаляем и назначение на проект
            if (Team[i] is Specialist specialist)
            {
                specialist.RemoveProject(Id);
            }

            if (Team[i] == ProjectManager)
            {
                ProjectManager = null;
            }

            Team.RemoveAt(i);
            return true;
        }
    }
    return false;
}


// Методы для управления проектом
public void UpdateStatus(ProjectStatus newStatus)
{
    Status = newStatus;
    if (newStatus == ProjectStatus.Completed)
    {
        EndDate = DateTime.Now;
    }
}

public int GetDuration()
{
    if (Status == ProjectStatus.Completed)
    {
        return (EndDate - StartDate).Days;
    }
    return (DateTime.Now - StartDate).Days;
}

#endregion

#region Реализация интерфейса IReportable
// Реализация интерфейса IReportable
public string GenerateReport(string format)
{
```

```
            if (format.ToLower() == "short")
            {
                return $"Проект: {Name} (ID: {Id}), Статус: {Status}, Сотрудников в
команде: {Team.Count}";
            }

            string teamReport = "Члены команды:\n";
            for (int i = 0; i < Team.Count; i++)
            {
                teamReport += $"  - {Team[i].Name} (ID: {Team[i].Id})\n";
            }

            return $"Проект ID: {Id}\nName: {Name}\n" +
                   $"Дата начала: {StartDate.ToShortDateString()}\nДата окончания:
{EndDate.ToShortDateString()}\n" +
                   $"Статус: {Status}\nБюджет: {Budget:C2}\n" +
                   $"Менеджер: {(ProjectManager != null ? ProjectManager.Name : "Не
назначен")}\n" +
                   $"Продолжительность: {GetDuration()} дней\n" +
                   $"Сотрудников в команде: {Team.Count}\n{teamReport}";
        }

        #endregion
    }
                                        }
```

Spelialist.cs

```
using EmployeeManagementSystem.Core.Enums;
using EmployeeManagementSystem.Core.Models.Structs;
using System;
using System.Collections.Generic;

namespace EmployeeManagementSystem.Core.Models
{
    /// <summary>
    /// Класс специалист наследует от класса Employee
    /// </summary>
    [Serializable]
    public class Specialist : Employee
    {
        #region Свойства
        // Дополнительные свойства
        public string Specialization { get; set; }
        public int QualificationLevel { get; set; }
        public List<ProjectAssignment> ProjectAssignments { get; private set; }

        #endregion

        #region Контруктор
        // Конструктор
        /// <summary>
        /// Инициализация специалиста
        /// </summary>
        /// <param name="id"></param>
        /// <param name="name"></param>
        /// <param name="birthDate"></param>
        /// <param name="hireDate"></param>
        /// <param name="baseSalary"></param>
        /// <param name="position"></param>
        /// <param name="contactInfo"></param>
        /// <param name="department"></param>
        /// <param name="specialization"></param>
        /// <param name="qualificationLevel"></param>
        public Specialist(int id, string name, DateTime birthDate, DateTime hireDate,
decimal baseSalary,
                          EmployeePosition position, ContactInfo contactInfo,
Department department,
                          string specialization, int qualificationLevel)
            : base(id, name, birthDate, hireDate, baseSalary, position, contactInfo,
department)
        {
            Specialization = specialization;
            QualificationLevel = qualificationLevel;
            ProjectAssignments = new List<ProjectAssignment>();
        }

        #endregion

        #region Переопределение метода из IPayable
        // Переопределение метода из IPayable
```

```
        /// <summary>
        /// Вычисление зарплаты
        /// </summary>
        /// <returns></returns>
        public override decimal CalculateSalary()
        {
            decimal experienceBonus = BaseSalary * (CalculateExperience() / 200m);
            decimal qualificationBonus = BaseSalary * (QualificationLevel / 10m);
            decimal projectsBonus = 0;

            for (int i = 0; i < ProjectAssignments.Count; i++)
            {
                projectsBonus += ProjectAssignments[i].HoursPerWeek * 0.2m;
            }

            return BaseSalary + experienceBonus + qualificationBonus + projectsBonus;
        }
        #endregion

        #region Методы для управления проектами
        // Методы для управления проектами

        /// <summary>
        /// Добавление проекта
        /// </summary>
        /// <param name="projectId"></param>
        /// <param name="hoursPerWeek"></param>
        public void AddProject(int projectId, Project project, decimal hoursPerWeek)
        {
            //if (!ProjectAssignments.Exists(p => p.ProjectId == projectId)
            ProjectAssignment assignment = new ProjectAssignment(
                projectId,
                project,
                Id,
                DateTime.Now,
                hoursPerWeek
            );

