1. ООП. Понятие класс, интерфейс, объект.

Объектно-ориентированное программирование (ООП) — это парадигма программирования, которая рассматривает программу, как множество объектов, которые взаимодействуют между собой. ООП «держится на трех китах» если можно это так выразить: наследование, инкапсуляция, и полиморфизм.

Наследование — концепция, согласно которой класс может частично или полностью повторить свойства и методы родителя (класса, от которого он наследуется).

Инкапсуляция — ограничение доступа до некоего кода и предоставление методов для использования этого кода.(геттеры сеттеры)

Полиморфизм — способность метода вести себя по разному при разных наборах параметров. Допустим, то что объекты состоят из атомов и есть то, что наследуют все объекты. Но все дополнения и правки - полиморфизм. Так, из атомов мы слепили колёса и приделали на доску - это скейт.

Класс — это и есть отображение объекта. Объект — это отображение некой сущности. В классе мы описали сам объект, его свойства и методы.

Абстрактный класс - есть ёмкость, что невозможно создать, для которой нет полного чертежа. А все чертежи, что дополнили до полного - есть наследованные классы от класса ёмкость. Абстрагируемся от низкоуровневой задачи "формирования ёмкостей посредством перемещения молекул" и приходим к "конструированию ёмкости посредством совмещения деталей, элементов". Это и есть абстракция.

Интерфейс — это конструкция похожая на класс. Только класс — это представление объекта с набором параметров и методов, а интерфейс — только с набором методов. Причем интерфейс задает поведение класса.

2. Программный агент. Назначение агента. Создание агента в JADE. Основные методы class Agent.

Агент – это вычислительный процесс, который населяет AgentPlatform и обычно предлагает одну или несколько вычислительных услуг, которые могут быть опубликованы в виде описания услуг. Он обладает независимым поведением и способен обмениваться сообщениями с другими агентами. Агент обладает набором поведений (функций, методов), для отработки заданной реакции на определенные события.

Агент JADE создается с помощью определения класса, наследуемого от класса *jade.core.Agent*, и реализации метода *setup()*:

```
import jade.core.Agent;
public class BookBuyerAgent extends Agent {

	protected void setup() {
		System.out.println(“Hello! Buyer-agent “+getAID().getName()+” is ready.”);
	}
}
```

Метод *setup()* выполняет инициализацию агента. Далее агент функционирует в рамках своего поведения.

3. Понятие платформы, контейнера, служебных агентов (AMS, DF)

Платформа JADE является распределенной и представляет собой набор *контейнеров. Контейнером* называется динамическая среда исполнения мультиагентных приложений, в которой находятся агенты. Каждый контейнер может содержать несколько агентов. Набор активных контейнеров называется *платформой*. Один из контейнеров всегда является главным (*Main container*), все остальные контейнеры связываются с ним и регистрируются в момент запуска. Поэтому первым контейнером при старте платформы должен быть главный, а все остальные контейнеры должны быть «обыкновенными» (т.е. неглавными) контейнерами и должны заранее «знать», как найти главный контейнер, на котором они будут регистрироваться, т.е. должны иметь данные о хосте и порте.

Другой главный контейнер, запущенный где-либо в сети, представляет собой другую платформу, на которой могут зарегистрироваться новые обычные контейнеры

Кроме возможности приёма регистраций от других контейнеров, главный контейнер отличается от обычного контейнера тем, что содержит два специальных агента, автоматически запускаемых одновременно с контейнером:

- **AMS (Agent Management System)** – данный агент управляет всей платформой. Этот агент необходим для взаимодействия агентов платформы, он обеспечивает доступ агентов к сервису белых страниц платформы и управляет их жизненным циклом. AMS обеспечивает различные операции платформы, такие как: создание и удаление агентов, удаление контейнеров, завершение работы платформы. Каждому агенту необходимо зарегистрироваться в AMS, для получения персонального идентификатора AID. Запуск агента AMS происходит внутри главного контейнера платформы, но может существовать в любом контейнере. Агент, совершающий действия связанные с функционированием платформы, должен вначале запросить подтверждение у агента AMS.

- **-DF (Directory Facilitator** – менеджер директорий) представляет собой службу «желтых страниц» (yellow pages), где агенты могут публиковать информацию о предоставляемых ими сервисах. С помощью DF агент может находить агентов, предоставляющих необходимые ему сервисы, и вступать с ними в переговоры. Внутри одной платформы может существовать несколько DF, предоставляющих информацию о различных группах сервисов или о сервисах различных групп агентов. Агенты платформы могут подписываться у DF-агента на получение информации о регистрации необходимого сервиса.

