Приложение

**ФОНД ОЦЕНОЧНЫХ СРЕДСТВ**

ТЕХНОЛОГИИ ПРОМЫШЛЕННОЙ РАЗРАБОТКИ

ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ.

ТРЕБОВАНИЯ

**2 Семестр**

**Раздел 2 Реинжиниринг**

**2.1 Лабораторная работа (ЛР) - 11 Неделя**

**Лабораторная работа №2**

**«Технологии промышленной разработки программного обеспечения. Требования»   
(2 семестр)**

**Приложение №1**

К Лабораторной работе №1

**Регистратор ресурсов (обзор базового варианта как проекта)**

1. Постановка задачи регистрации ресурсов
   1. На пороге разработки

Дан конечный набор ресурсов, предоставляемых во временное пользование неким клиентам. Имеется также лицо, далее «клерк», который контролирует процесс занятия и освобождения ресурсов, неким образом взаимодействуя с клиентами и ресурсами.

Такая система встречается во множестве частных случаев, каковыми являются любая вокзальная камера хранения, библиотека, гостиница, система резервирования мест на пассажирском транспорте. Она входит как элемент в множество более сложных хозяйственных и технических систем. С другой стороны, её можно обобщить ещё дальше, допуская, например, что контролёром и, может быть, регулятором ресурсов является не лицо – человек, а некоторый неодушевлённый механизм.

Положим, что данная система изначально не содержит средств автоматизации, но является «доменом проектирования» – частью реального или воображаемого мира, в пределах которой ставится и затем решается задача компьютерной поддержки определённых протекающих в ней процессов. Разумеется, их нужно указать прежде, чем можно будет приступить к созданию поддержки. Пусть это будут процессы принятия решений, которые приходится выполнять клерку, когда он контролирует занятие и освобождение ресурсов. При этом, чтобы уклониться от разработки аппаратуры и ограничиться исключительно программистской разработкой, мы сознательно воздерживаемся от полной автоматизации системы, устранения клерка как лица.

* 1. Анализ и моделирование области приложения

Будем считать, что с этого момента мы приступаем к разработке означенной поддержки, хотя ясно, что ни предмет поддержки, ни её цели, ни требования к ней не определены достаточно чётко. Однако современные методологии разработки не только допускают, но полагают закономерным и обязательным уточнение задачи разработке в ходе проектирования, главным образом, на ранних его этапах. При этом погружение в процесс разработки по конкретной технологии способствует успешному уточнению задачи, поскольку обеспечивает разработчика конкретными языками, инструментами и правилами их использования.

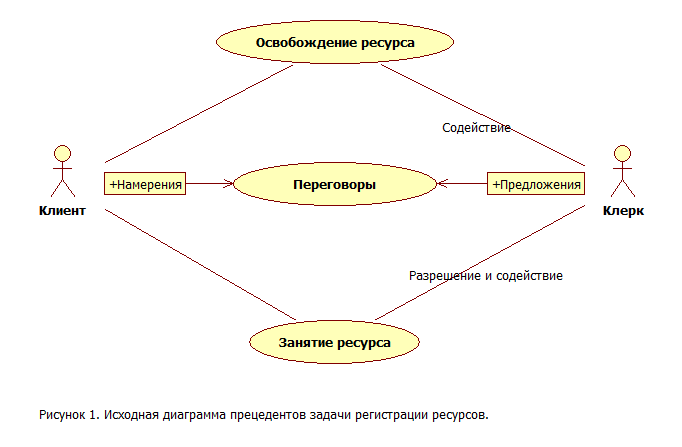
В данном пособии для проектирования отобрана технология (модель жизненного цикла программных продуктов) RUP, и по её правилам собственно проектированию программного продукта предшествует определение и исследование домена проектирования – среды, в которой этот продукт должен функционировать. В результате возникают концептуальная диаграммы классов, а также описания прецедентов в виде таблицы и соответствующей диаграммы. Эти документы («артефакты» разработки) выражают бизнес-модель домена и требования к разрабатываемому продукту.

К моменту начала разработки мы имеем её «идею», которая может быть лишена конкретных деталей, но так или иначе очерчивает домен проектирования, требования к разработке и цель разработки. Уже первая, исходная версия артефактов разработки несёт в себе определённую интерпретацию «идеи», детали, которые так или иначе уменьшают степень её общности. Эта версия не должна показывать проектные решения, относящиеся к разрабатываемому продукту и изменения, связанные с включением программного продукта в домен проектирования. Они появляются позже, в диаграмме системных взаимодействий и диаграмме (диаграммах) проектных классов, а также в связанных с ними артефактах.

Диаграмму системных взаимодействий и диаграмму проектных классов разрабатывают на основе описания прецедентов и концептуальной диаграммы классов, которая может уже в первой своей версии достигнуть детальности, достаточной для перехода к проектированию, или же первая версия подвергается ряду последовательных уточнений. Вторая возможность позволяет построить иерархию описаний доменов проектирования и обосновывать их выбор в терминах применяемой технологии.

В нашем случае Табл.1 описывает прецеденты. На Рис.1 это описание иллюстрировано исходной диаграммой прецедентов.

| Наименование прецедента | Переговоры |
| --- | --- |
| Исполнители | Клиент, Клерк |
| Предусловие | Система работает (предоставляет ресурсы). |
| Описание | **Основной сценарий (поток)**  Клиент высказывает запрос на использование ресурса.  Клерк сообщает клиенту, может ли быть его запрос выполнен и на каких условиях.  Между клиентом и клерком достигается соглашение об использовании ресурса. |
| Постусловие | Клиент может реализовать запрос |
| Описание | **Альтернативный сценарий (поток)**  Клиент высказывает запрос на использование ресурса.  Клерк сообщает клиенту, может ли быть его запрос выполнен и на каких условиях.  Клиент и клерк не достигают соглашения об использовании ресурса. |
| Постусловие | Клиент отзывает запрос. |
| Наименование прецедента | Занятие ресурса |
| Исполнители | Клиент, Клерк |
| Предусловия | Система работает (предоставляет ресурсы),  Между клиентом и клерком достигнуто соглашение об использовании ресурса. |
| Описание | **Основной сценарий (поток)**  Клиент вступает в использование ресурса. Клерк оказывает ему положенное содействие. |
| Постусловие | Клиент занимает запрошенный ресурс. |
| Наименование прецедента | Освобождение ресурса |
| Исполнители | Клиент, Клерк |
| Предусловие | Клиент, занимающий определённый ресурс, намерен его освободить. |
| Описание | **Основной сценарий (поток)**  Клиент освобождает ресурс. Клерк оказывает ему положенное содействие. |
| Постусловие | Занятый ресурс стал свободным. |

Таблица 1. Исходная таблица прецедентов задачи регистрации ресурсов.

Описание для “Рисунок 1. Исходная диаграмма прецедентов задачи регистрации ресурсов”

Данное изображение представляет собой \*\*диаграмму прецедентов (Use Case Diagram)\*\*, которая описывает взаимодействие между участниками системы и процессами в контексте задачи регистрации ресурсов. Давайте разберем изображение поэтапно:

### 1. \*\*Общая структура диаграммы\*\*

- Диаграмма содержит несколько элементов:

- \*\*Акторы (Actors)\*\*: Представляют внешние участников системы, которые взаимодействуют с ней.

- \*\*Прецеденты (Use Cases)\*\*: Определяют функциональные возможности или действия, которые система может выполнять для акторов.

- \*\*Связи\*\*: Показывают отношения между акторами и прецедентами.

### 2. \*\*Элементы диаграммы\*\*

#### \*\*Акторы\*\*

- \*\*Клиент\*\*:

- Изображен как человеческая фигурка слева.

- Участвует в процессе переговоров, предоставляя свои намерения.

- \*\*Клерк\*\*:

- Изображен как человеческая фигурка справа.

- Участвует в процессе переговоров, делая предложения.

#### \*\*Прецеденты\*\*

- \*\*Переговоры\*\*:

- Центральный прецедент на диаграмме.

- Связан с обоими акторами через связи "Намерения" и "Предложения".

- Это основной процесс, который включает взаимодействие клиента и клерка.

- \*\*Занятие ресурса\*\*:

- Прецедент в нижней части диаграммы.

- Связан с прецедентом "Переговоры" через связь "Разрешение и содействие".

- Этот прецедент описывает действие, которое происходит после успешного завершения переговоров.

- \*\*Освобождение ресурса\*\*:

- Прецедент в верхней части диаграммы.

- Связан с прецедентом "Переговоры" через связь "Содействие".

- Этот прецедент описывает действие, которое может происходить параллельно или после переговоров.

### 3. \*\*Связи между элементами\*\*

- \*\*Связь "Намерения"\*\*:

- Идет от актора "Клиент" к прецеденту "Переговоры".

- Значит, клиент выражает свои намерения в рамках процесса переговоров.

- \*\*Связь "+Предложения"\*\*:

- Идет от актора "Клерк" к прецеденту "Переговоры".

- Значит, клерк делает предложения в рамках процесса переговоров.

- \*\*Связь "Содействие"\*\*:

- Идет от прецедента "Переговоры" к прецеденту "Освобождение ресурса".

- Значит, процесс переговоров оказывает влияние на освобождение ресурса.

- \*\*Связь "Разрешение и содействие"\*\*:

- Идет от прецедента "Переговоры" к прецеденту "Занятие ресурса".

- Значит, процесс переговоров приводит к занятию ресурса.

### 4. \*\*Текстовые подписи\*\*

- Внизу под диаграммой написано:

- \*\*"Рисунок 1. Исходная диаграмма прецедентов задачи регистрации ресурсов."\*\*

- Это указывает на то, что диаграмма относится к задаче регистрации ресурсов и является исходным вариантом.

### 5. \*\*Цветовая палитра\*\*

- Прецеденты выделены овальными фигурами желтого цвета.

- Акторы представлены человеческими фигурками с красными очертаниями.

- Связи между элементами показаны красными линиями.

### 6. \*\*Общий смысл диаграммы\*\*

Диаграмма описывает следующий процесс:

1. Клиент и клерк участвуют в процессе \*\*переговоров\*\*, где клиент выражает свои намерения, а клерк делает предложения.

2. В результате переговоров может произойти:

- \*\*Занятие ресурса\*\*, если переговоры прошли успешно.

- \*\*Освобождение ресурса\*\*, если это необходимо в процессе переговоров.

### Заключение

Данная диаграмма четко демонстрирует взаимодействие между участниками системы (клиентом и клерком) и ключевыми процессами (переговоры, занятие/освобождение ресурса). Она помогает понять поток действий и ответственность каждого участника в контексте задачи регистрации ресурсов.

Интерпретация исходной идеи в этих описаниях характеризуется следующими особенностями:

клиент во всех сценариях один,

связь между клиентами различных прецедентов не предусмотрена,

ресурс освобождается только по инициативе клиента,

ресурсы запрашиваются, занимаются и освобождаются по одиночке,

содействие клерка клиенту осуществляется после принятия ими решения об использовании ресурса и с этим решением не связано.

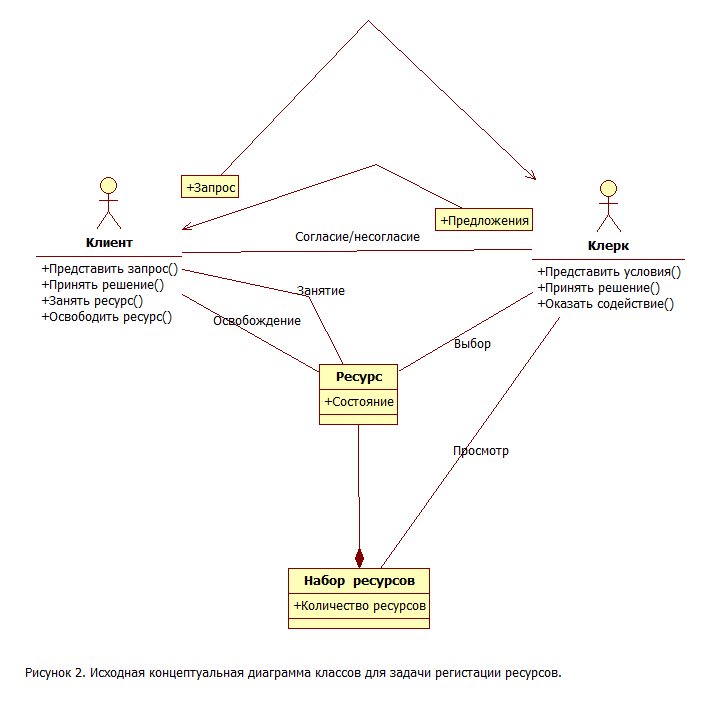
Исходная диаграмм классов представлена на Рис.2.

На этой диаграмме показано, что ресурсы имеют лишь два атрибута – Состояние и Номер, причём состояние принимает значение “ресурс свободен” либо “ресурс занят”. Соответственно, клиент может запрашивать или ресурс с конкретным номером, или любой. Клерк, со своей стороны, может предложить клиенту на выбор все или некоторое подмножество свободных ресурсов по их номерам. То есть, указанная интерпретация сводит свободу решений клерка к тривиальному случаю: выбору ресурса из свободных в данный момент.

Выбор такой интерпретации оправдывается потенциальной простотой программного решения, и далее можно будет увидеть, что эта интерпретация не связывает развития проекта по многим направлениям.

* 1. Обоснование требований к решению задачи

В контексте приведённых выше артефактов разработки исходная идея (требование) компьютерной поддержки процессов принятия решений клерка приводит разработчиков к решению возложить на компьютер реализацию ассоциативной связи “Выбор”. То есть, компьютер должен указывать клерку, какой ресурс может быть предоставлен очередному клиенту. Если реализовать автоматическое взаимодействие компьютера с ресурсами, при котором компьютер без участия клерка получает информацию о занятости ресурсов, то связь “Просмотр” между клерком и набором ресурсов не нужна. Но если разработчики отказываются реализовать автоматическое взаимодействие компьютера с ресурсами, то компьютер должен получать нужную информацию о состоянии ресурсов от клерка или клиента. Если стремиться к минимальному изменению диаграммы классов, то остаётся принять решение о преобразовании ассоциации “Просмотр” между клерком и набором ресурсов в ассоциацию между клерком и компьютером с наименованием “Опрос и актуализация” и соответствующей семантикой.

В результате будет получена первая диаграмма проектных классов, показанная на Рис.3. Эта диаграмма должна пополняться и уточняться, подводя к разработке соответствующих программ на языках программирования. Поскольку они преимущественно англоязычны, уже начиная с первой диаграммы проектных классов словесные обозначения образованы на основе английского языка и записаны латиницей. Компьютер, аппаратура и программы, впервые появившиеся на диаграмме на Рис.3, представлены подсистемой System. 