            ProjectAssignments.Add(assignment);
        }
        /// <summary>
        /// Удаление проекта
        /// </summary>
        /// <param name="projectId"></param>
        /// <returns></returns>
        public bool RemoveProject(int projectId)
        {
            for (int i = 0; i < ProjectAssignments.Count; i++)
            {
                if (ProjectAssignments[i].ProjectId == projectId)
                {
                    ProjectAssignments.RemoveAt(i);
                    return true;
                }
            }
            return false;
        }
```

```
        #endregion

        #region Метод повышения квалификации
        // Метод повышения квалификации

        /// <summary>
        /// Повышение квалификации
        /// </summary>
        /// <param name="newLevel"></param>
        public void UpdateQualification(int newLevel)
        {
            if (newLevel >= 1 && newLevel <= 10)
            {
                QualificationLevel = newLevel;
            }
        }
        #endregion

        #region Переопределение методов из Employee
        // Переопределение методов из Employee

        /// <summary>
        /// Генерация отчета
        /// </summary>
        /// <param name="format"></param>
        /// <returns>Строка отчета</returns>
        public override string GenerateReport(string format)
        {
            string baseReport = base.GenerateReport(format);

            if (format.ToLower() == "short")
            {
                return $"{baseReport}, Специализация: {Specialization}, Проекты:
{ProjectAssignments.Count}";
            }

            string projectsReport = "Задачи по проектам:\n";
            for (int i = 0; i < ProjectAssignments.Count; i++)
            {
                //projectsReport += $"  - Проект ID:
{ProjectAssignments[i].ProjectId}, " +
                //                  $"Часов в неделю:
{ProjectAssignments[i].HoursPerWeek}\n";
                projectsReport += $"  - Проект: {ProjectAssignments[i].Project.Name},
" +
                                  $"Часов в неделю:
{ProjectAssignments[i].HoursPerWeek}\n";

            }
            return $"{baseReport}\nСпециализация: {Specialization}\n" +
                   $"Квалификационный уровень:
{QualificationLevel}\n{projectsReport}";
        }

        #endregion
    }
}
```

IManagerial.cs

```
using EmployeeManagementSystem.Core.Models;
using System.Collections.Generic;

namespace EmployeeManagementSystem.Core.Interfaces
{
    /// <summary>
    /// Интерфейс для объектов с управленческими функциями
    /// </summary>
    public interface IManagerial
    {
        /// <summary>
        /// Добавление подчиненного
        /// </summary>
        /// <param name="employee">Сотрудник</param>
        bool AddSubordinate(Employee employee);

        /// <summary>
        /// Удаление подчиненного
        /// </summary>
        /// <param name="employeeId">ID сотрудника</param>
        /// <returns>Признак успешного удаления</returns>
        bool RemoveSubordinate(Employee employee);
        // bool RemoveSubordinate(int employeeId);

        /// <summary>
        /// Получение списка подчиненных
        /// </summary>
        /// <returns>Коллекция подчиненных</returns>
        IEnumerable<Employee> GetSubordinates();
    }
}
```

IPayable.cs

```
namespace EmployeeManagementSystem.Core.Interfaces
{
    /// <summary>
    /// Интерфейс для объектов, которым может начисляться оплата
    /// </summary>
    public interface IPayable
    {
        /// <summary>
        /// Расчет заработной платы
        /// </summary>
        /// <returns>Сумма к выплате</returns>
        decimal CalculateSalary();
    }
}
```

IReportable.cs

```
namespace EmployeeManagementSystem.Core.Interfaces
{
    /// <summary>
    /// Интерфейс для объектов, формирующих отчеты
    /// </summary>
    public interface IReportable
    {
        /// <summary>
        /// Создание отчета
        /// </summary>
        /// <param name="format">Формат отчета</param>
        /// <returns>Строка с отчетом в указанном формате</returns>
        string GenerateReport(string format);
    }
}
```

Test.cs

```
using Xunit;
using EmployeeManagementSystem.Core.Models;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using EmployeeManagementSystem.Core.Enums;
using EmployeeManagementSystem.Core.Models.Structs;
using EmployeeManagementSystem.Core.Services;

namespace EmployeeManagementSystem.Core.Models.Tests
{
    #region EmployeeTests
    public class EmployeeTests
    {
        private readonly ContactInfo _testContactInfo = new ContactInfo(
            "test@example.com",
            "+1234567890",
            "123 Test Street");

        [Fact]
        public void CalculateExperience_ShouldReturnCorrectMonths()
        {
            // Arrange
            var now = DateTime.Now;
            var hireDate = now.AddMonths(-15);

            var department = new Department(1, "Test Department", null, 100000);
            var specialist = new Specialist(
                1, "John Doe", DateTime.Now.AddYears(-30), hireDate, 5000,
                EmployeePosition.SeniorSpecialist, _testContactInfo, department,
                "Software Development", 8);

