Лабораторная работа № 2.
Основы программирования агентов в среде JADE.
Модели поведения агентов

Содержание работы:

Цель работы.
1. Теоретические сведения.
   1.1. Общие принципы построения пользовательских агентов в среде JADE.
   1.2. Жизненный цикл и состояния агента.
   1.3. Запуск агента на выполнение. Метод **setup()** класса **Agent**. Поведения агента.
   1.4. Типовые модели поведения.
2. Порядок выполнения работы.
3. Содержание отчета.
4. Вопросы для самопроверки.

Цель работы:

1. Получить знания о состояниях жизненного цикла агентов и принципах объектно-ориентированной реализации агентов в среде JADE.

2. Изучить типовые модели поведения, встроенные в JADE.

3. Получить практические навыки создания пользовательских агентов на платформе JADE, выполняющих простые модели поведения.

Теоретические сведения

1.1. Общие принципы построения пользовательских агентов в среде JADE.

Для создания прикладных агентов в JADE пользователю необходимо определить свой класс, производный от базового класса **Agent** (jade.core.Agent). Класс **Agent** поддерживает все основные функции взаимодействия с платформой (регистрация, конфигурирование, удаленное управление), а также базовое множество методов для реализации задаваемого пользователем поведения агента (отправка/прием сообщений, использование стандартных протоколов взаимодействия, регистрация в нескольких доменах и др.). Пользовательский агент создается как экземпляр пользовательского класса (например, “myAgent”), производного от класса **Agent**.

Для обозначения различных функций (сервисов), реализуемых агентом, используется понятие “*поведений*” (behaviours). Вычислительная модель агента является мультизадачной, т.е. задачи (*поведения*) агента выполняются параллельно.

Планирование поведений осуществляется скрытым от программиста внутренним планировщиком базового класса **Agent**.

1.2. Жизненный цикл и состояния агента.

В соответствии со спецификацией FIPA Agent Management (FIPA00023) JADE-агент может находится в одном из восьми состояний, которые представлены соответствующими константами в классе **Agent**:

- **AP_INITIATED -** объект Агент создан, но еще не зарегистрировался у AMS, не имеет имени и адреса и не может общаться с другими агентами;
- **AP_ACTIVE -** объект Агент зарегистрирован у AMS, имеет регулярное имя и адрес(а) и имеет доступ ко всему множеству возможностей JADE;
- **AP_SUSPENDED -** объект Агент в данный момент остановлен. Его внутренний поток вычислений (thread) приостановлен и никакое поведение агента не выполняется;
- **AP_WAITING -** объект Агент блокирован и ожидает некоторого события. Его внутренний поток вычислений находится в “спящем” состоянии для Java-монитора и будет активизирован при наступлении некоторого события (как правило, при поступлении сообщения);
- **AP_DELETED –** Агент “мертв”. Выполнение внутреннего потока вычислений Агента завершено и он больше не является зарегистрированным у AMS;
- **AP_TRANSIT -** мобильный агент переходит в это состояние, когда он находится в процессе миграции к новому месту нахождения. Система продолжает буферировать сообщения, которые затем будут пересылаться к новому месту нахождения агента;
- **AP_COPY -** это состояние используется внутри JADE при клонировании агента;
- **AP_GONE -** это состояние используется внутри JADE, когда мобильный агент переместился в новое место и перешел в устойчивое состояние.

Для выполнения переходов между состояниями класс **Agent** поддерживает открытые (public) методы, имена которых соответствуют переходам машины состояний, приведенной в спецификации FIPA “*Agent Management*” (рис. 1). Например, метод **doWait()** переводит агента в состояние AP_WAITING, метод

doSuspend() - в состояние AP_SUSPENDED из состояний AP_ACTIVE или AP_WAITING и т. д.