Описание для “Рисунок 2. Исходная концептуальная диаграмма классов для задачи регистрации ресурсов”

### Описание изображения

Данное изображение представляет собой \*\*концептуальную диаграмму классов (Class Diagram)\*\*, которая описывает структуру объектов и их взаимодействие в контексте задачи регистрации ресурсов. Давайте разберем изображение поэтапно:

---

#### \*\*1. Общая структура диаграммы\*\*

- Диаграмма содержит несколько элементов:

- \*\*Классы\*\*: Представляют собой сущности системы, которые имеют атрибуты и методы.

- \*\*Ассоциации\*\*: Показывают отношения между классами.

- \*\*Акторы\*\*: Внешние участники системы, которые взаимодействуют с классами.

---

#### \*\*2. Элементы диаграммы\*\*

##### \*\*Акторы\*\*

- \*\*Клиент\*\*:

- Изображен как человеческая фигурка слева.

- Участвует в процессе взаимодействия с системой через запросы и действия.

- \*\*Клерк\*\*:

- Изображен как человеческая фигурка справа.

- Участвует в процессе взаимодействия с системой через предложения и действия.

##### \*\*Классы\*\*

1. \*\*Ресурс\*\*:

- Центральный класс на диаграмме.

- Содержит следующие компоненты:

- \*\*Атрибуты\*\*:

- `+Состояние`: Атрибут, который хранит информацию о текущем состоянии ресурса (например, занят или свободен).

- \*\*Методы\*\*:

- `+Представить запрос()`: Метод для представления запроса на использование ресурса.

- `+Принять решение()`: Метод для принятия решения о доступности ресурса.

- `+Занять ресурс()`: Метод для занятия ресурса.

- `+Освободить ресурс()`: Метод для освобождения ресурса.

- Этот класс отвечает за управление индивидуальным ресурсом.

2. \*\*Набор ресурсов\*\*:

- Класс в нижней части диаграммы.

- Содержит следующие компоненты:

- \*\*Атрибуты\*\*:

- `+Количество ресурсов`: Атрибут, который хранит информацию о количестве доступных ресурсов.

- Этот класс отвечает за управление группой ресурсов.

---

##### \*\*Ассоциации\*\*

1. \*\*Связь между "Клиент" и "Ресурс"\*\*:

- Связь называется "+Запрос".

- Значит, клиент отправляет запрос на использование ресурса.

- Дополнительные методы клиента:

- `+Представить запрос()`: Метод для отправки запроса.

- `+Принять решение()`: Метод для принятия решения после получения ответа от системы.

- `+Занять ресурс()`: Метод для выполнения действий при успешном получении ресурса.

- `+Освободить ресурс()`: Метод для освобождения ресурса после использования.

2. \*\*Связь между "Клерк" и "Ресурс"\*\*:

- Связь называется "+Предложения".

- Значит, клерк делает предложения по использованию ресурса.

- Дополнительные методы клерка:

- `+Представить условия()`: Метод для предоставления условий использования ресурса.

- `+Принять решение()`: Метод для принятия решений относительно доступности ресурса.

- `+Оказать содействие()`: Метод для оказания помощи в процессе использования ресурса.

3. \*\*Связь между "Ресурс" и "Набор ресурсов"\*\*:

- Связь называется "Присоединение":

- Это отношение агрегации (aggregation), которое показывает, что ресурсы являются частью набора ресурсов.

- Дополнительные связи:

- `Занятие`: Отражает процесс выбора и использования ресурса.

- `Освобождение`: Отражает процесс освобождения ресурса после использования.

- `Выбор`: Процесс выбора конкретного ресурса из набора.

- `Просмотр`: Процесс просмотра информации о ресурсах.

---

#### \*\*3. Текстовые подписи\*\*

- Внизу под диаграммой написано:

- \*\*"Рисунок 2. Исходная концептуальная диаграмма классов для задачи регистрации ресурсов."\*\*

- Это указывает на то, что диаграмма является исходной концептуальной моделью для задачи регистрации ресурсов.

---

#### \*\*4. Цветовая палитра\*\*

- Классы выделены прямоугольными фигурами с разделением на три секции:

- Верхняя секция: Имя класса.

- Середина: Атрибуты класса.

- Нижняя секция: Методы класса.

- Акторы представлены человеческими фигурками с красными очертаниями.

- Связи между элементами показаны красными линиями с соответствующими метками.

---

#### \*\*5. Общий смысл диаграммы\*\*

Диаграмма описывает следующую структуру и взаимодействие:

1. \*\*Клиент\*\* отправляет запрос на использование ресурса.

2. \*\*Клерк\*\* анализирует запрос и предлагает условия использования ресурса.

3. \*\*Ресурс\*\* отвечает за управление своим состоянием (занят/освобожден) и выполняет операции по занятию/освобождению.

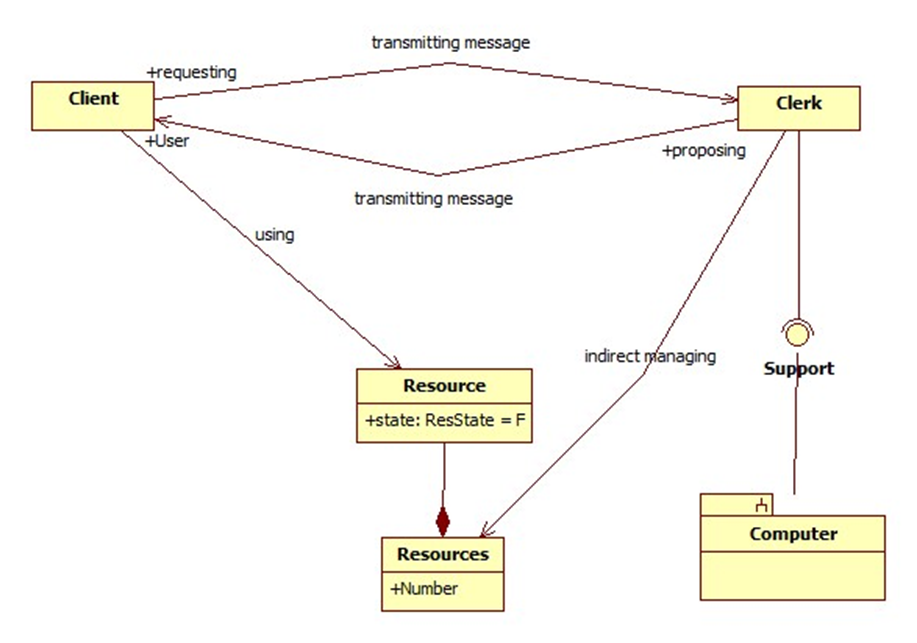
4. \*\*Набор ресурсов\*\* отвечает за управление группой ресурсов, включая просмотр и выбор конкретных ресурсов.

---

### \*\*Заключение\*\*

Данная диаграмма четко демонстрирует структуру классов и их взаимодействие в контексте задачи регистрации ресурсов. Она помогает понять, какие классы существуют в системе, какие у них свойства и методы, а также как они связаны друг с другом и с внешними акторами.

Рисунок 3. Начальная диаграмма классов проекта.

Описание для “Рисунок 3. Начальная диаграмма классов проекта.”

### Описание изображения

Данное изображение представляет собой \*\*диаграмму последовательности (Sequence Diagram)\*\*, которая описывает взаимодействие между различными участниками системы в ходе выполнения определенного процесса. Диаграмма показывает поток сообщений и взаимодействия между объектами на протяжении времени. Давайте разберем изображение поэтапно:

---

#### \*\*1. Общая структура диаграммы\*\*

- \*\*Участники (Participants)\*\*:

- Представлены вертикальными линиями слева направо.

- Каждый участник соответствует объекту или актору в системе.

- \*\*Сообщения (Messages)\*\*:

- Показаны горизонтальными стрелками, которые указывают направление передачи сообщений между участниками.

- Стрелки имеют метки, описывающие тип сообщения или действие.

---

#### \*\*2. Элементы диаграммы\*\*

##### \*\*Акторы/Объекты\*\*

1. \*\*Client\*\*:

- Изображен как прямоугольник с надписью `Client`.

- Это внешний актор, который взаимодействует с системой.

- Имеет метод: `+requesting`.

2. \*\*Clerk\*\*:

- Изображен как прямоугольник с надписью `Clerk`.

- Это внутренний участник системы, который обрабатывает запросы клиента.

- Имеет метод: `+proposing`.

3. \*\*Resource\*\*:

- Изображен как прямоугольник с надписью `Resource`.

- Представляет конкретный ресурс, который может быть использован клиентом.

- Имеет атрибут: `+state: ResState = F` (где `F` может означать "Free" — свободен).

4. \*\*Resources\*\*:

- Изображен как прямоугольник с надписью `Resources`.

- Представляет коллекцию или набор ресурсов.

- Имеет атрибут: `+Number`.

5. \*\*Computer\*\*:

- Изображен как прямоугольник с надписью `Computer`.

- Представляет компьютерную систему, которая поддерживает работу других компонентов.

6. \*\*Support\*\*:

- Изображен как круг с надписью `Support`.

- Представляет дополнительную поддержку или службу, которая может участвовать в процессе.

---

##### \*\*Связи и Сообщения\*\*

1. \*\*Связь между Client и Clerk\*\*:

- \*\*Message 1\*\*: `transmitting message`

- Клиент отправляет сообщение клерку (`+requesting`).

- \*\*Message 2\*\*: `transmitting message`

- Клерк отвечает клиенту (`+proposing`).

2. \*\*Связь между Clerk и Resource\*\*:

- \*\*Message 3\*\*: `using`

- Клерк использует ресурс для обработки запроса клиента.

3. \*\*Связь между Support и Computer\*\*:

- \*\*Message 4\*\*: `indirect managing`

- Поддержка управляет компьютерной системой косвенно.

4. \*\*Связь между Resource и Resources\*\*:

- Прямая связь без явного сообщения:

- Ресурс является частью коллекции ресурсов (`Resources`).

---

#### \*\*3. Текстовые подписи\*\*

- Внизу под диаграммой написано:

- \*\*"Рисунок [номер]. Исходная диаграмма последовательности для задачи регистрации ресурсов."\*\*

- Это указывает на то, что диаграмма относится к задаче регистрации ресурсов и является исходным вариантом.

---

#### \*\*4. Цветовая палитра\*\*

- Участники (объекты и акторы) выделены прямоугольниками желтого цвета.

- Сообщения представлены стрелками красного цвета с метками.

- Поддержка (Support) изображена как круг.

---

#### \*\*5. Общий смысл диаграммы\*\*

Диаграмма описывает следующий процесс:

1. \*\*Клиент\*\* отправляет запрос (`+requesting`) клерку через сообщение (`transmitting message`).

2. \*\*Клерк\*\* обрабатывает запрос и предлагает решение (`+proposing`) клиенту через ответное сообщение (`transmitting message`).

3. \*\*Клерк\*\* использует ресурс (`using`) для выполнения операций, связанных с запросом клиента.

4. \*\*Поддержка (Support)\*\* косвенно управляет компьютерной системой (`indirect managing`), обеспечивая ее работоспособность.

5. \*\*Ресурсы\*\* организованы в коллекцию (`Resources`), где каждый ресурс имеет состояние (`+state: ResState = F`).

---

### \*\*Заключение\*\*

Данная диаграмма последовательности четко демонстрирует поток взаимодействий между участниками системы в контексте задачи регистрации ресурсов. Она помогает понять, как информация передается между объектами и какие действия выполняются на каждом этапе процесса.

REQUEST – команда (без параметров), по которой программа выбирает первый свободный ресурс и выдаёт его номер, состояние ресурса не изменяется;

FREE – инициирует освобождение ресурса, номер освобождаемого ресурса указывается дополнительно;

OCCUPY – инициирует занятие ресурса, номер освобождаемого ресурса указывается дополнительно.

Соответствие между словарём Рис.2 и словарём Рис.3 показано в следующем перечне:

Клерк – clerk,

Клиент – customer,

Запрос – request,

Предложения – proposal,

Занятие – occupation,

Освобождение – releasing,

Согласие – agreement,

Несогласие – disagreement,

Ресурс – cell,

Набор ресурсов – cells,

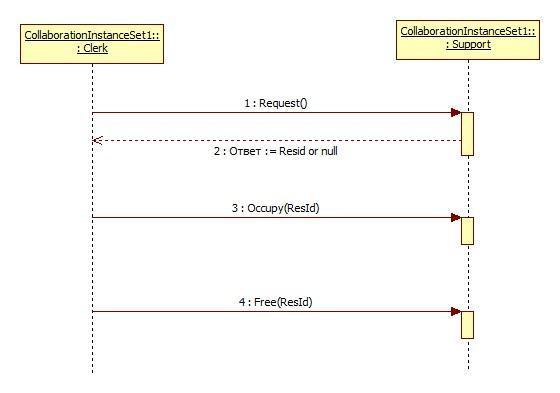
Состояние – state,

Количество ресурсов – cell number,

Выбор - selection,

Актуализация – updating.

Клерк выдаёт компьютеру команды и получает информацию об их выполнении. В модели это взаимодействие представлено интерфейсом. Команды определены как его операции и доступны в окне свойств. Взаимодействие между компьютером и клерком более детально показано на диаграмме системных взаимодействий на Рис.4.

Рисунок 4. Начальная диаграмма системных взаимодействий

Описание для “Рисунок 4. Начальная диаграмма системных взаимодействий”:  
### Описание изображения

Данное изображение представляет собой \*\*диаграмму последовательности (Sequence Diagram)\*\*, которая описывает взаимодействие между двумя участниками системы в ходе выполнения определенного процесса. Диаграмма показывает поток сообщений и их порядок выполнения на протяжении времени. Давайте разберем изображение поэтапно:

---

#### \*\*1. Общая структура диаграммы\*\*

- \*\*Участники (Participants)\*\*:

- Представлены вертикальными линиями слева направо.

- Каждый участник соответствует объекту или актору в системе.

- \*\*Сообщения (Messages)\*\*:

- Показаны горизонтальными стрелками, которые указывают направление передачи сообщений между участниками.

- Стрелки имеют метки, описывающие тип сообщения или действие.

- \*\*Лифты (Lifelines)\*\*:

- Вертикальные линии, представляющие жизненный цикл каждого участника.

- На этих линиях отмечаются события или действия, происходящие с каждым участником.

---

#### \*\*2. Элементы диаграммы\*\*

##### \*\*Акторы/Объекты\*\*

1. \*\*CollaborationInstanceSet1::Clerk\*\*:

- Изображен как прямоугольник с надписью `CollaborationInstanceSet1::Clerk`.

- Это первый участник системы, который играет роль клерка (`Clerk`).

- Лифт этого участника находится слева.

2. \*\*CollaborationInstanceSet1::Support\*\*:

- Изображен как прямоугольник с надписью `CollaborationInstanceSet1::Support`.

- Это второй участник системы, который играет роль поддержки (`Support`).

- Лифт этого участника находится справа.

