Мультиагентно-орієнтоване програмування

3-й рівень навчання, осінь 2021

- Доц. Баклан І.В.
- Email: iaa@ukr.net
- Web: baklaniv.at.ua

Лекція 2

Введення до платформи JaCaMo

У цій лекції ми представляємо JaCaMo, особливу платформу, прийняту в нашій дисципліні для практичного багатоагентного орієнтованого програмування. Ця платформа підтримує практичне програмування на основі абстракцій, представлених у попередньому розділі: програмування організованих агентів, розташованих у спільному середовищі. JaCaMo побудований поверх трьох існуючих платформ, які розроблялися роками (Boissier et al. 2013, 2019), а саме Jason (Bordini et al. 2007) для агентів програмування, CArtAgO (Ricci et al. 2009) для середовищ програмування та Moise (Hübner et al. 2007) для організацій з програмування.

У цій лекції класичний приклад Hello-World має версію з кількома агентами. Ми починаємо з найпростішої програми, яку ми можемо написати на JaCaMo, і вдосконалюємо її, щоб поступово показувати деякі найважливіші аспекти мови програмування та самої платформи.

Вказівки щодо створення, редагування та запуску додатків JaCaMo можна знайти у лабораторній роботі №1.

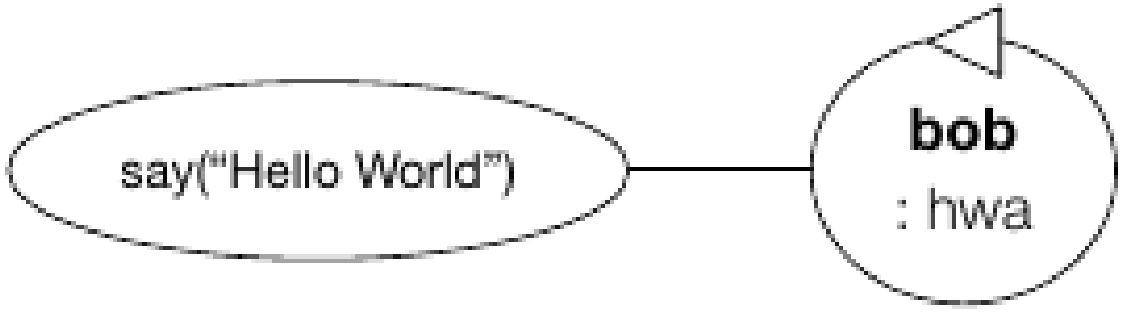
Одноагентний Hello-World

Ми починаємо з системи, яка має єдиного і дуже простого агента, який просто роздруковує повідомлення, використовуючи наступний план, написаний на Jason (і зберігається у файлі під назвою **`hwa.asl`**):

```
+!say(M) <- .print(M).
```

Цей план можна читати як "коли я маю мету! Скажіть (**`M`**), досягніть її, надрукувавши значення змінної **`M`**" (**`M`** - це змінна, оскільки вона починається з великої літери).

Для запуску агента JaCaMo використовує файли програм (імена яких закінчуються на **`.jcm`**). У нашому прикладі файлом програми є **`sag_hw.jcm`**, в якому ми даємо ім'я агенту (**`bob`**) та початкову мету **`(скажімо ("Hello World"))`**. Вміст цього файлу, зображеного графічно на малюнку 2.1, виглядає наступним чином:



Мал. 2.1 Конфігурація Hello-World для одного агента.

```
mas sag_hw { // the MAS is identified by sag_hw

agent bob: hwa.asl{// initial plans for bob are in
hwa.asl

goals: say("Hello World") // initial goal for bob

}

}
```

Результат виконання маємо такий

```
JaCaMo Http Server running on http://192.168.0.15:3272

[bob] Hello World
```

Для кращого розуміння результатів виконуються такі кроки у виконанні файлу програми (`.jcm`):

1. Агент на ім’я **`bob`** створюється з початковими переконаннями, цілями та планами, як це визначено зі змісту файлу з назвою **`hwa.asl`**.

2. Мета **`say ("Hello World")`** делегується **`bob`**, створюючи подію

```
+! say ("Hello World").
```

3. План у файлі **`hwa.asl`**, як показано раніше, запускається і використовується для обробки цієї події.