Агенту разрешено выполнять его поведения (т.е. задачи) только когда он находится в состоянии AP_ACTIVE. Следует иметь в виду, что если какое-то поведение вызывает метод **doWait()**, то блокируется весь агент и все его поведения. Для того, чтобы приостановить только одно поведение агента, следует использовать метод **block()** класса **Behaviour**.

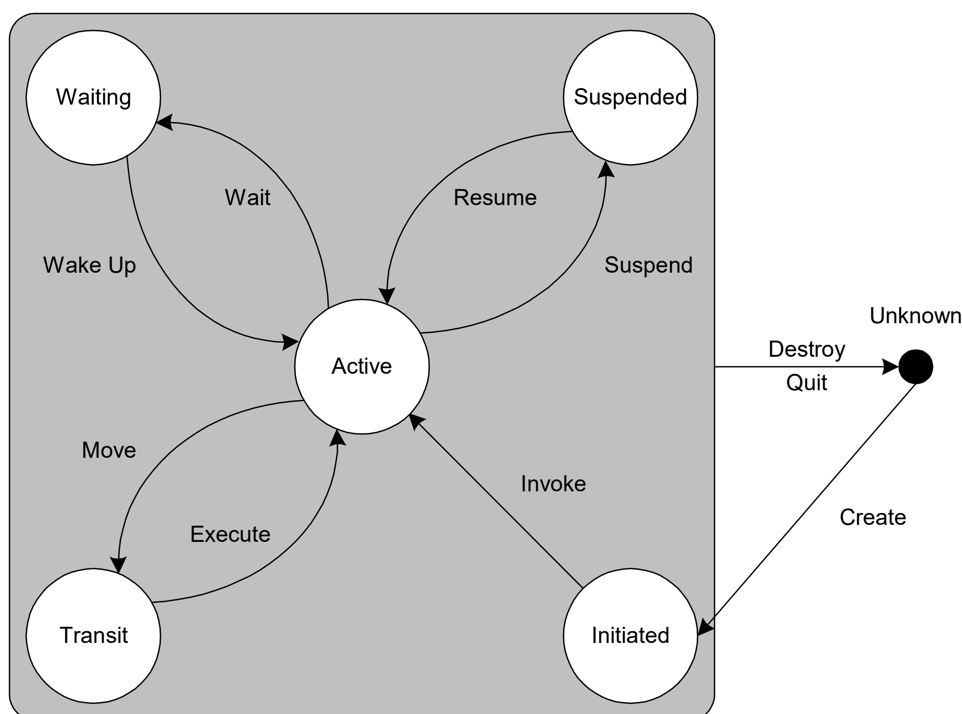

Рис. 1 - Диаграмма переходов между состояниями жизненного цикла FIPA агента

1.3. Запуск агента на выполнение. Метод **setup()** класса **Agent**.

При “рождении” нового агента на платформе выполняется следующая последовательность действий:

- выполняется конструктор агента;
- агенту присваивается идентификатор (AID);
- агент регистрируется у AMS;
- агент переводится в состояние AP_ACTIVE;
- вызывается метод **setup()** агента.

Идентификатор агента (AID) в соответствии со спецификацией FIPA имеет следующие атрибуты:

- *глобально уникальное имя*. В JADE это имя представляет собой конкатенацию локального имени агента (введенного в командной строке), символа '@' и идентификатора домашней платформы агента в формате:

```
<ЛокальноеИмяАгента>'@'<ИмяХоста>':'<НомерПортаРегистрации_JADE_ RMI>
'/' 'JADE'
```

(Пример уникального имени агента: myAgent@localhost:1099/JADE).

- *множество адресов агента*. Каждый агент наследует транспортные адреса своей домашней агентной платформы;
- *множество серверов разрешения имен*, т.е. сервисов “белых страниц” у которых агент регистрируется.

Для представления идентификатора агента в JADE используется класс jade.core.AID, конструктору которого передается в параметре строка-идентификатор: AID(String name).

