ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ
ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«САМАРСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ
УНИВЕРСИТЕТ»

**Д.Е. Новичков, В.А. Галузин, А.В. Галицкая, Р.Д. Гуськов,
В.Б. Ларюхин, П.О. Скобелев**

Разработка мультиагентных систем для управления мобильными ресурсами на языке программирования Python

Учебное пособие и лабораторный практикум

САМАРА 2023

УДК 004.9 (075)
ББК 32.97

Разработка мультиагентных систем для управления мобильными ресурсами на языке Python. Учебное пособие и лабораторный практикум / Д.Е.Новичков, В.А. Галузин, А.В. Галицкая, Б. Ларюхин, П.О. Скобелев. Самарский государственный технический университет. Самара, 2024. – 90 с.

Учебное пособие и лабораторный практикум предназначены для студентов, обучающихся по специальности «Программная инженерия» на кафедре «Вычислительная техника».

Настоящее пособие и лабораторный практикум позволяют студенту 1 курса, не обладающему глубокими знаниями по программированию, освоить принципы мультиагентных технологий и начать их применять для решения сложных задач управления ресурсами и ряда других.

Выбор языка Python для использования мультиагентных технологий обусловлен как быстрым развитием этого языка для решения сложных задач, так и тем фактом, что вчерашние школьники, приходя на первый курс в ВУЗ, в настоящее время уже имеют достаточные начальные знания именно этого языка.

В ходе обучения обучаемым предлагается разработать мультиагентные технологии, решая задачу управления мобильными

ресурсами с поэтапным повышением сложности разбора конфликтов на каждом шаге. В этих целях даются небольшие вводные фрагменты теории построения акторных систем, но основной упор сделан на большое число практических примеров, специально подготовленных и разработанных для решения задачи управления мобильными ресурсами.

Предусматривается использование настоящего пособия и лабораторного практикума после завершения первого модуля (уровня) курса «Эмерджентный интеллект», который базируется на применении разработанного ранеее конструктора онтологий и универсализированной онтологически-настраива-емой мультиагентной системы (Онто-МАС). Вместе с тем, разработанные материалы могут быть использованы и автономно, в параллель с указанными выше, для обучения основам мультиагентного прораммирования уже с первого семестра первого курса.

Полученные знания и навыки будут полезны в ходе проектной работы, курсового и дипломного проектирования.

Учебное пособие разработано на кафедре «Вычислительная техника» Самарского государственного технического университета.

Табл. 2. Ил. 17.

Печатается по решению редакционно-издательского совета Самарского государственного технического университета

Оглавление

ВВЕДЕНИЕ

Настоящее пособие и лабораторнй практикум являются частью курса «Эмерджентный интеллект», посвященного созданию нового класса систем искусственного коллективного интеллекта.

Эмерджентный интеллект определяется как способность открытой мультиагентной системы решать сложные задачи за счет конкуренции и кооперации прогроаммных агентов, выявляющих и разрешающих конфликты с взаимными уступками на виртуальном рынке. Решение любой сложной задачи при этом самоорганизуется до достижения неулучшаемого состояния «конкурентного равновесия», когда ни один из агентов более не может улучшить требуемые показатели работы системы.

Мультиагентные технологии, наряду с онтологиями, составляют основу систем эмерджентного интеллекта, определяя возможность программной реализации моделей, методов и алгоритмов коллективного согласованного принятия решений. В ходе курса обучаемым предстоит разработать мультиагентные технологии для обнаружения и резрешения конфликтов и перейти от построения вложенных циклов выполнения команд – к параллельной и аснихронной работе программных агентов.

Перед началом обучения рекомендуется изучить пособие: «Введение в системы «Эмерджентного интеллекта», которое содержит теоретические сведения о развитии систем «эмерджентного интеллекта» в контексте формирующейся новой Индустрии 5.0 / Общества 5.0, связываемых с цифровизацией знаний и созданием

колоний и экосистем автономных цифровых двойников предприятий и сложных техинических изделий.

Учебное пособие организовано следующим образом. В 1-й главе излагается теория акторных систем. Во 2-й главе описывается базовый каркас стандартной мультиагентной системы. В главах 3-6 описываются лабораторные работы, последовательное выполнение которых позволит изучить подходы к созданию мультиагентных систем планирования на языке программирования Python.

Авторы выражают надежду, что представленный материал станет для обучаемых отправной точкой на пути изучения и дальнейшего применения принципов эмерджентного интеллекта, онтологий и мультиагентных технологий как в задачах управления различными видами ресурсов в реальном времени, так и для других применений.

Данное пособие рекомендуется как студентам и аспирантам вузов, так и инженерам, занимающимся созданием и внедрением новых технологий управления производством, транспортом и другими видами ресурсов.

Цели, задачи и содержание пособия и лабораторного практикума

Основными целями обучения являются:

– изучение принципов разработки мультиагентных технологий, применяемых в интеллекктуальных системах эмерджентного интеллекта для решения задач управления ресурсами;

– получение практических навыков реализации мультиагентных технологий на языке Python;

– применение полученных навыков в задачах управления ресурсами.

Основные задачи обучения:

– изучение базовых понятий акторных систем;

– знакомство с одним из фреймворков акторных систем;

– применение выбранного фрейворка для реализации первой мультиагентной системы;

– реализация моделей и методов коллективного согласованного принятия решений агентами для планирования и оптимизациями ресурсов;

- матчинг заказов и ресурсов в сцене мира предприятия;
- разрешение конфликтов между агентами заказов и ресурсов;

– оценка получаемых решений.

Результаты обучения будут использованы для разработки более сложных моделей, методов и алгоритмов для использования в решении реальных практических задач.

Содержание:

- создание программных агентов в акторной системе;
- основные типы программных агентов;
- конструкция программных агентов;
- наделение агентов собственными целевыми функциями;
- сцена мира агентов;
- поиск взаимного соответствия (матчинг) агентов;
- передача сообщений между агентами;
- протоколы взаимодействия агентов;
- выявление и разрешение конфликтов в расписании;
- оценка получаемых решений.

Кроме задач, пособие содержит контрольные вопросы для самопроверки обучаемых или проведения контрольных опросов.

Требования к обучаемым: знание языка программирования Python, знание ООП, знание теории эмерджентного интеллекта, онтологий и мультиагентных технологий.

Технические требования: Python версии 3.11, IDE для разработки (PyCharm CE или VS Code), доступ в интернет для скачивания необходимых внешних библиотек. Требования к ПК: объем оперативной памяти не менее 8 Гб, объем места на жестком диске – не менее 10 Гб.

Полный исходный код каждого урока приведен в архиве, соответствующем данному уроку

1. Акторы и акторные системы. Создание простейшей акторной системы.

Цель занятия - получить представление о понятии актора, как первооснове создания программных агентов, и акторной системы, особенностях их функционирования.

Рекомендуемая для ознакомления перед началом занятия теория:

- https://ru.wikipedia.org/wiki/Модель_акторов

В первом занятии следует познакомиться с понятием актора и системой, основанных на акторном взаимодействии.

Актор - это программный объект, функционирующий в системе путем обмена сообщениями с другими акторами. Это значит, что он не может вызывать методы других акторов напрямую; его способности к этому ограничены только отправкой сообщения. Но и другие акторы, в свою очередь, не могут вызывать его методы - только отправлять ему сообщения.

Это значит, что актор в общем случае при получении сообщения может совершить только три действия:

1. Изменить свое внутреннее состояние;
2. Породить другого актора;
3. Отправить сообщение.

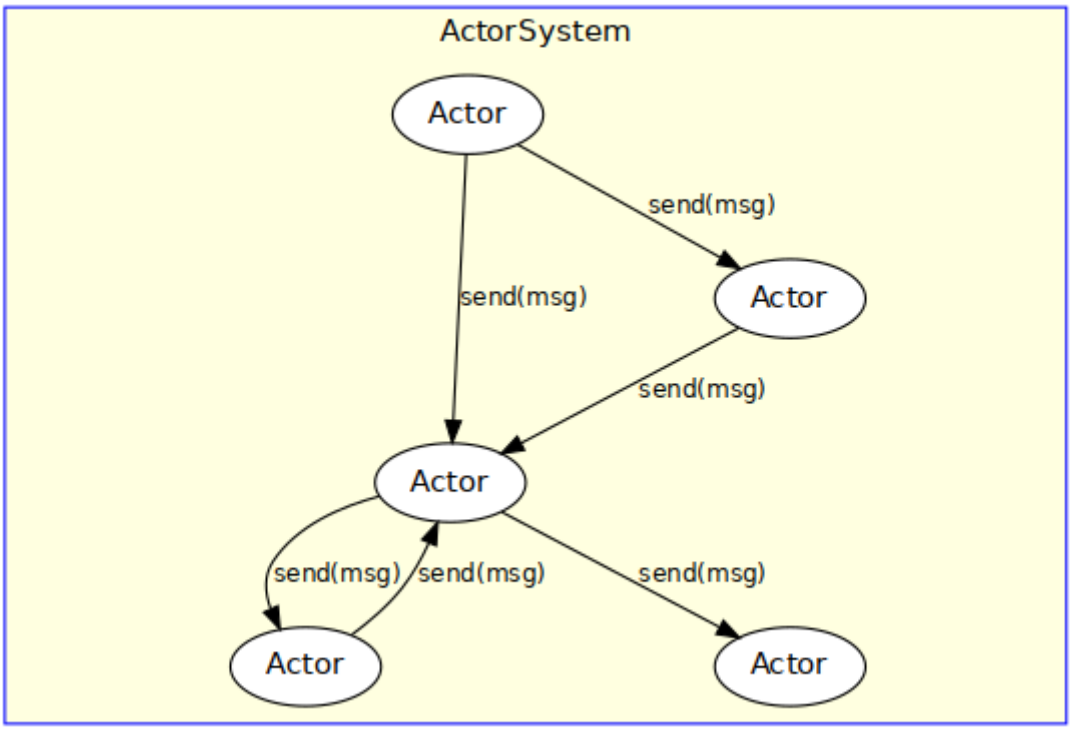

Рис. 1.1. Общая схема взаимодействия акторов

Источник схемы - https://thespianpy.com/doc/in_depth

В данном курсе мы будем использовать фреймворк для построения акторных систем под названием thespian [https://thespianpy.com/doc].

При этом описанный подход может быть применен и к другим фреймворкам при адаптации примеров исходного кода.

1.1. Создание проекта

Создадим директорию проекта, в которой будут храниться исходные файлы, назвав ее *SimpleActorsSystem*.

Вся дальнейшая работа будет выполняться в этой директории.

1.2. Установка фреймворка

Для корректной работы нескольких Python-проектов в операционной системе рекомендуется использовать виртуальное окружение.

Официальная документация по работе с виртуальным окружением приведена по ссылке - https://docs.python.org/3/library/venv.html

Для компьютеров под управлением ОС Windows виртуальное окружение можно настроить следующими командами:

python -m venv venv

call venv/scripts/activate.bat

Примечание: если код выше указан для интерпретатора cmd. Если в системе (или IDE) используется PowerShell, то вторая команды выглядит проще: *venv/scripts/activate.bat*

После установки и активации виртуального окружения необходимо установить фреймворк, это можно сделать путем вызова команды *pip install thespian*

1.3. Создание первого актора

Напишем простейшего актора, который будет принимать сообщение и печатать его в консоль вызовом функции print.

Весь код будем писать в файле *main.py*, который нужно создать, если он не появился при создании проекта в IDE.

Первым шагом мы импортирум все пространство имен из используемого фреймворка:

*from thespian.actors import **

Важно: импорт пространства имен полностью в рабочих проектах не рекомендуется, в данном примере так делается только для обучающих целей.

После этого создадим класс, который назовем *SimplestActor*, от наследовавшись от класса *Actor* фреймворка *thespian*. У этого класса мы переопределим метод обработки сообщений, называющийся *receiveMessage*.

```
class SimplestActor(Actor):
    """
    Объявляем класс - Простой актор, который в будущем станет агентом
    """
    def receiveMessage(self, msg, sender):
        print(f'Актор с адресом {self.myAddress} получил {msg} от {sender}')
```

Этот метод принимает три аргумента. Сам объект класса (*self*), сообщение (*msg*) и адрес отправителя (*sender*). Данный метод вызывается каждый раз при получении актором сообщения. В реализации метода мы написали вывод в консоль строки, содержащей адрес актора, содержимое сообщение и адрес отправителя.

После этого необходимо реализовать вход в программу.

Для этого напишем проверку, запускается ли данный файл, и реализуем логику по созданию актора и отправки ему сообщения.

```
if __name__ == "__main__":
    # Создаем систему акторов, внутри которой они будут жить
    actorSystem = ActorSystem()
    # Создаем экземпляр созданного нами класса и сохраняем его адрес
```

```
actorAddress1 = actorSystem.createActor(SimplestActor)
# Отправляем по сохраненному адресу сообщение.
actorSystem.tell(actorAddress1, "Первое сообщение")
```

ActorSystem (акторная система) - это программный объект, реализующий акторное взаимодействие. Такое взаимодействие - создание и удаление акторов, обмен сообщениями - может быть реализовано в разных потоках операционной системы и даже на разных компьютерах, для этого необходимо настроить акторную систему соответствующим образом.

В данном пособии мы рассмотрим работу акторов в одном потоке, но примеры здесь и далее могут быть запущены и в распределенном режиме.

Далее мы создаём экземпляр описанного нами класса *SimplestActor* внутри акторной системы. Акторная система не возвращает этот экземпляр, а возвращает его адрес, который мы сохраняем внутри переменной.

После создания актора мы не можем вызывать его методы, дальнейшее взаимодействие реализуется с помощью отправки сообщений. В данном случае мы вызываем метод акторной системы *tell*, то есть отправляем сообщение по адресу актора и не ждём ответа от него. Этот метод принимает адрес актора-получателя и само сообщение.

Актор, получив данное сообщение, выведет в консоль сообщение из тела метода. Убедимся в этом, выполнив написанный код командой *python main.py*

Ожидаем, что вывод будет примерно следующим:

Актор с адресом ActorAddr-/A~a получил Первое сообщение от ActorAddr-System:ExternalRequester

Такие переговоры можно представить в виде диаграммы последовательности:

@startuml

"Акторная система" -> SimplestActor : "Первое сообщение"

@enduml

Рис. 1.2. Диаграмма последовательности отправки сообщения

Сообщения, которые может принимать актор, не типизированы, т.е. могут иметь произвольную структуру. В используемом в примерах фреймворке ограничения на сообщения появляются при функционировании акторов в распределенной системе: они должны быть сериализуемы в формат *pickle* [https://docs.python.org/3/library/pickle.html].

Убедимся в этом, отправив актору сообщения разных типов:

actorSystem.tell(actorAddress1, 132)

actorSystem.tell(actorAddress1, [1, 3, 5])

actorSystem.tell(actorAddress1, {'key': 'value'})

Ожидается, что при запуске программы будут выведены строки следующего вида:

Актор с адресом ActorAddr-/A~a получил Первое сообщение от ActorAddr-System:ExternalRequester Актор с адресом ActorAddr-/A~a получил 132 от ActorAddr-System:ExternalRequester Актор с адресом ActorAddr-/A~a получил [1, 3, 5] от ActorAddr-System:ExternalRequester Актор с адресом ActorAddr-/A~a получил {'key': 'value'} от ActorAddr-System:ExternalRequester

Проверим возможности актора по изменению собственного состояния. Для этого модифицируем класс, добавив в него сохранение всех полученных сообщений и вывод всех их при получении любого сообщения. Добавим конструктор класса, в котором вызовем сначала конструктор базового класса, а затем объявим поле *messages* - список, в котором и будем сохранять сообщения.

```
def __init__(self):
    super().__init__()
    print('Создан новый актор')
    self.messages = []
```

После этого метод получения сообщений можно расширить выводом всех имеющихся сообщений:

```
def receiveMessage(self, msg, sender):
    print(f'Актор с адресом {self.myAddress} получил {msg} от {sender}')
    self.messages.append(msg)
    for message in self.messages:
        print(f'Актор с адресом {self.myAddress} ранее получал сообщение {message}')
```

Выполним код, убедившись, что сообщения, отправляемые нами, повторяются по несколько раз. Логику работы можно представить следующим образом:

@startuml

"Акторная система" -> SimplestActor : "Первое сообщение"

note over SimplestActor: "Вывод всех сообщений"

@enduml

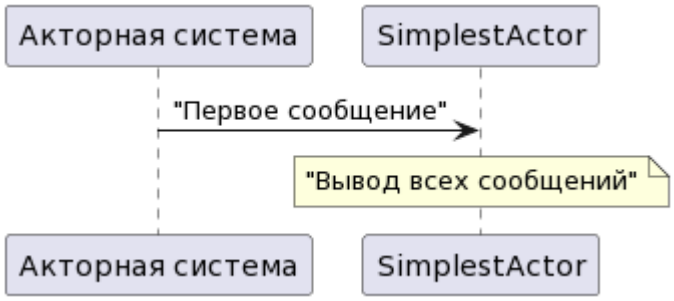

Рис. 1. 3. Диаграмма последовательности с выводом сообщений

1.4. Порождение новых акторов

Доработаем логику акторов - пусть при получении в сообщении текста “*CREATE_ACTOR*” он будет создавать дочернего актора.