---

##### \*\*Связи и Сообщения\*\*

1. \*\*Message 1: Request()\*\*:

- Отправитель: `CollaborationInstanceSet1::Clerk`.

- Получатель: `CollaborationInstanceSet1::Support`.

- Сообщение отправляется от клерка к поддержке.

- Метка: `1: Request()` — это метод или действие, которое выполняется для отправки запроса.

2. \*\*Message 2: Order := ResId or null\*\*:

- Отправитель: `CollaborationInstanceSet1::Support`.

- Получатель: `CollaborationInstanceSet1::Clerk`.

- Поддержка отправляет ответ клерку.

- Метка: `2: Order := ResId or null` — это действие, при котором поддержка возвращает идентификатор ресурса (`ResId`) или значение `null`, если ресурс недоступен.

3. \*\*Message 3: Occupy(ResId)\*\*:

- Отправитель: `CollaborationInstanceSet1::Clerk`.

- Получатель: `CollaborationInstanceSet1::Support`.

- Клерк отправляет запрос на захват ресурса (`Occupy`) с использованием идентификатора ресурса (`ResId`), полученного ранее.

- Метка: `3: Occupy(ResId)`.

4. \*\*Message 4: Free(ResId)\*\*:

- Отправитель: `CollaborationInstanceSet1::Clerk`.

- Получатель: `CollaborationInstanceSet1::Support`.

- Клерк отправляет запрос на освобождение ресурса (`Free`) с использованием идентификатора ресурса (`ResId`).

- Метка: `4: Free(ResId)`.

---

##### \*\*Временная ось\*\*

- Горизонтальные линии на диаграмме представляют время.

- Сообщения располагаются сверху вниз в порядке их выполнения:

1. `Request()`

2. `Order := ResId or null`

3. `Occupy(ResId)`

4. `Free(ResId)`

---

#### \*\*3. Текстовые подписи\*\*

- В данном изображении нет дополнительных текстовых подписей внизу, но можно предположить, что это часть документации или описание задачи, связанной с управлением ресурсами.

---

#### \*\*4. Цветовая палитра\*\*

- Участники (объекты) выделены прямоугольниками желтого цвета.

- Сообщения представлены стрелками красного цвета с метками.

- Лифты участников обозначены вертикальными линиями.

---

#### \*\*5. Общий смысл диаграммы\*\*

Диаграмма описывает следующий процесс:

1. \*\*Клерк\*\* отправляет запрос (`Request()`) поддержке для получения доступа к ресурсу.

2. \*\*Поддержка\*\* анализирует запрос и возвращает идентификатор ресурса (`ResId`) или `null`, если ресурс недоступен.

3. Если ресурс доступен, \*\*клерк\*\* отправляет запрос на захват ресурса (`Occupy(ResId)`).

4. После использования ресурса \*\*клерк\*\* отправляет запрос на освобождение ресурса (`Free(ResId)`).

---

### \*\*Заключение\*\*

Данная диаграмма последовательности четко демонстрирует взаимодействие между клерком и поддержкой в контексте управления ресурсами. Она помогает понять порядок выполнения действий и поток сообщений между участниками системы.

Диаграммы на Рис.3 и Рис.4 фактически представляют формулировку задачи регистрации ресурсов в нотации языка UML. Решение этой задачи состоит в последовательном уточнении и детализации этих диаграмм вплоть до получения текста прикладной программы на исходном языке программирования, реализующей решение задачи в среде, которая также должна быть точно специфицирована в ходе разработки.

Требования к дальнейшему проектированию и конструированию класса Program:

Выбор простейших решений,

Сценарии, выраженные последовательностями команд, независимы.

Приложение должно быть готово к исполнению сценариев от запуска до завершения.

Приложение не обязано работать в режиме “non stop”, но сеансы “работы” приложения, от запуска до завершения, не ограничиваются по длительности, равно как и количество запусков.

Преимущественно использовать средства, предоставляемые средой, а не воспроизводить их на низком уровне.

Приложение не требует каких-либо дополнительных средств или действия для подготовки к работе. В частности, внешнее хранилище данных создаётся самим приложением (в частности, не нужно отдельно создавать базу данных, но инсталлирование СУБД, конечно, нужно, если вариант приложения использует базу данных).

Из проведённого анализа и выполненных построений нетрудно заключить, что разрабатываемое приложение может быть отнесено к классу приложений, которым свойственна структура Модель – Вид – Контроллер (MVC). В соответствии с этим заключением делим приложение на следующие части:

диалоговая часть, реализующая интерфейс приложения и пользователя (соответствует “виду” структуры MVC;

модель, представляющая в приложении набор регистрируемых ресурсов;

контроллер, реализующий команды пользователя, анализируя и актуализируя состояние модели.

Такое разделение соответствует шаблону GRASP, подсказывающему декомпозицию приложения на фасадную часть, контроллер и информационный эксперт.

Нетрудно видеть, что модель вряд ли можно построить независимой от конкретных условий использования приложения. В частности, она будет зависеть от количества ресурсов. Этого соображения достаточно, чтобы ввести в структуру приложения часть, которая в диалоге с пользователем “выясняет” условия применения и выполняет необходимую настройку.

Уже сейчас, на предварительном уровне проектирования, можно выделить в модели часть, отвечающую за хранение состояния модели между запусками приложения.

Эти решения будут отражены на диаграммах UML в ходе проектирования.

1.4. Переход от постановки задачи к её решению

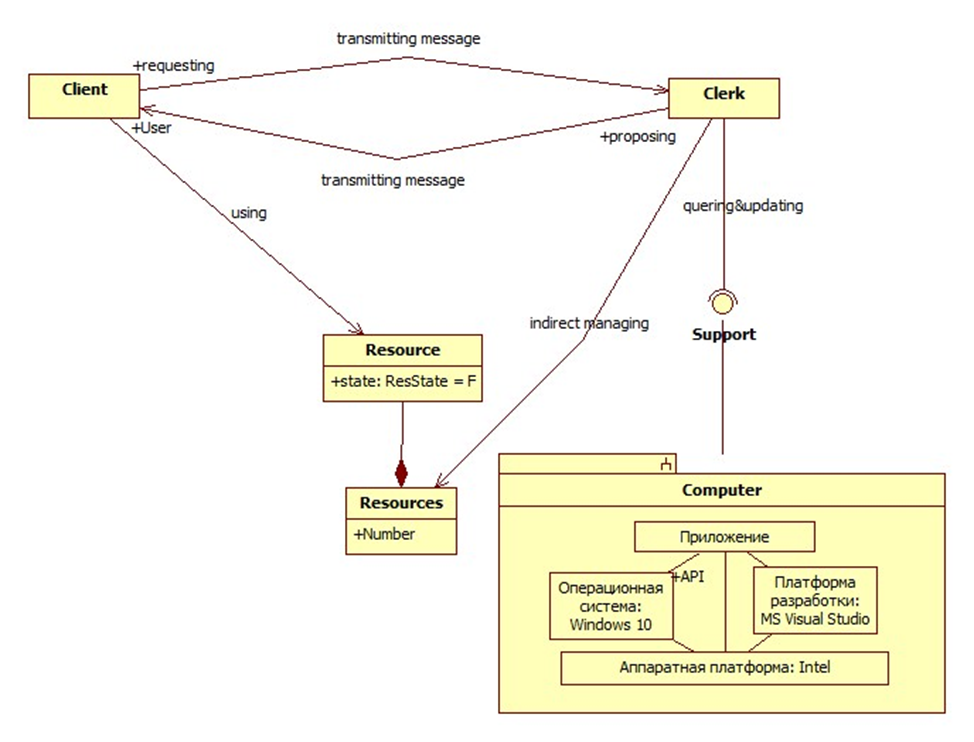
Проектирование складывается из последовательности решений.

Первые решения относятся к выбору компьютера и операционной системы. Очевидно, что решаемая задача не предъявляет никаких особых требований к производительности компьютера и его архитектуре. Задачу равно успешно можно решить в среде любой из общеизвестных операционных систем, применяемых на персональных компьютерах: Windows, Unix/Linux, Mac OS. Выбор разумно остановить на таком сочетании аппаратурной платформы и операционной системы, которое с большой вероятностью найдётся “под рукой” у читателя, то есть на сочетании персонального компьютера, оснащённого микропроцессором с архитектурой Intel Pentium или выше, и операционной системы Microsoft Windows уровня не ниже XP. В этой связи следует заметить, что, начиная с этого момента, читателю полезно все рассмотрения и построения данного пособия повторять на имеющемся в его распоряжении компьютере.

Следующее решение – выбор интегральной среды разработки (IDE). Предлагается остановиться на Microsoft Visual Studio, которая включает в себя ряд технологий, интересных для выполняемого проекта. Показанные далее примеры программ составлены и отлажены в Visual Studio издания 13, но, за единичными исключениями, они выполнимы и в Visual Studio издания не ниже 8. С не меньшим успехом можно было бы развивать проект в среде Delphi, однако новейшие версии этой IDE вряд ли можно считать доступными для читателя-студента.

Общий подход при принятии последующих решений: предпочтение отдаётся простейшим решениям, более сложные могут быть реализованы дополнительно.

Эти решения отражены на диаграмме классов, представленной на Рис.5. Она является развитием модели, показанной на Рис.4.

Рисунок 5. Диаграмма классов с первыми проектными решениями.

Описание для “Рисунок 5. Диаграмма классов с первыми проектными решениями”:

### Описание изображения

Данное изображение представляет собой \*\*диаграмму последовательности (Sequence Diagram)\*\*, которая описывает взаимодействие между различными участниками системы в ходе выполнения определенного процесса. Диаграмма показывает поток сообщений и взаимодействия между объектами на протяжении времени. Давайте разберем изображение поэтапно:

---

#### \*\*1. Общая структура диаграммы\*\*

- \*\*Участники (Participants)\*\*:

- Представлены вертикальными линиями слева направо.

- Каждый участник соответствует объекту или актору в системе.

- \*\*Сообщения (Messages)\*\*:

- Показаны горизонтальными стрелками, которые указывают направление передачи сообщений между участниками.

- Стрелки имеют метки, описывающие тип сообщения или действие.

---

#### \*\*2. Элементы диаграммы\*\*

##### \*\*Акторы/Объекты\*\*

1. \*\*Client\*\*:

- Изображен как прямоугольник с надписью `Client`.

- Это внешний актор, который взаимодействует с системой.

- Имеет метод: `+requesting`.

2. \*\*Clerk\*\*:

- Изображен как прямоугольник с надписью `Clerk`.

- Это внутренний участник системы, который обрабатывает запросы клиента.

- Имеет метод: `+proposing`.

3. \*\*Resource\*\*:

- Изображен как прямоугольник с надписью `Resource`.

- Представляет конкретный ресурс, который может быть использован клиентом.

- Имеет атрибут: `+state: ResState = F` (где `F` может означать "Free" — свободен).

4. \*\*Resources\*\*:

- Изображен как прямоугольник с надписью `Resources`.

- Представляет коллекцию или набор ресурсов.

- Имеет атрибут: `+Number`.

5. \*\*Computer\*\*:

- Изображен как прямоугольник с надписью `Computer`.

- Представляет компьютерную систему, которая поддерживает работу других компонентов.

- Внутри компьютера указаны следующие детали:

- \*\*Приложение\*\*: Приложение, работающее на компьютере.

- \*\*Операционная система\*\*: Windows 10.

- \*\*Платформа разработки\*\*: MS Visual Studio.

- \*\*Аппаратная платформа\*\*: Intel.

6. \*\*Support\*\*:

- Изображен как круг с надписью `Support`.

- Представляет дополнительную поддержку или службу, которая может участвовать в процессе.

---

##### \*\*Связи и Сообщения\*\*

1. \*\*Связь между Client и Clerk\*\*:

- \*\*Message 1\*\*: `transmitting message`

- Клиент отправляет сообщение клерку (`+requesting`).

- \*\*Message 2\*\*: `transmitting message`

- Клерк отвечает клиенту (`+proposing`).

2. \*\*Связь между Clerk и Resource\*\*:

- \*\*Message 3\*\*: `using`

- Клерк использует ресурс для обработки запроса клиента.

3. \*\*Связь между Support и Computer\*\*:

- \*\*Message 4\*\*: `indirect managing`

- Поддержка управляет компьютерной системой косвенно.

4. \*\*Связь между Resource и Resources\*\*:

- Прямая связь без явного сообщения:

- Ресурс является частью коллекции ресурсов (`Resources`).

5. \*\*Связь между Clerk и Support\*\*:

- \*\*Message 5\*\*: `querying&updating`

- Клерк отправляет запросы и обновления поддержке.

---

#### \*\*3. Текстовые подписи\*\*

- Внизу под диаграммой написано:

- \*\*"Рисунок [номер]. Исходная диаграмма последовательности для задачи регистрации ресурсов."\*\*

- Это указывает на то, что диаграмма относится к задаче регистрации ресурсов и является исходным вариантом.

---

#### \*\*4. Цветовая палитра\*\*

- Участники (объекты и акторы) выделены прямоугольниками желтого цвета.

- Сообщения представлены стрелками красного цвета с метками.

- Поддержка (Support) изображена как круг.

---

#### \*\*5. Общий смысл диаграммы\*\*

Диаграмма описывает следующий процесс:

1. \*\*Клиент\*\* отправляет запрос (`+requesting`) клерку через сообщение (`transmitting message`).

2. \*\*Клерк\*\* обрабатывает запрос и предлагает решение (`+proposing`) клиенту через ответное сообщение (`transmitting message`).

3. \*\*Клерк\*\* использует ресурс (`using`) для выполнения операций, связанных с запросом клиента.

4. \*\*Поддержка (Support)\*\* косвенно управляет компьютерной системой (`indirect managing`), обеспечивая ее работоспособность.

5. \*\*Ресурсы\*\* организованы в коллекцию (`Resources`), где каждый ресурс имеет состояние (`+state: ResState = F`).

6. \*\*Клерк\*\* также взаимодействует с поддержкой через запросы и обновления (`querying&updating`).

---

#### \*\*6. Особенности компьютерной системы\*\*

Внутри блока `Computer` подробно описаны характеристики системы:

- \*\*Приложение\*\*: Базовое приложение, которое работает на компьютере.

- \*\*Операционная система\*\*: Windows 10 — это платформа, на которой выполняется приложение.

- \*\*Платформа разработки\*\*: MS Visual Studio — инструментарий, используемый для создания и управления приложением.

- \*\*Аппаратная платформа\*\*: Intel — процессорная архитектура, на которой работает система.

---

### \*\*Заключение\*\*

Данная диаграмма последовательности четко демонстрирует поток взаимодействий между участниками системы в контексте задачи регистрации ресурсов. Она помогает понять, как информация передается между объектами и какие действия выполняются на каждом этапе процесса.