4. Виконання плану дає результат `[bob] Hello World` як результат виконання внутрішньої дії `.print`.

5. Агент продовжує працювати, але не має нічого робити, оскільки він є єдиним агентом у системі і сам не генерував жодних подальших цілей або змін у середовищі, які могли б привести його до подальших дій.

6. Як показано у результатах виконання, існує URL-адреса для перевірки поточного стану агентів (що включає їх переконання, наміри та плани), і згодом ми побачимо, що те саме стосується середовища та організацій.

Мультиагентний Hello-World

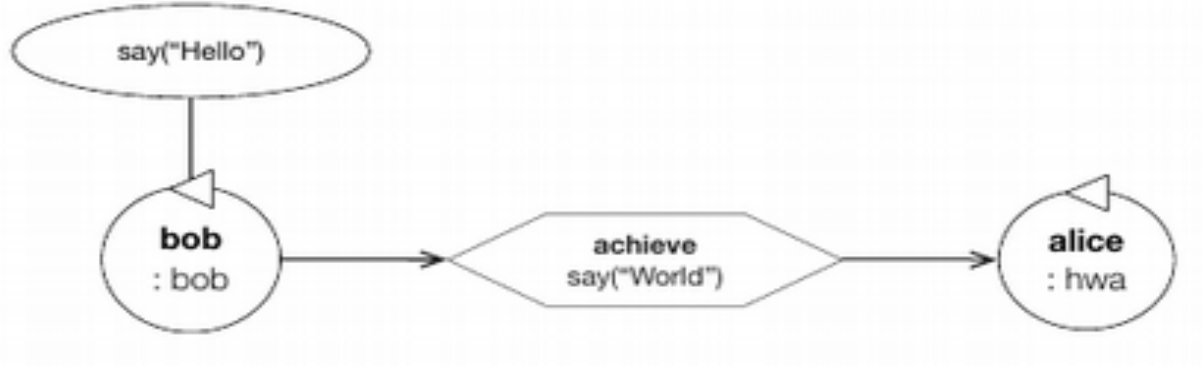
Тепер у нас є два агенти - Боб і Аліса. Агент Боб друкує "Hello", а Аліса - "World". Для того, щоб створити обидва агента з одного коду (як і в попередньому прикладі, що має лише один план), ми можемо використовувати такий файл програми:

```
mas mag_hw {
      agent bob: hwa.asl {
            goals: say("Hello")
      }
      agent alice: hwa.asl {
            goals: say("World")
      }
}
```

Однак результат виконання може бути таким:

```
[alice] World
[bob] Hello
```

Агенти працюють одночасно і асинхронно переслідують свої цілі, і тому така початкова реалізація не може гарантувати порядок надрукованих повідомлень. Потрібна деяка координація, щоб боб друкував першим, а потім Аліса. Ми можемо вирішити проблему, коли Боб надішле повідомлення Алісі, як тільки його повідомлення буде надруковане (див. Малюнок 2.2).



2.2 Координація комунікацій

Нова програма виглядає наступним чином (буде включена у файл з назвою bob.asl):

```
+!say(M) <- .print(M);
.send(alice,achieve,say("World")).
```

Надсилаючи повідомлення про досягнення Alicb, Боб делегує Алісі ціль `say("World")`. Вона використовує план `+! say (M) <- .print (M)`. для досягнення мети, як і раніше.

Оскільки мета Аліси тепер походить від Боба, а не від ініціалізації системи, файл програми потрібно змінити таким чином:

```
mas mag_hw {
agent bob { // file bob.asl is used
goals: say("Hello")
}
agent alice: hwa.asl
}
```