Для этого расширим метод получения сообщений, добавив туда следующий код:

```
if msg == 'CREATE_ACTOR':
        child_actor_address = self.createActor(SimplestActor)
        print(f'Создали нового актора, его адрес - {child_actor_address}')
```

Теперь изменим код точки входа в программу, добавив отправку запроса на создание нового актора:

actorSystem.tell(actorAddress1, 'CREATE_ACTOR')

Убедимся, что при вызовые программы создается новый актор - в выводе должны появиться строки вида

Создан новый актор Создали нового актора, его адрес - ActorAddr-/A~a~a

Такой процесс можно визуализировать так:

@startuml

"Акторная система" -> SimplestActor : "Первое сообщение"

note over SimplestActor: "Вывод всех сообщений"

note over SimplestActor: "Если сообщение - CREATE_ACTOR, то создаем нового актора"

@enduml

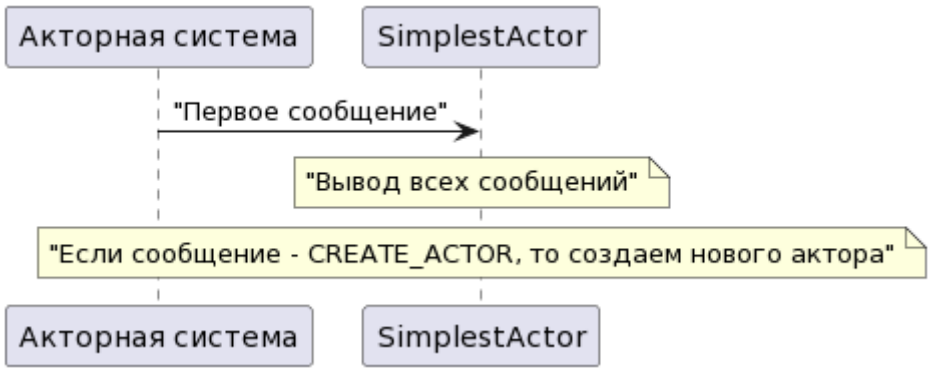

Рис. 1.4. Диаграмма последовательности с отображением условия

1.5. Отправка сообщений

Доработаем логику порождения таким образом, чтобы актор-родитель сохранял адреса всех порожденных им акторов, чтобы ретранслировать им все входящие сообщения. Для этого добавим в

конструкторе автора ещё один список, в котором будут храниться адреса.

```
def __init__(self):
    super().__init__()
    print('Создан новый актор')
    self.messages = []
    self.child_addresses = []
```

Метод обработки сообщений будет выглядеть следующим образом:

```
def receiveMessage(self, msg, sender):
    print(f'Актор с адресом {self.myAddress} получил {msg} от {sender}')
    self.messages.append(msg)
    for message in self.messages:
        print(f'Актор с адресом {self.myAddress} ранее получал сообщение {message}')
    for child in self.child_addresses:
        print(f'Ретранслируем сообщение дочернему актору с адресом {child}')
        self.send(child, msg)
    if msg == 'CREATE_ACTOR':
        child_actor_address = self.createActor(SimplestActor)
        print(f'Создали нового актора, его адрес - {child_actor_address}')
        self.child_addresses.append(child_actor_address)
```

При отправке сообщения, отличного от CREATE_ACTOR, мы перенаправляем сообщение всем порожденным акторам. Проверим это, отправив сообщение:

actorSystem.tell(actorAddress1, 'RETRANSLATED MESSAGE')

При запуске программы должна появиться информация о ретрансляции сообщений:

Актор с адресом ActorAddr-/A~a содержит получал сообщение RETRANSLATED MESSAGE Ретранслируем сообщение дочернему актору с адресом ActorAddr-/A~a~a Актор с адресом ActorAddr-/A~a~a получил RETRANSLATED MESSAGE от ActorAddr-/A~a Актор с адресом ActorAddr-/A~a~a содержит получал сообщение RETRANSLATED MESSAGE

В виде диаграммы последовательности это можно представить так:

@startuml

"Акторная система" -> SimplestActor : Первое сообщение

note over SimplestActor: Вывод всех сообщений

"Акторная система" -> SimplestActor : CREATE_ACTOR

note over SimplestActor: Сообщение - CREATE_ACTOR, \n значит, создаем нового актора

"Акторная система" -> SimplestActor : RETRANSLATED MESSAGE

SimplestActor -> ChildActor : RETRANSLATED MESSAGE

@enduml

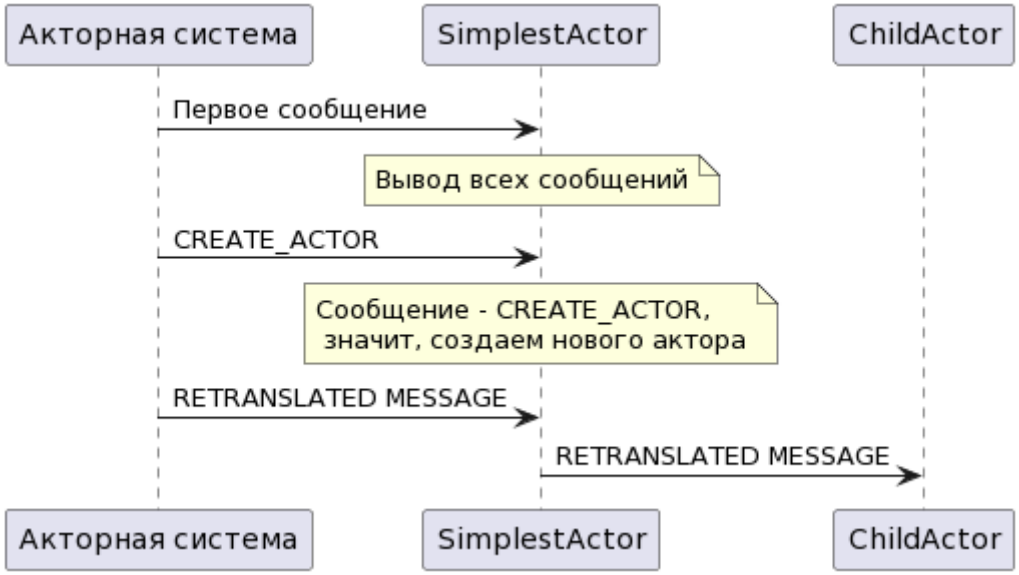

Рис. 1.5. Диаграмма последовательности с перенаправлением сообщений

1.6. Взаимодействие акторов

Акторное взаимодействие может быть продемонстрировано при работе в системе разных типов акторов. Создадим два класса акторов: актор-число и актор-калькулятор. Актор-число будет инициализирован каким-либо числом и принимать адрес калькулятора, а актор-калькулятор будет суммировать все числа, которые отправили ему другие акторы.

```
class NumberActor(Actor):
    def __init__(self):
        super().__init__()
        # В это поле будем сохранять полученное значение.
        self.value = 0

    def receiveMessage(self, msg, sender):
```

print(f'Актор числа с адресом {self.myAddress} получил {msg} от {sender}')

 # Ожидаем, что в сообщении будет содержаться словарь со значением числа и адресом актора-калькулятора.

self.value = msg.get('init_value')

calculator_address = msg.get('calculator_address')

 # Отправляем калькулятору свое значение.

self.send(calculator_address, self.value)

Актор-калькулятор будет сохранять все полученные числа и выводить в консоль их сумму.

```
class CalculatorActor(Actor):
    def __init__(self):
        super().__init__()
        # Все полученные числа будем сохранять в списке.
        self.values = []

    def receiveMessage(self, msg, sender):
        # Ожидаем, что в сообщении будет содержаться число. Добавляем его в список.
        self.values.append(msg)
        # Суммируем все числа и выводим их в консоль.
        total_sum = sum(self.values)
        print(f'Актор-калькулятор с адресом {self.myAddress} получил {msg} от {sender}, итоговая сумма: {total_sum}')
```

В функцию входа в программу добавим такие строки:

```
        print('_____')
    print('Запускаем работу системы с калькулятором')

    # Создаем актор-калькулятор, сохраняем его адрес.
    calculator_address = actorSystem.createActor(CalculatorActor)

    # Создаем актор-число
    number_agent_1 = actorSystem.createActor(NumberActor)
    # Формируем сообщение со значением числа и адресом калькулятора.
    init_message_1 = {'init_value': 1, 'calculator_address': calculator_address}

    # Отправляем актору числа сообщение, после которого он должен отправить другое сообщение калькулятору.
    actorSystem.tell(number_agent_1, init_message_1)

    number_agent_2 = actorSystem.createActor(NumberActor)
    init_message_2 = {'init_value': 2, 'calculator_address': calculator_address}
    actorSystem.tell(number_agent_2, init_message_2)

    number_agent_3 = actorSystem.createActor(NumberActor)
    init_message_3 = {'init_value': 3, 'calculator_address': calculator_address}
    actorSystem.tell(number_agent_3, init_message_3)
```

В окне вывода должны появиться следующие строки:

Запускаем работу системы с калькулятором Актор числа с адресом ActorAddr-/A~c получил {'init_value': 1, 'calculator_address': <thespian.actors.ActorAddress object at 0x0000024FA5A13E10>} от ActorAddr-System:ExternalRequester Актор-калькулятор с адресом ActorAddr-/A~b получил 1 от ActorAddr-/A~c, итоговая сумма: 1 Актор числа с адресом ActorAddr-/A~d получил {'init_value': 2, 'calculator_address': <thespian.actors.ActorAddress object at 0x0000024FA5A13E10>} от ActorAddr-System:ExternalRequester Актор-калькулятор с адресом ActorAddr-/A~b получил 2 от ActorAddr-/A~d, итоговая сумма: 3 Актор числа с адресом ActorAddr-/A~e получил {'init_value': 3, 'calculator_address': <thespian.actors.ActorAddress object at 0x0000024FA5A13E10>} от ActorAddr-System:ExternalRequester Актор-калькулятор с адресом ActorAddr-/A~b получил 3 от ActorAddr-/A~e, итоговая сумма: 6

Такой процесс соответствует следующей логике переговором между акторами:

@startuml

"Акторная система" -> NumberActor1 : {'init_value': 1,\n 'calculator_address': calculator_address}

NumberActor1 -> CalculatorActor : 1

note over CalculatorActor: Сохраняем число,\n считаем и выводим сумму

"Акторная система" -> NumberActor2 : {'init_value': 2,\n 'calculator_address': calculator_address}

NumberActor2 -> CalculatorActor : 2

note over CalculatorActor: Выводим 3

"Акторная система" -> NumberActor3 : {'init_value': 3,\n 'calculator_address': calculator_address}

NumberActor3 -> CalculatorActor : 3

note over CalculatorActor: Выводим 6

@enduml

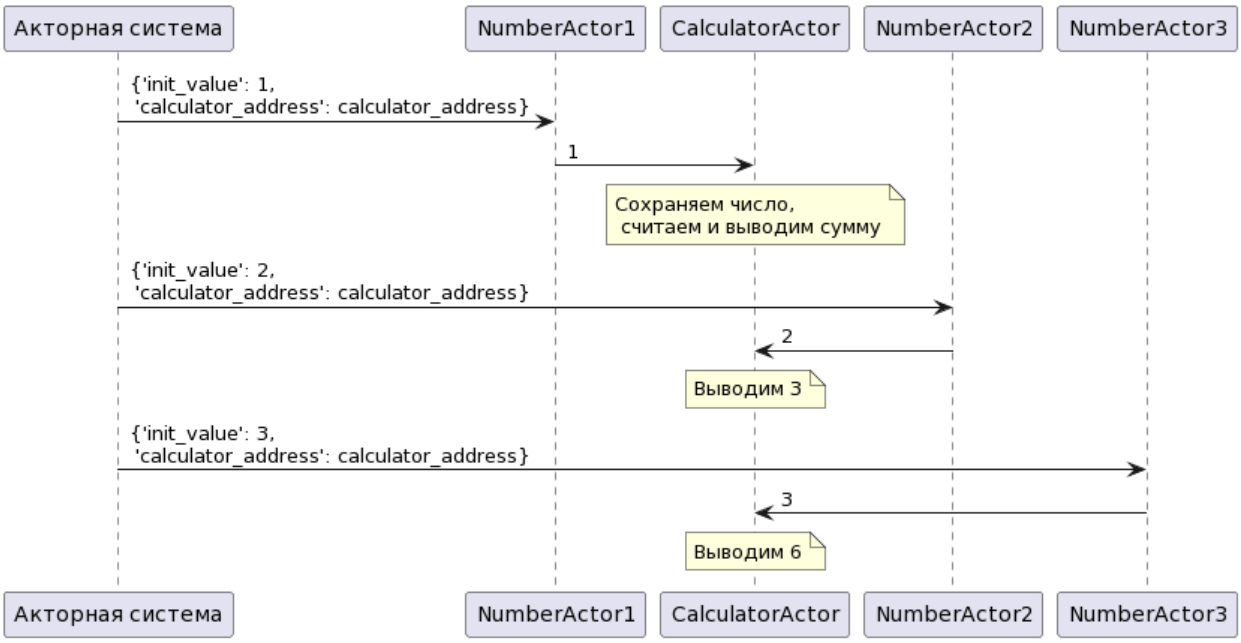

Рис.1.6. Диаграмма последовательности подсчета суммы акторами

1.7. Заключение

В рамках данной работы была создана программа, демонстрирующая основные принципы акторных систем. Следующими шагами будет создание и использование

вспомогательного кода, упрощающего разработку и отладку акторных мультиагентных систем.

Исходный код, написанный в рамках данного занятия, доступен в приложении к уроку.

1.8. Самостоятельная работа

В качестве самостоятельной работы предлагается расширить логику функционирования актора-калькулятора - реализовать в нем возможность не только суммирования, но и других арифметических действий. Для этого необходимо реализовать отправку еще одного типа сообщений и логику по его обработке.

Вопросы для самопроверки:

- Как создаются акторы?
- Как строится простейшая акторная система?
- Чем отличаются акторы от обычных программ или программных объектов?
- Как передаются сообщения между акторами?
- Как строятся переговоры между акторами?

2. Каркас мультиагентной системы

В первом занятии была создана простейшая акторная система, акторы внутри нее обрабатывали сообщения строго определенных типов.

Вся логика по обработке этих сообщений была расположена в одном методе, но при создании реального приложения такой подход значительно увеличит сложность кода.

В рамках данного занятия будут предложены методы по упрощению написания кода акторов.

2.1. Формат сообщений

В предыдущем занятии написанные акторы могли принимать сообщения любых типов. На практике же, если мы хотим создать и поддерживать стандартизированный протокол переговоров, необходимо перечень сообщений типизировать. Для этого создадим файл с описанием классов сообщений: message.py. Типы сообщений, которыми будут обмениваться акторы, вынесем в Enum [https://docs.python.org/3/library/enum.html]. Тогда каждое сообщение может характеризоваться типом и содержать данные. Опишем это как dataclass [https://docs.python.org/3/library/dataclasses.html] из двух полей. Позже, в случае необходимости, можно будет внести дополнительную информацию: например, время формирования или обработки сообщения.

```
from enum import Enum
from dataclasses import dataclass
from typing import Any
```

```
class MessageType(Enum):
    INITIALIZATION = 'Инициализация'
    CREATE_ACTOR = 'Создание актора'
    HELLO_WORLD = 'Приветствие'


@dataclass
class Message:
    """Класс для хранения сообщений"""
    msg_type: MessageType
    msg_body: Any
```

2.2. Иерархия наследования

Создадим файл agent_base.py, в котором создадим промежуточный класс агента - AgentBase. Этот класс будет реализовывать вспомогательную логику взаимодействия в акторной среде. Остальные акторы должны будут наследоваться от этого класса. Базовый класс объявим абстрактным [https://docs.python.org/3/library/abc.html], чтобы избежать возможности инстанцировать его напрямую.

```
from abc import ABC


from thespian.actors import Actor
```

```
class AgentBase(ABC, Actor):
    """
    Базовая реализация агента
    """


    def __init__(self):
        self.name = 'Базовый агент'
```

2.3. Подписка на сообщения

Теперь, когда у каждого сообщения внутри системы есть тип, можно каждому типу сообщения поставить в соответствие обработчик. Для этого реализуем механизм подписки на сообщения по типу. Каждый обработчик будет принимать сообщение и отправителя, поэтому такой обработчик можно типизировать как функцию с двумя параметрами: Callable[[Any, ActorAddress], None]

Тогда для хранения всех обработчиков удобно использовать ассоциативный контейнер: Dict[MessageType, Callable[[Any, ActorAddress], None]] = {}

Этот контейнер определим членом класса, объявив его в методе *__init__*

Добавим в такой контейнер первый обработчик. Он может быть максимально простым:

```
def handle_hello_world(self, message, sender):
    print("Hello, world")
```

Для добавления обработчика напишем метод подписки:

```
def subscribe(self, msg_type: MessageType, handler: Callable[[Any, ActorAddress], None]):
    """
    Подписка на события определенного типа
    :param msg_type: Тип события
    :param handler: Обработчик сообщения заданного типа
    :return:
    """
    if msg_type in self.handlers:
        logging.warning('Повторная подписка на сообщение: %s', msg_type)
    self.handlers[msg_type] = handler
```

Подписка может быть осуществлена, например, в конструкторе агента: self.subscribe(MessageType.HELLO_WORLD, self.handle_hello_world)