Следующее решение – выбор языка программирования и шаблона проекта. Простейший шаблон проекта – “консольный” – оставляет две возможности для выбора языка: C++ и C#. Далее он реализуется в виде “консольного приложения” на обоих языках, но в качестве основного принимается проект на языке C#, поскольку он позволяет, не меняя языка, перейти к более богатому возможностями проекту – проекту типа WindowsForms.

Протокол взаимодействия пользователя и программы определён на диаграмме последовательностей на Рис.4. Он должен реализовываться в течение всего промежутка времени от запуска программы до завершения её как приложения Windows. В ответ на введённую команду программа выдаёт сообщение. После запуска и после выдачи ответного сообщения программа должна быть готова к вводу команды.

Для обеспечения интерфейса между приложением и пользователем консольное приложение предоставляет консольное окно, открытое с момента запуска приложения до момента его завершения, а для ввода – клавиатуру. Пользователь вводит информацию с клавиатуры при наличии запроса в консольном окне приложения. При наличии запроса, пока ввод не закончен нажатием клавиши ENTER, приложение как процесс находится в состоянии ожидания.

Класс Console пространства имён System содержит методы, реализующие вывод из приложения в окно, ввод с клавиатуры и синхронизацию действий пользователя и приложения. Метод WriteLine выводит строку, метод ReadLine переводит запрос ввода на следующую строку и ждёт до тех пор, пока пользователь завершил ввод. При этом метод возвращает введённую информацию, позволяя приложению её обрабатывать.

Консольное приложение, представленное в шаблоне, содержит единственный модуль на языке C#, имеющий следующий вид:

using System;

namespace Cons2ReAr

{

class Program

{

static void Main(string[] args)

{

}

}

}

При запуске приложения первым получит управление метод Main. Соответственно, дальнейшие шаги по конструированию приложения можно планировать как формирование тела метода Main и, в случае необходимости, других компонентов класса Program.

Разработка класса Program должна обеспечить реализацию взаимодействий, показанных на Рис.4. В консольных проектах диалог между компьютером и работником (в нашем случае – клерком) организуется исключительно в виде обмена сообщениями, часть из которых может задерживаться на экране дисплея.

1. “Консольное” решение задачи регистрации ресурсов
   1. Организация диалога с пользователем

Методы класса Console устроены так, что готовность консольного приложения к вводу команды с клавиатуры может быть реализована вызовом метода ReadKey или ReadLine. Поскольку во многих случаях вводить понятнее более одной литеры, из этих двух методов будет использоваться ReadLine. Перед ней целесообразно выводить сообщение, напоминающее пользователю, какую информацию ему предлагается вводить.

Пусть вводимая команда присваивается переменной command. Тогда элемент интерфейса “ввод команды – получение ответного сообщения” начинается следующей парой предложений, помеченной для удобства ссылок на них:

L: Console.WriteLine("Введите команду:");

Command = Console.ReadLine();

Затем в зависимости от того, какая команда введена, выполняется та или иная обработка и выдаётся соответствующее сообщение. После этого, не теряя общности программы, можно вернуться к метке L. То есть, диалог реализуется циклом, который действует на протяжении всего сеанса работы приложения. Простейший его вариант может быть представлен так:

do

{

Console.WriteLine("Введите команду:");

Command = Console.ReadLine();

/\* анализ и выполнение команды \*/

}

while(true);

Для удобства пользователя расширим спецификацию, введя два новых положения:

- Если в качестве команды вводится пустая строка, то приложение завершается.

- Анализ и выполнение введённой команды не изменяют значения переменной Command.

Тогда фрагмент приложения, реализующий диалог с пользователем, будет выглядеть так:

string Command;

do

{

Console.WriteLine("Введите команду:");

Command = Console.ReadLine();

/\* анализ и выполнение команды \*/

}

while(Command != "");

Положим, что этот фрагмент помещён в тело Main класса Program, а команды исполняются методами того же класса, заголовок которых совпадает с заголовком команд на диаграмме Рис. 4.

Тогда приложение приобретёт следующий вид:

using System;

namespace Cons2ReAr

{

class Program

{

static string Request(){}

static void Occupy(string cn){}

static void Free(string cn){}

static void Main(string[] args)

{

string Command;

do

{

Console.WriteLine("Введите команду:");

Command = Console.ReadLine();

if (Command == "REQUEST") Console.WriteLine(Request());

if (Command == "OCCUPY")

{

Console.WriteLine("Введите номер ячейки:");

Occupy(Console.ReadLine());

Console.WriteLine("Ячейка стала занята.");

};

if (Command == "FREE")

{

Console.WriteLine("Введите номер ячейки:");

Free(Console.ReadLine());

Console.WriteLine("Ячейка освобождена.");

};

}

while(Command != "");

}

}

}

Этот текст успешно компилируется и запускается на выполнение, хотя тела методов Request, Occupy и Free пусты и при их вызове никакие действия не выполняются. Тем не менее, начиная с этого текста, можно проводить автономную отладку диалоговой части разрабатываемого приложения. Для того, чтобы считать эту часть полностью написанной, необходимо добавить операторы обработки особых случаев и соответствующие классы согласно спецификации. В результате будет получен следующий текст;

using System;

namespace ResRegV1cons

{

class ResAreBusy : Exception { }

class ResIdInvalid : Exception { }

class UnRecommended : Exception { }

class ResIsBusy : Exception { }

class ResWasFree : Exception { }

class Program

{

static string Request(){}

static void Occupy(string cn){}

static void Free(string cn){}

static void Main(string[] args)

{

string Command;

do

{

Console.WriteLine("Введите команду:");

Command = Console.ReadLine();

Command = Command.ToUpper();

try

{

if (Command == "REQUEST") Console.WriteLine(Request());

if (Command == "OCCUPY")

{

Console.WriteLine("Введите номер ячейки:");

Occupy(Console.ReadLine());

Console.WriteLine("Ячейка стала занята.");

};

if (Command == "FREE")

{

Console.WriteLine("Введите номер ячейки:");

Free(Console.ReadLine());

Console.WriteLine("Ячейка освобождена.");

};

}

catch (OverflowException) { Console.WriteLine("Такой ячейки нет."); }

catch (FormatException) { Console.WriteLine("Такой ячейки нет."); }

catch (ResIdInvalid) { Console.WriteLine("Такой ячейки нет."); }

catch (ResWasFree) { Console.WriteLine("Ячейка была свободна."); }

catch (ResAreBusy) { Console.WriteLine("Все ячейки заняты."); }

catch (ResIsBusy) { Console.WriteLine("Ячейка уже занята."); }

}

while(Command != "");

}

}

Тело диалогового цикла взято погружено в try-блок, и особые случаи, распознаваемые при выполнении команд, будут обрабатываться в операторах catch. Обработка состоит в выдаче соответствующего сообщения, после чего приложение продолжит функционировать.

Попутно спецификация расширена разрешением вводить команды как строчными, так и заглавными буквами в любом сочетании. Для этого в тело диалогового цикла перед анализом команды введено предложение, переводящее введённую строку в верхний регистр (заглавные буквы).

Несистемные особые случаи должны возбуждаться операторами **throw** в методах, исполняющих команды.

Выполненное построение можно расценивать как первую итерацию в ходе выполнения данного проекта.

Вместо использования механизма try … catch для обработки несистемных особых случаев можно было бы использовать возвращаемый параметр методов обработки команд, и на некоторые обусловленные значения реагировать в теле диалогового цикла выдачей соответствующего сообщения. Однако этот вариант отклоняется как решение более низкого уровня, которое, к тому же, заметно усложняет текст диалоговой части.

Анализ команды реализуется последовательным сравнением введённой строки с эталонными названиями команд. При совпадении вызываются соответствующие методы. Для обеспечения отладки диалога эти, пока ещё не разработанные методы заменяются отладочными.

if (Command == "REQUEST") Console.WriteLine(Request());

if (Command == "OCCUPY")

{

Console.WriteLine("Введите номер ячейки:");

Occupy(Console.ReadLine());

};

if (Command == "FREE")

{

Console.WriteLine("Введите номер ячейки:");

Free(Console.ReadLine());

};

Диаграмма последовательностей на Рис.6 описывает диалог между классом Program и классом Console.

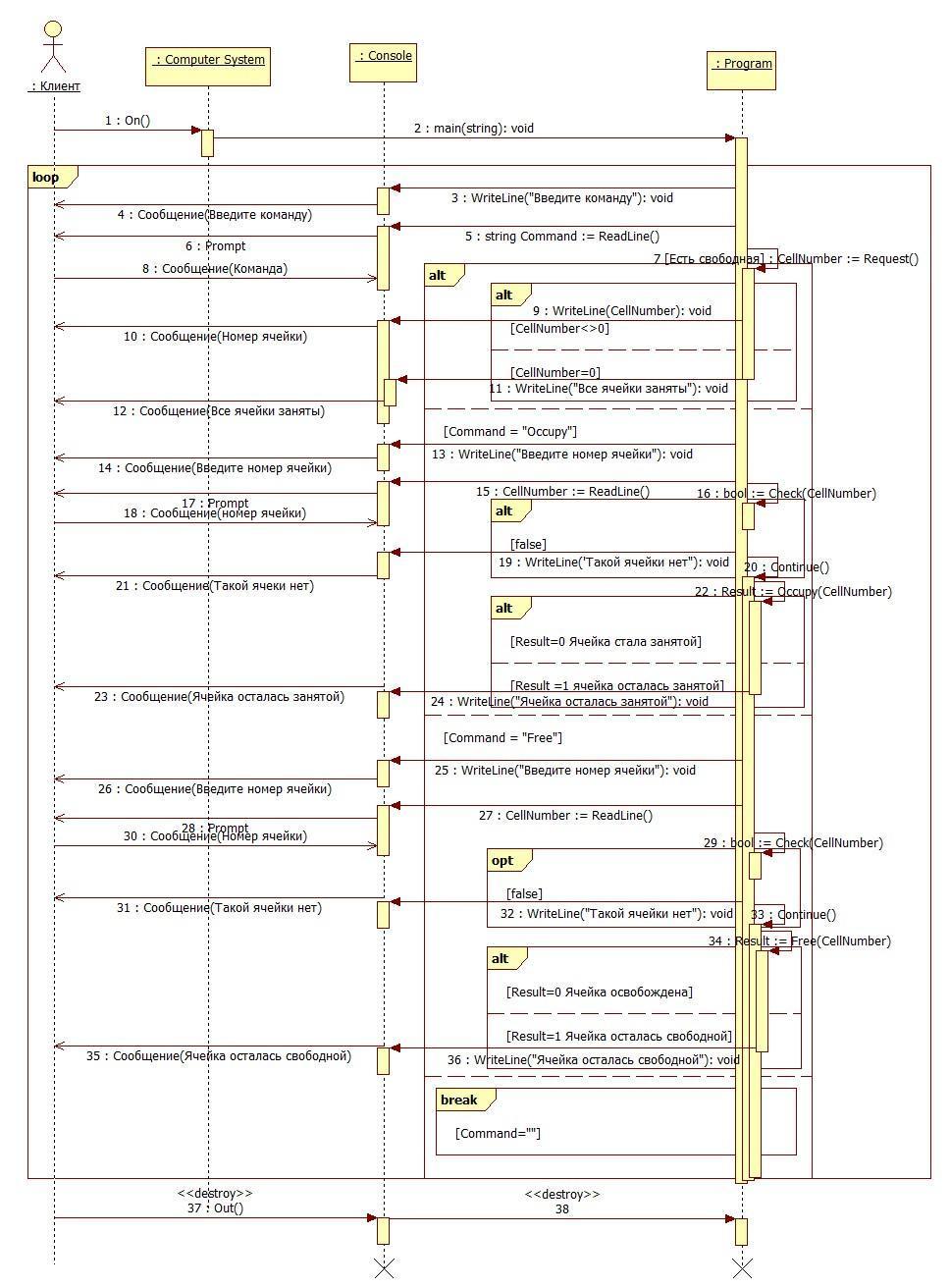


Рисунок 6. Диаграмма последовательностей консольного диалога.

Описание для “Рисунок 6. Диаграмма последовательностей консольного диалога.”:

### Описание изображения

Данное изображение представляет собой \*\*диаграмму последовательности (Sequence Diagram)\*\*, которая описывает взаимодействие между различными участниками системы в ходе выполнения определенного процесса. Диаграмма показывает поток сообщений и их порядок выполнения на протяжении времени. Давайте разберем изображение поэтапно:

---

#### \*\*1. Общая структура диаграммы\*\*

- \*\*Участники (Participants)\*\*:

- Представлены вертикальными линиями слева направо.

- Каждый участник соответствует объекту или актору в системе.

- \*\*Сообщения (Messages)\*\*:

- Показаны горизонтальными стрелками, которые указывают направление передачи сообщений между участниками.

- Стрелки имеют метки, описывающие тип сообщения или действие.

- \*\*Лифты (Lifelines)\*\*:

- Вертикальные линии, представляющие жизненный цикл каждого участника.

- На этих линиях отмечаются события или действия, происходящие с каждым участником.

---

#### \*\*2. Элементы диаграммы\*\*

##### \*\*Акторы/Объекты\*\*

1. \*\*Клиент\*\*:

- Изображен как человеческая фигурка слева.

- Это внешний актор, который взаимодействует с системой через консоль.

2. \*\*Computer System\*\*:

- Изображен как прямоугольник с надписью `: Computer System`.

- Представляет компьютерную систему, которая управляет процессом.

3. \*\*Console\*\*:

- Изображен как прямоугольник с надписью `: Console`.

- Представляет консольный интерфейс для взаимодействия с пользователем.

4. \*\*Program\*\*:

- Изображен как прямоугольник с надписью `: Program`.

- Представляет программу, которая реализует основной функционал системы.

---

##### \*\*Связи и Сообщения\*\*

Диаграмма содержит несколько ключевых этапов взаимодействия:

1. \*\*Инициализация системы\*\*:

- \*\*Message 1\*\*: `On()`

- Клиент отправляет команду `On()` компьютерной системе для запуска программы.

2. \*\*Запуск программы\*\*:

- \*\*Message 2\*\*: `main(string): void`

- Компьютерная система запускает метод `main` программы.

3. \*\*Взаимодействие с пользователем\*\*:

- \*\*Message 3\*\*: `WriteLine("Введите команду")`

- Программа выводит приветственное сообщение в консоль, приглашая пользователя ввести команду.

- \*\*Message 4\*\*: `Prompt`

- Консоль ожидает ввода от пользователя.

- \*\*Message 5\*\*: `string Command := ReadLine()`

- Программа читает введенную команду от пользователя.

4. \*\*Обработка команды\*\*:

- После получения команды программа начинает обрабатывать различные варианты действий:

- \*\*Если команда "Occupy"\*\*:

- \*\*Message 6\*\*: `CellNumber := Request()`

- Программа запрашивает номер ячейки у пользователя.

- \*\*Message 7\*\*: `Check(CellNumber)`

- Программа проверяет доступность указанной ячейки.