4. Простые поведения OneShotBehaviour, Behaviour, WakerBehaviour, TickerBehaviour. Их назначения, отличия, основные методы

Для отработки реакции на события агент имеет поведения.

Типы поведений:

- простое поведение:
    - единожды исполняемое поведение;

    - циклически исполняемое поведение;
- сложное поведение:
    - поведение-триггер;
    - последовательное поведение;
    - параллельное поведение.

«Одноразовое» поведение завершается сразу и его метод action() выполняется только один раз. jade.core.behaviours.OneShotBehaviour уже реализовывает метод done(), возвращая true и может быть удобным образом наследоваться, чтобы реализовывать данный тип поведения.

```
public class  MyOneShotBehaviour extends OneShotBehaviour {
  public void action() {
  // perform operation X
  }
 }
```

В данном примере операция X выполнится единожды.

«Циклическое» поведение никогда не завершается и его метод action() выполняет одни и те же операции каждый раз, когда он вызывается. jade.core.behaviours.CyclicBehaviour уже реализовывает метод done() всегда возвращая false и может наследоваться для реализации циклических моделей поведения.

```
public class  MyCyclicBehaviour extends CyclicBehaviour {
        public void action() {
                // perform operation Y
        }
}
```

Операция Y будет выполняться в цикле бесконечно (пока агент, имеющий это поведение, не будет завершён).

Общий случай поведения включает в себя статус, в зависимости от которого выполняются различные операции. Выполнение действий завершается, когда встречается данное условие.

```
public  class MyThreeStepBehaviour extends Behaviour {
        private int step = 0;
        public void action() {
                switch (step) {
                case 0:
                        // perform operation X
                        step++;
                break;
                case 1:
                        // perform operation Y
                        step++;
                break;
                case 2:
                        // perform operation Z
                        step++;
                break;
                }
```

```
        }

        public boolean done() {
        return step == 3;
        }
}
```

Jade предоставляет возможность соединять простые формы поведения для создания более сложных. (SequentialBehaviour, ParallelBehaviour и FSMBehaviour).
Jade предоставляет два готовых класса (в пакете jade.core.behaviours), с помощью которых можно легко реализовать поведение, позволяющее выполнять определённые действия в заданные моменты времени.

- WakerBehaviour, в котором методы action() и done() уже реализованы таким образом, что вызывается абстрактный метод handleElapsedTimeout() после того, как заданный промежутка времени (указанный в конструкторе) истечёт.

```
public class  MyAgent extends Agent {
  protected void setup() {
  System.out.println(“Adding waker  behaviour”);
  addBehaviour(new WakerBehaviour(this,  10000) {
  protected void handleElapsedTimeout() {
  // perform operation X
  }
  } );
  }
  }
```

В данном случае операция X выполнится через 10 секунд после того, как была выведена надпись «Adding waker behaviour»

- В TickerBehaviour методы action() и done() реализованы таким образом, что вызывают циклически абстрактный метод onTick(), ожидая некоторое (заданное в конструкторе) время после каждого выполнения. TickerBehaviour никогда не завершится.

```
public class  MyAgent extends Agent {
  protected void setup() {
  addBehaviour(new TickerBehaviour(this,  10000) {
  protected void onTick() {
  // perform operation Y
  }
  } );
  }
  }
```

Операция Y выполняется с периодом в 10 секунд.

5. Язык ACL, структура ACLmessage, методы для обмена сообщениями, фильтрация сообщений

Сообщения, которыми обмениваются JADE агенты имеют языковой формат ACL определенный FIPA(http://www.fipa.org) - международным стандартом взаимодействия агентов. Этот формат сообщения включает в себя несколько обязательных полей и, в частности:

- Отправитель сообщения
- Список получателей
- коммуникативные действия («пожелания», Перформатив), указывающие цель отправки сообщения. Типы сообщений:
  - REQUEST (запрос), если отправитель желает, чтобы получатель произвѐл некоторое действие,
  - INFORM (информирование), если отправитель желает, чтобы получатель был извещѐн о некотором факте,
  - QUERY_IF, если отправитель желает знать, достигнуто или нет заданное условие,
  - CFP (извещение о предложении), PROPOSE (предложение), ACCEPT_PROPOSAL (принятие предложения), REJECT_PROPOSAL (отклонение предложения), если отправитель и получатель ведут переговоры,
  - другие типы коммуникативных действий;
- Содержание т.е. фактическая информация, содержащаяся в сообщении (т.е. действия, которые будут выполняться в REQUEST сообщении, то, что отправитель желает раскрыть в INFORM сообщение ...).
- Язык содержания, т.е. какой синтаксис используется, чтобы выразить содержание (как отправителю, так и получателю необходимо иметь возможность кодировать / раскодировать выражения).
- Онтология, то есть словарь символов, используемых в их содержании и их значения(как отправитель так и получатель должны приписывать тот же смысл символам в сообщении , чтобы корректно понимать его).
- Некоторые поля, используемые для контроля нескольких одновременных разговоров и указания таймаутов для получения ответов, таких, как conversation-ID,reply-with,in-reply-to,reply-by.

С сообщением в JADE работают как с объектом класса jade.lang.acl.ACLMessage, который имеет get и set методы обработки всех полей сообщения.

Отправка сообщения другому агенту, состоит в заполнении ACLMessage объекта и вызове метода send() класса Agent.Код, приведенный ниже, информирует агента по имени Peter о том, что сегодня дождь — «today it's raining».

```
ACLMessage msg  = new ACLMessage(ACLMessage.INFORM);
  msg.addReceiver(new  AID(“Peter”, AID.ISLOCALNAME));
  msg.setLanguage(“English”);
  msg.setOntology(“Weather-forecast-ontology”);
  msg.setContent(“Today  it’s raining”);
  send(msg);
```

JADE автоматически размещает сообщения в личной очереди сообщений получателя, как только они приходят. Агент может забрать сообщения из своей очереди сообщений с использованием метода receive(). Этот метод возвращает первое сообщение в очереди сообщений (удаляет его) или null, если очереди сообщений пуста.

```
ACLMessage msg  = receive();
  if (msg !=  null) {
  // Process the message }
```

Фильтрация сообщений

Когда шаблон указан, метод receive() возвращает первое сообщение (если оно есть), соответствующее этому шаблону, игнорирую все неподходящие сообщения. Подобные шаблоны реализованы как экземпляры класса jade.lang.acl.MessageTemplate, который предоставляет набор производящих методов, позволяющих просто и гибко создавать шаблоны.

```
public void  action() {
  MessageTemplate  mt = MessageTemplate.MatchPerformative(ACLMessage.CFP);
  ACLMessage msg  = myAgent.receive(mt);
  if  (msg != null) {
  // Получено сообщение типа CFP. Обработка сообщения
  ...
  }
  else  {
  block();
  }
  }
```

Метод createReply () класса ACLMessage автоматически создает новую настройку ACLMessageproperly получателя и все поля, используемые для управления сеансом (conversation-id, reply-with, in-reply-to).

6. Конечный автомат. FSMbehaviour

7. Сервис желтых страниц. Взаимодействие и назначение

Сервис «жёлтых страниц» в JADE (согласно спецификации FIPA) предоставлен агентом, названным DF (Directory Facilitator, Менеджер Директорий). Каждая соответствующая FIPA платформа поддерживает по умолчанию агент DF (его локальное имя — «df»). Другие DF-агенты могут быть запущены и несколько DF-агентов (включая те, что находятся в платформе по умолчанию) и объединены для обеспечения единого распределённого каталога «жёлтых страниц».

Поскольку DF является агентом, имеется возможность взаимодействовать с ним привычным образом обмениваясь ACL-сообщениями, используя соответствующий язык содержания (язык SLo) и соответствующую онтологию (онтологию FIPA-agent-management), согласно спецификациям FIPA. Для упрощения этих взаимоотношений JADE предоставляет класс jade.domain.DFService с помощью которого можно опубликовывать и искать сервисы через вызовы методов.

Агент, желающий опубликовать (сделать доступными публично) один или более сервисов должен предоставить DF описание, включающее, AID этого агента, список возможных языков и онтологий, которые должны знать другие агенты для взаимодействия с ним и список сервисов для публикации. Для каждого опубликовываемого сервиса предоставляется описание, включающее тип сервиса, его имя, языки и онтологии, необходимые для его использования и ещё некоторое количество свойств, специфичных для данного сервиса. Классы DFAgentDescription, ServiceDescription и Property, включённые в пакет jade.domain.FIPAAgentManagement, соответственно представляют собой три упомянутые абстракции.