И теперь в базовом классе при получении сообщения будем пытаться найти подписку на данный тип сообщения

```
def receiveMessage(self, msg, sender):
    """Обрабатывает сообщения - запускает их обработку в зависимости от типа.
    :param msg:
    :param sender:
    :return:
    """
    logging.debug('%s получил сообщение: %s', self.name, msg)
```

```
        if isinstance(msg, Message):
            message_type = msg.msg_type
            if message_type in self.handlers:
                try:
                    self.handlers[message_type](msg, sender)
                except Exception as ex:
                    traceback.print_exc()
                    logging.error(ex)
            else:
                logging.warning('%s Отсутствует подписка на сообщение: %s', self.name, message_type)
```

Итоговый класс базового агента выглядит следующим образом:

```
"""Содержит базовую реализацию агента с обработкой сообщений"""

from abc import ABC
from typing import Dict, Callable, Any, List
import logging
import traceback

from thespian.actors import Actor, ActorAddress

from message import MessageType, Message


class AgentBase(ABC, Actor):
```

```
        """
        Базовая реализация агента
        """

        def __init__(self):
            self.name = 'Базовый агент'
            self.handlers: Dict[MessageType, Callable[[Any, ActorAddress], None]] = {}
            self.subscribe(MessageType.HELLO_WORLD, self.handle_hello_world)

        def subscribe(self, msg_type: MessageType, handler: Callable[[Any, ActorAddress], None]):
            """
            Подписка на события определенного типа
            :param msg_type: Тип события
            :param handler: Обработчик сообщения заданного типа
            :return:
            """
            if msg_type in self.handlers:
                logging.warning('Повторная подписка на сообщение: %s', msg_type)
            self.handlers[msg_type] = handler

        def receiveMessage(self, msg, sender):
```

```
"""Обрабатывает сообщения - запускает их обработку в зависимости от типа.
:param msg:
:param sender:
:return:
"""
logging.debug('%s получил сообщение: %s', self.name, msg)

if isinstance(msg, Message):
    message_type = msg.msg_type
    if message_type in self.handlers:
        try:
            self.handlers[message_type](msg, sender)
        except Exception as ex:
            traceback.print_exc()
            logging.error(ex)
    else:
        logging.warning('%s Отсутствует подписка на сообщение: %s', self.name, message_type)

def handle_hello_world(self, message, sender):
    print("Hello, world")
```

2.4. Перенос приложения с прошлого занятия

Реализуем на созданном каркасе приложение-калькулятор. Создадим файл number_agent.py, в котором реализуем логику по получению числа в сообщении об инициализации:

```
from thespian.actors import ActorAddress

from agent_base import AgentBase
from message import Message, MessageType


class NumberAgent(AgentBase):
    def __init__(self):
        super().__init__()
        # В это поле будем сохранять полученное значение.
        self.value = 0
        self.subscribe(MessageType.INITIALIZATION,
self.handle_initialization)

    def handle_initialization(self, message: Message, sender:
ActorAddress):
        print(f'Актор числа с адресом {self.myAddress} получил
{message} от {sender}')
        self.value = message.msg_body.get('init_value')
        calculator_address =
message.msg_body.get('calculator_address')
        # Отправляем калькулятору свое значение.
        new_message = Message(MessageType.HELLO_WORLD,
self.value)
        self.send(calculator_address, new_message)
```

Агент-калькулятор будет выглядеть схожим образом:

```
from thespian.actors import ActorAddress

from agent_base import AgentBase
from message import Message


class CalculatorAgent(AgentBase):
    def __init__(self):
        super().__init__()
        # Все полученные числа будем сохранять в списке.
        self.values = []

    def handle_hello_world(self, message: Message, sender: ActorAddress):
        # Ожидаем, что в сообщении будет содержаться число. Добавляем его в список.
        new_value = message.msg_body
        self.values.append(new_value)
        # Суммируем все числа и выводим их в консоль.
        total_sum = sum(self.values)
        print(f'Актор-калькулятор с адресом {self.myAddress} получил {message} от {sender},'
              f' итоговая сумма: {total_sum}')
```

Обращаем внимание, что у этого класса нет явной подписки на обработчик сообщения - метод handle_hello_world, т.к. мы переопределили обработчик базового класса.

Логику по взаимодействию акторов можно написать в файле main.py:

```
from thespian.actors import *
from calculator_agent import CalculatorAgent
from number_agent import NumberAgent
from message import Message, MessageType


if __name__ == '__main__':
    # Создаем систему акторов, внутри которой они будут жить
    actorSystem = ActorSystem()
    # Создаем актор-калькулятор, сохраняем его адрес.
    calculator_address = actorSystem.createActor(CalculatorAgent)
    # Создаем актор-число
    number_agent_1 = actorSystem.createActor(NumberAgent)
    # Формируем сообщение со значением числа и адресом калькулятора.
    init_message_1_data = {'init_value': 1, 'calculator_address': calculator_address}
    # Отправляем актору числа сообщение, после которого он должен отправить другое сообщение калькулятору.
    init_message_1 = Message(MessageType.INITIALIZATION, init_message_1_data)
    actorSystem.tell(number_agent_1, init_message_1)

    number_agent_2 = actorSystem.createActor(NumberAgent)
```

init_message_2_data = {'init_value': 3, 'calculator_address': calculator_address}

init_message_2 = Message(MessageType.INITIALIZATION, init_message_2_data)

actorSystem.tell(number_agent_2, init_message_2)

Ожидается, что вывод приложения будет примерно следующим:

Актор числа с адресом ActorAddr-/A~b получил Message(msg_type=<MessageType.INITIALIZATION: 'Инициализация'>, msg_body={'init_value': 1, 'calculator_address': <thespian.actors.ActorAddress object at 0x0000015409320390>}) от ActorAddr-System:ExternalRequester Актор-калькулятор с адресом ActorAddr-/A~a получил Message(msg_type=<MessageType.HELLO_WORLD: 'Приветствие'>, msg_body=1) от ActorAddr-/A~b, итоговая сумма: 1 Актор числа с адресом ActorAddr-/A~c получил Message(msg_type=<MessageType.INITIALIZATION: 'Инициализация'>, msg_body={'init_value': 3, 'calculator_address': <thespian.actors.ActorAddress object at 0x0000015409320390>}) от ActorAddr-System:ExternalRequester Актор-калькулятор с адресом ActorAddr-/A~a получил Message(msg_type=<MessageType.HELLO_WORLD: 'Приветствие'>, msg_body=3) от ActorAddr-/A~c, итоговая сумма: 4

2.5. Заключение

В этом занятии был создан каркас мультиагентной системы, упрощающий реализацию сложных систем.

Следующими шагами будут расширение созданного каркаса и реализация систем планирования на его основе.

Исходный код данного занятия находится в приложении к уроку.

2.6. Самостоятельная работа

В качестве самостоятельной работы необходимо расширить функции калькулятора, добавив туда возможности по выполнению других арифметических операций.

Вопросы для самопроверки:

- В чем состоит переход от акторов – к программным агентам?
- Как устроена мультиагентная система?
- При каких условиях мультиагентная система становится самоорганизующейся системой?
- Что дает агентам механизм подписки на сообщения?

3. Простейшее решение логистической задачи планирования

3.1. Постановка задачи

Рассмотрим простейшую задачу планирования логистической компании.

Необходимо доставить грузы из одной точки в другую с помощью курьеров.

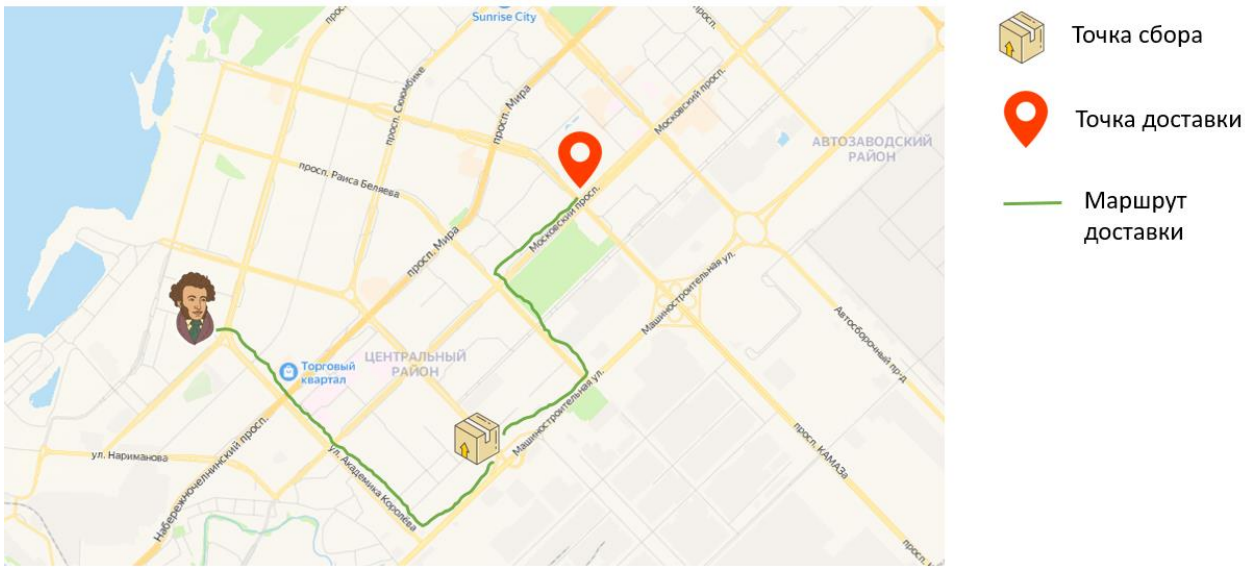

Рис. 3.1. Маршрут доставки груза

Напомним, что решение такой задачи формируется за счет переговоров в мультиагентной среде. Протокол переговоров для поиска решения можно представить в следующем виде:

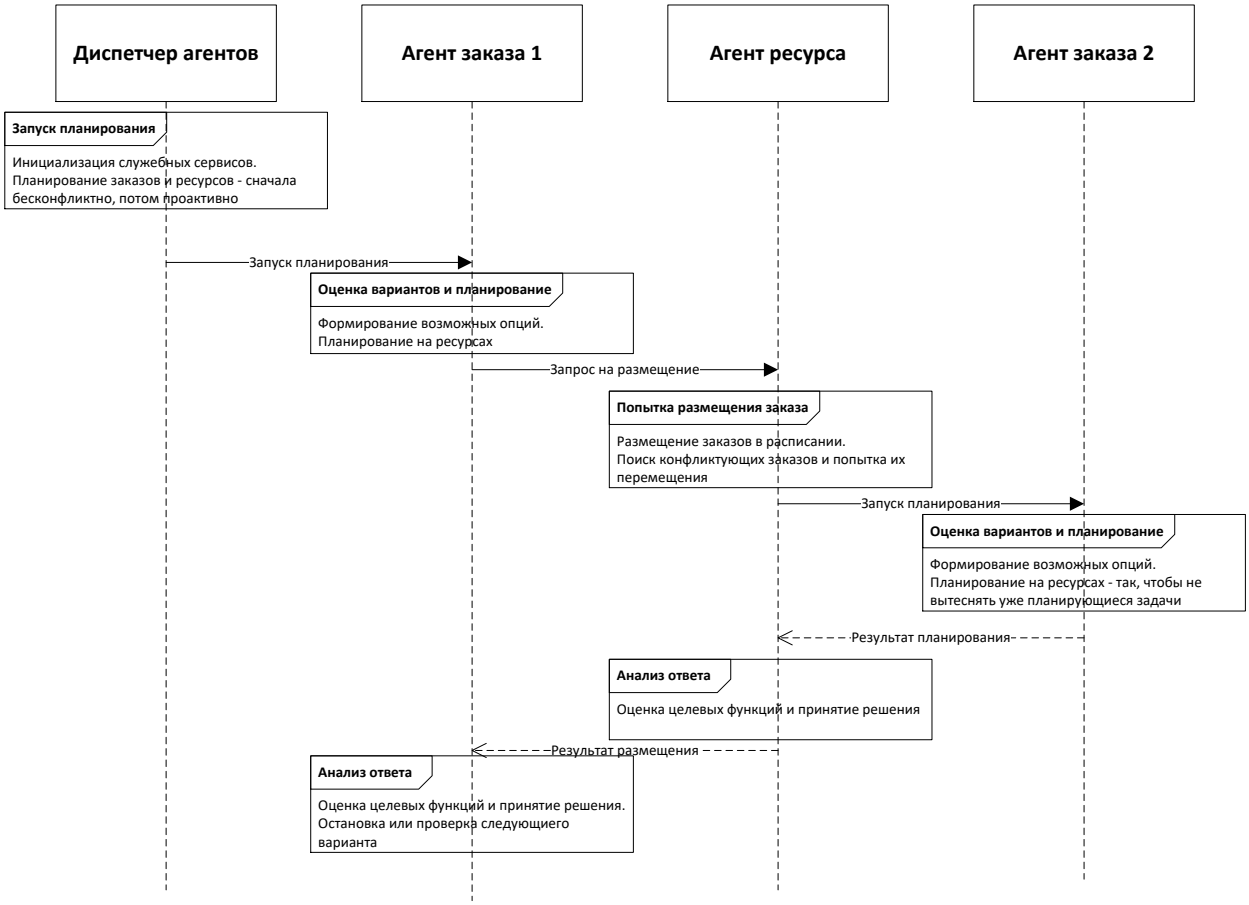

Рис. 3.2. Общая схема переговоров

Каждый курьер будет формировать свой собственный график доставки грузов, который можно представить как набор записей в расписании.

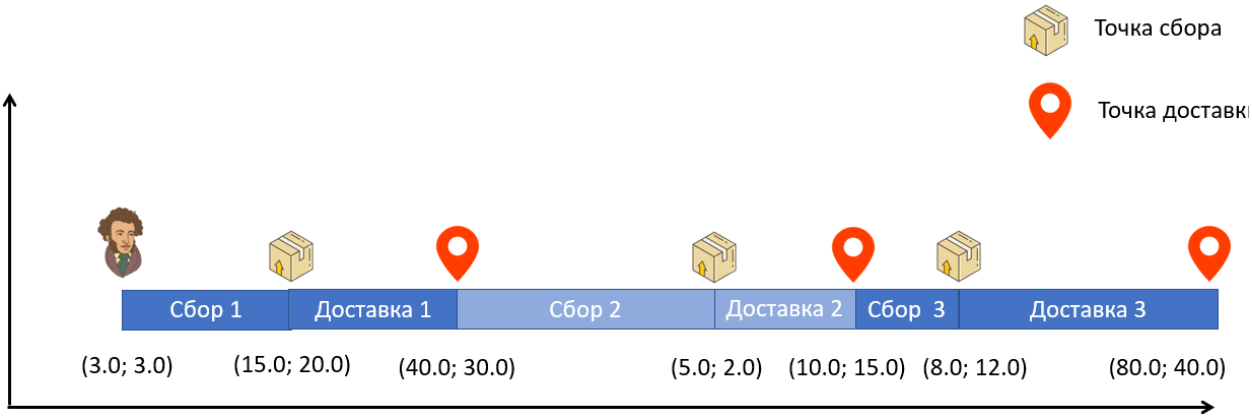

Рис. 3.3. График доставки грузов одного курьера

В качестве входных данных будем рассматривать набор курьеров компании и перечень заказов, которые необходимо запланировать.

Эта информация будет читаться из файла MS Excel и содержать следующие поля:

1. Заказы:
   a. Номер
   b. Наименование
   c. Масса
   d. Стоимость
   e. Координаты получения заказа - x, y
   f. Координаты доставки заказа - x, y
   g. Временной интервал получения заказа
   h. Временной интервал доставки заказа
   i. Тип - строка
2. Курьеры:
   a. Номер
   b. ФИО
   c. Координаты первоначального положения - x, y
   d. Типы доставляемых заказов
   e. Стоимость выхода на работу
   f. Цена работы за единицу времени
   g. Скорость

Этот перечень максимально упрощен и может быть расширен.

В качестве временных интервалов рассматривается относительное время от момента t0.

В рамках данного занятия рассмотрим простейший вариант построения плана - при котором заказ будет планироваться на первого возможного курьера.

3.2. Чтение входных данных

Для чтения данных из файла и последующего сохранения можно использовать фреймворки *pandas* и *openpyxl*, необходимо установить их в активированном виртуальном окружении в соответствии с первым уроком.

Информацию о курьерах и заказах можно записать на разные листы исходного файла.

В таком случае код для чтения данных можно реализовать в виде функции, принимающей наименование файла и название листа.

Возвращать эта функция будет список словарей, содержащих информацию об атрибутах заказов и курьеров.

```
import pandas as pd



def get_excel_data(filename, sheet_name) -> typing.List:
    """
    Возвращает данные из Excel-файла
    :param filename:
    :param sheet_name:
    :return:
    """
    df = pd.read_excel(filename, sheet_name=sheet_name)
```

```
    df_index = df.to_dict('index')
    resulted_list = list(df_index.values())
    return resulted_list
```

Ключом в каждом словаре будет являться наименование атрибута, прочитанное из первой строки, а значением - значение указанного атрибута.

3.3. Расширение каркаса мультиагентной системы классами предметной области

Все сущности планирования будут жить в **сцене мира агентов** - виртуальном представлении реального мира.