            // Act
            int experience = specialist.CalculateExperience();

            // Assert
            Assert.Equal(15, experience);
        }

        [Fact]
        public void CalculateExperienceString_ShouldReturnCorrectFormat()
        {
            // Arrange
            var now = DateTime.Now;
            var hireDate = now.AddYears(-1).AddMonths(-3);

            var department = new Department(1, "Test Department", null, 100000);
            var specialist = new Specialist(
                1, "John Doe", DateTime.Now.AddYears(-30), hireDate, 5000,
                EmployeePosition.SeniorSpecialist, _testContactInfo, department,
                "Software Development", 8);

            // Act
```

```
            string experienceString = specialist.CalculateExperienceString();

            // Assert
            Assert.Equal("1г,3м", experienceString);
        }
    }
#endregion

    #region ManagerTests
    public class ManagerTests
    {
        private readonly ContactInfo _testContactInfo = new ContactInfo(
            "test@example.com",
            "+1234567890",
            "123 Test Street");

        [Fact]
        public void AddSubordinate_ShouldAddEmployeeToList()
        {
            // Arrange
            var department = new Department(1, "Test Department", null, 100000);
            var manager = new Manager(
                1, "Jane Manager", DateTime.Now.AddYears(-40),
DateTime.Now.AddYears(-5), 10000,
                EmployeePosition.DepartmentHead, _testContactInfo,  department,
                ManagementLevel.MiddleManager, 500);

            var specialist = new Specialist(
                2, "John Specialist", DateTime.Now.AddYears(-30),
DateTime.Now.AddYears(-2), 5000,
                EmployeePosition.SeniorSpecialist, _testContactInfo, department,
                "Software Development", 8);

            // Act
            var result = manager.AddSubordinate(specialist);

            // Assert
            Assert.True(result);
            Assert.Single(manager.Subordinates);
            Assert.Equal(specialist, manager.Subordinates[0]);
        }

        [Fact]
        public void AddSubordinate_ShouldReturnFalse_WhenAddingSelf()
        {
            // Arrange
            var department = new Department(1, "Test Department", null, 100000);
            var manager = new Manager(
                1, "Jane Manager", DateTime.Now.AddYears(-40),
DateTime.Now.AddYears(-5), 10000,
                EmployeePosition.DepartmentHead, _testContactInfo, department,
                ManagementLevel.MiddleManager, 500);

            // Act
            var result = manager.AddSubordinate(manager);

            // Assert
```

```
            Assert.False(result);
            Assert.Empty(manager.Subordinates);
        }

        [Fact]
        public void CalculateSalary_ShouldIncludeExperienceAndSubordinatesBonus()
        {
            // Arrange
            var department = new Department(1, "Test Department", null, 100000);
            var manager = new Manager(
                1, "Jane Manager", DateTime.Now.AddYears(-40),
DateTime.Now.AddMonths(-60), 10000,
                EmployeePosition.DepartmentHead, _testContactInfo,  department,
                ManagementLevel.MiddleManager, 500);

            var specialist1 = new Specialist(
                2, "John Specialist", DateTime.Now.AddYears(-30),
DateTime.Now.AddYears(-2), 5000,
                EmployeePosition.SeniorSpecialist, _testContactInfo, department,
                "Software Development", 8);

            var specialist2 = new Specialist(
                3, "Alice Developer", DateTime.Now.AddYears(-25),
DateTime.Now.AddYears(-1), 4500,
                EmployeePosition.MiddleSpecialist, _testContactInfo, department,
                "Software Development", 6);

            manager.AddSubordinate(specialist1);
            manager.AddSubordinate(specialist2);

            // Act
            decimal salary = manager.CalculateSalary();

            // Assert
            decimal expectedExperienceBonus = 10000 * (60m / 100m); // 60 months /
100
            decimal expectedSubordinatesBonus = 500 * 2; // 500 per subordinate
            decimal expectedSalary = 10000 + expectedExperienceBonus +
expectedSubordinatesBonus;

            Assert.Equal(expectedSalary, salary);
        }

        [Fact]
        public void RemoveSubordinate_ShouldRemoveEmployeeFromList()
        {
            // Arrange
            var department = new Department(1, "Test Department", null, 100000);
            var manager = new Manager(
                1, "Jane Manager", DateTime.Now.AddYears(-40),
DateTime.Now.AddYears(-5), 10000,
                EmployeePosition.DepartmentHead, _testContactInfo, department,
                ManagementLevel.MiddleManager, 500);

            var specialist = new Specialist(
                2, "John Specialist", DateTime.Now.AddYears(-30),
DateTime.Now.AddYears(-2), 5000,
```