Метод **setup()** является точкой входа, в которой стартует активность прикладного агента. Программист должен реализовать метод **setup()**, который используется для инициализации агента. Этот метод начинает выполняться, когда агент уже зарегистрирован у AMS и находится в состоянии AP_ACTIVE. Программист может использовать процедуру инициализации для выполнения следующих действий:

- (необязательной) модификации зарегистрированных у AMS данных;
- (необязательного) задания описания агента и предоставляемых им сервисов, а также регистрации его в одном или более доменах, т. е. у DF;
- (обязательного) добавления задач (поведений) к очереди готовых задач с помощью метода **addBehaviour()**. Эти поведения планируются сразу же после завершения метода **setup()**.

В методе **setup()** должно быть создано и добавлено в агента хотя бы одно *поведение* (Behaviour). Конкретные поведения реализуются как экземпляры соответствующих классов, которые пользователь должен объявить в создаваемом агентном классе, расширяя соответствующие базовые классы JADE. Таким образом, минимальная реализация метода setup() имеет вид:

```
public void setup()
{
   myBehaviour b=new myBehaviour();
   addBehaviour(b);
}
```

Здесь myBehaviour – определяемый пользователем класс поведения, а b - экземпляр этого класса, который добавляется в данного агента (например, myAgent) методом addBehaviour(b).

После завершения метода **setup()** JADE автоматически выполняет первое поведение в очереди готовых задач, а затем переключается на другие поведения в очереди используя циклический (round-robin) планировщик без приоритетов. В данной лабораторной работе предполагается, что создаваемые агенты будут иметь несколько *поведений*.

1.4. Типовые модели поведения.

Каждый агент в процессе работы может решать некоторое множество задач. Для представления отдельной задачи, решаемой агентом, в среде JADE используется термин «*поведение*» (Behaviour). Каждый агент может реализовывать любое число *поведений*. *Поведения* могут добавляться агенту как при его инициализации (в теле метода `Agent.setup()`), так и в процессе последующей работы. Таким образом, агент является по существу контейнером поведений. Кроме того, из одного поведения можно запустить любое количество новых поведений.

Так как на одного агента (и, следовательно, на одну нить, выполняемую виртуальной Java-машиной) может приходиться несколько поведений, то эти поведения должны исполняться поочередно, а не одновременно. Таким образом, пока не будет выполнена одна модель поведения, управление не будет передано другим моделям поведения.

В JADE для реализации поведений используется абстрактный класс Behaviour, а все пользовательские поведения являются расширениями этого класса. Чтобы добавить агенту некоторое поведение используется метод `addBehaviour(Behaviour)` класса `jade.core.Agent.` Для удаления поведения, используется метод `removeBehaviour(Behaviour)`.

В теле любого *поведения* необходимо определить метод public void action(), в котором выполняются все действия данного поведения. Если в агенте запущено одновременно несколько поведений, то планировщик заданий агента по очереди запускает методы action(), поэтому если, например, в теле метода action() реализовать бесконечный цикл, то это заблокирует исполнение других поведений. По окончании исполнения action() вызывается метод public boolean done() данного поведения. Если этот метод возвращает true, то данное поведение считается выполненным, если false, то планировщик заданий снова включает данное поведение в очередь.

Следует отметить, что поведение не сохраняет никаких данных в стеке, поэтому все вычисления необходимо производить в переменных объектов Behaviour или Agent.

Если поведение находится в состоянии ожидания (например, прихода ACL сообщения), то для исключения холостой работы процессора используется метод block(). Этот метод начинает действовать после того, как выполнится метод action() и переводит данную модель поведения в очередь блокированных моделей поведений. Блокированные поведения запускаются при получении ACL-сообщений. Кроме того, метод block(long) позволяет переводить поведение в блокированное состояние на определенный промежуток времени, передаваемый в качестве параметра.

Для удобства разработчиков в JADE предусмотрены встроенные классы поведений, отражающие различные типы задач. Ниже приведена иерархия классов Behaviour (рисунок 2).