Для простоты можно рассматривать сцену как простой контейнер, содержащий сущности с разделением по типу:

```
class Scene:
    """
    Класс сцены
    """
    def __init__(self):
        self.entities = defaultdict(list)
```

Реализуем модель предметной области - классы курьеров и заказов. Для этого создадим базовый класс, от которого они будут наследоваться:

```
class BaseEntity:
    """Базовая сущность"""


    def __init__(self, onto_desc, scene=None):
```

```
        self.onto_description = onto_desc
        self.name = onto_desc.get('data', {}).get('label', {}).get('value')
        # self.uri = self.onto_description.get('uri')
        self.uri = 'UNKNOWN_URI'
        self.scene = scene
```

Онтологическое описание, которое принимает конструктор сущности, может быть прочитано на следующих уроках из базы знаний. Поле *uri* - это уникальный идентификатор ресурса (URI - https://ru.wikipedia.org/wiki/URI), атрибут сцены хранит ссылку на общую сцену мира для доступа к другим сущностям.

В таком случае классы курьера и заказа можно реализовать таким образом:

```
class OrderEntity(BaseEntity):
    """
    Класс заказа
    """
    def __init__(self, onto_desc: {}, init_dict_data, scene=None):
        super().__init__(onto_desc, scene)
        self.number = init_dict_data.get('Номер')
        self.name = init_dict_data.get('Наименование')
        self.weight = init_dict_data.get('Масса')
        self.volume = init_dict_data.get('Объем')
        self.price = init_dict_data.get('Стоимость')
        x1 = init_dict_data.get('Координата получения x')
        y1 = init_dict_data.get('Координата получения y')
        self.point_from = Point(x1, y1)
```

```
        x2 = init_dict_data.get('Координата доставки x')
        y2 = init_dict_data.get('Координата доставки y')
        self.point_to = Point(x2, y2)
        self.time_from = init_dict_data.get('Время получения заказа')
        self.time_to = init_dict_data.get('Время доставки заказа')
        self.order_type = init_dict_data.get('Тип заказа')

        self.uri = 'Order' + str(self.number)


    class CourierEntity(BaseEntity):
        """
        Класс заказа
        """
        def __init__(self, onto_desc: {}, init_dict_data, scene=None):
            super().__init__(onto_desc, scene)
            self.number = init_dict_data.get('Табельный номер')
            self.name = init_dict_data.get('ФИО')
            x1 = init_dict_data.get('Координата начального положения x')
            y1 = init_dict_data.get('Координата начального положения y')
            self.init_point = Point(x1, y1)
            self.types = [_type.lstrip() for _type in init_dict_data.get('Типы доставляемых заказов', '').split(';')]
            self.cost = init_dict_data.get('Стоимость выхода на работу')
            self.rate = init_dict_data.get('Цена работы за единицу времени')
```

```
self.velocity = init_dict_data.get('Скорость')
self.max_volume = init_dict_data.get('Объем ранца')
self.max_mass = init_dict_data.get('Грузоподъемность')

self.uri = 'Courier' + str(self.number)
```

Выше показаны только конструкторы классов. Для удобства для дальнейшей работы классы рекомендуется расположить в разных файлах, расширить функциональность классов, реализуя вспомогательные методы.

3.4. Структура расписания

Заказы, которые будет доставлять курьер, размешаются в его расписании. В примерах ниже расписание будет иметь простейший вид и являться список записей. Каждая запись характеризуется временем начала и окончания, типом, заказом, к которому она относится, и дополнительными параметрами. В коде это можно представить в виде простого класса:

```
@dataclass
class ScheduleItem:
    """
    Класс записи расписания
    """
    order: OrderEntity
    rec_type: str
    start_time: int
```

```
end_time: int
point_from: Point
point_to: Point
cost: float
all_params: dict
```

Тогда весь класс расписания может выглядеть как список таких записей:

```
self.schedule: typing.List[ScheduleItem] = []
```

В графическом виде каждую запись можно представить следующим образом:

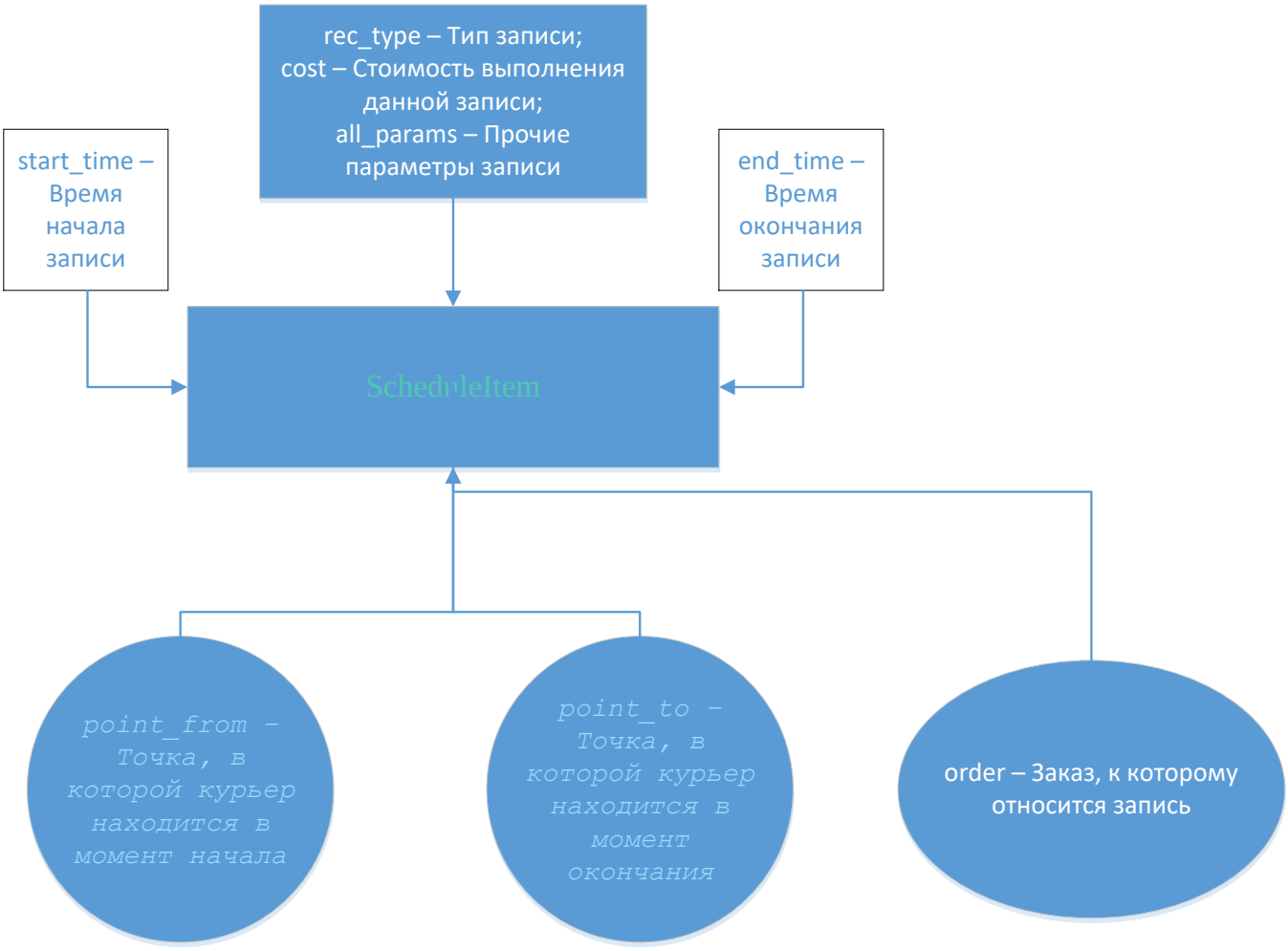

Рис. 3. 4. Структура записи расписания

3.5. Реализация адресной книги агентов

У сущностей реального мира появляются их представители в мире виртуальном - агенты. Каждый агент будет привязан к сущности, понять адрес другого агента можно с помощью вспомогательного класса - адресной книги:

```
class ReferenceBook:
    """Адресная книга агентов с привязкой к сущностям"""
    def __init__(self):
        self.agents_entities = {}

    def add_agent(self, entity, agent_address):
        """
        Сохраняет адрес агента с привязкой с сущности
        :param entity:
        :param agent_address:
        :return:
        """
        self.agents_entities[entity] = agent_address

    def get_address(self, entity):
        """
        Возвращает адрес агента указанной сущности.
        :param entity:
        :return:
        """
        if entity not in self.agents_entities:
```

```
            logging.error(f'Агент {entity} отсутствует в адресной книге')
            return None
        return self.agents_entities[entity]
```

Создадим класс диспетчера агентов, который содержит акторную систему для переговоров, адресную книгу и сцену. Этот класс будет реализовывать логику по взаимодействию агентов:

```
class AgentsDispatcher:
    """
    Класс диспетчера агентов
    """
    def __init__(self, scene):
        self.actor_system = ActorSystem()
        self.reference_book = ReferenceBook()
        self.scene = scene


    def add_entity(self, entity: BaseEntity):
        """
        Добавляет сущность в сцену, создает агента и отправляет ему сообщение об инициализации
        :param entity:
        :return:
        """
        entity_type = entity.get_type()
        agent_type = TYPES_AGENTS.get(entity_type)
        if not agent_type:
```

```
            logging.warning(f'Для сущности типа {entity_type} не указан агент')
            return False
        self.scene.entities[entity_type].append(entity)
        self.create_agent(agent_type, entity)
        return True

    def create_agent(self, agent_class, entity):
        """
        Создает агента заданного класса с привязкой к сущности
        :param agent_class:
        :param entity:
        :return:
        """
        agent = self.actor_system.createActor(agent_class)
        self.reference_book.add_agent(entity=entity, agent_address=agent)
        init_data = {'dispatcher': self, 'scene': self.scene, 'entity': entity}
        init_message = Message(MessageType.INIT_MESSAGE, init_data)
        self.actor_system.tell(agent, init_message)
```

Полный код этого и других классов содержится в архиве, являющегося приложением к уроку.

Расширим базовую реализацию агента, добавив туда подписку на сообщение об инициализации:

```
def handle_init_message(self, message, sender):
```

```
        """
        Обработка сообщения с инициализацией - сохранение присланных данных в агенте.
        :param message:
        :param sender:
        :return:
        """
        message_data = message.msg_body
        self.scene = message_data.get('scene')
        self.dispatcher = message_data.get('dispatcher')
        self.entity = message_data.get('entity')
        self.name = self.name + ' ' + self.entity.name
        logging.info(f'{self} проинициализирован')
```

Агенты, которые мы будут создаваться, относятся к заказам и курьерам:

```
class OrderAgent(AgentBase):
    """
    Класс агента заказа
    """
    def __init__(self):
        super().__init__()
        self.entity: OrderEntity
        self.name = 'Агент заказа'


class CourierAgent(AgentBase):
```

```
    """
    Класс агента курьера
    """
    def __init__(self):
        super().__init__()
        self.entity: CourierEntity
        self.name = 'Агент курьера'
```

После этого можно объединить все реализованные части кода - чтение исходных данных, создание сущностей и инициализация агентов. Для этого в файле main.py реализуем такую логику:

```
if __name__ == "__main__":
    logging.basicConfig(level=logging.INFO, format="%(asctime)s %(levelname)s %(message)s")

    logging.info("Добро пожаловать в мир агентов")
    scene = Scene()
    dispatcher = AgentsDispatcher(scene)

    couriers = get_excel_data('Исходные данные.xlsx', 'Курьеры')
    logging.info(f'Прочитаны курьеры: {couriers}')
    for courier in couriers:
        onto_description = {}
        entity = CourierEntity(onto_description, courier, scene)
        dispatcher.add_entity(entity)

    orders = get_excel_data('Исходные данные.xlsx', 'Заказы')
```

```
logging.info(f'Прочитаны заказы: {orders}')
for order in orders:
    onto_description = {}
    entity = OrderEntity(onto_description, order, scene)
    dispatcher.add_entity(entity)
```

3.6. Реализация базового простейшего алгоритма планирования

Следующим шагом необходимо реализовать логику по планированию заказов на курьерах. Её можно представить в виде следующих шагов:

1. Агенты заказов запрашивают перечень курьеров
2. Агенты заказов запрашивают у курьеров список возможных интервалов доставки
3. Агенты курьеров формируют перечень всех возможных вариантов доставки, исходя из своего расписания
4. Агенты заказов выбирают наиболее подходящий для себя вариант и пробуют запланироваться на нем

Запрос списка курьеров может быть реализован максимально просто - путем обращения к сцене мира.

```
all_couriers: typing.List[CourierEntity] = self.scene.get_entities_by_type('COURIER')
```

В реальной системе, работающей в нескольких потоках или же на разных компьютерах, использование общей памяти недопустимо, такой подход можно использовать только для учебных задач.

Важно: примеры кода в этом уроке могут не работать при вставке их в явном виде - например, необходимо добавить дополнительные импорты. Полный код есть в примере, являющимся приложением к уроку.

После этого можно отправить всем курьерам запрос на получение возможных параметров доставки. Для этого необходимо расширить перечень возможных сообщений:

```
PRICE_REQUEST = 'Запрос цены'
```

Код по запросу параметров реализуется в агенте заказа и выглядит следующим образом:

```
def __send_params_request(self):
    all_couriers: typing.List[CourierEntity] = self.scene.get_entities_by_type('COURIER')
    logging.info(f'{self} - список курьеров: {all_couriers}')
    for courier in all_couriers:
        courier_address = self.dispatcher.reference_book.get_address(courier)
        logging.info(f'{self} - адрес {courier}: {courier_address}')
        request_message = Message(MessageType.PRICE_REQUEST, self.entity)
        self.send(courier_address, request_message)
        self.unchecked_couriers.append(courier_address)
```

Агент курьера должен подписаться на это сообщение в конструкторе и реализовать логику по его обработке:

```
def handle_price_request(self, message, sender):
```

```
        """
        Обработка сообщения с запросом параметров заказа.
        Выполняет расчет в зависимости от текущего расписания курьера.
        :param message:
        :param sender:
        :return:
        """
        order: OrderEntity = message.msg_body
        params = self.__get_params(order)
        price_message = Message(MessageType.PRICE_RESPONSE, params)
        self.send(sender, price_message)

    def __get_params(self, order: OrderEntity) -> typing.List:
        """
        Формирует возможные варианты размещения заказа
        :param order:
        :return:
        """
        p1 = order.point_from
        # Надо посчитать стоимость выполнения заказа, сроки доставки
        last_point: Point = self.entity.get_last_point()
        distance_to_order = last_point.get_distance_to_other(p1)
```

```
        p2 = order.point_to
        distance_with_order = p1.get_distance_to_other(p2)
        time_to_order = distance_to_order / self.entity.velocity
        time_with_order = distance_with_order / self.entity.velocity
        duration = time_to_order + time_with_order

        price = duration * self.entity.rate
        logging.info(f'{self} - заказ {order} надо пронести {distance_with_order},'
                     f' к нему идти {distance_to_order}'
                     f'это займет {duration} и будет стоить {price}')

        last_time = self.entity.get_last_time()

        asap_time_from = last_time
        asap_time_to = asap_time_from + time_to_order + time_with_order
        # Вариант, при котором мы выполняем заказ как только можем
        asap_variant = {'courier': self.entity, 'time_from': asap_time_from, 'time_to': asap_time_to,
                        'price': price, 'order': order, 'variant_name': 'asap'}
        params = [asap_variant]

        return params
```

Для работы вышеуказанного кода необходимо расширить перечень сообщений, добавив туда

PRICE_RESPONSE = 'Ответ цены'

Этот метод может быть расширен другими вариантами доставки - например, при котором заказ забирается и/или доставляется вовремя.