```
                EmployeePosition.SeniorSpecialist, _testContactInfo, department,
                "Software Development", 8);

            manager.AddSubordinate(specialist);

            // Act
            var result = manager.RemoveSubordinate(specialist);

            // Assert
            Assert.True(result);
            Assert.Empty(manager.Subordinates);
        }

        [Fact]
        public void CalculateTeamSalary_ShouldSumManagerAndSubordinatesSalaries()
        {
            // Arrange
            var department = new Department(1, "Test Department", null, 100000);
            var manager = new Manager(
                1, "Jane Manager", DateTime.Now.AddYears(-40),
DateTime.Now.AddYears(-5), 10000,
                EmployeePosition.DepartmentHead, _testContactInfo,  department,
                ManagementLevel.MiddleManager, 500);

            var specialist = new Specialist(
                2, "John Specialist", DateTime.Now.AddYears(-30),
DateTime.Now.AddYears(-2), 5000,
                EmployeePosition.SeniorSpecialist, _testContactInfo, department,
                "Software Development", 8);

            manager.AddSubordinate(specialist);

            // Act
            decimal teamSalary = manager.CalculateTeamSalary();

            // Assert
            decimal managerSalary = manager.CalculateSalary();
            decimal specialistSalary = specialist.CalculateSalary();

            Assert.Equal(managerSalary + specialistSalary, teamSalary);
        }
    }

#endregion

    #region SpecialistTests
    public class SpecialistTests
    {
        private readonly ContactInfo _testContactInfo = new ContactInfo(
            "test@example.com",
            "+1234567890",
            "123 Test Street");

        [Fact]
        public void CalculateSalary_ShouldIncludeQualificationAndProjectBonuses()
        {
            // Arrange
```

```
            var department = new Department(1, "Test Department", null, 100000);
            var specialist = new Specialist(
                1, "John Specialist", DateTime.Now.AddYears(-30),
DateTime.Now.AddMonths(-24), 5000,
                EmployeePosition.SeniorSpecialist, _testContactInfo,  department,
                "Software Development", 8);

            specialist.AddProject(1, null, 20); // 20 hours per week
            specialist.AddProject(2, null, 10); // 10 hours per week

            // Act
            decimal salary = specialist.CalculateSalary();

            // Assert
            decimal expectedExperienceBonus = 5000 * (24m / 200m); // 24 months / 200
            decimal expectedQualificationBonus = 5000 * (8m / 10m); // Level 8 / 10
            decimal expectedProjectBonus = (20 + 10) * 0.2m; // 30 hours * 0.2
            decimal expectedSalary = 5000 + expectedExperienceBonus +
expectedQualificationBonus + expectedProjectBonus;

            Assert.Equal(expectedSalary, salary);
        }

        [Fact]
        public void AddProject_ShouldAddToProjectAssignments()
        {
            // Arrange
            var department = new Department(1, "Test Department", null, 100000);
            var specialist = new Specialist(
                1, "John Specialist", DateTime.Now.AddYears(-30),
DateTime.Now.AddYears(-2), 5000,
                EmployeePosition.SeniorSpecialist, _testContactInfo,  department,
                "Software Development", 8);

            // Act
            specialist.AddProject(1, null, 20);

            // Assert
            Assert.Single(specialist.ProjectAssignments);
            Assert.Equal(1, specialist.ProjectAssignments[0].ProjectId);
            Assert.Equal(20, specialist.ProjectAssignments[0].HoursPerWeek);
        }

        [Fact]
        public void RemoveProject_ShouldRemoveFromProjectAssignments()
        {
            // Arrange
            var department = new Department(1, "Test Department", null, 100000);
            var specialist = new Specialist(
                1, "John Specialist", DateTime.Now.AddYears(-30),
DateTime.Now.AddYears(-2), 5000,
                EmployeePosition.SeniorSpecialist, _testContactInfo,  department,
                "Software Development", 8);

            specialist.AddProject(1, null, 20);
            specialist.AddProject(2, null, 10);
```

```
            // Act
            var result = specialist.RemoveProject(1);

            // Assert
            Assert.True(result);
            Assert.Single(specialist.ProjectAssignments);
            Assert.Equal(2, specialist.ProjectAssignments[0].ProjectId);
        }

        [Fact]
        public void UpdateQualification_ShouldChangeQualificationLevel()
        {
            // Arrange
            var department = new Department(1, "Test Department", null, 100000);
            var specialist = new Specialist(
                1, "John Specialist", DateTime.Now.AddYears(-30),
DateTime.Now.AddYears(-2), 5000,
                EmployeePosition.SeniorSpecialist, _testContactInfo, department,
                "Software Development", 5);

            // Act
            specialist.UpdateQualification(9);

            // Assert
            Assert.Equal(9, specialist.QualificationLevel);
        }
    }