Чтобы опубликовать сервис агент должен создать подходящее описание (представленное экземпляром класса DFAgentDescription) и вызвать статический метод register() класса DFService. Типично (но не обязательно) регистрация сервиса проходит в методе setup() как показано ниже в случае с агентом-продавцом книг.

```
protected void  setup() {
  ...
  // Register the  book-selling service in the yellow pages
  DFAgentDescription  dfd = new DFAgentDescription();
  dfd.setName(getAID());
  ServiceDescription  sd = new ServiceDescription();
  sd.setType(“book-selling”);
  sd.setName(“JADE-book-trading”);
  dfd.addServices(sd);
  try {
  DFService.register(this,  dfd);
  }
  catch  (FIPAException fe) {
  fe.printStackTrace();
  }
  ...
  }
```

Агент, которому требуется найти некоторые сервисы, должен предоставить агенту DF описание необходимого шаблона. Результатом поиска является список всех описаний, которые удовлетворяют указанному шаблону. Описание соответствует шаблону, если все поля, указанные в шаблоне, присутствуют с теми же значениями и в описании. Использование статического метод *search()* класса *DFService* можно проиллюстрировать примером, в котором агент-покупатель книг динамически находит всех агентов, предоставлямых сервисом *«book-selling»*

```
DFAgentDescription template = new DFAgentDescription();
ServiceDescription sd = new ServiceDescription();
sd.setType(“book-selling”);
template.addServices(sd);
try {
        DFAgentDescription[] result = DFService.search(myAgent, template);
        sellerAgents = new AID[result.length];
        for (int i = 0; i < result.length; ++i) {
```

```
        sellerAgents[i] = result[i].getName();
        }
}
catch (FIPAException fe) {
        fe.printStackTrace();
}
```

Класс *DFService* также предоставляет поддержку подписки на уведомления от DF о появлении агента, предоставляющего указанный сервис (методы *searchUntilFound()* и *createSubscriptionMessage()*).

8. AMS агент. Назначение и способы взаимодействия с ним

AMS является обязательным компонентом AgentPlatform, и в одной AgentPlatform будет существовать только одна AMS. AMS отвечает за управление работой AgentPlatform, такой как создание агентов, удаление агентов и наблюдение за миграцией агентов в и из AgentPlatform (если AgentPlatform поддерживает мобильность агентов). Поскольку разные AgentPlatform имеют разные возможности, AMS может быть запрошен для получения описания своей AgentPlatform. (дополнительно см.Вопрос 3).

AMS имеет возможность выполнять следующие функции:

В части ams-agent-description:

-регистрация
-отмена регистрации
-изменение
-поиск
-получение описания

В части ap-description:

-приостановление действий агента(отправить в режим ожидания)
-уничтожение агента
-создание агента
-возобновление действия агента
-вызов агента
- выполнение агентом действий
-управление ресурсами

Проще говоря AMS – это агент с максимально широкими полномочиями(управления платформой). Все операции по управления платформой моделируются дейсвтиями JADE-Agent-Management ontology.

Наименование действия	Класс	описание
Создание агента	CreateAgent	
Убийство агента	KillAgent	
Убийство контейнера	KillContainer	
Отключение платформы	ShutdownPlatform	
Запрос контейнеров в платформе	QueryPlatformLocationsAction	Позволяет получить список всех контейнеров на платформе. Возвращает List of ConteinerID.

Запрос Агентов в контейнере	QueryAgentsOnLocation	Позволяет получить список агентов, в настоящее время живущих в данном контейнере. Возвращает List of AID.
Поиск Агнета	WhereIsAgentAction	Позволяет получить ID контейнера, в котором агент на данный момент живет.
Установка MTP	InstallMTP	Активирует новый MTP данного класса в данном контейнере. В результате установки MTP создается URL
УдалениеMTP	UninstallMTP	Деактивирует MTP с заданным URL в заданном контейнере

Помимо обработки запросов на выполнение операций управления платформой, JADE AMS также поддерживает механизм подписки, с помощью которого заинтересованные агенты могут получать уведомления о событиях платформы, таких как создание и завершение работы агентов, завершение работы контейнера и так далее.