- Если ячейка свободна:

- \*\*Message 8\*\*: `Occupy(CellNumber)`

- Программа занимает указанную ячейку.

- Если ячейка занята:

- \*\*Message 9\*\*: `WriteLine("Такой ячейки нет")`

- Программа выводит сообщение об ошибке.

- \*\*Если команда "Free"\*\*:

- \*\*Message 10\*\*: `CellNumber := Request()`

- Программа запрашивает номер ячейки у пользователя.

- \*\*Message 11\*\*: `Check(CellNumber)`

- Программа проверяет состояние указанной ячейки.

- Если ячейка занята:

- \*\*Message 12\*\*: `Free(CellNumber)`

- Программа освобождает указанную ячейку.

- Если ячейка уже свободна:

- \*\*Message 13\*\*: `WriteLine("Такой ячейки нет")`

- Программа выводит сообщение об ошибке.

5. \*\*Циклическое взаимодействие\*\*:

- Процесс повторяется в цикле (`loop`), пока пользователь не завершит работу программы.

6. \*\*Завершение работы\*\*:

- Когда пользователь вводит пустую строку или специальную команду для выхода:

- \*\*Message 37\*\*: `Out()`

- Программа завершает работу.

- \*\*Message 38\*\*: `<destroy>`

- Все объекты уничтожаются, завершая жизненный цикл системы.

---

##### \*\*Альтернативные ветви (Alternative Branches)\*\*

- В некоторых случаях диаграмма показывает альтернативные пути выполнения:

- Например, при проверке доступности ячейки (`Check(CellNumber)`):

- Если ячейка свободна (`Result = 1`), выполняется дальнейшая обработка.

- Если ячейка занята (`Result = 0`), выводится соответствующее сообщение об ошибке.

---

##### \*\*Опциональные ветви (Optional Branches)\*\*

- Некоторые шаги могут быть опциональными:

- Например, если пользователь вводит некорректную команду, программа может игнорировать ее или вывести сообщение об ошибке.

---

##### \*\*Циклы\*\*

- Диаграмма использует оператор `loop`, чтобы показать, что процесс взаимодействия с пользователем повторяется до тех пор, пока не будет введена команда выхода.

---

##### \*\*Условные ветви\*\*

- В нескольких местах используются условные ветви (`alt`), которые показывают, какие действия выполняются в зависимости от результатов проверок:

- Например, проверка состояния ячейки (`Check(CellNumber)`).

---

#### \*\*3. Текстовые подписи\*\*

- Внизу под диаграммой написано:

- \*\*"Рисунок [номер]. Исходная диаграмма последовательности для задачи управления ресурсами."\*\*

- Это указывает на то, что диаграмма относится к задаче управления ресурсами и является исходным вариантом.

---

#### \*\*4. Цветовая палитра\*\*

- Участники (объекты и акторы) выделены прямоугольниками желтого цвета.

- Сообщения представлены стрелками красного цвета с метками.

- Альтернативные ветви (`alt`) и опциональные ветви (`opt`) помечены соответствующими ключевыми словами.

- Циклы помечены оператором `loop`.

---

#### \*\*5. Общий смысл диаграммы\*\*

Диаграмма описывает следующий процесс:

1. \*\*Клиент\*\* запускает компьютерную систему.

2. \*\*Программа\*\* начинает работу и предлагает пользователю ввести команду.

3. Пользователь вводит команду, например, `"Occupy"` или `"Free"`.

4. Программа обрабатывает команду:

- Если команда `"Occupy"`, программа запрашивает номер ячейки, проверяет ее доступность и занимает ячейку, если она свободна.

- Если команда `"Free"`, программа запрашивает номер ячейки, проверяет ее состояние и освобождает ячейку, если она занята.

5. Процесс повторяется в цикле до тех пор, пока пользователь не введет команду выхода.

6. После завершения работы программа уничтожает все объекты и завершает работу.

---

### \*\*Заключение\*\*

Данная диаграмма последовательности четко демонстрирует поток взаимодействий между участниками системы в контексте задачи управления ресурсами. Она помогает понять, как информация передается между объектами и какие действия выполняются на каждом этапе процесса.

Все принятые решения позволяют трансформировать диаграмму классов проекта, как показано на Рис.7. На трансформированной диаграмме показан также клерк (класс clerk), остальные классы и их связи, которые находились на диаграмме классов на Рис.3, далее показываться не будут, поскольку разработка их не затрагивает.

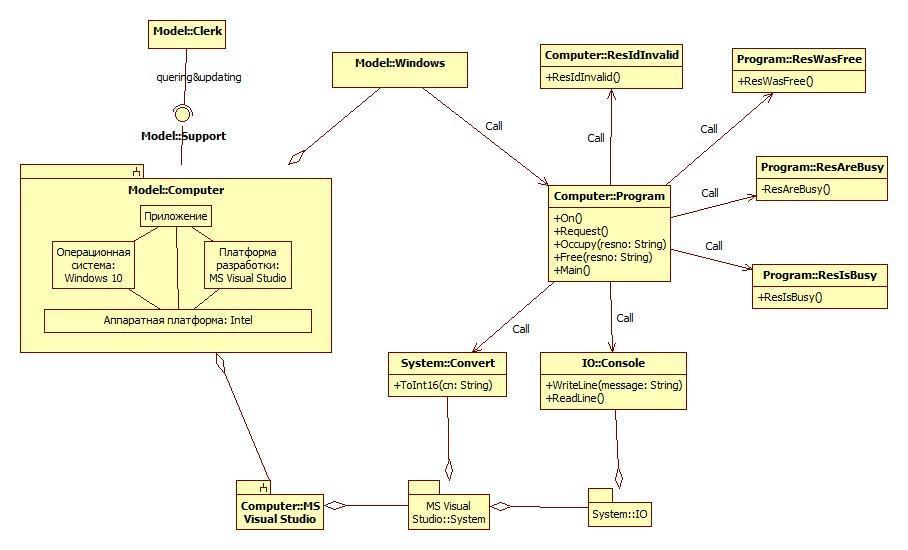


Рисунок 7. Диаграмма классов с обеспечением консольного диалога.

Описание для “Рисунок 7. Диаграмма классов с обеспечением консольного диалога.”:

### Описание изображения

Данное изображение представляет собой \*\*диаграмму компонентов (Component Diagram)\*\*, которая описывает структуру системы и взаимодействие между ее компонентами. Диаграмма показывает как компоненты связаны друг с другом через интерфейсы или вызовы методов. Давайте разберем изображение поэтапно:

---

#### \*\*1. Общая структура диаграммы\*\*

- \*\*Компоненты (Components)\*\*:

- Представлены прямоугольниками с закругленными углами.

- Каждый компонент соответствует определенной части системы или библиотеке.

- \*\*Интерфейсы (Interfaces)\*\*:

- Показывают точки взаимодействия между компонентами.

- \*\*Связи\*\*:

- Показаны стрелками, указывающими направление взаимодействия между компонентами.

---

#### \*\*2. Элементы диаграммы\*\*

##### \*\*Компоненты\*\*

1. \*\*Model:Clerk\*\*:

- Изображен как прямоугольник с надписью `Model:Clerk`.

- Представляет модель клерка, который может выполнять запросы и обновления (`querying&updating`).

2. \*\*Model:Support\*\*:

- Изображен как круг с надписью `Model:Support`.

- Представляет модель поддержки, которая взаимодействует с другими компонентами.

3. \*\*Model:Windows\*\*:

- Изображен как прямоугольник с надписью `Model:Windows`.

- Представляет модель операционной системы Windows.

4. \*\*Computer::ResIdInvalid\*\*:

- Изображен как прямоугольник с надписью `Computer::ResIdInvalid`.

- Представляет компонент, который проверяет, является ли идентификатор ресурса (`ResId`) недопустимым.

- Метод: `+ResIdInvalid()`.

5. \*\*Program::ResWasFree\*\*:

- Изображен как прямоугольник с надписью `Program::ResWasFree`.

- Представляет компонент, который отслеживает состояние ресурса (был ли он свободен).

- Метод: `+ResWasFree()`.

6. \*\*Program::ResAreBusy\*\*:

- Изображен как прямоугольник с надписью `Program::ResAreBusy`.

- Представляет компонент, который проверяет, заняты ли все ресурсы.

- Метод: `+ResAreBusy()`.

7. \*\*Program::ResIsBusy\*\*:

- Изображен как прямоугольник с надписью `Program::ResIsBusy`.

- Представляет компонент, который проверяет, занят ли конкретный ресурс.

- Методы:

- `+ResIsBusy()`

- `-ResIsBusy()` (возможно, это метод для освобождения ресурса).

8. \*\*Computer::Program\*\*:

- Изображен как прямоугольник с надписью `Computer::Program`.

- Представляет основную программу компьютера.

- Методы:

- `+On()`: Запуск программы.

- `+Request()`: Метод для отправки запроса.

- `+Occupy(resno: String)`: Занять ресурс с указанным идентификатором.

- `+Free(resno: String)`: Освободить ресурс с указанным идентификатором.

- `+Main()`: Главный метод программы.

9. \*\*System::Convert\*\*:

- Изображен как прямоугольник с надписью `System::Convert`.

- Представляет системную библиотеку для конвертации данных.

- Метод: `+ToInt16(cv: String)` — преобразование строки в целое число типа `Int16`.

10. \*\*IO::Console\*\*:

- Изображен как прямоугольник с надписью `IO::Console`.

- Представляет компонент для работы с консолью.

- Методы:

- `+WriteLine(message: String)`: Вывод текста на консоль.

- `+ReadLine()`: Чтение строки с консоли.

11. \*\*Computer::MS Visual Studio\*\*:

- Изображен как прямоугольник с надписью `Computer::MS Visual Studio`.

- Представляет интегрированную среду разработки (IDE) Microsoft Visual Studio.

12. \*\*MS Visual Studio::System\*\*:

- Изображен как прямоугольник с надписью `MS Visual Studio::System`.

- Представляет системные компоненты IDE MS Visual Studio.

13. \*\*System::IO\*\*:

- Изображен как прямоугольник с надписью `System::IO`.

- Представляет системную библиотеку для ввода-вывода.

---

##### \*\*Связи между компонентами\*\*

1. \*\*Model:Clerk → Model:Support\*\*:

- Связь называется `querying&updating`.

- Это указывает на то, что модель клерка взаимодействует с моделью поддержки для выполнения запросов и обновлений.

2. \*\*Model:Windows → Computer::ResIdInvalid\*\*:

- Связь называется `Call`.

- Это указывает на то, что модель Windows вызывает метод `ResIdInvalid()` в компоненте `Computer::ResIdInvalid`.

3. \*\*Computer::ResIdInvalid → Program::ResWasFree\*\*:

- Связь называется `Call`.

- Это указывает на то, что компонент `Computer::ResIdInvalid` вызывает метод `ResWasFree()` в компоненте `Program::ResWasFree`.

4. \*\*Computer::Program → Program::ResAreBusy\*\*:

- Связь называется `Call`.

- Это указывает на то, что программа вызывает метод `ResAreBusy()` в компоненте `Program::ResAreBusy`.

5. \*\*Computer::Program → Program::ResIsBusy\*\*:

- Связь называется `Call`.

- Это указывает на то, что программа вызывает методы `ResIsBusy()` и `-ResIsBusy()` в компоненте `Program::ResIsBusy`.

6. \*\*Computer::Program → System::Convert\*\*:

- Связь называется `Call`.

- Это указывает на то, что программа использует метод `ToInt16(cv: String)` из компонента `System::Convert`.

7. \*\*Computer::Program → IO::Console\*\*:

- Связь называется `Call`.

- Это указывает на то, что программа использует методы `WriteLine(message: String)` и `ReadLine()` из компонента `IO::Console`.

8. \*\*Computer::MS Visual Studio → MS Visual Studio::System\*\*:

- Связь показывает, что среда разработки MS Visual Studio использует системные компоненты IDE.

9. \*\*MS Visual Studio::System → System::IO\*\*:

- Связь показывает, что системные компоненты IDE используют общую системную библиотеку ввода-вывода (`System::IO`).

---

#### \*\*3. Текстовые подписи\*\*

- Внизу под диаграммой написано:

- \*\*"Рисунок [номер]. Исходная диаграмма компонентов для задачи управления ресурсами."\*\*

- Это указывает на то, что диаграмма относится к задаче управления ресурсами и является исходным вариантом.

---

#### \*\*4. Цветовая палитра\*\*

- Компоненты выделены прямоугольниками желтого цвета.

- Интерфейсы и связи представлены стрелками красного цвета с метками.

---

#### \*\*5. Общий смысл диаграммы\*\*

Диаграмма описывает следующее:

1. \*\*Модель клерка\*\* взаимодействует с моделью поддержки для выполнения запросов и обновлений.

2. \*\*Модель Windows\*\* взаимодействует с компонентами, которые проверяют состояние ресурсов.

3. \*\*Основная программа (`Computer::Program`)\*\* управляет ресурсами, используя методы для их занятия и освобождения.

4. \*\*Программа использует различные компоненты\*\*:

- `System::Convert` для конвертации данных.

- `IO::Console` для взаимодействия с пользователем через консоль.

5. \*\*Среда разработки MS Visual Studio\*\* связана с системными компонентами IDE и использует общую системную библиотеку ввода-вывода.

---

### \*\*Заключение\*\*

Данная диаграмма компонентов четко демонстрирует структуру системы и взаимодействие между ее компонентами в контексте задачи управления ресурсами. Она помогает понять, какие компоненты существуют в системе, как они связаны и какие функции выполняют.

* 1. Реализация бизнес-логики и выбор модели данных

Реализация бизнес-логики и выбор модели данных оказываются тесно связанными. Согласно спецификации набор ресурсов моделируется множеством объектов – моделями ресурсов. Команды Occupy и Free изменяют состояние ресурса, указанного параметром команды, а команда Request – выдаёт один из “свободных” ресурсов, выбранных по некоторому правилу, которое спецификацией не установлено.

Считаем очевидным, что все три команды могут быть реализованы перебором множества моделей ресурсов, причём команды Occupy и Free выбирают ресурс, сравнивая его идентификатор со своим входным параметром, а Request выбирает, например, первый попавшийся “свободный”. Перебор в командах Occupy и Free может быть заменён прямым доступом, если он обеспечен на множестве моделей ресурсов. Эти рассуждения выявляют два независящих от платформы подхода к реализации бизнес-логики разрабатываемого приложения, дальнейшая конкретизация которых требует рассмотрения возможностей выбранной платформы и среды проектирования.

В Microsoft Visual Studio прямой доступ по индексу или набору индексов обеспечен для встроенного в C# массива и, соответственно, класса Array в пространстве имён System. Это лишь частное представление множества, частное, прежде всего в двух отношениях:

* ёмкость ограничивается при инициировании,
* доступ к элементу по номеру.