#endregion

    #region DepartmentTests
    public class DepartmentTests
    {
        private readonly ContactInfo _testContactInfo = new ContactInfo(
            "test@example.com",
            "+1234567890",
            "123 Test Street");

        [Fact]
        public void AddEmployee_ShouldAddToEmployeesList()
        {
            // Arrange
            var department = new Department(1, "Test Department", null, 100000);
            var specialist = new Specialist(
                1, "John Specialist", DateTime.Now.AddYears(-30),
DateTime.Now.AddYears(-2), 5000,
                EmployeePosition.SeniorSpecialist, _testContactInfo, null,
                "Software Development", 8);

            // Act
            department.AddEmployee(specialist);

            // Assert
            Assert.Single(department.Employees);
            Assert.Equal(specialist, department.Employees[0]);
            Assert.Equal(department, specialist.Department);
        }
```

```
        [Fact]
        public void SetHead_ShouldUpdateHeadAndAddToSubordinates()
        {
            // Arrange
            var department = new Department(1, "Test Department", null, 100000);
            var manager = new Manager(
                1, "Jane Manager", DateTime.Now.AddYears(-40),
DateTime.Now.AddYears(-5), 10000,
                EmployeePosition.DepartmentHead, _testContactInfo, null,
                ManagementLevel.MiddleManager, 500);

            var specialist = new Specialist(
                2, "John Specialist", DateTime.Now.AddYears(-30),
DateTime.Now.AddYears(-2), 5000,
                EmployeePosition.SeniorSpecialist, _testContactInfo, null,
                "Software Development", 8);

            department.AddEmployee(specialist);

            // Act
            department.SetHead(manager);

            // Assert
            Assert.Equal(manager, department.Head);
            Assert.Contains(manager, department.Employees);
            Assert.Contains(specialist, manager.Subordinates);
        }

        [Fact]
        public void RemoveEmployee_ShouldRemoveFromEmployeesList()
        {
            // Arrange
            var department = new Department(1, "Test Department", null, 100000);
            var specialist = new Specialist(
                1, "John Specialist", DateTime.Now.AddYears(-30),
DateTime.Now.AddYears(-2), 5000,
                EmployeePosition.SeniorSpecialist, _testContactInfo, null,
                "Software Development", 8);

            var manager = new Manager(
                2, "Jane Manager", DateTime.Now.AddYears(-40),
DateTime.Now.AddYears(-5), 10000,
                EmployeePosition.DepartmentHead, _testContactInfo, null,
                ManagementLevel.MiddleManager, 500);


            department.AddEmployee(manager);
            department.SetHead(manager);

            department.AddEmployee(specialist);

            // Act
            //var result = department.RemoveEmployee(1);
            var result = department.RemoveEmployee(specialist);

            // Assert
```

```
            Assert.True(result);

            result = department.RemoveEmployee(manager);
            Assert.True(result);


            Assert.Empty(department.Employees);
        }



        [Fact]
        public void GetSpecialistsBySpecialization_ShouldReturnMatchingSpecialists()
        {
            // Arrange
            var department = new Department(1, "Test Department", null, 100000);

            var specialist1 = new Specialist(
                1, "John Developer", DateTime.Now.AddYears(-30),
DateTime.Now.AddYears(-2), 5000,
                EmployeePosition.SeniorSpecialist, _testContactInfo,  null,
                "Software Development", 8);

            var specialist2 = new Specialist(
                2, "Alice Developer", DateTime.Now.AddYears(-28),
DateTime.Now.AddYears(-1), 4500,
                EmployeePosition.MiddleSpecialist, _testContactInfo, null,
                "Software Development", 6);

            var specialist3 = new Specialist(
                3, "Bob Designer", DateTime.Now.AddYears(-25),
DateTime.Now.AddMonths(-6), 4000,
                EmployeePosition.JuniorSpecialist, _testContactInfo, null,
                "UI/UX Design", 4);

            department.AddEmployee(specialist1);
            department.AddEmployee(specialist2);
            department.AddEmployee(specialist3);

            // Act
            var devSpecialists = department.GetSpecialistsBySpecialization("Software
Development").ToList();

            // Assert
            Assert.Equal(2, devSpecialists.Count);
            Assert.Contains(specialist1, devSpecialists);
            Assert.Contains(specialist2, devSpecialists);
        }
    }
#endregion

    #region ProjectTests
    public class ProjectTests
    {
        private readonly ContactInfo _testContactInfo = new ContactInfo(
            "test@example.com",
```

```
            "+1234567890",
            "123 Test Street");