Класс SimpleBehaviour – абстрактный класс для поведений, не требующих прерывания своей работы.

Класс OneShotBehaviour – класс для поведений, которые запускаются только один раз. У данного класса метод done() всегда возвращает значение true.

Класс CyclicBehaviour – класс для моделей поведения, которые должны запускаться постоянно (циклически). У данного класса метод done() всегда возвращает значение false.

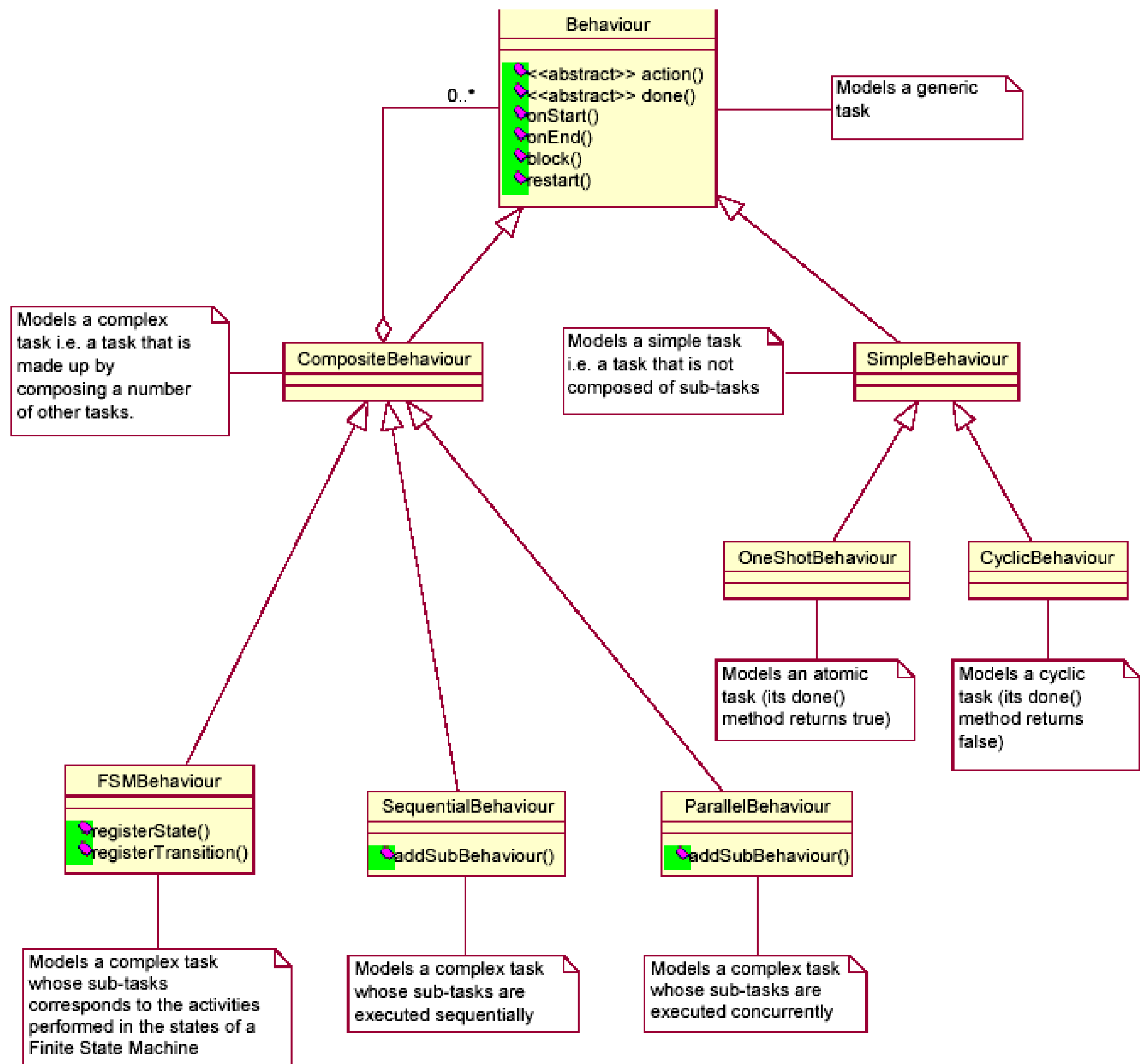

Рис. 2 - Иерархия классов Behaviour (Поведений)