Агент заказа должен дождаться, пока ему ответят все курьеры, после чего выбрать наилучший для себя вариант:

```
def handle_price_response(self, message, sender):
    logging.info(f'{self} - получил сообщение {message}')
    courier_variants = message.msg_body

    self.possible_variants.extend(courier_variants)
    self.unchecked_couriers.remove(sender)
    if not self.unchecked_couriers:
        self.__run_planning()

def __run_planning(self):
    if not self.possible_variants:
        logging.info(f'{self} - нет возможных вариантов для планирования')
        return
    # Оцениваем варианты
    self.__evaluate_variants()
    # TODO: необходимо сортировать варианты
    sorted_vars = self.possible_variants
```

```
        # Наилучший
        best_variant = sorted_vars[0]
        # Адрес лучшего варианта
        best_variant_address = self.dispatcher.reference_book.get_address(best_variant.get('courier'))

        logging.info(f'{self} - лучшим вариантом признан {best_variant}, '
                     f'адрес - {best_variant_address}')
        request_message = Message(MessageType.PLANNING_REQUEST, best_variant)
        self.send(best_variant_address, request_message)
```

Агент курьера при получении этого типа сообщения должен попытаться добавить заказ в свое расписание в соответствии с указанными параметрами:

```
    def handle_planning_request(self, message, sender):
        """
        Обработка сообщения с запросом на планирования.
        :param message:
        :param sender:
        :return:
        """
        params = message.msg_body
        # Пытаемся добавить заказ в свое расписание
        adding_result = self.entity.add_order_to_schedule(params.get('order'),
```

```
                                        params.get('time_from'),
                                        params.get('time_to'),
                                        params.get('price'),
                                        params)

        conflicted_records = self.entity.get_conflicts(params.get('time_from'), params.get('time_to'))
        # Что будем делать с конфликтами?

        logging.info(f'{self} получил запрос на размещение {params}, '
                     f'результат - {adding_result}, конфликты - {conflicted_records}')

        result_msg = Message(MessageType.PLANNING_RESPONSE, adding_result)
        self.send(sender, result_msg)
```

Для работы вышеуказанного кода необходимо расширить перечень сообщений, добавив туда

```
PLANNING_RESPONSE = 'Ответ на размещение'
```

И наконец, чтобы запустить переговоры, расширим подписку на инициализацию агента заказа:

```
    def handle_init_message(self, message, sender):
        super().handle_init_message(message, sender)
        # Ищем в системе ресурсы и отправляем им запросы
        self.__send_params_request()
```

Решение можно представить в виде диаграммы последовательности:

@startuml

note over Заказ1 : Формирование списка курьеров

Заказ1 -> Курьер1: Запрос параметров

Заказ1 -> Курьер2: Запрос параметров

note over Заказ2 : Формирование списка курьеров

Заказ2 -> Курьер1: Запрос параметров

Заказ2 -> Курьер2: Запрос параметров

Курьер1 -> Заказ1: Параметры: время, цена

note over Заказ1 : Все курьеры ответили? \n Нет, ожидаем всех ответов

Курьер1 -> Заказ2 : Параметры: время, цена

note over Заказ2 : Все курьеры ответили? \n Нет, ожидаем всех ответов

Курьер2 -> Заказ1: Параметры: время, цена

note over Заказ1 : Все курьеры ответили? \n Да, оцениваем варианты

note over Заказ1 : Выбор лучшего варианта

Заказ1 -> Курьер1: Прошу запланировать по варианту

Курьер1 -> Заказ1: Принято, вариант запланирован

Курьер2 -> Заказ2 : Параметры: время, цена

note over Заказ2 : Все курьеры ответили? \n Да, оцениваем варианты

note over Заказ2 : Выбор лучшего варианта

Заказ2-> Курьер1: Прошу запланировать по варианту

Курьер1 -> Заказ2: Отказ, вариант уже невыполним

note over Заказ2: Выбор лучшего варианта

Заказ2-> Курьер2 : Прошу запланировать по варианту

Курьер2 -> Заказ2: Принято, вариант запланирован

@enduml

Визуализация таких переговоров выглядит так:

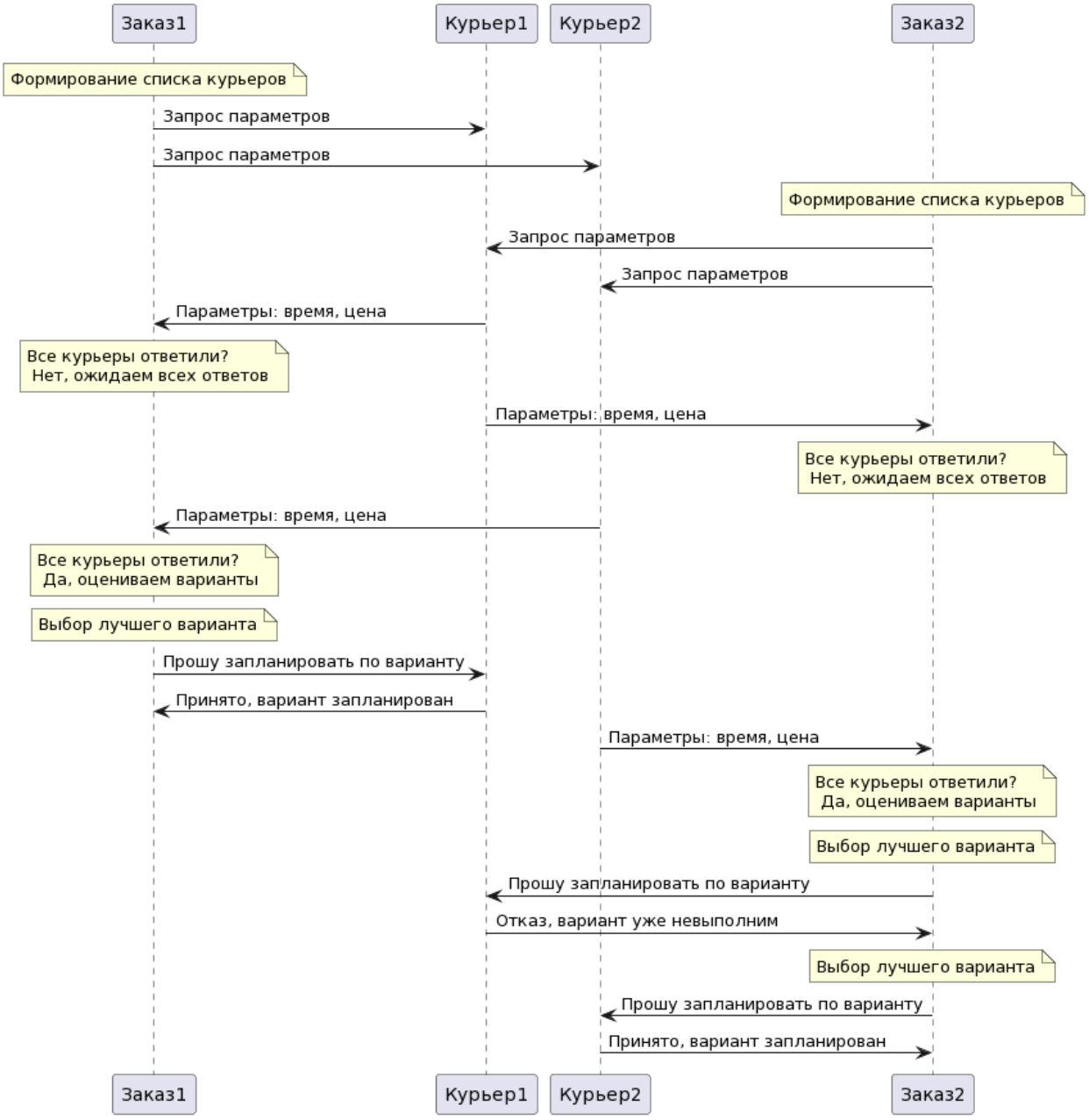

Рис. 3.5. Пример переговоров по поиску курьера для заказа

Полный код приложения, реализующего заданную логику, приведен в приложении к уроку.

3.7. Заключение

В рамках данной работы была создана программа, реализующая мультиагентный подход к решению логистической задачи. Данное

решение не является оптимизированным, т.к. строится на первых возможных вариантах.

3.8. Самостоятельная работа

В качестве самостоятельной работы предлагается расширить логику функционирования программы - разделить записи расписания курьера в зависимости от выполняемых действий: движение за грузом, ожидание или же движение с грузом.

Вопросы для самоконтроля:

- Опишите постановку задачи логистической компании;
- Определите основные классы агентов;
- Определите понятие сцены и поясните, как она используется;
- Поясните принципы формирования протоколов взаимодействия между агентами;
- Дайте примеры взаимодействия различных классов агентов.

4. Матчинг заказов и ресурсов. Критерии планирования. Многокритериальное планирование. Реализация линейной свёртки критериев.

В предыдущем занятии мы реализовали простейший вариант распределения заказов между курьерами.

Это решение не является оптимальным, т.к. не учитывает никакие из критериев планирования.

В качестве таких критериев можно выделить:

- Время получения заказа - чем позже, тем лучше;
- Время доставки заказа – чем раньше, тем лучше;
- Стоимость - чем меньше, тем лучше.

Дополнительной особенностью планирования является ряд ограничений - например, по товарной совместимости.

В исходных данных уже есть поля “Тип заказа” и “Типы доставляемых заказов” - в первой версии планирования эти данные никак не учитываются.

4.1. Матчинг заказов и ресурсов

Реализуем базовый матчинг заказов и ресурсов - то есть поиск соответствия между ними по каким-либо параметрам: тип заказа, объем и масса. Как было сказано выше, такое соответствие можно построить, например, по типу заказов. Для этого модифицируем метод агента заказа, который работает со списком возможных курьеров:

def __send_params_request(self):

```
        all_couriers: typing.List[CourierEntity] = self.scene.get_entities_by_type('COURIER')
        logging.info(f'{self} - список ресурсов: {all_couriers}')
        for courier in all_couriers:
            if self.entity.order_type not in courier.types:
                logging.info(f'{self} - типы грузов {courier} - {courier.types} '
                             f'не включают {self.entity.order_type}')
                continue
            courier_address = self.dispatcher.reference_book.get_address(courier)
            logging.info(f'{self} - адрес {courier}: {courier_address}')
            request_message = Message(MessageType.PRICE_REQUEST, self.entity)
            self.send(courier_address, request_message)
            self.unchecked_couriers.append(courier_address)
```

Расширим логику матчинга. Если масса и объем заказа будут больше максимально возможных значений курьера, то он при получении запроса параметров сразу может отвечать отрицательно. Для этого изменим метод __get_params агента курьера, добавив туда следующие строки:

```
        if order.weight > self.entity.max_mass:
            return []
        if order.volume > self.entity.max_volume:
            return []
```

Такое разделение логики матчинга позволит в будущем дополнить логику - например, курьер может отказаться от заведомо невыгодных (очень далеких от него) заказов.

4.2. Критерии планирования

Каждый вариант планирования может быть оценен по критериям, указанных в начале планирования. Но для сравнения величин разной размерности (времени и стоимости) необходимо провести их нормирование в диапазон от 0 до 1. Для этого вычислить минимальное и максимальное значение указанных величин и оценить их в зависимости от типа критерия (будет ли функция возрастающей или убывающей). Для этого в базовой реализации агента создадим две функции, осуществляющие данную логику:

```
@staticmethod
    def get_decreasing_kpi_value(value: float, min_value: float, max_value: float):
        """
        Функция возвращает значение убывающей линейной функции (f(x)) с областью определения [minValue, maxValue], нормированное к единице.
        f(minValue) === 1. f(maxValue) === 0
        :param value:
        :param min_value:
        :param max_value:
        :return:
        """
```

```
        if max_value == min_value:
            return 1

        if value > max_value or value < min_value:
            return -1

        if isinstance(value, float) and abs(value - min_value) < 0.000001:
            # Работаем с минимальным значением => возвращаем 1
            return 1
        if isinstance(value, float) and abs(value - max_value) < 0.000001:
            # Работаем с максимальным значением => возвращаем 0
            return 0
        # Решаем уравнение прямой по двум точкам. f(minValue) == 1; f(maxValue) == 0
        return 1 - (value - min_value) / (max_value - min_value)

    @staticmethod
    def get_increasing_kpi_value(value: float, min_value: float, max_value: float):
        """
        Функция возвращает значение возрастающей линейной функции
        (f(x)) с областью определения [minValue, maxValue], нормированное к единице.
        f(minValue) === 0. f(maxValue) === 1
        :param value:
```

```
        :param min_value:
        :param max_value:
        :return:
        """

        if max_value == min_value:
            # Работаем с прямой, все значения - единицы
            return 1

        if value > max_value or value < min_value:
            # Некорректные данные, выходят из диапазона
            return -1
        if isinstance(value, float) and abs(value - min_value) < 0.000001:
            # Работаем с минимальным значением => возвращаем 0
            return 0
        if isinstance(value, float) and abs(value - max_value) < 0.000001:
            # Работаем с максимальным значением => возвращаем 1
            return 1

        # Решаем уравнение прямой по двум точкам. f(minValue) == 0; f(maxValue) == 1
        return (value - min_value) / (max_value - min_value)
```

Для корректной работы этих функций агент заказа должен минимальное и максимальное значение интересующих нас величин (времени получения и доставки, стоимости):

```
    def __evaluate_variants(self):
```

```
        """
        Оцениваем варианты по критериям и расширяем информацию о них
        :return:
        """
        if not self.possible_variants:
            return
        all_start_times = [var.get('time_from') for var in self.possible_variants]
        min_start_time = min(all_start_times)
        max_start_time = max(all_start_times)
        all_finish_times = [var.get('time_to') for var in self.possible_variants]
        min_finish_time = min(all_finish_times)
        max_finish_time = max(all_finish_times)
        all_prices = [var.get('price') for var in self.possible_variants]
        min_price = min(all_prices)
        max_price = max(all_prices)
        logging.info(f'{self} минимальный старт: {min_start_time}, минимальное завершение - {min_finish_time}, '
                     f'минимальная цена - {min_price}')
```

После этого можно дополнить возможные варианты размещения необходимыми параметрами:

```
        for variant in self.possible_variants:
```

start_efficiency = self.get_decreasing_kpi_value(variant.get('time_from'), min_start_time, max_start_time)

finish_efficiency = self.get_increasing_kpi_value(variant.get('time_to'), min_finish_time, max_finish_time)

price_efficiency = self.get_decreasing_kpi_value(variant.get('price'), min_price, max_price)

variant['start_efficiency'] = start_efficiency # [0; 1]

variant['finish_efficiency'] = finish_efficiency # [0; 1]

variant['price_efficiency'] = price_efficiency # [0; 1]

В результате, каждый вариант планирования являться словарем следующего вида:

{

'courier': Курьер Пушкин А. С. ,

'time_from': 0, 'time_to': 0.7885618083164126,

'price': 11.82842712474619,

'order': Заказ Доставка ноутбука,

'variant_name': 'asap',

'start_efficiency': 1.0,

'finish_efficiency': 0.05685180632430151,

'price_efficiency': 1

}

4.3. Выбор критерия для планирования

Система планирования может считать главным какой-то один из приведенных выше критериев (start_efficiency, finish_efficiency,

price_efficiency). В таком случае метод запуска планирования __run_planning можно модифицировать следующим образом:

```
def __run_planning(self):
    if not self.possible_variants:
        logging.info(f'{self} - нет возможных вариантов для планирования')
        return
    # Оцениваем варианты
    self.__evaluate_variants()
    # Сортируем варианты
    sorted_vars = sorted(self.possible_variants, key=lambda x: x.get('finish_efficiency'))
    # Наилучший
    best_variant = sorted_vars[0]
    # Адрес лучшего варианта
    best_variant_address = self.dispatcher.reference_book.get_address(best_variant.get('courier'))

    logging.info(f'{self} - лучшим вариантом признан {best_variant}, '
                 f'адрес - {best_variant_address}')
    request_message = Message(MessageType.PLANNING_REQUEST, best_variant)
    self.send(best_variant_address, request_message)
```

В коде выше выбирается вариант, у которого значение критерия finish_efficiency максимально, и попытка размещения идет именно на

нем. Если нам важен другой критерий, то код можно модифицировать - и будет выбран другой вариант!

4.4. Свертка критериев

На практике важных для планирования критериев может быть больше одного. В таком случае необходимо реализовать многокритериальное планирование - например, с помощью линейной свертки критериев. Для этого введем веса каждого критерия - таким образом, чтобы сумма всех весов была равна 1. В таком случае можно дополнить варианты размещения еще одним полем - значением свертки, являющейся суммой произведения веса критериев на значение целевой функции по этому критерию:

```
# Итоговая оценка варианта должна учитывать все критерии
        # (в какой-то пропорции)
        finish_weight = 0.2
        start_weight = 0.3
        price_weight = 0.5
        variant['total_efficiency'] = finish_weight * finish_efficiency + start_weight * start_efficiency +\
                                      price_weight * price_efficiency
```

Тогда выбор лучшего варианта должен идти по параметру total_efficiency:

```
        # Сортируем варианты
        sorted_vars = sorted(self.possible_variants, key=lambda x: x.get('total_efficiency'))
```

4.5. Опции планирования

В текущей версии заказы добавляются в “хвост” расписания курьера. Такой подход формирует бесконфликтный план, который не учитывает “пожелания” заказов. Соответственно, выбор вариантов существенно ограничен: каждый курьер предлагает только одну опцию размещения. Расширим эту логику, добавив в метод __get_params дополнительный вариант:

```
if asap_time_from < order.time_from:
        # Генерируем вариант, при котором мы заберем заказ вовремя
        jit_time_from = order.time_from - time_to_order
        # JIT-вариант можно генерировать и при наличии записей, но надо расширить
        # проверки на возможность доставки.
        if jit_time_from > 0 and not self.entity.schedule:
            jit_time_to = jit_time_from + time_to_order + time_with_order
            jit_from_variant = {'courier': self.entity, 'time_from': jit_time_from, 'time_to': jit_time_to,
                                'price': price, 'order': order, 'variant_name': 'jit'}
            params.append(jit_from_variant)
```

Этот код предлагает (если это физически допустимо) разместить заказ не сразу, как только это возможно, а забрать его в заданный момент времени. Тогда у агента заказа появляется больше вариантов - и план может стать более качественным.

4.6. Заключение

В рамках данной работы программа планирования стала учитывать ограничения заказов и ресурсов, оптимизировать решение по заданному набору критериев.

Исходный код, написанный в рамках данного занятия, доступен в приложении к уроку.

4.7. Самостоятельная работа

В качестве самостоятельной работы необходимо:

1. Добавить еще один вариант размещения заказа - при котором заказ будет доставлен вовремя.
2. Предложить и реализовать еще один критерий планирования.

Вопросы для самоконтроля:

- Сформулируйте примеры критериев для планирования ресурсов;
- Что такое свертка критериев?
- Поясните понятие матчинга;
- Приведите примеры матчинга агентов;
- Что такое опции (варианты), которые возникают в ходе планирования?
- Как агенты выбирают лучшее решение из множества различных решений?