        [Fact]
        public void AssignTeamMember_ShouldAddEmployeeToTeam()
        {
            // Arrange
            var department = new Department(1, "Test Department", null, 100000);
            var manager = new Manager(
                1, "Jane Manager", DateTime.Now.AddYears(-40),
DateTime.Now.AddYears(-5), 10000,
                EmployeePosition.DepartmentHead, _testContactInfo,  department,
                ManagementLevel.MiddleManager, 500);

            var project = new Project(
                1, "Test Project", DateTime.Now, DateTime.Now.AddMonths(6), 50000,
                ProjectStatus.Planning, manager);

            var specialist = new Specialist(
                2, "John Specialist", DateTime.Now.AddYears(-30),
DateTime.Now.AddYears(-2), 5000,
                EmployeePosition.SeniorSpecialist, _testContactInfo, department,
                "Software Development", 8);

            // Act
            project.AssignTeamMember(specialist, 20);

            // Assert
            Assert.Equal(2, project.Team.Count); // Manager + Specialist
            Assert.Contains(specialist, project.Team);

            // Check that the project was added to specialist's assignments
            Assert.Single(specialist.ProjectAssignments);
            Assert.Equal(1, specialist.ProjectAssignments[0].ProjectId);
            Assert.Equal(20, specialist.ProjectAssignments[0].HoursPerWeek);
        }

        [Fact]
        public void RemoveTeamMember_ShouldRemoveEmployeeFromTeam()
        {
            // Arrange
            var department = new Department(1, "Test Department", null, 100000);
            var manager = new Manager(
                1, "Jane Manager", DateTime.Now.AddYears(-40),
DateTime.Now.AddYears(-5), 10000,
                EmployeePosition.DepartmentHead, _testContactInfo, department,
                ManagementLevel.MiddleManager, 500);

            var project = new Project(
                1, "Test Project", DateTime.Now, DateTime.Now.AddMonths(6), 50000,
                ProjectStatus.Planning, manager);

            var specialist = new Specialist(
                2, "John Specialist", DateTime.Now.AddYears(-30),
DateTime.Now.AddYears(-2), 5000,
                EmployeePosition.SeniorSpecialist, _testContactInfo, department,
                "Software Development", 8);
```

```
            project.AssignTeamMember(specialist, 20);

            // Act
            var result = project.RemoveTeamMember(2);

            // Assert
            Assert.True(result);
            Assert.Single(project.Team); // Only the manager remains
            Assert.DoesNotContain(specialist, project.Team);

            // Check that the project was removed from specialist's assignments
            Assert.Empty(specialist.ProjectAssignments);
        }

        [Fact]
        public void UpdateStatus_ShouldChangeProjectStatus()
        {
            // Arrange
            var department = new Department(1, "Test Department", null, 100000);
            var manager = new Manager(
                1, "Jane Manager", DateTime.Now.AddYears(-40),
DateTime.Now.AddYears(-5), 10000,
                EmployeePosition.DepartmentHead, _testContactInfo, department,
                ManagementLevel.MiddleManager, 500);

            var project = new Project(
                1, "Test Project", DateTime.Now.AddDays(-30),
DateTime.Now.AddMonths(6), 50000,
                ProjectStatus.Planning, manager);

            // Act
            project.UpdateStatus(ProjectStatus.InProgress);

            // Assert
            Assert.Equal(ProjectStatus.InProgress, project.Status);
        }

        [Fact]
        public void UpdateStatus_Completed_ShouldSetEndDateToNow()
        {
            // Arrange
            var department = new Department(1, "Test Department", null, 100000);
            var manager = new Manager(
                1, "Jane Manager", DateTime.Now.AddYears(-40),
DateTime.Now.AddYears(-5), 10000,
                EmployeePosition.DepartmentHead, _testContactInfo, department,
                ManagementLevel.MiddleManager, 500);

            var originalEndDate = DateTime.Now.AddMonths(6);
            var project = new Project(
                1, "Test Project", DateTime.Now.AddDays(-30), originalEndDate, 50000,
                ProjectStatus.InProgress, manager);

            // Act
            project.UpdateStatus(ProjectStatus.Completed);
```

```
            // Assert
            Assert.Equal(ProjectStatus.Completed, project.Status);
            Assert.NotEqual(originalEndDate, project.EndDate);

            // The end date should be close to now (within a second)
            var difference = Math.Abs((DateTime.Now - project.EndDate).TotalSeconds);
            Assert.True(difference < 1);
        }

        [Fact]
        public void GetDuration_ShouldReturnCorrectNumberOfDays()
        {
            // Arrange
            var department = new Department(1, "Test Department", null, 100000);
            var manager = new Manager(
                1, "Jane Manager", DateTime.Now.AddYears(-40),
DateTime.Now.AddYears(-5), 10000,
                EmployeePosition.DepartmentHead, _testContactInfo, department,
                ManagementLevel.MiddleManager, 500);

            var startDate = DateTime.Now.AddDays(-30);
            var project = new Project(
                1, "Test Project", startDate, DateTime.Now.AddMonths(6), 50000,
                ProjectStatus.InProgress, manager);