Класс CompositeBehaviour – абстрактный класс, предназначенный для реализации сложных моделей поведения, состоящих из нескольких вложенных моделей поведения. Перед запуском моделей поведений – потомков от класса ComplexBehaviour выполняется метод onStart(), а после выполнения всех потомков – метод onEnd(). Эти два метода могут быть переопределены пользователем для реализации конкретных задач. Если необходимо бесконечное повторение исполнения модели поведения ComplexBehaviour, в методе onEnd() можно вызвать метод reset(). Так же имеется ряд методов которые необходимо переопределить:

1. void scheduleFirst() – определяет какие поведения запускать первыми
2. void scheduleNext(boolean parm1, int parm2) - определяет какие поведения запускать следующими
3. Collection getChildren() – получить список дочерних поведений

4. boolean checkTermination(boolean parm1, int parm2) – проверка на необходимость завершения выполнения поведения.
5. Behaviour getCurrent() – получить текущее поведение, которое выполняется.

Класс SequentialBehaviour – класс, предназначенный для последовательного исполнения моделей поведений – потомков. SequentialBehaviour заканчивает свою работу, когда все поведения – потомки выполнены. Модели поведения – потомки управляются методами addSubBehaviour(Behaviour) и removeSubBehaviour(Behaviour).

Класс ParallelBehaviour – класс, исполняющий модели поведения потомков параллельно. Исполнение ParallelBehaviour заканчивается при достижении какого-то заданного условия.

Класс FSMBehaviour реализует механизм машины с конечным числом состояний где сложная задача разбивается на ряд более мелких.

Таким образом шаблон объявления пользовательского класса агента имеет вид:

```
public class myAgent extends Agent
{
  public void setup()
  {
    // реализация метода setup()
  }
  class myBehaviour extends SimpleBehaviour
  {
    public void action()
    {
      // реализация метода action()
    }
    public boolean done()
    {
        // реализация метода done()
    }
  }
}
```

В данной лабораторной работе будет рассмотрены следующие поведения: SimpleBehaviour, OneShotBehaviour, CyclicBehaviour, SequentialBehaviour, ParallelBehaviour.

Остановка выполнения агента. Любое поведение может вызвать метод **Agent.doDelete()** для того, чтобы остановить выполнение агента. Когда агент готов перейти в состояние AP_DELETED (т.е. собирается прекратить свое

существование), выполняется метод **Agent.takeDown()**. Метод **takeDown()** может быть переопределен программистом, чтобы реализовать любые специфичные для приложения действия, необходимые для корректной выгрузки агента, такие как разрегистрация его у DF-агентов. Когда этот метод выполняется, агент еще зарегистрирован у AMS и, следовательно, может посылать сообщения другим агентам, но сразу же после завершения метода **takeDown()** агент будет разрегистрирован а его вычислительный поток уничтожен.

2. Порядок выполнения работы

2.1. Взять задание по вариантам из таблицы 2.1.