5. Разбор конфликтов

В предыдущем занятии все заказы добавлялись в свободные интервалы расписания курьера.

Такой подход формирует бесконфликтный план, который учитывает “пожелания” заказов весьма ограниченно и зависит от порядка добавления заказов и курьеров.

При этом, если курьеры будут модифицировать план в зависимости от новых предложений, то итоговое распределение заказов может стать более выгодным или лучше и по другим критериям. Пример такой ситуации приведен на рисунке ниже – новый заказ может быть «подхвачен» одним курьером в нескольких вариантах.

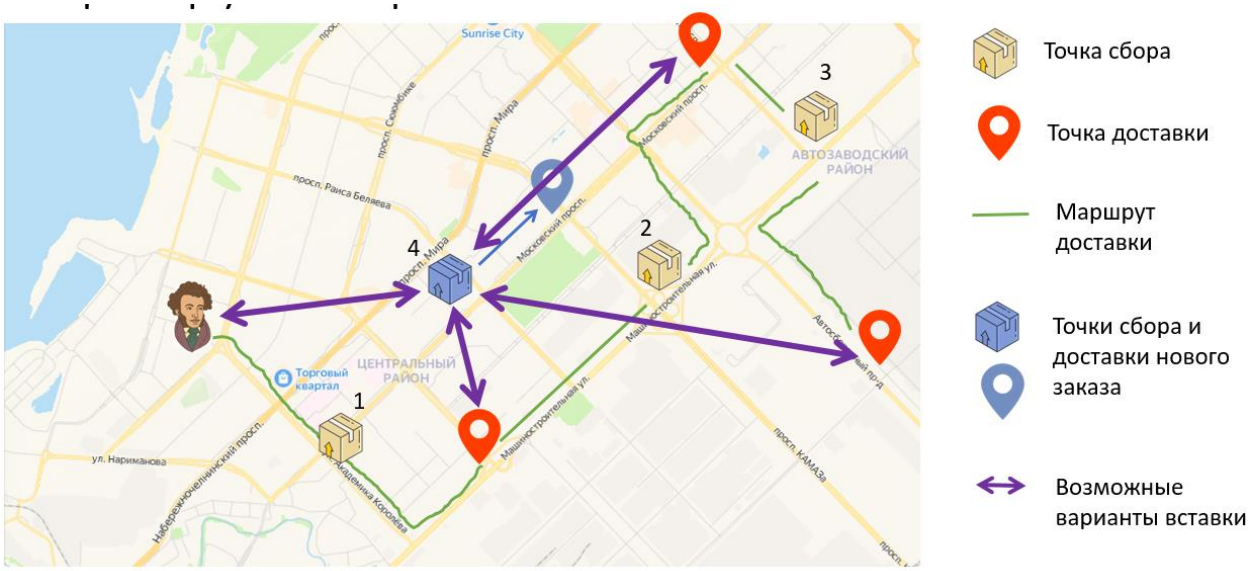

Рис. 5.1. Добавление нового заказа

В рамках этого урока рассмотрим решение простейших конфликтов между заказами.

5.1. Подготовка тестового сценария для проверки конфликтного варианта размещения

Рассмотрим поиск конфликтного варианта для размещения заказа на примере. Пусть изначально в расписании курьера уже было размещено два заказа.

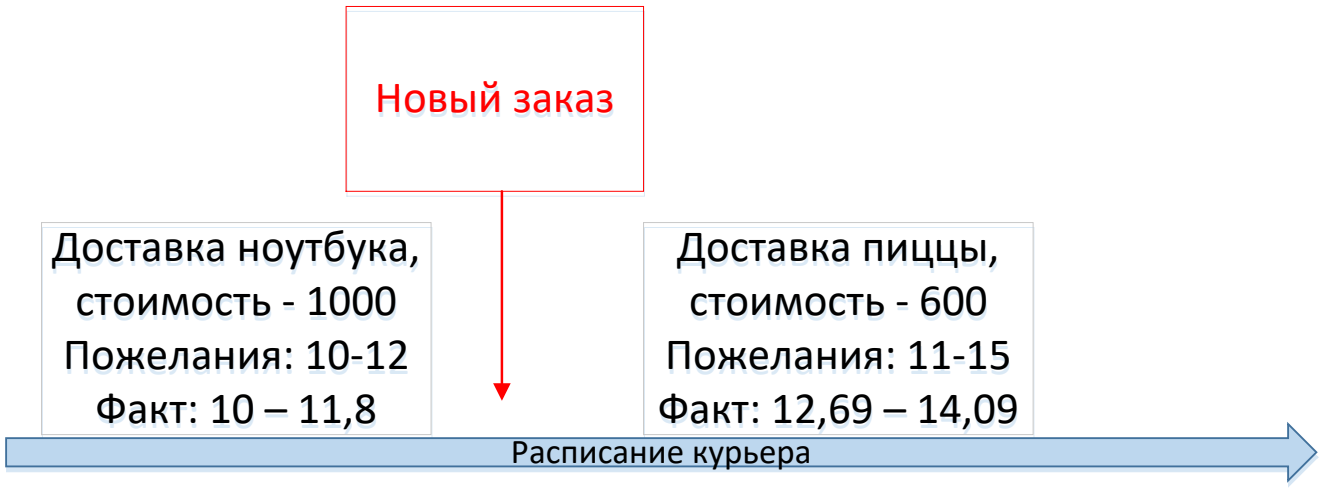

Рис. 5.2. Размещение в расписание нового заказа

Пусть также новый, добавляемый заказ, хочет разместиться так, что возникает пересечение с уже имеющимися записями.

Смоделируем такую ситуацию в исходных данных. Для этого создадим копию файла с исходными данными, назвав ее “Исходные данные_конфликт”.

На вкладке “Курьеры” оставим одного курьера - Лермонтова М. Ю., остальных удалим. Скорость этому курьеру изменим, указав значение 5. В файле main.py изменим чтение исходных данных, вписав туда новое название файла:

couriers = get_excel_data('Исходные данные_конфликт.xlsx', 'Курьеры')

 # Остальной код не меняем

orders = get_excel_data('Исходные данные_конфликт.xlsx', 'Заказы')

Установим веса критериев (в методе __evaluate_variants) и запустим планирование:

finish_weight = 0.3

start_weight = 0.2

price_weight = 0.5

Расписание будет выглядеть следующим образом:

Таблица 1.1.

Расписание курьера

resource_name	task_name	type	from	to	start_time	end_time	cost
Лермонтов М. Ю.	Доставка ноутбука	Ожидание	(3.0; 3.0)	(3.0; 3.0)	0	9,43432	0
Лермонтов М. Ю.	Доставка ноутбука	Движение за грузом	(3.0; 3.0)	(1.0; 1.0)	9,434315	10	0
Лермонтов М. Ю.	Доставка ноутбука	Движение с грузом	(1.0; 1.0)	(10.0; 1.0)	10	11,8	47,31371
Лермонтов М. Ю.	Доставка пиццы	Движение за грузом	(10.0; 1.0)	(6.0; 3.0)	11,8	12,6944	0
Лермонтов М. Ю.	Доставка пиццы	Движение с грузом	(6.0; 3.0)	(6.0; 10.0)	12,69443	14,0944	45,88854
Лермонтов М. Ю.	Доставка документов	Движение за грузом	(6.0; 10.0)	(5.0; 2.0)	14,09443	15,7069	0

Лермонтов М. Ю.	Доставка документов	Движение с грузом	(5.0; 2.0)	(4.0; 2.0)	15,70688	15,9069	36,24903

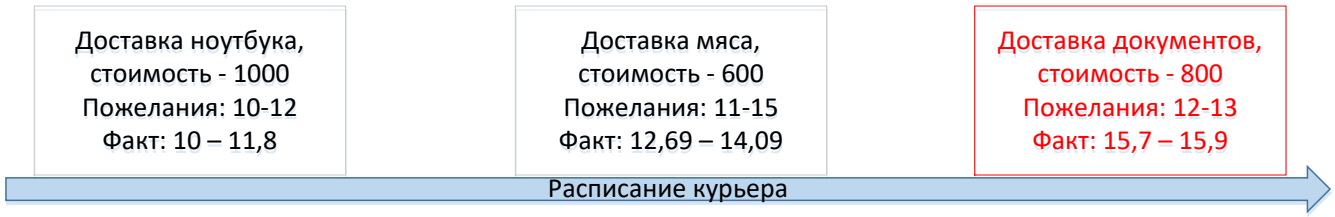

Рис. 5.3. Расписание курьера при появлении нового заказа

Процесс переговоров можно представить следующим образом:

@startuml

participant "Доставка ноутбука"

participant "Доставка пиццы"

participant "Доставка документов"

note over "Доставка ноутбука" : Формирование списка курьеров

actor "Лермонтов М. Ю." #red

"Доставка ноутбука"-> "Лермонтов М. Ю.": PRICE_REQUEST - Запрос параметров

"Лермонтов М. Ю."-> "Доставка ноутбука": PRICE_RESPONSE - Параметры: \n [\n time_from: 0, time_to: 2.36, price: 47.31, var_name: asap,\n time_from: 9.43, time_to: 11.8, price: 47.31, variant_name: jit \n]

note over "Доставка ноутбука" : Выбор наилучшего \nдля себя варианта - jit

"Доставка ноутбука"-> "Лермонтов М. Ю.": PLANNING_REQUEST- по варианту jit

"Лермонтов М. Ю."-> "Доставка ноутбука": PLANNING_RESPONSE - принято

note over "Доставка пиццы" : Формирование списка курьеров

"Доставка пиццы"-> "Лермонтов М. Ю.": PRICE_REQUEST - Запрос параметров

"Лермонтов М. Ю."-> "Доставка пиццы": PRICE_RESPONSE - Параметры: \n [\n time_from: 11.79, time_to: 14.09, prce: 45.88, var_name: asap\n]

note over "Доставка пиццы" : Выбор наилучшего для себя варианта - asap

"Доставка пиццы"-> "Лермонтов М. Ю.": PLANNING_REQUEST- по варианту asap

"Лермонтов М. Ю."-> "Доставка пиццы": PLANNING_RESPONSE - принято

note over "Доставка документов" : Формирование списка курьеров

"Доставка документов"-> "Лермонтов М. Ю.": PRICE_REQUEST - Запрос параметров

"Лермонтов М. Ю."-> "Доставка документов": PRICE_RESPONSE - Параметры: \n [\n time_from: 14.09, time_to: 15.90, prce: 36.24, var_name: asap\n]

note over "Доставка документов" : Выбор наилучшего \nдля себя варианта - asap

"Доставка документов"-> "Лермонтов М. Ю.": PLANNING_REQUEST- по варианту asap

"Лермонтов М. Ю."-> "Доставка документов": PLANNING_RESPONSE - принято

@enduml

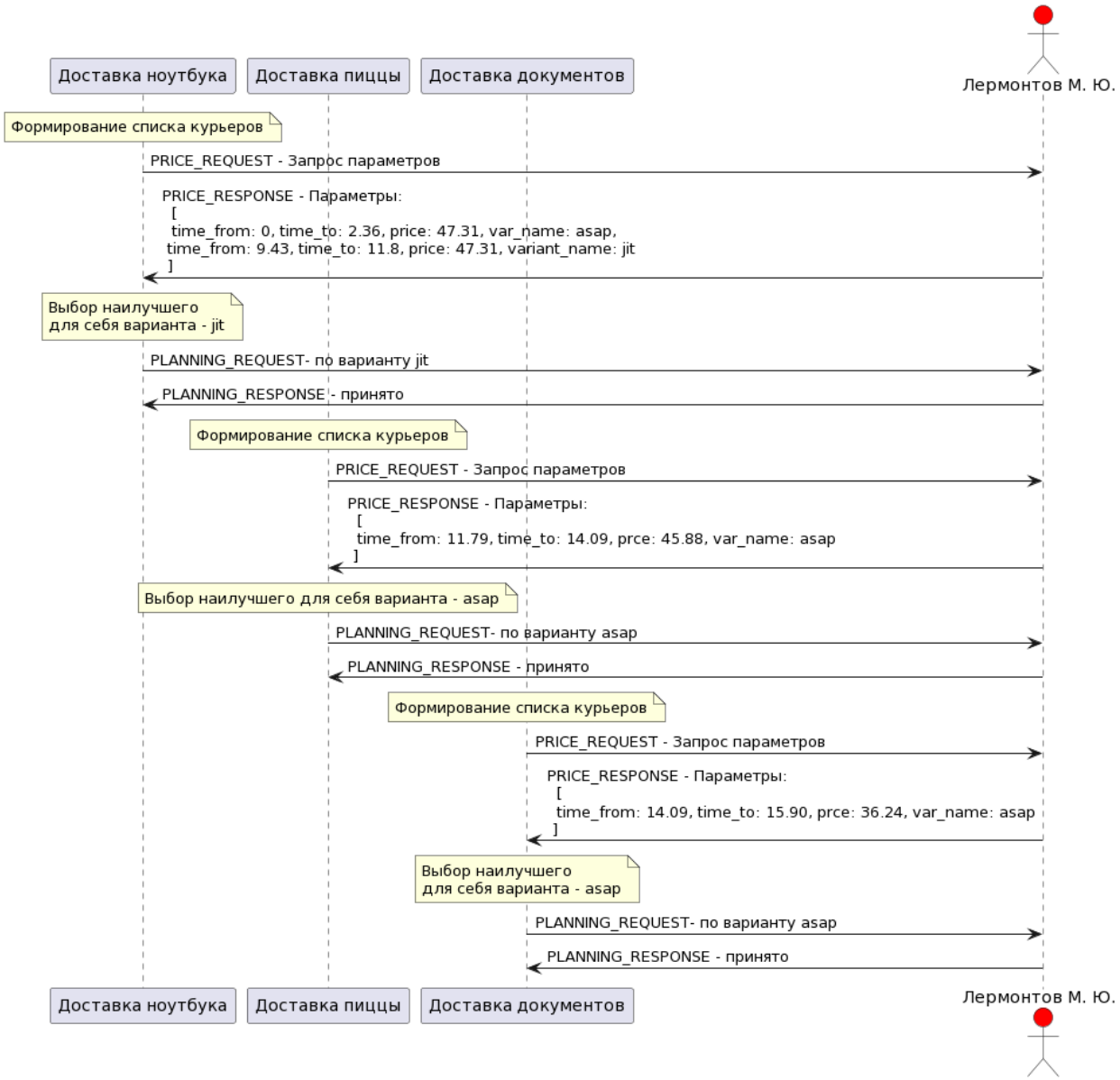

Рис. 5.4 . Диаграмма последовательности с выбором курьера

Можно заметить, что третий заказ разместился сильно позже желаемого. Альтернативное размещение могло бы быть таким:

Таблица 1.2

Альтернативное расписание курьера

resource_name	task_name	type	from	to	start_time	end_time	cost
Лермонтов М. Ю.	Доставка ноутбука	Ожидание	(3.0; 3.0)	(3.0; 3.0)	0	9,43432	0
Лермонтов М. Ю.	Доставка ноутбука	Движение за грузом	(3.0; 3.0)	(1.0; 1.0)	9,434315	10	0
Лермонтов М. Ю.	Доставка ноутбука	Движение с грузом	(1.0; 1.0)	(10.0; 1.0)	10	11,8	47,31371
Лермонтов М. Ю.	Доставка документов	Движение за грузом	(10.0; 1.0)	(5.0; 2.0)	11,8	12,8198	0
Лермонтов М. Ю.	Доставка документов	Движение с грузом	(5.0; 2.0)	(4.0; 2.0)	12,8198	13,0198	24,39608
Лермонтов М. Ю.	Доставка пиццы	Движение за грузом	(4.0; 2.0)	(6.0; 3.0)	13,0198	13,467	0
Лермонтов М. Ю.	Доставка пиццы	Движение с грузом	(6.0; 3.0)	(6.0; 10.0)	13,46702	14,867	36,94427

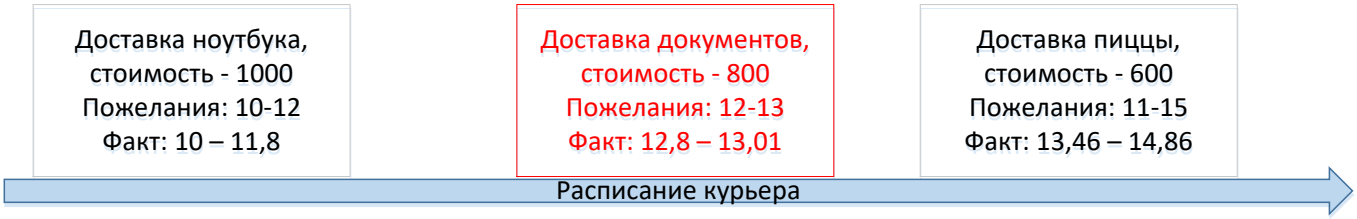

Рис. 5.5. Альтернативное расписание курьера

Такой план является более качественным: все заказы в нем разместились практически в диапазоне своих пожеланий, суммарная стоимость работы курьера в нем тоже оказалась ниже: в первой версии было 129.45, во второй - 108.65.

5.2. Поиск конфликтного варианта размещения

Такой план может быть получен, если агент заказа сможет решить возникающий конфликт за место в расписании. Для этого в момент получения запроса на опции планирования ему необходимо найти потенциальный конфликт и возможности его разрешения.