Ресурсы пространства имён System.Collections.Generic снимают первое из указанных ограничений и ослабляют второе.

Ресурсы пространствах имён System.Collections.Generic обеспечивают в программе сущности, соответствующие множествам как в общеязыковом, неформальном понимании этого слова, так и в строгом математическом.

Интерфейс ICollection<T> можно расценивать как центральный по универсальности и функциональной наполненности.

Методы Add, Remove и Contains c параметром (Object value), а также Clear() обеспечивают чисто теоретико-множественные операции с множеством (“коллекцией”). Мощность (количество элементов) выражается значением свойства Count.

Перебор коллекции обеспечивается методом GetEnumerator, унаследованным от интерфейса IEnumerable<T> и реализующим интерфейс IEnumerator<T>.

Реализации интерфейса ICollection<T> дополняют его и специализируют. Основные реализации:

классом LinkedList<T>,

классом HashSet<T>, причём этот класс реализует не только непосредственно ICollection<T>, а IHashSet<T>, потомок ICollection<T>;

классом List<T>, причём этот класс реализует не только непосредственно ICollection<T>, а IList<T >, потомок ICollection<T>. Интерфейс IList<T> обеспечивает индексирование методами IndexOf, Insert и RemoveAt, а также свойством Item. Класс List<T> содержит также методы, обеспечивающие поиск по заданному предикату и сортировку.

Интерфейс ICollection<T> наследует интерфейс IEnumerable<T> (и IEnumerable), представляющий методы, которые обеспечивают функциональность, похожую на функциональность оператора Select языка SQL. Эти методы (более 130 методов) реализуются в статическом классе Enumerable, находящемся в пространстве имён System.Linq. Эти методы вызываются в порядке расширения интерфейса ICollection<T> и других.

Для представления можества можно также использовать класс Stack.

Приведённы обзор обосновывает в качестве простейшей следующую реализацию модели в виде массива:

static string[] vRes\_s;//Модель набора ресурсов

и команд Occupy и Free:

static void Occupy(string cn)//cn –идентификатор (номер) ресурса

{

vRes\_s[Convert.ToInt16(cn) - 1] = "B";//Ресурс становится “свободным”

}

static void Free(string cn) //cn – идентификатор (номер) ресурса

{

vRes\_s[Convert.ToInt16(cn) - 1] = "F";//Ресурс становится “занятым”

}

Спецификация уточняется: идентификатором ресурса служит его номер от 1. Поэтому значение входного параметра уменьшается на единицу, поскольку элементы массива нумеруются с 0.

Ресурс моделируется литерой (“B” или “F”), а идентификатор, определённый в спецификации как поле, в модели представлен неявно – номером элемента массива. Таким образом реализация оказывается проще спецификации.

Реализация команды Request возвращает номер первого свободного элемента массива, попавшегося при его переборе от начала.

static string Request()

{

for(int i=0;i<vRes\_s.Length;i++)

{

if(vRes\_s[i] == "F") return Convert.ToString(i + 1);

}

}

Включение в эти методы распознавания особых случаев позволяет получить их окончательный вид:

static void Occupy(string cn)

{

if ((Convert.ToInt16(cn) > vRes\_s.Length) | (Convert.ToInt16(cn) < 0)) throw new ResIdInvalid();

if(vRes\_s[Convert.ToInt16(cn) - 1] == "B")throw new ResIsBusy();

vRes\_s[Convert.ToInt16(cn) - 1] = "B";

}

static void Free(string cn)

{

if ((Convert.ToInt16(cn) > vRes\_s.Length) | (Convert.ToInt16(cn) < 0)) throw new ResIdInvalid();

if (vRes\_s[Convert.ToInt16(cn) - 1] == "F") throw new ResWasFree();

vRes\_s[Convert.ToInt16(cn) - 1] = "F";

}

static string Request()

{

for(int i=0;i<vRes\_s.Length;i++)

{

if(vRes\_s[i] == "F") return Convert.ToString(i + 1);

}

throw new ResAreBusy(); ;

}

* 1. Сохранение модели ресурсов

Выбор способа сохранения модели ресурсов.

Модель ресурсов сохраняется как последовательный файл методом WriteAllLines и восстанавливается методом ReadAllLines класса File пространства имён IO. Причём метод WriteAllLines превращает каждый строковый элемент массива в строку файла, а метод ReadAllLines – строку файла в строковый элемент массива.

*Почему?* Вызов этих методов является простейшим способом (нужен всего по одному их вызову) пересылки массива строк в файл и обратно. Иные способы, например, использование потоков для сохранения в файле или обращение к базе данных, требуют более сложных программ.

Выбор файла во время работы приложения.

Предлагаются два варианта: а) файл со стандартным именем в директории, в которой находится исполняющийся модуль приложения, и б) пользователь указывает имя файла (включающее в себя полный путь к файлу).

Имя директории, содержащей исполняемый модуль приложения, может быть найдено методом GetCurrentDirectory. Находящимся в пространстве имён IO.

*Почему?* Вариант (а) предусмотрен спецификацией. Вариант (б) как дополнительный увеличивает степень дружественности приложения, позволяет использовать несколько фалов, в которых сохраняется модель.

Когда пользователь может выбирать файл для хранения модели.

Один раз при запуске приложения.

*Почему?* Чтобы не усложнять основной протокол взаимодействия пользователя с приложением.

Возможности воздействия пользователя на файл, предназначенный для хранения модели.

При выборе файла или не изменять файл или “переустановить модель”, то есть указать количество ресурсов и установить их в модели в свободное состояние.

*Почему?* Обеспечиваются возможности: а) продолжить обслуживание ресурсов после перезапуска приложения и б) установить естественное (свободное) начальное состояние набора ресурсов.

Резюме: модель сохраняется предложением программы:

File.WriteAllLines(Path, vRes\_s);

и восстанавливается предложением:

vRes\_s = File.ReadAllLines(Path);

Path – переменная, содержащая полный путь к файлу. Путь к стандартному файлу, пусть его имя будет Resmod00.txt, может быть получено выполнением предложения:

Path = Directory.GetCurrentDirectory() + @"\Resmod00";

Если файл при записи отсутствует, то он создаётся.

Интеграция бизнес-логики и сохранения ресурсов.

Позволит провести автономную отладку разрабатываемого приложения.

Включение бизнес-логики и модели в приложение.

Поле vRres\_s (модель ресурсов), а также методы Request, Occupy и Free образуют новый статическй класс Model. Временные определения этих методов из класса Program удаляются. Методы объявляются public для обеспечения их доступности их других классов. Поле vRres\_s инициировано.

Выделение класса Model продиктовано принципиальными соображениями

Обеспечение сохранения модели.

В начало цикла вводится предложение, сохраняющее модель в файле по адресу Path.

В класс Program вводится определение переменной Path, а в начало метода Main вводится предложение, которое присваивает этой переменной значение – путь к файлу, предназначенному для сохранения модели.

Расположение определения переменной Path и присвоения ей значения временно, служит целям отладки. Так же временно реализовано инициирование модели.

Сохранение модели ресурсов в начале каждого диалогового цикла позволяет сохранять модель после каждой выполненной команды и проверять возможность сохранения перед началом работы. Иначе можно было бы ввести специальную команду сохранения, но это усложнило бы пользовательский интерфейс приложения.

Резюме:

Модуль ResRegV1cons приложения приобретает следующий вид:

using System;

using System.IO;

namespace ResRegV1cons

{

class ResAreBusy : Exception { }

class ResIdInvalid : Exception { }

class UnRecommended : Exception { }

class ResIsBusy : Exception { }

class ReswasFree : Exception { }

static class Model

{

static string[] vRes\_s = new[] { "1", "2", "3" };//Модель набора ресурсов

public static void Occupy(string cn)

{

if ((Convert.ToInt16(cn) > vRes\_s.Length) | (Convert.ToInt16(cn) < 0)) throw new CellIdInvalid();

if(vRes\_s[Convert.ToInt16(cn) - 1] == "B")throw new CellIsBusy();

vRes\_s[Convert.ToInt16(cn) - 1] = "B";

}

public static void Free(string cn)

{

if ((Convert.ToInt16(cn) > vRes\_s.Length) | (Convert.ToInt16(cn) < 0)) throw new CellIdInvalid();

if (vRes\_s[Convert.ToInt16(cn) - 1] == "F") throw new CellwasFree();

vRes\_s[Convert.ToInt16(cn) - 1] = "F";

}

public static string Request()

{

for(int i=0;i<vRes\_s.Length;i++)

{

if(vRes\_s[i] == "F") return Convert.ToString(i + 1);

}

throw new CellsAreBusy(); ;

}

}

class Program

{

// static string Request(){}

// static void Occupy(string cn){}

// static void Free(string cn){}

static string Path; //путь к файлу, сохраняющему модель

static void Main(string[] args)

{

string Command;

Path = Directory.GetCurrentDirectory() + @"\Resmod00";

do

{

File.WriteAllLines(Path, Model.vRes\_s);//сохранение модели

Console.WriteLine("Введите команду:");

Command = Console.ReadLine();

Command = Command.ToUpper();

try

{

if (Command == "REQUEST") Console.WriteLine(Model.Request());

if (Command == "OCCUPY")

{

Console.WriteLine("Введите номер ресурса:");

Model.Occupy(Console.ReadLine());

Console.WriteLine("Ресурс стал занятым.");

};

if (Command == "FREE")

{

Console.WriteLine("Введите номер ресурса:");

Model.Free(Console.ReadLine());

Console.WriteLine("Ресурс освобождён.");

};

}

catch (OverflowException) { Console.WriteLine("Такого ресурса нет."); }

catch (FormatException) { Console.WriteLine("Такого ресурса нет."); }

catch (ResIdInvalid) { Console.WriteLine("Такого ресурса нет."); }

catch (ResWasFree) { Console.WriteLine("Ресурс был свободен."); }

catch (ResAreBusy) { Console.WriteLine("Все ресурсы заняты."); }

catch (ResIsBusy) { Console.WriteLine("ресурс уже занят."); }

}

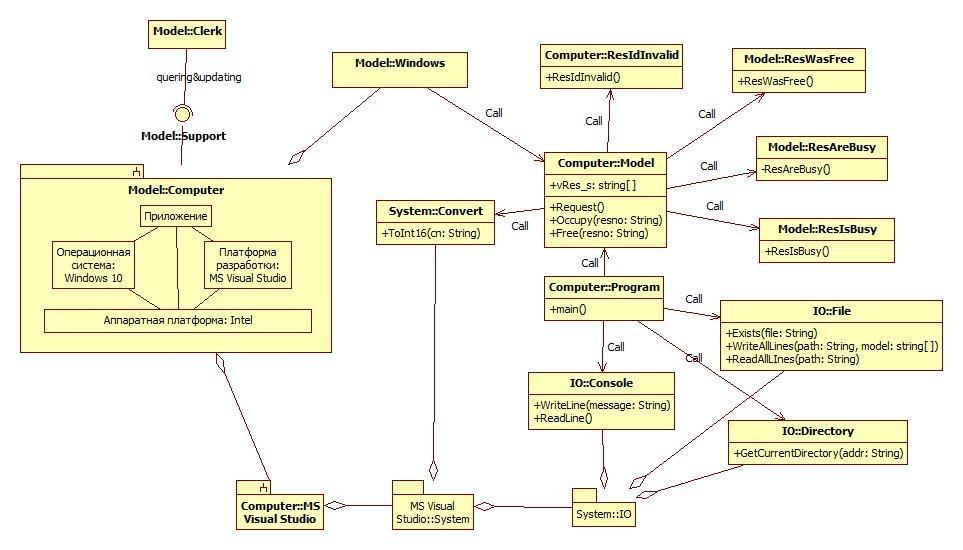
while(Command != "");

}

}

}

Соответственно, в диаграмму проектных классов добавляются классы ввода-вывода с используемыми методами, а в класс Pogram – поле, представляющее модель ресурсов. Актульное состояние диаграммы классов показано на Рис. 7.

Рисунок 7. Диаграмма классов с выбором и сохранением модели данных.

Описание для “Рисунок 7. Диаграмма классов с выбором и сохранением модели данных.”:

### Описание изображения

Данное изображение представляет собой \*\*диаграмму компонентов (Component Diagram)\*\*, которая описывает структуру системы и взаимодействие между ее компонентами. Диаграмма показывает как компоненты связаны друг с другом через интерфейсы или вызовы методов. Давайте разберем изображение поэтапно:

---

#### \*\*1. Общая структура диаграммы\*\*

- \*\*Компоненты (Components)\*\*:

- Представлены прямоугольниками с закругленными углами.

- Каждый компонент соответствует определенной части системы или библиотеке.

- \*\*Интерфейсы (Interfaces)\*\*:

- Показывают точки взаимодействия между компонентами.

- \*\*Связи\*\*:

- Показаны стрелками, указывающими направление взаимодействия между компонентами.

---

#### \*\*2. Элементы диаграммы\*\*

##### \*\*Компоненты\*\*

1. \*\*Model:Clerk\*\*:

- Изображен как прямоугольник с надписью `Model:Clerk`.

- Представляет модель клерка, который может выполнять запросы и обновления (`querying&updating`).

2. \*\*Model:Support\*\*:

- Изображен как круг с надписью `Model:Support`.

- Представляет модель поддержки, которая взаимодействует с другими компонентами.

3. \*\*Model:Windows\*\*:

- Изображен как прямоугольник с надписью `Model:Windows`.

- Представляет модель операционной системы Windows.

4. \*\*Computer::ResIdInvalid\*\*:

- Изображен как прямоугольник с надписью `Computer::ResIdInvalid`.

- Представляет компонент, который проверяет, является ли идентификатор ресурса (`ResId`) недопустимым.

- Метод: `+ResIdInvalid()`.

5. \*\*Model::ResWasFree\*\*:

- Изображен как прямоугольник с надписью `Model::ResWasFree`.

- Представляет компонент, который отслеживает состояние ресурса (был ли он свободен).

- Метод: `+ResWasFree()`.

6. \*\*Model::ResAreBusy\*\*:

- Изображен как прямоугольник с надписью `Model::ResAreBusy`.

- Представляет компонент, который проверяет, заняты ли все ресурсы.

- Методы:

- `+ResAreBusy()`

- `-ResAreBusy()` (возможно, это метод для освобождения ресурса).

7. \*\*Model::ResIsBusy\*\*:

- Изображен как прямоугольник с надписью `Model::ResIsBusy`.

- Представляет компонент, который проверяет, занят ли конкретный ресурс.

- Метод: `+ResIsBusy()`.

8. \*\*Computer::Model\*\*:

- Изображен как прямоугольник с надписью `Computer::Model`.

- Представляет модель компьютера.

- Атрибут: `+r\_Res\_s: string[]`.

9. \*\*Computer::Program\*\*:

- Изображен как прямоугольник с надписью `Computer::Program`.

- Представляет основную программу компьютера.