Наименование действия	Класс	описание
	AddedContainer	Оповещает о присоединении нового контейнера
	RemovedContainer	Оповещает о том что контейнер исчез из платформы
	BornAgent	Оповещает о создании агента на данной платформе
	DeadAgent	Оповещает о смерти агента на данной платформе
	SuspendedAgent	Оповещает о приостановлении действий Агента
	ResumedAgent	Оповещает о возобновлении действий Агента
	MovedAgent	Оповещает о том что агент переместился из одной платформы в другую
	ClonedAgent	Оповещает о том что агент был клонирован на данной

		платформы
	KillContainerRequested	Указывает, что AMS собирается обработать запрос на удаление заданного контейнера
	ShutdownPlatformRequested	Указывает, что AMS собирается обработать запрос на отключение заданной платформы

Графические инструменты JADE, такие как RMA и Sniffer, используют данные подписки.

9. Сервисы. Назначение, способ использования встроенных сервисов. Topic service

Версия 3.5 JADE также поддерживает тематическое общение, то есть, помимо отправки сообщений одному или нескольким получателям (с указанием имени), можно отправлять сообщения на заданную тему. Такие сообщения будут доставлены всем агентам, которые зарегистрировали свой интерес к этой теме. Если ни один агент не зарегистрировал свою заинтересованность в какой-либо теме, отправка сообщения в эту тему вообще не имеет никакого эффекта (т. Е. Ответ FAILURE не получен). Следует отметить, что агенту отправителя не нужно знать, какой агент зарегистрирован в теме.

Тематическое взаимодействие осуществляется с помощью jade.core.messaging.TopicManagementService, поэтому его необходимо активировать во всех контейнерах платформы.

Чтобы иметь полностью унифицированный API для отправки как сообщений получателям, так и сообщений по заданной теме, темы представлены объектами AID. Интерфейс TopicManagementHelper, включенный в пакет jade.core.messaging, предоставляет удобные методы для создания AID тем (createTopic ()) и проверки, представляет ли данный AID тему (isTopic ()). Чтобы отправить сообщение в заданную тему, достаточно добавить AID темы среди получателей сообщений.

10. Встроенные способы коммуникации. AchiveReInitiator, AchiveReResponder. Структура взаимодействия.

JADE предоставляет пару AchieveREInitiator / Responder классов, которые являются единой однородной реализацией всех этих протоколов взаимодействия.

На рисунке 5 показана структура этих протоколов взаимодействия. Инициатор отправляет сообщение (как правило, он выполняет коммуникативное действие, как показано в белом поле). Ответчик может затем ответить отправкой непонятного или отказа в достижении рационального эффекта от коммуникативного действия, либо также сообщения о согласии для передачи соглашения о выполнении (возможно, в будущем) коммуникативного действия, как показано в первый ряд затененных коробок. Респондент выполняет действие и, наконец, должен ответить сообщением о результате действия (в конечном итоге просто о том, что действие было выполнено) или с ошибкой, если что-то пошло не так. Обратите внимание, что мы расширили протокол, чтобы сделать факультативной передачу сообщения о согласии. Фактически, в большинстве случаев выполнение действия занимает настолько короткое время, что отправка сообщения о согласии - это просто бесполезная и неэффективная служебная нагрузка; в таких случаях

согласие на совершение коммуникативного акта включается в прием следующего сообщения в протоколе

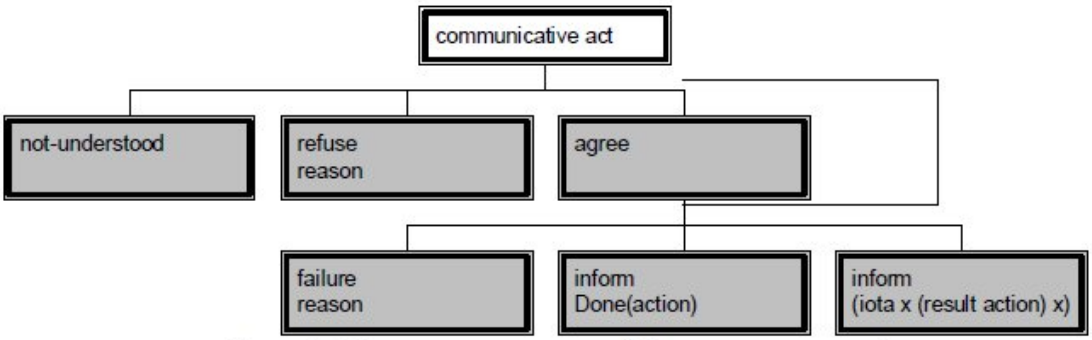

Figure 5 - Homogeneous structure of the interaction protocols.

AchieveREInitiator

Конструктор принимает сообщение, которое должно быть отправлено, и ссылку на агента, который отправляет сообщение. Поведение заботится об отправке самого сообщения. Единственный метод, который должен быть реализован - это handleInform, который вызывается при получении сообщения INFORM от ответчика.