@startuml

participant "Доставка ноутбука"

participant "Доставка пиццы"

participant "Доставка документов"

actor "Лермонтов М. Ю." #red

hnote across: Предыдущие шаги планирования

note over "Доставка документов" : Формирование списка курьеров

"Доставка документов"-> "Лермонтов М. Ю.": PRICE_REQUEST - Запрос параметров

note over "Лермонтов М. Ю." : Поиск потенциальных конфликтов\n для расширения списка вариантов

@enduml

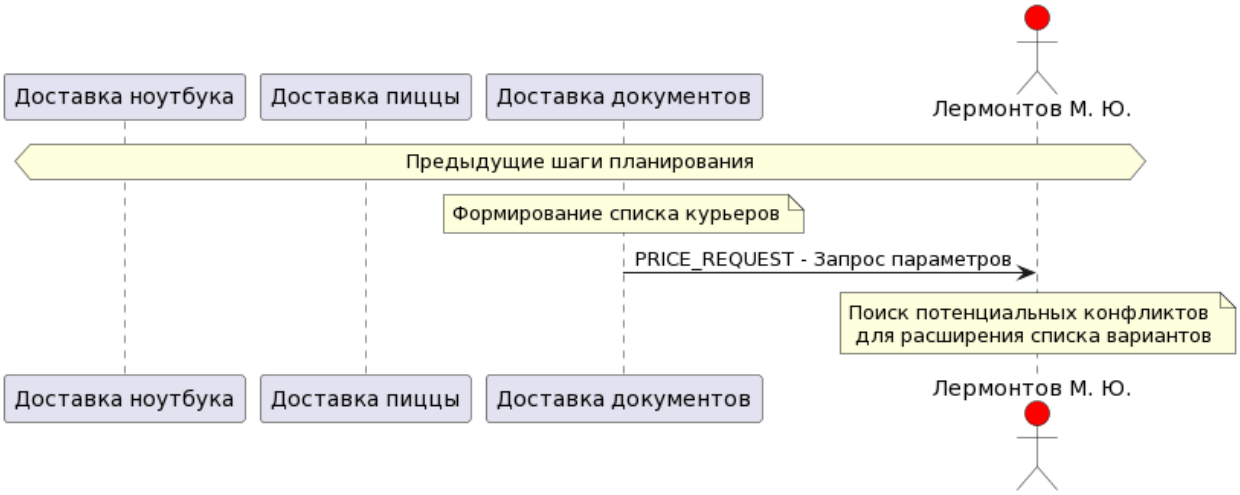

Рис. 5. 6. Диаграмма последовательности с поиском конфликтов

Добавим в код агента курьера - метод __get_params - поиск конфликтующих записей:

jit_time_from = order.time_from - time_to_order

jit_time_to = jit_time_from + time_to_order + time_with_order

if jit_time_from > 0:

conflicted_records = self.entity.get_conflicts(jit_time_from, jit_time_to)

logging.info(f'{self} - {conflicted_records=}')

Тогда при запуске программы можно увидеть следующий вывод:

Агент курьера Лермонтов М. Ю. - заказ Заказ Доставка документов надо пронести 1.0, к нему идти 8.06225774829855это займет 1.8124515496597098 и будет стоить 36.2490309931942 [courier_agent.py:59] Агент курьера Лермонтов М. Ю. - conflicted_records= [ScheduleItem(order=Заказ Доставка ноутбука, rec_type='Движение с грузом', start_time=10.0, end_time=11.799999999999999, point_from=(1.0; 1.0), point_to=(10.0; 1.0), cost=47.31370849898476, all_params={'courier': Курьер

Лермонтов М. Ю., 'time_from': 9.434314575050761, 'time_to': 11.8, 'price': 47.31370849898476, 'order': Заказ Доставка ноутбука, 'variant_name': 'jit', 'start_efficiency': 0, 'finish_efficiency': 1, 'price_efficiency': 1, 'total_efficiency': 0.7, 'success': True}), ScheduleItem(order=Заказ Доставка пиццы, rec_type='Движение за грузом', start_time=11.799999999999999, end_time=12.694427190999916, point_from=(10.0; 1.0), point_to=(6.0; 3.0), cost=0, all_params={'courier': Курьер Лермонтов М. Ю., 'time_from': 11.799999999999999, 'time_to': 14.094427190999916, 'price': 45.88854381999832, 'order': Заказ Доставка пиццы, 'variant_name': 'asap', 'start_efficiency': 1, 'finish_efficiency': 1, 'price_efficiency': 1, 'total_efficiency': 1.0, 'success': True})]

Объединим информацию о заказах из расписания - так, чтобы видеть сводную информацию. Для этого реализуем новый метод в сущности курьера:

```
def get_all_order_records(self, order: OrderEntity) -> typing.List[ScheduleItem]:
    """
    Возвращает все записи указанного заказа
    :param order:
    :return:
    """
    result: typing.List[ScheduleItem] = [rec for rec in self.schedule if rec.order == order]
    return result
```

И вызовем этот метод в агенте курьера в методе __get_params:

```
    # Формируем перечень заказов, которые есть в конфликтных записях
    conflicted_orders = set([rec.order for rec in conflicted_records])

    all_conflicted_records: typing.List[ScheduleItem] = []
    for _order in conflicted_orders:

all_conflicted_records.extend(self.entity.get_all_order_records(_order))
    conflicted_tasks = defaultdict(dict)
    for rec in all_conflicted_records:
        task = rec.order
        min_start = rec.start_time
        max_end = rec.end_time
        max_cost = rec.cost
        if task in conflicted_tasks:
            min_start = min(conflicted_tasks[task].get('start'), min_start)
            max_end = max(conflicted_tasks[task].get('end'), max_end)
            max_cost = conflicted_tasks[task].get('cost', 0) + max_cost

        conflicted_tasks[task] = {'start': min_start, 'end': max_end, 'cost': max_cost}
    logging.info(f'{self} ищет конфликтные варианты для заказа {order} {order.time_from=}, {order.time_to=} - '
                 f'{jit_time_from=}, {conflicted_tasks=}')
```

Вывод программы в этом случае будет примерно таким:

Агент курьера Лермонтов М. Ю. ищет конфликтные варианты для заказа Заказ Доставка документов order.time_from=12.0, order.time_to=13.0 - jit_time_from=10.38754845034029, conflicted_tasks=defaultdict(<class 'dict'>, { Заказ Доставка ноутбука: {'start': 9.434314575050761, 'end': 11.799999999999999, 'cost': 47.31370849898476}, Заказ Доставка пиццы: {'start': 11.799999999999999, 'end': 14.094427190999916, 'cost': 45.88854381999832} })

Напомним параметры заказа “Доставка документов”. Желаемый интервал доставки - [12-13], стоимость - 800.

Очевидно, что для заказа “Доставка документов” нет смысла вытеснять из расписания заказ “Доставка ноутбука” - интервалы их желаемого и фактического выполнения слабо пересекаются, и уже запланированный заказ дороже, т.е. более выгоден курьеру. А вот доставка заказа “Доставка пиццы” может быть смещена в расписании вправо, т.е. на более поздний срок: этот заказ дешевле и интервалы его выполнения значительно сильнее мешают добавляемому заказу.

5.3. Реализация вытеснения заказов по цене

Реализуем в коде логику вытеснения более дешевого заказа - для простоты, только одного. Для этого нам надо вычислить интервал, когда курьер сможет отправиться за грузом, сколько он будет идти и когда он его доставит. Добавим код в метод __get_params агента курьера:

 # В коде ниже смотрим только на цену, но можно оценивать и другие параметры.

```
        poss_removing_orders: typing.List[OrderEntity] = [_order for _order
in conflicted_orders
                                                    if _order.price < order.price]

        if not poss_removing_orders:
            logging.info(f'{self} не смог найти более дешевые заказы по
сравнению с {order}')
            return params

        # Для простоты оцениваем только один
        cheapest_order = min(poss_removing_orders, key=lambda x: x.price)
        logging.info(f'{self} нашел более дешевый заказ: {cheapest_order}')

        cheapest_order_records: typing.List[ScheduleItem] =
self.entity.get_all_order_records(cheapest_order)
        # Ожидаем, что первая запись - это движение за грузом.
        start_record = cheapest_order_records[0]
        start_time = start_record.start_time
        start_point = start_record.point_from
        conflicted_distance_to_order = start_point.get_distance_to_other(p1)
        conflicted_time_to_order = conflicted_distance_to_order /
self.entity.velocity
        conflicted_finish = start_time + conflicted_time_to_order +
time_with_order
```

```
logging.info(f'{self} в случае вытеснения отправится из точки {start_point} в {start_time}, '
             f'за заказом придет в {start_time + conflicted_time_to_order} '
             f'и доставит в {conflicted_finish}')

new_conflicts = self.entity.get_conflicts(start_time, conflicted_finish)
other_order_conflicts = [_rec for _rec in new_conflicts if _rec.order != cheapest_order]
if other_order_conflicts:
    logging.info(f'{self} не сможет доставить заказ, есть конфликты {other_order_conflicts}')
    return params
conflicted_price = (conflicted_time_to_order + time_with_order) * self.entity.rate
conflict_variant = {'courier': self.entity, 'time_from': start_time, 'time_to': conflicted_finish,
                    'price': conflicted_price, 'order': order, 'variant_name': 'conflict'}
params.append(conflict_variant)
```

Теперь агент заказа получает еще один вариант размещения, который он может выбрать. В случае, если лучшим будет выбран именно он, то агент курьера получит сообщение с этим вариантом и должен будет удалить конфликтующий заказ. В первой версии решения конфликтов такое удаление может происходить безусловно,

дальше же можно реализовать логику по запросу на удаление. Вынесем логику добавления заказа в отдельный метод агента курьера:

```
    def add_order(self, params: dict) -> bool:
        """
        Добавление заказа с параметрами в расписание ресурса.
        :param params:
        :return:
        """
        variant_name = params.get('variant_name')
        if variant_name == 'conflict':
            conflicted_records: typing.List[ScheduleItem] = self.entity.get_conflicts(params.get('time_from'),

params.get('time_to'))
            logging.info(f'{self} - {conflicted_records=}')
            # Формируем перечень заказов, которые есть в конфликтных записях
            conflicted_orders = set([rec.order for rec in conflicted_records])
            if len(conflicted_orders) > 1:
                logging.info(f'{self} получил запрос на размещение {params}, '
                             f'много конфликтов - {conflicted_orders}')
                return False
```

```
            # Делаем копию расписания. Если заказ разместить не удастся, то восстановим его.
            old_schedule = copy.copy(self.entity.schedule)
            for rec in conflicted_records:
                self.entity.schedule.remove(rec)
            adding_result = self.entity.add_order_to_schedule(params.get('order'),
                                                              params.get('time_from'),
                                                              params.get('time_to'),
                                                              params.get('price'),
                                                              params)
            if not adding_result:
                self.entity.schedule = old_schedule
            else:
                # Удаление заказа произошло успешно, надо сообщить удаленному заказу
                # о необходимости перепланироваться
                remove_message = Message(MessageType.REMOVE_ORDER, self.entity)
                removed_order_address = self.dispatcher.reference_book.get_address(conflicted_orders.pop())
                self.send(removed_order_address, remove_message)
            return adding_result

        adding_result = self.entity.add_order_to_schedule(params.get('order'),
```

```
                                        params.get('time_from'),
                                        params.get('time_to'),
                                        params.get('price'),
                                        params)
        return adding_result
```

Этот метод должен быть вызван в методе агента курьера, который примет следующий вид:

```
    def handle_planning_request(self, message, sender):
        """
        Обработка сообщения с запросом на планирования.
        :param message:
        :param sender:
        :return:
        """
        params = message.msg_body
        # Пытаемся добавить заказ в свое расписание
        adding_result = self.add_order(params)

        logging.info(f'{self} получил запрос на размещение {params}, '
                     f'результат - {adding_result}')
        params['success'] = adding_result
        result_msg = Message(MessageType.PLANNING_RESPONSE,
params)
        self.send(sender, result_msg)
```

Агент заказа должен научиться обрабатывать запрос на удаление, при этом логично пересчитать варианты и запустить планирование.

```
def handle_remove_message(self, message, sender):
    """
    Обработка сообщения об удалении с курьера
    :param message:
    :param sender:
    :return:
    """
    courier = message.msg_body
    logging.info(f'{self} - удален из расписания курьера {courier}')
    # Считаем, что всем параметры старые их необходимо пересчитать.
    self.possible_variants.clear()
    self.__send_params_request()
```

Мы реализуем логику по обработке нового типа сообщений. Необходимо объявить тип сообщения и подписаться на него.

В результате работы новой версии программы расписание будет сформировано более эффективно, а в логах можно увидеть следующие строки:

Агент заказа Доставка документов - лучшим вариантом признан { 'courier': Курьер Лермонтов М. Ю., 'time_from': 11.799999999999999, 'time_to': 13.019803902718555, 'price': 24.396078054371134, 'order': Заказ Доставка документов, 'variant_name': 'conflict', 'start_efficiency':

1, 'finish_efficiency': 0, 'price_efficiency': 1, 'total_efficiency': 0.7}, адрес - ActorAddr-/A~a

Агент курьера Лермонтов М. Ю. - conflicted_records=[ScheduleItem(order=Заказ Доставка пиццы, rec_type='Движение за грузом', start_time=11.799999999999999, end_time=12.694427190999916, point_from=(10.0; 1.0), point_to=(6.0; 3.0), cost=0, all_params={'courier': Курьер Лермонтов М. Ю., 'time_from': 11.799999999999999, 'time_to': 14.094427190999916, 'price': 45.88854381999832, 'order': Заказ Доставка пиццы, 'variant_name': 'asap', 'start_efficiency': 1, 'finish_efficiency': 1, 'price_efficiency': 1, 'total_efficiency': 1.0, 'success': True}), ScheduleItem(order=Заказ Доставка пиццы, rec_type='Движение с грузом', start_time=12.694427190999916, end_time=14.094427190999916, point_from=(6.0; 3.0), point_to=(6.0; 10.0), cost=45.88854381999832, all_params={'courier': Курьер Лермонтов М. Ю., 'time_from': 11.799999999999999, 'time_to': 14.094427190999916, 'price': 45.88854381999832, 'order': Заказ Доставка пиццы, 'variant_name': 'asap', 'start_efficiency': 1, 'finish_efficiency': 1, 'price_efficiency': 1, 'total_efficiency': 1.0, 'success': True})]

Агент курьера Лермонтов М. Ю. получил запрос на размещение {'courier': Курьер Лермонтов М. Ю., 'time_from': 11.799999999999999, 'time_to': 13.019803902718555, 'price': 24.396078054371134, 'order': Заказ Доставка документов, 'variant_name': 'conflict', 'start_efficiency': 1, 'finish_efficiency': 0, 'price_efficiency': 1, 'total_efficiency': 0.7}, результат - True

Агент заказа Доставка пиццы - удален из расписания курьера Курьер Лермонтов М. Ю.

Агент заказа Доставка пиццы - список ресурсов: [Курьер Лермонтов М. Ю.]