- Методы:

- `+main()`: Главный метод программы.

- `+Request()`: Метод для отправки запроса.

- `+Occupy(resno: String)`: Занять ресурс с указанным идентификатором.

- `+Free(resno: String)`: Освободить ресурс с указанным идентификатором.

10. \*\*System::Convert\*\*:

- Изображен как прямоугольник с надписью `System::Convert`.

- Представляет системную библиотеку для конвертации данных.

- Метод: `+ToInt16(cn: String)` — преобразование строки в целое число типа `Int16`.

11. \*\*IO::File\*\*:

- Изображен как прямоугольник с надписью `IO::File`.

- Представляет компонент для работы с файлами.

- Методы:

- `+Exists(file: String)`: Проверка существования файла.

- `+WriteAllLines(path: String, model: string[])`: Запись всех строк в файл.

- `+ReadAllLines(path: String)`: Чтение всех строк из файла.

12. \*\*IO::Console\*\*:

- Изображен как прямоугольник с надписью `IO::Console`.

- Представляет компонент для работы с консолью.

- Методы:

- `+WriteLine(message: String)`: Вывод текста на консоль.

- `+ReadLine()`: Чтение строки с консоли.

13. \*\*IO::Directory\*\*:

- Изображен как прямоугольник с надписью `IO::Directory`.

- Представляет компонент для работы с директориями.

- Метод: `+GetCurrentDirectory(addr: String)` — получение текущей директории.

14. \*\*Computer::MS Visual Studio\*\*:

- Изображен как прямоугольник с надписью `Computer::MS Visual Studio`.

- Представляет интегрированную среду разработки (IDE) Microsoft Visual Studio.

15. \*\*MS Visual Studio::System\*\*:

- Изображен как прямоугольник с надписью `MS Visual Studio::System`.

- Представляет системные компоненты IDE MS Visual Studio.

16. \*\*System::IO\*\*:

- Изображен как прямоугольник с надписью `System::IO`.

- Представляет системную библиотеку для ввода-вывода.

---

##### \*\*Связи между компонентами\*\*

1. \*\*Model:Clerk → Model:Support\*\*:

- Связь называется `querying&updating`.

- Это указывает на то, что модель клерка взаимодействует с моделью поддержки для выполнения запросов и обновлений.

2. \*\*Model:Windows → Computer::ResIdInvalid\*\*:

- Связь называется `Call`.

- Это указывает на то, что модель Windows вызывает метод `ResIdInvalid()` в компоненте `Computer::ResIdInvalid`.

3. \*\*Computer::ResIdInvalid → Model::ResWasFree\*\*:

- Связь называется `Call`.

- Это указывает на то, что компонент `Computer::ResIdInvalid` вызывает метод `ResWasFree()` в компоненте `Model::ResWasFree`.

4. \*\*Computer::Model → Model::ResAreBusy\*\*:

- Связь называется `Call`.

- Это указывает на то, что компонент `Computer::Model` вызывает метод `ResAreBusy()` в компоненте `Model::ResAreBusy`.

5. \*\*Computer::Model → Model::ResIsBusy\*\*:

- Связь называется `Call`.

- Это указывает на то, что компонент `Computer::Model` вызывает метод `ResIsBusy()` в компоненте `Model::ResIsBusy`.

6. \*\*Computer::Program → System::Convert\*\*:

- Связь называется `Call`.

- Это указывает на то, что программа использует метод `ToInt16(cn: String)` из компонента `System::Convert`.

7. \*\*Computer::Program → IO::File\*\*:

- Связь называется `Call`.

- Это указывает на то, что программа использует методы работы с файлами из компонента `IO::File`.

8. \*\*Computer::Program → IO::Console\*\*:

- Связь называется `Call`.

- Это указывает на то, что программа использует методы консоли из компонента `IO::Console`.

9. \*\*Computer::Program → IO::Directory\*\*:

- Связь называется `Call`.

- Это указывает на то, что программа использует методы работы с директориями из компонента `IO::Directory`.

10. \*\*Computer::MS Visual Studio → MS Visual Studio::System\*\*:

- Связь показывает, что среда разработки MS Visual Studio использует системные компоненты IDE.

11. \*\*MS Visual Studio::System → System::IO\*\*:

- Связь показывает, что системные компоненты IDE используют общую системную библиотеку ввода-вывода (`System::IO`).

---

#### \*\*3. Текстовые подписи\*\*

- Внизу под диаграммой написано:

- \*\*"Рисунок [номер]. Исходная диаграмма компонентов для задачи управления ресурсами."\*\*

- Это указывает на то, что диаграмма относится к задаче управления ресурсами и является исходным вариантом.

---

#### \*\*4. Цветовая палитра\*\*

- Компоненты выделены прямоугольниками желтого цвета.

- Интерфейсы и связи представлены стрелками красного цвета с метками.

---

#### \*\*5. Общий смысл диаграммы\*\*

Диаграмма описывает следующее:

1. \*\*Модель клерка\*\* взаимодействует с моделью поддержки для выполнения запросов и обновлений.

2. \*\*Модель Windows\*\* взаимодействует с компонентами, которые проверяют состояние ресурсов.

3. \*\*Основная программа (`Computer::Program`)\*\* управляет ресурсами, используя методы для их занятия и освобождения.

4. \*\*Программа использует различные компоненты\*\*:

- `System::Convert` для конвертации данных.

- `IO::File`, `IO::Console`, и `IO::Directory` для работы с файлами, консолью и директориями.

5. \*\*Среда разработки MS Visual Studio\*\* связана с системными компонентами IDE и использует общую системную библиотеку ввода-вывода.

---

### \*\*Заключение\*\*

Данная диаграмма компонентов четко демонстрирует структуру системы и взаимодействие между ее компонентами в контексте задачи управления ресурсами. Она помогает понять, какие компоненты существуют в системе, как они связаны и какие функции выполняют.

* 1. Начальные действия после запуска программы

Организуется диалог с пользователем, в ходе которого принимаются следующие решения:

- какой файл будет использоваться для сохранения модели ресурсов,

- каково начальное состояние модели (обновлённое (все ресурсы “свободны”) или сохранённое в файле),

- если модель обновляется, то каково количество ресурсов.

Начальные действия определяются принятыми ранее решениями и произведёнными построениями.

Обновление модели.

Статический метод ClearModel

static void ClearModel()

{

Console.WriteLine("Укажите количество ресурсов:");

try

{

int Capacity = Convert.ToInt32(Console.ReadLine());

Model.vRes\_s = new string[Capacity];

for (int i = 0; i < Capacity; i++) Model.vRes\_s [i] = "F";

}

catch

{

Console.WriteLine("Введено некорректное число!");

ClearModel();

}

}

В случае некорректного ввода числа ресурсов запрос к пользователю повторяется.

Восстановление модели.

Статический метод GetModel запрашивает пользователя, нужно ли обновить, и если “да”, то вызывает метод ClearModel. В противном случае восстанавливает модель из файла.

static void GetModel()

{

Console.WriteLine("Обновить файл?");

if (Console.ReadLine().ToUpper() == "Y") ClearModel();

else

{

Model.vRes\_s = File.ReadAllLines(Path);

}

}

Диалог выбора файла для сохранения модели.

Статический метод On проверяет наличие стандартного файла для сохранения модели и, если он имеется, то пользователю направляется запрос, согласен ли тот его использовать. В случае положительного ответа вызывается метод GetModel. Иначе пользователю направляется запрос, желает ли он в данном запуске приложения работать со стандартным файлом. В случае положительного ответа вызывается модуль ClearModel.

Если пользователь не намерен работать со стандартным файлом, ему предлагается ввести адрес нестандартного файла. Если указанный файл существует, то вызывается метод GetModel. В противном случае вызывается.

Метод ClearModel не создаёт файла, но инициирует модель в оперативной памяти, затем её инициированное состояние будет сохранено перед запросом команды в методе Main класса Program. Таким образом работоспособность выбранного стандартного или нестандартного файла будет проверена. Если будет обнаружено, что в файл невозможно писать, то пользователю получит соответствующее сообщение и ему придётся перезапустить приложение. Если указанный нестандартный файл не существует, то метод On возвращает значение false, чтобы модуль On мог быть перезапущен.

static bool On()

{

try

{

if (File.Exists(Directory.GetCurrentDirectory() + @"\Resmod00"))

{

Console.WriteLine("Использовать существующий стандартный файл Resmod00?");

if (Console.ReadLine().ToUpper() == "Y")

{

Path = Directory.GetCurrentDirectory() + @"\Resmod00";

GetModel();

return true;

}

}

else

{

Console.WriteLine("Создать стандартный файл?");

if (Console.ReadLine().ToUpper() == "Y")

{

Path = Directory.GetCurrentDirectory() + @"\Resmod00";

ClearModel();

return true;

}

};

Console.WriteLine("Введите полный адрес нестандартного файла:");

Path = Console.ReadLine();

if (File.Exists(Path))

{

GetModel();

return true;

}

else

{

ClearModel();

return true;

}

}

catch (IOException) { Console.WriteLine("Файл не открылся."); return false; }

catch (Exception) { Console.WriteLine("Ошибка ввода-вывода."); return false; }

}

* 1. Интеграция и тестирование варианта V1 приложения.

Интеграция.

В модуль ResRegV1.cs вводится новый статический класс SetUp, включающий методыClearModel, GetModel. Публичный метод On вызывается в начале метода Main класса Program в цикле, который заканчивает, если On возвращает значение true.

Поле Path удаляется из класса Program и становится полем класса SetUp. Инициирование поля Path из метода Main удаляется. При обращении к полю Path в методе On необходимо уточнить, что Setup следует брать из класса SetUp, а не из пространства имён System.IO, в котором оно обозначает тип.

Образование отдельного класса для выполнения начальных действий при запуске приложения обусловлено соблюдением шаблонов GRASP.

На Рис.8 приведена актуальная диаграмма классов приложения.

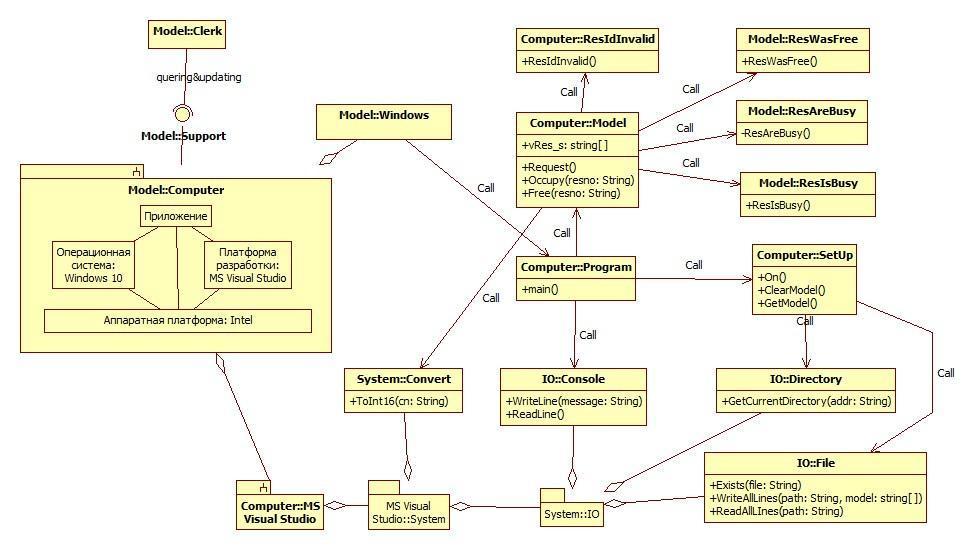


Рисунок 8. Финальная диаграмма классов первой версии Регистратора ресурсов.

Описание для “Рисунок 8. Финальная диаграмма классов первой версии Регистратора ресурсов.”:   
### Описание изображения

Данное изображение представляет собой \*\*диаграмму компонентов (Component Diagram)\*\*, которая описывает структуру системы и взаимодействие между ее компонентами. Диаграмма показывает как компоненты связаны друг с другом через интерфейсы или вызовы методов. Давайте разберем изображение поэтапно:

---

#### \*\*1. Общая структура диаграммы\*\*

- \*\*Компоненты (Components)\*\*:

- Представлены прямоугольниками с закругленными углами.

- Каждый компонент соответствует определенной части системы или библиотеке.

- \*\*Интерфейсы (Interfaces)\*\*:

- Показывают точки взаимодействия между компонентами.

- \*\*Связи\*\*:

- Показаны стрелками, указывающими направление взаимодействия между компонентами.

---

#### \*\*2. Элементы диаграммы\*\*

##### \*\*Акторы/Объекты\*\*

1. \*\*Model:Clerk\*\*:

- Изображен как прямоугольник с надписью `Model:Clerk`.

- Представляет модель клерка, который может выполнять запросы и обновления (`querying&updating`).

2. \*\*Model:Support\*\*:

- Изображен как круг с надписью `Model:Support`.

- Представляет модель поддержки, которая взаимодействует с другими компонентами.

3. \*\*Model:Windows\*\*:

- Изображен как прямоугольник с надписью `Model:Windows`.

- Представляет модель операционной системы Windows.

4. \*\*Computer::ResIdInvalid\*\*:

- Изображен как прямоугольник с надписью `Computer::ResIdInvalid`.

- Представляет компонент, который проверяет, является ли идентификатор ресурса (`ResId`) недопустимым.

- Метод: `+ResIdInvalid()`.

5. \*\*Model::ResWasFree\*\*:

- Изображен как прямоугольник с надписью `Model::ResWasFree`.

- Представляет компонент, который отслеживает состояние ресурса (был ли он свободен).

- Метод: `+ResWasFree()`.

6. \*\*Model::ResAreBusy\*\*:

- Изображен как прямоугольник с надписью `Model::ResAreBusy`.

- Представляет компонент, который проверяет, заняты ли все ресурсы.

- Методы:

- `+ResAreBusy()`

- `-ResAreBusy()` (возможно, это метод для освобождения ресурса).

7. \*\*Model::ResIsBusy\*\*:

- Изображен как прямоугольник с надписью `Model::ResIsBusy`.

- Представляет компонент, который проверяет, занят ли конкретный ресурс.

- Метод: `+ResIsBusy()`.

8. \*\*Computer::Model\*\*:

- Изображен как прямоугольник с надписью `Computer::Model`.

- Представляет модель компьютера.

- Атрибут: `+r\_Res\_s: string[]`.

9. \*\*Computer::Program\*\*:

- Изображен как прямоугольник с надписью `Computer::Program`.

- Представляет основную программу компьютера.

- Методы:

- `+main()`: Главный метод программы.

- `+Request()`: Метод для отправки запроса.

- `+Occupy(resno: String)`: Занять ресурс с указанным идентификатором.

- `+Free(resno: String)`: Освободить ресурс с указанным идентификатором.

10. \*\*Computer::SetUp\*\*:

- Изображен как прямоугольник с надписью `Computer::SetUp`.