AchieveREInitiator может также отправлять запрос более чем одному агенту за раз, агенты просто добавляются как получатели к сообщению, переданному в конструкторе. Ответы от Ответчиков могут быть обработаны или один за другим, или в методе handleAllResultNotifications, который получает все ответы сразу.

Экземпляр этого класса может быть легко сконструирован путем передачи в качестве аргумента своего конструктора сообщения, используемого для инициирования протокола. Важно, чтобы это сообщение имело правильное значение для слота протокола ACLMessage, как определено константами в интерфейсе FIPANames.InteractionProtocols, включенном в пакет jade.domain. Обратите внимание, что этот объект ACLMessage также может быть неполным, когда создается конструктор этого класса; метод обратного вызова prepareRequests может быть переопределен для возврата завершенного ACLMessage или, точнее (потому что этот инициатор позволяет управлять диалогом 1: N) вектора объектов ACLMessage для отправки.

Класс может быть легко расширен путем переопределения одного (или всех) его методов обратного вызова handle ..., которые предоставляют хуки для обработки всех состояний протокола. Например, метод handleRefuse вызывается при получении сообщения об отказе.

Опытные программисты могут найти полезным, вместо того, чтобы расширять этот класс и переопределять некоторые из его методов, регистрируя специфичные для приложения Поведения как обработчик состояний протокола, включая, например, другое поведение AchieveREInitiator для запроса пароля, прежде чем соглашаться выполнять коммуникативный акт. Методы registerHandle ... позволяют это сделать. Сочетание переопределенных методов и зарегистрированных поведений часто может быть лучшим решением. Стоит уточнить различие между следующими тремя обработчиками:

- handleOutOfSequence обрабатывает все неожиданные полученные сообщения, которые имеют правильный идентификатор диалога или значение in-reply-to

- handleAllResponses обрабатывает все полученные первые ответы (то есть не понято, отказываются, соглашаются) и вызывается после вызова handleNotUnderstood /

Refuse / Agreement для каждого полученного отдельного ответа. В случае разговоров 1: N переопределение этого метода может быть более полезным, чем переопределение других методов, поскольку этот позволяет обрабатывать все сообщения за один вызов.

- handleAllResultNotifications обрабатывает все полученные вторые ответы (т.е. сбой, информирование) и вызывается после вызова handleFailure / Inform для каждого полученного ответа. В случае разговоров 1: N переопределение этого метода может быть более полезным, чем переопределение других методов, поскольку этот позволяет обрабатывать все сообщения за один вызов.

Доступен набор переменных (они не являются константами!) (..._ KEY), которые предоставляют ключи для получения следующей информации из хранилища данных этого поведения:

- getDataStore (). get (ALL_RESPONSES_KEY) возвращает объект Vector объекта ACLMessage со всеми первыми ответами (т. е. непонятно, отказать, согласиться)

- getDataStore (). get (ALL_RESULT_NOTIFICATIONS_KEY) возвращает объект Vector объекта ACLMessage со всеми вторыми ответами (т. е. сбой, информирование)

- getDataStore (). get (REQUEST_KEY) возвращает объект ACLMessage, переданный в конструктор класса

- getDataStore (). get (ALL_REQUESTS_KEY) возвращает вектор объектов ACLMessage, возвращаемых методом prepareRequests. Напомним, что если поведение зарегистрировано как обработчик состояния PrepareRequests, то это поведение отвечает за помещение в хранилище данных соответствующего вектора объектов ACLMessage (привязанного к правому ключу), который должен быть отправлен этим инициатором.

Эта реализация управляет истечением времени ожидания, выраженного значением интервала ответа отправленных объектов ACLMessage. В случае диалога 1: N минимум оценивается и используется между значениями всех интервалов ответа отправленных объектов ACLMessage. Обратите внимание, что, как определено FIPA, это время ожидания относится ко времени, когда должен быть получен первый ответ (например, сообщение о согласии). Если приложениям необходимо ограничить время ожидания для получения последнего информационного сообщения, они должны встроить это ограничение в содержимое сообщения, используя прикладные онтологии.