Таблица 2.1 - Варианты заданий

№	Родительский класс	Тип	Кол-во повторов поведения	Выводимая строка	Время ожидания	Примечание
1	Агент	SimpleBehaviour	5	Hi! Step <номер шага>!	5000	
	SimpleBehaviour	CyclicBehaviour	-	Cyclic step!!!	5000	Запустить на 1 шаге
	SimpleBehaviour	OneShotBehaviour	-	One text	1000	Запустить на 3 шаге
2	Агент	ParallelBehaviour	-	I am started!	-	Перед запуском поведений потомков
	Parallel Behaviour	SimpleBehaviour	5	First Behaviour	5000	
	Parallel Behaviour	SimpleBehaviour	8	Second Behaviour	2000	
3	Агент	SequentialBehaviour	-	End work!	-	После окончания поведений потомков
	SequentialBehaviour	SimpleBehaviour	10	Go one step!	3000	
	SequentialBehaviour	CyclicBehaviour	-	Cyclic Behaviour	8000	
4	Агент	ParallelBehaviour	-	-	-	
	Parallel Behaviour	OneShotBehaviour	-	Shot!!!	10000	
	Parallel Behaviour	CyclicBehaviour	-	Cyclic Behaviour done one more step!	8000	
5	Агент	SimpleBehaviour	10	Behaviour Single	4800	
	Агент	OneShotBehaviour	-	Behaviour One	4500	
	OneShotBehaviour	CyclicBehaviour	-	Behaviour Two	3300	

Исходный текст примера выполнения задания для варианта представленного в таблице 2.2 находится в приложении А.

Таблица 2.2 – Вариант примера

Родительский класс	Тип	Количество повторов поведения	Выводимая строка	Время ожидания	Примечание
Агент	CyclicBehaviour	-	First Behaviour	1000	
Агент	SimpleBehaviour (1)	3	Second Behaviour	3000	
SimpleBehaviour (1)	SimpleBehaviour (2)	1	SubBehaviour of First	3000	Запустить на 2 шаге

2.2. Написать тест программы на JAVA.

Примечание. Откомпилировать класс агента можно, используя команду:

```
javac -<classpath> MyClass
```

здесь MyClass – имя класса пользовательского агента, код которого содержится в файле MyClass.java.

2.3. Поправить программу так, чтобы протокол работы агента изменился. В вариантах, где поведение агента находится в бесконечном цикле добавить условие корректного выхода из цикла (например, после 10 шагов).

2.4. Запустить на исполнение несколько одинаковых агентов с набором моделей поведения и посмотреть, как работают в независимых нитях исполняемые поочередно модели поведения нескольких агентов. Например, использовать команду:

```
java jade.Boot first: maslabs.lab2.BehAgent.java second: maslabs.lab2.BehAgent.java
```

3. Содержание отчета

3.1. Цель работы.

3.2. Определение модели поведения в JADE. Для чего необходима модель поведения?

3.3. Виды моделей поведения в JADE. Иерархия классов с объяснением назначения каждой модели поведения.

3.4. Исходный подробно документированный текст программы.

3.5. Протокол работы агента. Если агент уходит в бесконечный цикл, то приводить протокол необходимо от начала работы программы и до момента когда зацикленное поведение совершит 3-5 выводов сообщений на экран.

3.6. Изменённый подробно документированный исходный текст программы.

3.7. Протокол работы изменённого агента.

3.8. Выводы.

4. Вопросы для самопроверки

4.1. Какие основные возможности предоставляет базовый класс Agent?

4.2. Что понимается под *поведением* агента?

4.3. Для чего необходимы модели поведения?

4.4. Перечислите и охарактеризуйте состояния жизненного цикла агента.

4.5. Какие действия выполняются при запуске нового агента на платформе?

4.6. Для чего используется метод setup() класса Agent?

4.7. Что такое модели поведения в JADE? Для чего необходимы модели поведения?

4.8. Как создать и добавить агенту новое поведение? Как используются методы action() и done() объекта *поведение*?

4.9. Как добавить поведению дочернее поведение? Как используются методы onStart() и onEnd() объекта *поведение*?

4.10. Как можно разветвлять поведение агента?

4.11. Какие есть встроенные модели поведения в JADE?

4.12. Как работает планировщик заданий в JADE?