            // Act
            int duration = project.GetDuration();

            // Assert
            Assert.Equal(30, duration);
        }
    }
#endregion

    #region CompanyTests
    public class CompanyTests
    {
        private readonly ContactInfo _testContactInfo = new ContactInfo(
            "test@example.com",
            "+1234567890",
            "123 Test Street");

        [Fact]
        public void AddDepartment_ShouldAddToDepartmentsList()
        {
            // Arrange
            var ceo = new Manager(
                1, "John CEO", DateTime.Now.AddYears(-50), DateTime.Now.AddYears(-
10), 20000,
                EmployeePosition.CEO, _testContactInfo, null,
                ManagementLevel.TopManager, 1000);

            var company = new Company("Test Company", ceo);
            var department = new Department(1, "HR Department", null, 100000);

            // Act
            company.AddDepartment(department);
```

```
            // Assert
            Assert.Single(company.Departments);
            Assert.Equal(department, company.Departments[0]);
        }

        [Fact]
        public void RemoveDepartment_ShouldRemoveFromDepartmentsList()
        {
            // Arrange
            var ceo = new Manager(
                1, "John CEO", DateTime.Now.AddYears(-50), DateTime.Now.AddYears(-
10), 20000,
                EmployeePosition.CEO, _testContactInfo, null,
                ManagementLevel.TopManager, 1000);

            var company = new Company("Test Company", ceo);
            var department = new Department(1, "HR Department", null, 100000);
            company.AddDepartment(department);

            var developer1 = new Specialist(
                5, "Charlie Dev", DateTime.Now.AddYears(-30), DateTime.Now.AddYears(-
2), 8000,
                EmployeePosition.SeniorSpecialist, _testContactInfo, null,
                "Software Development", 8);

            var developer2 = new Specialist(
                6, "Dave Dev", DateTime.Now.AddYears(-28), DateTime.Now.AddYears(-1),
6000,
                EmployeePosition.MiddleSpecialist, _testContactInfo, null,
                "Software Development", 6);

            var manager = new Manager(
                7, "Jane Manager", DateTime.Now.AddYears(-40),
DateTime.Now.AddYears(-5), 10000,
                EmployeePosition.DepartmentHead, _testContactInfo, null,
                ManagementLevel.MiddleManager, 500);

            department.SetHead(manager);
            department.AddEmployee(developer1);
            department.AddEmployee(developer2);

            // Act
            var result = company.RemoveDepartment(department);

            // Assert
            Assert.False(result); // Department is not empty

            department.RemoveEmployee(developer1);
            department.RemoveEmployee(developer2);
            department.RemoveEmployee(manager);

            // Remove empty department
            result = company.RemoveDepartment(department);

            // Assert
            Assert.True(result);
```

```
            Assert.Empty(company.Departments);
        }

        [Fact]
        public void AddProject_ShouldAddToProjectsList()
        {
            // Arrange
            var ceo = new Manager(
                1, "John CEO", DateTime.Now.AddYears(-50), DateTime.Now.AddYears(-
10), 20000,
                EmployeePosition.CEO, _testContactInfo, null,
                ManagementLevel.TopManager, 1000);

            var company = new Company("Test Company", ceo);
            var department = new Department(1, "IT Department", null, 200000);

            var manager = new Manager(
                2, "Jane Manager", DateTime.Now.AddYears(-40),
DateTime.Now.AddYears(-5), 10000,
                EmployeePosition.DepartmentHead, _testContactInfo, department,
                ManagementLevel.MiddleManager, 500);

            var project = new Project(
                1, "New System", DateTime.Now, DateTime.Now.AddMonths(6), 50000,
                ProjectStatus.Planning, manager);

            // Act
            company.AddProject(project);

            // Assert
            Assert.Single(company.Projects);
            Assert.Equal(project, company.Projects[0]);
        }

        [Fact]
        public void RemoveProject_ShouldRemoveFromProjectsList()
        {
            // Arrange
            var ceo = new Manager(
                1, "John CEO", DateTime.Now.AddYears(-50), DateTime.Now.AddYears(-
10), 20000,
                EmployeePosition.CEO, _testContactInfo, null,
                ManagementLevel.TopManager, 1000);

            var company = new Company("Test Company", ceo);
            var department = new Department(1, "IT Department", null, 200000);

            var manager = new Manager(
                2, "Jane Manager", DateTime.Now.AddYears(-40),
DateTime.Now.AddYears(-5), 10000,
                EmployeePosition.DepartmentHead, _testContactInfo, department,
                ManagementLevel.MiddleManager, 500);

            var project = new Project(
                1, "New System", DateTime.Now, DateTime.Now.AddMonths(6), 50000,
                ProjectStatus.Planning, manager);
```