AchieveREResponder

Этот класс является реализацией роли респондента. Чтобы создать класс AchieveREResponder, нам нужно настроить MessageTemplate, основанный на типе сообщений, на которые мы хотим реагировать, чаще всего это сообщения REQUEST протокола FIPA_REQUEST., фактически он используется для выбора того, какой полученный ACLMessage должен быть обработан. Метод createMessageTemplate может использоваться для создания шаблона сообщения для данного протокола взаимодействия, но в некоторых случаях могут быть полезны более выборочные шаблоны, например, для создания экземпляра этого класса для каждого возможного агента отправителя.

```
MessageTemplate template = MessageTemplate.and(
MessageTemplate.MatchProtocol(FIPANames.InteractionProtocol.FIPA_REQUEST),
MessageTemplate.MatchPerformative(ACLMessage.REQUEST) );
```

Затем нам нужно реализовать метод handleRequest, который обрабатывает запрос. Этот метод возвращает сообщение, которое должно быть отправлено в качестве ответа. Если он возвращает null или сообщение AGREE (это необязательно в протоколе, однако его следует отправлять, особенно в тех случаях, когда обработка запроса может занять больше времени), мы должны реализовать метод prepareResultsNotification, который возвращает результаты задачи (как сообщение INFORM).

Класс может быть легко расширен путем переопределения одного (или всех) его методов prepare ..., которые предоставляют хуки для обработки состояний протокола и, в частности, для подготовки ответных сообщений. Метод prepareResponse вызывается, когда сообщение инициатора получено и первый ответ (например, согласие) должен быть отправлен обратно; вместо этого вызывается methodprepareResultNotification, когда должен быть достигнут рациональный эффект (например, действие должно быть выполнено в случае протокола FIPa-Request), и сообщение об окончательном ответе должно быть отправлено обратно (например, inform (done)). Будьте внимательны, возвращая правильное сообщение и устанавливая все необходимые слоты сообщения ACLMessage; в общем случае настоятельно рекомендуется создать ответное сообщение с помощью метода createReply () класса ACLMessage.

Опытные программисты могут оказаться полезными, вместо того, чтобы расширять этот класс и переопределять некоторые из его методов, регистрируя специфичные для приложения Поведения как обработчик состояний протокола.

Методы registerPrepare ... позволяют это сделать. Сочетание переопределенных методов и зарегистрированных поведений часто может быть лучшим решением.

Доступен набор переменных (они не являются константами!) (..._ KEY), которые предоставляют ключи для получения следующей информации из хранилища данных этого поведения:

- getDataStore (). get (REQUEST_KEY) возвращает объект ACLMessage, полученный инициатором
- getDataStore (). get (RESPONSE_KEY) возвращает первый объект ACLMessage, отправленный

инициатор

- getDataStore (). get (RESULT_NOTIFICATION_KEY) возвращает второй объект ACLMessage, отправленный инициатору. Напомним, что если поведение зарегистрировано как обработчик состояний Prepare ..., это поведение является обязанностью поместить его в хранилище данных (привязанное к правый ключ) правильный объект ACLMessage для отправки этим респондентом

11. ContractNetInitiator и ContractNetResponder. Описание коммуникации.

Сетевой протокол немного сложнее. В этом случае Инициатор отправляет сообщение CFP (Call for Proposal) нескольким Ответчикам. Это сообщение содержит действие, которое должен выполнить Ответчик.

Каждый Ответчик отвечает сообщением (ПРЕДЛОЖЕНИЕ), в котором он описывает условия (например, цену) для выполнения действия. Он также может ответить, что не будет выполнять действие вообще (ОТКАЗАТЬ). Инициатор выбирает из ответов

несколько Ответчиков для выполнения действия и отправляет им сообщение ACCEPT_PROPOSAL. другим ответчикам он отправляет REJECT_PROPOSAL.

Первый шаг протокола такой же, как в протоколе запроса, только инициатор отправляет CFP вместо REQUEST. Таким образом, Ответчик знает, что он не должен выполнять действие, а отправлять его условия. Действие выполняется после того, как Ответчик получает сообщение ACCEPT_PROPOSAL.