- Представляет компонент для настройки системы.

- Методы:

- `+On0()`: Инициализация системы.

- `+ClearModel()`: Очистка модели.

- `+GetModel()`: Получение модели.

11. \*\*System::Convert\*\*:

- Изображен как прямоугольник с надписью `System::Convert`.

- Представляет системную библиотеку для конвертации данных.

- Метод: `+ToInt16(cn: String)` — преобразование строки в целое число типа `Int16`.

12. \*\*IO::Console\*\*:

- Изображен как прямоугольник с надписью `IO::Console`.

- Представляет компонент для работы с консолью.

- Методы:

- `+WriteLine(message: String)`: Вывод текста на консоль.

- `+ReadLine()`: Чтение строки с консоли.

13. \*\*IO::Directory\*\*:

- Изображен как прямоугольник с надписью `IO::Directory`.

- Представляет компонент для работы с директориями.

- Метод: `+GetCurrentDirectory(addr: String)` — получение текущей директории.

14. \*\*IO::File\*\*:

- Изображен как прямоугольник с надписью `IO::File`.

- Представляет компонент для работы с файлами.

- Методы:

- `+Exists(file: String)`: Проверка существования файла.

- `+WriteAllLines(path: String, model: string[])`: Запись всех строк в файл.

- `+ReadAllLines(path: String)`: Чтение всех строк из файла.

15. \*\*Computer::MS Visual Studio\*\*:

- Изображен как прямоугольник с надписью `Computer::MS Visual Studio`.

- Представляет интегрированную среду разработки (IDE) Microsoft Visual Studio.

16. \*\*MS Visual Studio::System\*\*:

- Изображен как прямоугольник с надписью `MS Visual Studio::System`.

- Представляет системные компоненты IDE MS Visual Studio.

17. \*\*System::IO\*\*:

- Изображен как прямоугольник с надписью `System::IO`.

- Представляет системную библиотеку для ввода-вывода.

---

##### \*\*Связи между компонентами\*\*

1. \*\*Model:Clerk → Model:Support\*\*:

- Связь называется `querying&updating`.

- Это указывает на то, что модель клерка взаимодействует с моделью поддержки для выполнения запросов и обновлений.

2. \*\*Model:Windows → Computer::ResIdInvalid\*\*:

- Связь называется `Call`.

- Это указывает на то, что модель Windows вызывает метод `ResIdInvalid()` в компоненте `Computer::ResIdInvalid`.

3. \*\*Computer::ResIdInvalid → Model::ResWasFree\*\*:

- Связь называется `Call`.

- Это указывает на то, что компонент `Computer::ResIdInvalid` вызывает метод `ResWasFree()` в компоненте `Model::ResWasFree`.

4. \*\*Computer::Model → Model::ResAreBusy\*\*:

- Связь называется `Call`.

- Это указывает на то, что компонент `Computer::Model` вызывает метод `ResAreBusy()` в компоненте `Model::ResAreBusy`.

5. \*\*Computer::Model → Model::ResIsBusy\*\*:

- Связь называется `Call`.

- Это указывает на то, что компонент `Computer::Model` вызывает метод `ResIsBusy()` в компоненте `Model::ResIsBusy`.

6. \*\*Computer::Program → System::Convert\*\*:

- Связь называется `Call`.

- Это указывает на то, что программа использует метод `ToInt16(cn: String)` из компонента `System::Convert`.

7. \*\*Computer::Program → IO::Console\*\*:

- Связь называется `Call`.

- Это указывает на то, что программа использует методы консоли из компонента `IO::Console`.

8. \*\*Computer::Program → IO::Directory\*\*:

- Связь называется `Call`.

- Это указывает на то, что программа использует методы работы с директориями из компонента `IO::Directory`.

9. \*\*Computer::Program → IO::File\*\*:

- Связь называется `Call`.

- Это указывает на то, что программа использует методы работы с файлами из компонента `IO::File`.

10. \*\*Computer::Program → Computer::SetUp\*\*:

- Связь называется `Call`.

- Это указывает на то, что программа вызывает методы настройки системы из компонента `Computer::SetUp`.

11. \*\*Computer::MS Visual Studio → MS Visual Studio::System\*\*:

- Связь показывает, что среда разработки MS Visual Studio использует системные компоненты IDE.

12. \*\*MS Visual Studio::System → System::IO\*\*:

- Связь показывает, что системные компоненты IDE используют общую системную библиотеку ввода-вывода (`System::IO`).

---

#### \*\*3. Текстовые подписи\*\*

- Внизу под диаграммой написано:

- \*\*"Рисунок [номер]. Исходная диаграмма компонентов для задачи управления ресурсами."\*\*

- Это указывает на то, что диаграмма относится к задаче управления ресурсами и является исходным вариантом.

---

#### \*\*4. Цветовая палитра\*\*

- Компоненты выделены прямоугольниками желтого цвета.

- Интерфейсы и связи представлены стрелками красного цвета с метками.

---

#### \*\*5. Общий смысл диаграммы\*\*

Диаграмма описывает следующее:

1. \*\*Модель клерка\*\* взаимодействует с моделью поддержки для выполнения запросов и обновлений.

2. \*\*Модель Windows\*\* взаимодействует с компонентами, которые проверяют состояние ресурсов.

3. \*\*Основная программа (`Computer::Program`)\*\* управляет ресурсами, используя методы для их занятия и освобождения.

4. \*\*Программа использует различные компоненты\*\*:

- `System::Convert` для конвертации данных.

- `IO::Console`, `IO::Directory`, и `IO::File` для работы с консолью, директориями и файлами.

- `Computer::SetUp` для настройки системы.

5. \*\*Среда разработки MS Visual Studio\*\* связана с системными компонентами IDE и использует общую системную библиотеку ввода-вывода.

---

### \*\*Заключение\*\*

Данная диаграмма компонентов четко демонстрирует структуру системы и взаимодействие между ее компонентами в контексте задачи управления ресурсами. Она помогает понять, какие компоненты существуют в системе, как они связаны и какие функции выполняют.

Текст модуля и протокол его тестирования приведён в Приложении.

Тестирование.

Тестирование протоколируется «Средством записи действий» операционной системы.

План тестирования:

Приложение №2

Приложение 2.

Решение задачи регистрации ресурсов, вариант V1, на языке C#

using System.IO;

namespace ResRegV1cons

{

class ResAreBusy : Exception { }

class ResIdInvalid : Exception { }

class UnRecommended : Exception { }

class ResIsBusy : Exception { }

class ResWasFree : Exception { }

static class SetUp

{

public static string Path; //путь к файлу, сохраняющему модель

private static void ClearModel()

{

Console.WriteLine("Укажите количество ресурсов:");

try

{

Model.vRes\_s = new string[Convert.ToInt32(Console.ReadLine())];

for (int i = 0; i < Model.vRes\_s.Length; i++) Model.vRes\_s[i] = "F"; }

catch

{

Console.WriteLine("Введено некорректное число!");

ClearModel();

}

}

private static void GetModel()

{

Console.WriteLine("Обновить файл?");

if (Console.ReadLine().ToUpper() == "Y") ClearModel();

else

{

Model.vRes\_s = File.ReadAllLines(Path);

}

}

public static bool On()

{

try

{

if (File.Exists(Directory.GetCurrentDirectory() + @"\Resmod00"))

{

Console.WriteLine("Использовать существующий стандартный файл Resmod00?");

if (Console.ReadLine().ToUpper() == "Y")

{

Path = Directory.GetCurrentDirectory() + @"\Resmod00";

GetModel();

return true;

}

}

else

{

Console.WriteLine("Создать стандартный файл?");

if (Console.ReadLine().ToUpper() == "Y")

{

Path = Directory.GetCurrentDirectory() + @"\Resmod00";

ClearModel();

return true;

}

};

Console.WriteLine("Введите полный адрес нестандартного файла:");

Path = Console.ReadLine();

if (File.Exists(Path))

{

GetModel();

return true;

}

else

{

ClearModel();

return true;

}

}

catch (IOException) { Console.WriteLine("Файл не открылся."); return false; }

catch (Exception) { Console.WriteLine("Ошибка ввода-вывода."); return false; }

}

}

static class Model

{

public static string[] vRes\_s;//Модель набора ресурсов

public static void Occupy(string cn)

{

if ((Convert.ToInt16(cn) > vRes\_s.Length) | (Convert.ToInt16(cn) < 0)) throw new ResIdInvalid();

if(vRes\_s[Convert.ToInt16(cn) - 1] == "B")throw new ResIsBusy();

vRes\_s[Convert.ToInt16(cn) - 1] = "B";

}

public static void Free(string cn)

{

if ((Convert.ToInt16(cn) > vRes\_s.Length) | (Convert.ToInt16(cn) < 0)) throw new ResIdInvalid();

if (vRes\_s[Convert.ToInt16(cn) - 1] == "F") throw new ResWasFree();

vRes\_s[Convert.ToInt16(cn) - 1] = "F";

}

public static string Request()

{

for(int i=0;i<vRes\_s.Length;i++)

{

if(vRes\_s[i] == "F") return Convert.ToString(i + 1);

}

throw new ResAreBusy(); ;

}

}

class Program

{

static void Main(string[] args)

{

string Command;

while (!SetUp.On());

do

{

File.WriteAllLines(SetUp.Path, Model.vRes\_s);//сохранение модели

Console.WriteLine("Введите команду:");

Command = Console.ReadLine();

Command = Command.ToUpper();

try

{

if (Command == "REQUEST") Console.WriteLine(Model.Request());

if (Command == "OCCUPY")

{

Console.WriteLine("Введите номер ресурса:");

Model.Occupy(Console.ReadLine());

Console.WriteLine("Ресурс стал занятым.");

};

if (Command == "FREE")

{

Console.WriteLine("Введите номер ресурса:");

Model.Free(Console.ReadLine());

Console.WriteLine("Ресурс освобождён.");

};

}

catch (OverflowException) { Console.WriteLine("Такого ресурса нет."); }

catch (FormatException) { Console.WriteLine("Такого ресурса нет."); }

catch (ResIdInvalid) { Console.WriteLine("Такого ресурса нет."); }

catch (ResWasFree) { Console.WriteLine("Ресурс был свободен."); }

catch (ResAreBusy) { Console.WriteLine("Все ресурсы заняты."); }

catch (ResIsBusy) { Console.WriteLine("ресурс уже занят."); }

}

while(Command != "");

}

}

}

Приложение 3.

Решение задачи регистрации ресурсов, вариант V2, на языке C++

// ResRegV2cpp.cpp: главный файл проекта.

#include "stdafx.h"

using namespace System;

using namespace System::IO;

ref class ResAreBusy : Exception { };

ref class ResIsBusy : Exception { };

ref class ResIdInvalid : Exception { };

ref class ResWasFree : Exception { };

public ref class SetUp

{

public: static System::String ^Path;//путь к файлу, сохраняющему модель

private: static void ClearModel()

{

Console::WriteLine("Укажите количество ресурсов:");

try

{

vRes\_s = gcnew array<System::String^>(Convert::ToInt32(Console::ReadLine()));

for (int i = 0; i < vRes\_s->Length; i++) vRes\_s[i] = "F";

}

catch ( ... )

{

Console::WriteLine("Введено некорректное число!");

ClearModel();

}

}

private: static void GetModel()

{

Console::WriteLine("Обновить файл?");

if (Console::ReadLine()->ToUpper() == "Y") ClearModel();

else

{

vRes\_s = File::ReadAllLines(Path);

}

}

public: static bool On()

{

try

{

if (File::Exists(Directory::GetCurrentDirectory() + "Resmod00"))

{

Console::WriteLine("Использовать существующий стандартный файл Resmod00?");

if (Console::ReadLine()->ToUpper() == "Y")

{

Path = Directory::GetCurrentDirectory() + "Resmod00";

GetModel();

return true;

}

}

else

{

Console::WriteLine("Создать стандартный файл?");

if (Console::ReadLine()->ToUpper() == "Y")

{

Path = Directory::GetCurrentDirectory() + "Resmod00";

ClearModel();

return true;

}

};

Console::WriteLine("Введите полный адрес нестандартного файла:");

Path = Console::ReadLine();

if (File::Exists(Path))

{

GetModel();

return true;

}

else

{

ClearModel();

return true;

}

}

catch (const IOException^ e ) { Console::WriteLine("Файл не открылся."); return false; }

catch (const Exception^ e) { Console::WriteLine("Ошибка ввода-вывода."); return false; }

}

public: static array<System::String^> ^vRes\_s; //Модель набора ресурсов

public: static void Occupy(String^ cn)

{

if ((Convert::ToInt16(cn) > vRes\_s->Length) | (Convert::ToInt16(cn) < 0)) throw gcnew ResIdInvalid();

if (vRes\_s[Convert::ToInt16(cn) - 1] == "B")throw gcnew ResIsBusy();

vRes\_s[Convert::ToInt16(cn) - 1] = "B";

}

public: static void Free(String^ cn)

{

if ((Convert::ToInt16(cn) > vRes\_s->Length) | (Convert::ToInt16(cn) < 0)) throw gcnew ResIdInvalid();

if (vRes\_s[Convert::ToInt16(cn) - 1] == "F") throw gcnew ResWasFree();

vRes\_s[Convert::ToInt16(cn) - 1] = "F";

}

public: static String^ Request()

{

for (int i = 0; i<vRes\_s->Length; i++)

{

if (vRes\_s[i] == "F") return Convert::ToString(i + 1);

}

throw gcnew ResAreBusy();

}

};

int main(array<System::String ^> ^args)

{

String ^Command;

while ((!SetUp::On()));

do

{

File::WriteAllLines(SetUp::Path, SetUp::vRes\_s);//сохранение модели

Console::WriteLine("Введите команду:");

Command = Console::ReadLine();

Command = Command->ToUpper();

try

{

if (Command == "REQUEST") Console::WriteLine(SetUp::Request());

if (Command == "OCCUPY")

{

Console::WriteLine("Введите номер ячейки:");

SetUp::Occupy(Console::ReadLine());

Console::WriteLine("Ячейка стала занята.");

};

if (Command == "FREE")

{

Console::WriteLine("Введите номер ячейки:");

SetUp::Free(Console::ReadLine());

Console::WriteLine("Ячейка освобождена.");

};

}

catch (const OverflowException^ e) { Console::WriteLine("Такого ресурса нет."); }

catch (const FormatException^ e) { Console::WriteLine("Такого ресурса нет."); }

catch (const ResIdInvalid^ e) { Console::WriteLine("Такого ресурса нет."); }

catch (const ResWasFree^ e) { Console::WriteLine("Ресурс был свободен."); }

catch (const ResAreBusy^ e) { Console::WriteLine("Все ресурсы заняты."); }

catch (const ResIsBusy^ e) { Console::WriteLine("ресурс уже занят."); }

}

while ((Command != ""));

return 0;

}