```
            company.AddProject(project);

            // Act
            var result = company.RemoveProject(1);

            // Assert
            Assert.True(result);
            Assert.Empty(company.Projects);
        }

        [Fact]
        public void GetTotalEmployees_ShouldReturnCorrectCount()
        {
            // Arrange
            var ceo = new Manager(
                1, "John CEO", DateTime.Now.AddYears(-50), DateTime.Now.AddYears(-
10), 20000,
                EmployeePosition.CEO, _testContactInfo,  null,
                ManagementLevel.TopManager, 1000);

            var company = new Company("Test Company", ceo);

            var hrDepartment = new Department(1, "HR Department", null, 100000);
            var itDepartment = new Department(2, "IT Department", null, 200000);

            var hrManager = new Manager(
                2, "Jane HR", DateTime.Now.AddYears(-45), DateTime.Now.AddYears(-7),
12000,
                EmployeePosition.DepartmentHead, _testContactInfo, hrDepartment,
                ManagementLevel.MiddleManager, 600);

            var itManager = new Manager(
                3, "Bob IT", DateTime.Now.AddYears(-40), DateTime.Now.AddYears(-5),
15000,
                EmployeePosition.DepartmentHead, _testContactInfo, itDepartment,
                ManagementLevel.MiddleManager, 700);

            var hrSpecialist = new Specialist(
                4, "Alice HR", DateTime.Now.AddYears(-35), DateTime.Now.AddYears(-3),
7000,
                EmployeePosition.SeniorSpecialist, _testContactInfo, hrDepartment,
                "HR Management", 7);

            var developer1 = new Specialist(
                5, "Charlie Dev", DateTime.Now.AddYears(-30), DateTime.Now.AddYears(-
2), 8000,
                EmployeePosition.SeniorSpecialist, _testContactInfo, itDepartment,
                "Software Development", 8);

            var developer2 = new Specialist(
                6, "Dave Dev", DateTime.Now.AddYears(-28), DateTime.Now.AddYears(-1),
6000,
                EmployeePosition.MiddleSpecialist, _testContactInfo, itDepartment,
                "Software Development", 6);

            hrDepartment.SetHead(hrManager);
            itDepartment.SetHead(itManager);
```

```
            company.AddDepartment(hrDepartment);
            company.AddDepartment(itDepartment);

            // Act
            int totalEmployees = company.GetTotalEmployees();

            // Assert
            Assert.Equal(6, totalEmployees); // CEO + 2 managers + 3 specialists
        }

        [Fact]
        public void GetAllEmployees_ShouldReturnAllEmployees()
        {
            // Arrange
            var ceo = new Manager(
                1, "John CEO", DateTime.Now.AddYears(-50), DateTime.Now.AddYears(-
10), 20000,
                EmployeePosition.CEO, _testContactInfo, null,
                ManagementLevel.TopManager, 1000);

            var company = new Company("Test Company", ceo);

            var department = new Department(1, "HR Department", null, 100000);

            var manager = new Manager(
                2, "Jane Manager", DateTime.Now.AddYears(-40),
DateTime.Now.AddYears(-5), 10000,
                EmployeePosition.DepartmentHead, _testContactInfo, department,
                ManagementLevel.MiddleManager, 500);

            var specialist = new Specialist(
                3, "Alice Specialist", DateTime.Now.AddYears(-30),
DateTime.Now.AddYears(-2), 5000,
                EmployeePosition.SeniorSpecialist, _testContactInfo, department,
                "HR Management", 8);

            company.AddDepartment(department);

            department.SetHead(manager);
            //department.AddEmployee(specialist);

            // Act
            var allEmployees = company.GetAllEmployees().ToList();

            // Assert
            Assert.Equal(3, allEmployees.Count); // CEO + Manager + Specialist
            Assert.Contains(ceo, allEmployees);
            Assert.Contains(manager, allEmployees);
            Assert.Contains(specialist, allEmployees);
        }
    }
#endregion

    #region DataServiceTests
    public class DataServiceTests
    {
```

```
        [Fact]
        public void SaveAndLoad_ShouldPreserveCompanyData()
        {
            var dataService = new DataService();
            var company = new Company("Test Company", null);
            var department = new Department(1, "Test Dept", null, 100000m);
            company.AddDepartment(department);

            dataService.SaveData(company, "test.bin");
            var loaded = dataService.LoadData("test.bin");

            Assert.NotNull(loaded);
            Assert.Equal(company.Name, loaded.Name);
            Assert.Equal(company.Departments.Count, loaded.Departments.Count);
        }
    }

#endregion
}
```