Агент заказа Доставка пиццы - адрес Курьер Лермонтов М. Ю.: ActorAddr-/A~a Агент заказа Доставка документов - получил Message(msg_type=<MessageType.PLANNING_RESPONSE: 'Ответ на размещение'>, msg_body={'courier': Курьер Лермонтов М. Ю., 'time_from': 11.799999999999999, 'time_to': 13.019803902718555, 'price': 24.396078054371134, 'order': Заказ Доставка документов, 'variant_name': 'conflict', 'start_efficiency': 1, 'finish_efficiency': 0, 'price_efficiency': 1, 'total_efficiency': 0.7, 'success': True}), результат - {'courier': Курьер Лермонтов М. Ю., 'time_from': 11.799999999999999, 'time_to': 13.019803902718555, 'price': 24.396078054371134, 'order': Заказ Доставка документов, 'variant_name': 'conflict', 'start_efficiency': 1, 'finish_efficiency': 0, 'price_efficiency': 1, 'total_efficiency': 0.7, 'success': True}

Агент заказа Доставка документов доволен, ничего делать не надо

Агент курьера Лермонтов М. Ю. - заказ Заказ Доставка пиццы надо пронести 7.0, к нему идти 2.23606797749979, это займет 1.8472135954999578 и будет стоить 36.94427190999916

Агент курьера Лермонтов М. Ю. - conflicted_records=[ScheduleItem(order=Заказ Доставка ноутбука, rec_type='Движение с грузом', start_time=10.0, end_time=11.799999999999999, point_from=(1.0; 1.0), point_to=(10.0; 1.0), cost=47.31370849898476,

all_params={'courier': Курьер Лермонтов М. Ю., 'time_from': 9.434314575050761, 'time_to': 11.8, 'price': 47.31370849898476, 'order': Заказ Доставка ноутбука, 'variant_name': 'jit', 'start_efficiency': 0, 'finish_efficiency': 1, 'price_efficiency': 1, 'total_efficiency': 0.8, 'success': True}), ScheduleItem(order=Заказ Доставка документов, rec_type='Движение за грузом', start_time=11.799999999999999, end_time=12.819803902718556, point_from=(10.0; 1.0), point_to=(5.0; 2.0), cost=0, all_params={'courier': Курьер Лермонтов М. Ю., 'time_from': 11.799999999999999, 'time_to': 13.019803902718555, 'price': 24.396078054371134, 'order': Заказ Доставка документов, 'variant_name': 'conflict', 'start_efficiency': 1, 'finish_efficiency': 0, 'price_efficiency': 1, 'total_efficiency': 0.7, 'success': True})]

Агент курьера Лермонтов М. Ю. ищет конфликтные варианты для заказа Заказ Доставка пиццы order.time_from=11.0, order.time_to=15.0 - jit_time_from=10.552786404500042, jit_time_to=12.4, conflicted_tasks=defaultdict(<class 'dict'>, { Заказ Доставка документов: {'start': 11.799999999999999, 'end': 13.019803902718555, 'cost': 24.396078054371134}, Заказ Доставка ноутбука: {'start': 9.434314575050761, 'end': 11.799999999999999, 'cost': 47.31370849898476}})

Агент курьера Лермонтов М. Ю. не смог найти более дешевые заказы по сравнению с Заказ Доставка пиццы

Агент заказа Доставка пиццы - получил сообщение Message(msg_type=<MessageType.PRICE_RESPONSE: 'Ответ цены'>, msg_body=[{'courier': Курьер Лермонтов М. Ю., 'time_from': 13.019803902718555, 'time_to': 14.867017498218514, 'price':

36.94427190999916, 'order': Заказ Доставка пиццы, 'variant_name': 'asap'}])

Агент заказа Доставка пиццы минимальный старт: 13.019803902718555, минимальное завершение - 14.867017498218514, минимальная цена - 36.94427190999916

Агент заказа Доставка пиццы - sorted_vars=[{'courier': Курьер Лермонтов М. Ю., 'time_from': 13.019803902718555, 'time_to': 14.867017498218514, 'price': 36.94427190999916, 'order': Заказ Доставка пиццы, 'variant_name': 'asap', 'start_efficiency': 1, 'finish_efficiency': 1, 'price_efficiency': 1, 'total_efficiency': 1.0}] Агент заказа Доставка пиццы - лучшим вариантом признан {'courier': Курьер Лермонтов М. Ю., 'time_from': 13.019803902718555, 'time_to': 14.867017498218514, 'price': 36.94427190999916, 'order': Заказ Доставка пиццы, 'variant_name': 'asap', 'start_efficiency': 1, 'finish_efficiency': 1, 'price_efficiency': 1, 'total_efficiency': 1.0}, адрес - ActorAddr-/A~a

Агент курьера Лермонтов М. Ю. получил запрос на размещение {'courier': Курьер Лермонтов М. Ю., 'time_from': 13.019803902718555, 'time_to': 14.867017498218514, 'price': 36.94427190999916, 'order': Заказ Доставка пиццы, 'variant_name': 'asap', 'start_efficiency': 1, 'finish_efficiency': 1, 'price_efficiency': 1, 'total_efficiency': 1.0}, результат - True

Агент заказа Доставка пиццы - получил Message(msg_type=<MessageType.PLANNING_RESPONSE: 'Ответ на размещение'>, msg_body={'courier': Курьер Лермонтов М. Ю., 'time_from': 13.019803902718555, 'time_to': 14.867017498218514,

'price': 36.94427190999916, 'order': Заказ Доставка пиццы, 'variant_name': 'asap', 'start_efficiency': 1, 'finish_efficiency': 1, 'price_efficiency': 1, 'total_efficiency': 1.0, 'success': True}), результат - {'courier': Курьер Лермонтов М. Ю., 'time_from': 13.019803902718555, 'time_to': 14.867017498218514, 'price': 36.94427190999916, 'order': Заказ Доставка пиццы, 'variant_name': 'asap', 'start_efficiency': 1, 'finish_efficiency': 1, 'price_efficiency': 1, 'total_efficiency': 1.0, 'success': True} Агент заказа Доставка пиццы доволен, ничего делать не надо

5.4. Заключение

В данном уроке реализована простейшая версия решения конфликтов, при которой агент курьера выбирает наиболее выгодный для себя заказ.

Дальнейшее развитие этого метода может включать в себя рекурсивные переговоры: когда удаляемый заказ сначала ищет для себя размещение, а не просто удаляется из плана.

Полный исходный код урока находится в приложении к уроку.

5.5. Самостоятельная работа

В этом уроке мы решаем конфликт, только если “мешает” один заказ. Необходимо расширить метод так, чтобы можно было решать конфликты с несколькими заказами.

Вопросы для самоконтроля:

- Какого рода конфликты между заказами и ресурсами могут возникать в расписании?
- Каким образом возможно разрешать конфликты между заказами и ресурсами?

- Почему так важно как можно быстрее построить первое расписание, даже не решая сразу конликтов?
- Каковы условия останова процедуры выявления и разрешения конфликтов?
- Какие преимущества можно выявить у такого класса распределенных алгоритмов?
- Какие есть ограничения у таких алгоритмов и как их можно преодолевать?

6. Адаптивное планирование

В уроках выше было реализовано т.н. “пакетное” планирование - план формируется один раз, все дальнейшие изменения в окружающем мире можно учесть только с помощью перезапуска программы.

В реально работающих системах поток событий, влияющих на план, слишком большой.

Для решения задачи логистики можно рассмотреть такие типы событий:

- появление нового заказа,
- появление нового курьера,
- удаление заказа, удаление курьера.

Реализуем обработку событий. Для начала расширим интерфейс класса диспетчера - AgentsDispatcher, добавив туда возможность удалять сущности.

```
def remove_entity(self, entity_type: str, entity_name: str) -> bool:
    """
    Удаляет сущность по типу и имени
    :param entity_type:
    :param entity_name:
    :return:
    """
    entities: typing.List[BaseEntity] = self.scene.get_entities_by_type(entity_type)
    for entity in entities:
        if entity.name == entity_name:
```

```
            agent_address = self.reference_book.get_address(entity)
            if not agent_address:
                logging.error(f'Агент сущности {entity} не найден')
                return False
            self.actor_system.tell(agent_address, ActorExitRequest())
            self.scene.entities[entity_type].remove(entity)
            return True
        return False
```

Переделаем запуск программы, реализовав там бесконечный цикл и чтение команд из консоли. Создание новых заказов и курьеров реализуем в отдельном методе:

```
def add_entity(scene: Scene):
    type_input = input("Выберите тип. C - курьер, O - заказ: ")
    required_params = []
    entity_class = None
    if type_input.upper() in ["C", "С"]:
        required_params = [
            'Табельный номер', 'ФИО', 'Координата начального положения x',
            'Координата начального положения y', 'Типы доставляемых заказов',
            'Стоимость выхода на работу', 'Цена работы за единицу времени', 'Скорость',
            'Объем ранца', 'Грузоподъемность',
        ]
        entity_class = CourierEntity
```

```
        if type_input.upper() in ["О", "O"]:
            required_params = [
                'Номер', 'Наименование', 'Масса', 'Объем', 'Стоимость', 'Координата получения x', 'Координата получения y',
                'Координата доставки x', 'Координата доставки y', 'Время получения заказа', 'Время доставки заказа',
                'Тип заказа',
            ]
            entity_class = OrderEntity
        if required_params and entity_class:
            result = {}
            for param in required_params:
                param_input = input(f'Введите параметр {param}: ')
                result[param] = param_input
            if result:
                onto_description = {}
                # TODO: Тут не хватает проверки на дублирование идентификаторов.
                entity = entity_class(onto_description, result, scene)
                return entity
            return result
        return None
```

Тогда запуск программы можно модифицировать, добавив туда следующие строки:

```
    while True:
        logging.info("A - добавить агента")
```

```
            logging.info("D - удалить агента")
            logging.info("L - посмотреть список агентов")
            logging.info("Q - Выход")
            user_input = input(": ")
            was_events_added = False
            if user_input.upper() == "Q":
                break
            if user_input.upper() == "A":
                entity = add_entity(scene)
                if entity:
                    dispatcher.add_entity(entity)
                    was_events_added = True
            elif user_input.upper() == "D":
                logging.info("Запускаем удаление агента")
                type_input = input("Выберите тип. C - курьер, O - заказ: ")
                name_input = input("Введите имя сущности: ")
                entity_type = ''
                if type_input.upper() in ["C", "С"]:
                    entity_type = 'COURIER'
                elif type_input.upper() in ["O", "О"]:
                    entity_type = 'ORDER'

                remove_result = dispatcher.remove_entity(entity_type,
name_input)
                logging.info(f'Результат удаления агента: {remove_result}')
                was_events_added = remove_result
```

```
        elif user_input.upper() == "L":
            agents_addresses = dispatcher.get_agents_id()
            logging.info(agents_addresses)

        if was_events_added:
            new_schedule_records = []
            for courier in scene.get_entities_by_type('COURIER'):
                new_schedule_records.extend(courier.get_schedule_json())
            save_schedule_to_excel(new_schedule_records,
'Результаты.xlsx')
```

Теперь у нас появилась возможность добавления новых событий. Однако обработка этих событий пока что отсутствует - необходимо ее реализовать.

6.1. Обработка появления нового заказа

Проверим создание нового заказа. Для этого запустим программу, после чего в консоли выберем “Добавить агента”. Заполним параметры следующим образом:

Выберите тип. C - курьер, O - заказ: o

Введите параметр Номер: 6

Введите параметр Наименование: Новый заказ

Введите параметр Масса: 12

Введите параметр Объем: 12

Введите параметр Стоимость: 1200

Введите параметр Координата получения x: 0

Введите параметр Координата получения y: 0

Введите параметр Координата доставки x: 10

Введите параметр Координата доставки y: 10

Введите параметр Время получения заказа: 16

Введите параметр Время доставки заказа: 18

Введите параметр Тип заказа: Обычный

Новый заказ, созданный таким образом, запланируется без внесения изменений в код. Это происходит благодаря тому, что после создания заказа его агент сразу формирует список курьеров и пытается выполнить планирование.

Создадим еще один заказ - с новым типом доставки.

Выберите тип. C - курьер, O - заказ: o

Введите параметр Номер: 8

Введите параметр Наименование: Новый заказ 2

Введите параметр Масса: 10

Введите параметр Объем: 12

Введите параметр Стоимость: 900

Введите параметр Координата получения x: 0

Введите параметр Координата получения y: 0

Введите параметр Координата доставки x: 12

Введите параметр Координата доставки y: 12

Введите параметр Время получения заказа: 10

Введите параметр Время доставки заказа: 19

Введите параметр Тип заказа: Срочный

Такой заказ будет не запланирован, т.к. в системе нет курьеров, способных доставлять заказы с типом “Срочный”.

6.2. Обработка появления нового курьера

Добавим нового курьера с типом “Срочный”.

Выберите тип. C - курьер, O - заказ: c

Введите параметр Табельный номер: 10

Введите параметр ФИО: Ivanov P.I.

Введите параметр Координата начального положения x: 0

Введите параметр Координата начального положения y: 0

Введите параметр Типы доставляемых заказов: Срочный

Введите параметр Стоимость выхода на работу: 10

Введите параметр Цена работы за единицу времени: 12

Введите параметр Скорость: 120

Введите параметр Объем ранца: 20

Введите параметр Грузоподъемность: 30

2023-12-10 18:36:00,391 INFO => Агент курьера Ivanov P.I. проинициализирован [agent_base.py:67]

2023-12-10 18:36:00,424 INFO => A - добавить агента [main.py:70]

2023-12-10 18:36:00,424 INFO => D - удалить агента [main.py:71]

2023-12-10 18:36:00,424 INFO => L - посмотреть список агентов [main.py:72]

2023-12-10 18:36:00,424 INFO => Q - Выход [main.py:73]

Как видно из окна вывода, агент курьера проинициализировался, но заказ, который подходит ему по типу, остался незапланированным. Проверим, запланируется ли новый заказ этого же типа, создав его.

Выберите тип. C - курьер, O - заказ: o

Введите параметр Номер: 13

Введите параметр Наименование: Новый заказ 3

Введите параметр Масса: 2

Введите параметр Объем: 2

Введите параметр Стоимость: 1000

Введите параметр Координата получения x: 0

Введите параметр Координата получения y: 0

Введите параметр Координата доставки x: 12

Введите параметр Координата доставки y: 12

Введите параметр Время получения заказа: 10

Введите параметр Время доставки заказа: 16

Введите параметр Тип заказа: Срочный

Этот заказ окажется запланированным. Значит, курьер “работает” и принимать заказы в свой план может, но первый заказ с типом “Срочный” так и остался неразмещенным. Причиной такого поведения является то, что курьер при создании никак не оповещает потенциальные заказы о своем появлении. Это можно реализовать, введя новый тип сообщений (*NEW_COURIER = 'Появление нового курьера'*), отправляя его и реализуя логику по его обработке.

Отправка сообщений реализуется в методе агента курьера и выглядит так:

```
def handle_init_message(self, message, sender):
    super().handle_init_message(message, sender)
    all_orders = self.scene.get_entities_by_type('ORDER')
```

matched_orders = [order for order in all_orders if order.order_type in self.entity.types]

for order in matched_orders:

order_address = self.dispatcher.reference_book.get_address(order)

new_courier_message = Message(MessageType.NEW_COURIER, self.entity)

self.send(order_address, new_courier_message)

Агент заказа должен подписаться на этот тип сообщений и корректно его обрабатывать.

```
def handle_new_resource_message(self, message, sender):
    """
    Обработка сообщения о появлении нового курьера
    :param message:
    :param sender:
    :return:
    """
    courier = message.msg_body
    logging.info(f'{self} - узнал о новом курьере {courier}')
    # Если заказ уже запланирован, то ничего делать не надо.
    if not self.entity.delivery_data.get('courier'):
        # Считаем, что всем параметры старые их необходимо пересчитать.
        self.possible_variants.clear()
        self.__send_params_request()
```

Можно убедиться, что теперь заказы с новым типом будут планироваться независимо от того, добавлены ли они до появления нужных курьеров или после.

6.3. Обработка удаления курьера

Запустим программу и удалим курьера, на которого запланированы все заказы:

INFO => Запускаем удаление агента [main.py:84]

Выберите тип. C - курьер, O - заказ: c

Введите имя сущности: Лермонтов М. Ю.

INFO => Агент курьера Лермонтов М. Ю. получил сообщение - ActorExitRequest [agent_base.py:36]

INFO => Результат удаления агента: True [main.py:94]

Можно видеть ситуацию, аналогичную предыдущей - событие в систему попало, но изменений в плане нет. Причина та же: не хватает обработки сообщения об удалении. Введем новый тип сообщения (*DELETED_COURIER = 'Удаление курьера'*), напишем его отправку и обработку.

Отправлять сообщение будет агент курьера в переопределенном методе базового класса:

```
def handle_delete_message(self):
    super().handle_delete_message()
    all_orders = self.scene.get_entities_by_type('ORDER')
    for order in all_orders:
        order_address = self.dispatcher.reference_book.get_address(order)
```

deleted_courier_message = Message(MessageType.DELETED_COURIER, self.entity)

self.send(order_address, deleted_courier_message)

Агент заказа должен посмотреть, влияет ли это сообщение на него и, если влияет, то попытаться запланироваться на других курьерах.

```
def handle_remove_message(self, message, sender):
    """
    Обработка сообщения об удалении с курьера
    :param message:
    :param sender:
    :return:
    """
    courier = message.msg_body
    logging.info(f'{self} - удален из расписания курьера {courier}')
    # Считаем, что всем параметры старые их необходимо пересчитать.
    self.entity.delivery_data = {
        'courier': None,
        'price': None,
        'time_from': None,
        'time_to': None,
    }
    self.possible_variants.clear()
    self.__send_params_request()
```

й

💡

Для предотвращения попыток на курьерах, которые находятся в процессе удаления, в базовую сущность был добавлен флаг is_deleting. А метод по получению сущностей из сцены был модифицирован таким образом, чтобы этот флаг учитывать.

6.4. Заключение

В этом уроке была реализована первая модель реакции системы на события из реального мира. Список событий может быть расширен: например, могут измениться параметры заказов или ресурсов, веса критериев, и т. д. Предложенный механизм может быть расширен и на такие типы событий.

Полный исходный код урока находится в приложении к уроку.

6.5. Самостоятельная работа

В качестве самостоятельной работы необходимо реализовать событие удаления заказа.

Важно обратить внимание, что если у курьера были записи в расписании после удаленного заказа, то их, возможно, придется модифицировать.

Вопросы для самоконтроля:

- Приведите примеры основных классов событий;
- Определите термин «адаптивность»;
- Что «адаптивность» означает для управления ресурсами?
- Какие модели и методы взаимодействий агентов обеспечивают адаптивность при планировании?

- Как адаптивность системы планирования способствует повышению эффективности использования ресурсов?

Заключение

В настоящем пособии и лабораторном практикуме даются начальные знания о том, как создавать мультиагентные системы и лежащие в их основе распределнные алгоритмы принятия и координации решений, базирующиеся на протоколах переговоров и взаимных уступок агентов в ходе адаптивного перепланирования при появлении различных непредвиденных событий в реальном времени.

Указанные распределенные интеллектуальные системы коллективного согласованного принятия решений, с переходом к Индустрии 5.0 и Обществу 5.0, будут составлять основу построений и масштабирования любой крупной промышленной системы управления ресурсами.

Освоение указанных приницпов студентами уже на первых курсах ВУЗа будет способствовать их скорейшей интеграции как в практические бизнес-проекты, так и научные проекты передовых исследований и разработок в области коллективного искусственного интеллекта.