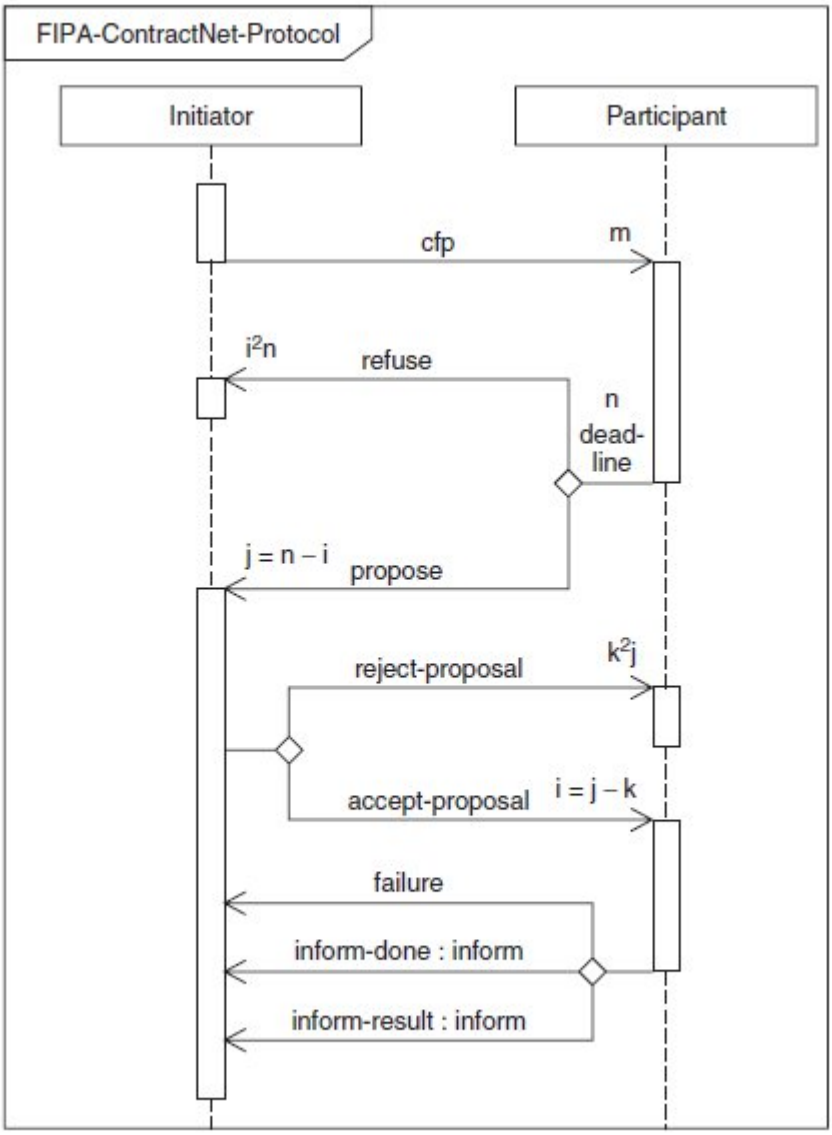

Figure 5.8 The FIPA-Contract-Net interaction protocol

Участвуя в диалоге, управляемом протоколом взаимодействия, агент может играть роль инициатора или ответчика. Как следствие, классы в пакете jade.proto делятся на инициаторов и респондентов. Есть ContractNetInitiator и ContractNetResponder, SubscriptionInitiator и SubscriptionResponder и так далее.

Инициатор ContractNet В Джейде Инициатор внедряется в ContractNetInitiator поведение. Сначала нам нужно пройти сообщение CFP, которое должно быть отправлено Ответчикам, конструктору. Затем агент ждет ответов. При каждом получении ответа вызываются методы handlePropose или handleRefuse. После получения всех ответов вызывается метод handleAllResponses. Затем инициатор выбирает ответчиков, которые будут выполнять действие, и отправляет им сообщение ACCEPT_PROPOSAL. Он

отправляет REJECT_PROPOSAL остальным ответчикам. Инициатор может отправлять эти сообщения по мере их поступления из handlePropose или ждать всех ответов и отправлять решения из метода handleAllResponses. После того как решения отправлены, агент ожидает сообщения INFORM о завершении действия и может обработать их в методе handleInform.

Все конструкторы класса инициатора содержат параметр ACLMessage, представляющий сообщение, используемое для инициации протокола. Например, класс ContractNetInitiator получает сообщение CFP для отправки респондентам, чтобы инициировать процедуру делегирования задачи.

Contract-Net Responder

Ответчик протокола Contract-Net реализован в ContractNetResponder. После получения CFP он обрабатывает его с помощью метода handleCfpp. Здесь он указывает свое условие (ПРЕДЛОЖИТЬ) или сразу же отклоняет задачу (ОТКАЗАТЬ).

После получения сообщения ACCEPT_PROPOSAL он выполняет действие, указанное в CFP. ACCEPT_PROPOSAL может быть обработан в методе handleAcceptProposal. Агент также может обрабатывать отклонение его предложения в handleRejectProposal