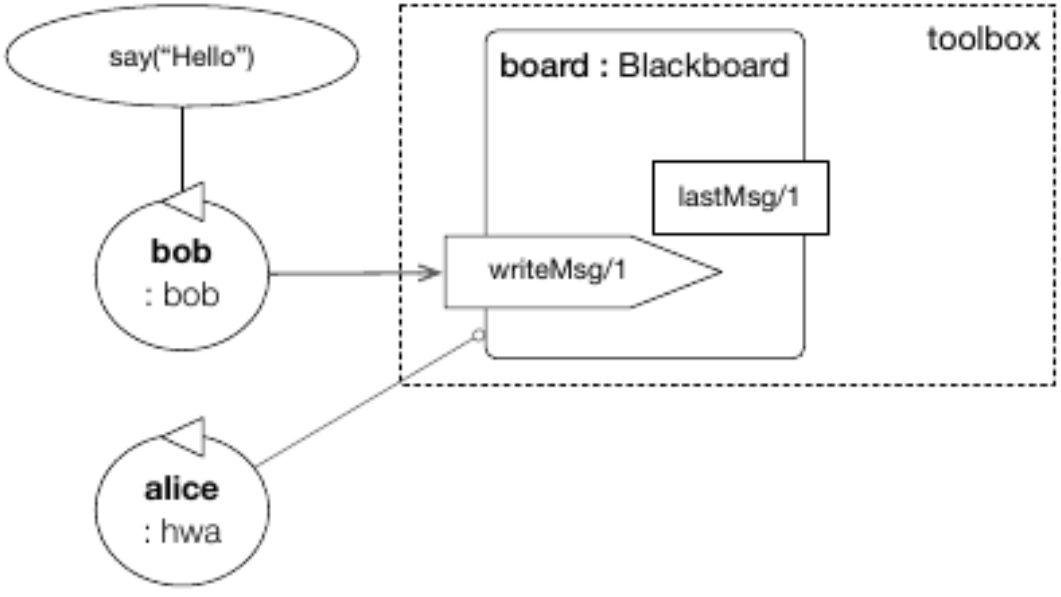
Середовище Hello-World

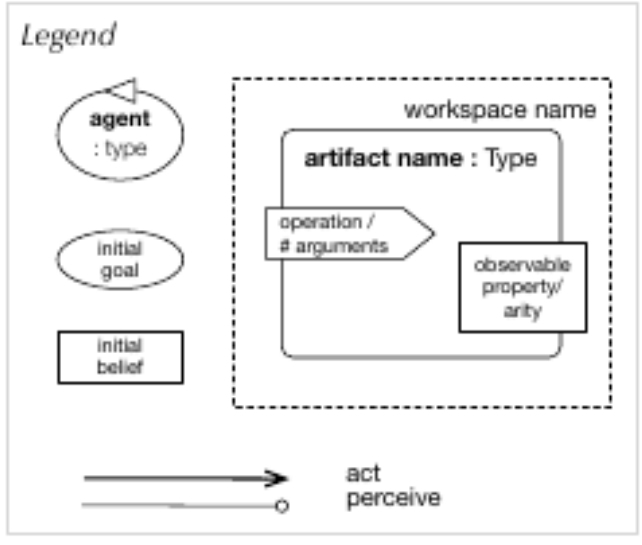
Приклад тепер розглядає середовище з артефактом дошки як дошку, яку агенти можуть використовувати для написання повідомлень та сприйняття повідомлень, написаних на ньому. У цій версії прикладу Hello-World Боб пише повідомлення “Hello” на дошці; і Аліса, яка спостерігає за дошкою, пише повідомлення «World», як тільки вона переконається, що повідомлення «Hello» написано.

Середовища структуровані у робочі області; всі агенти в робочій області мають спільний доступ до всіх екземплярів артефактів у цій робочій області. У файлі програми ми можемо вказати початковий набір артефактів та робочих областей, які слід створити під час створення MAS. У цьому випадку файл `sit_hw.jcm` має такий вигляд:

```
mas sit_hw {
  agent bob {
    join: room  // bob joins workspace toolbox
    goals: say("Hello")
  }
  agent alice {
    join: room // alice also joins workspace toolbox
    focus: room.board  // and focus on artifact board
  }
  workspace room {  // creates the workspace toolbox
    artifact board: tools.Blackboard // with artifact
board
  }
}
```

Початкова конфігурація включає робочу область під назвою room, де розміщено артефакт дошки з інструментами типу. Чорна дошка (див. малюнок 2.3). Обидва агенти приєднуються до кімнати робочої області під час ініціалізації, щоб отримати доступ до артефакту плати. Агент Аліса зосереджується на (тобто спостерігає) артефакті. Необхідно зосередитися на тому, щоб агент Аліса була уважною до змін у спостережуваних властивостях цього артефакту: коли щось написано наступного разу, коли Аліса відчує навколишнє середовище, переконання, що відповідає властивості спостережуваного артефакту, буде автоматично створено, і вона зможе реагувати на це.





2.3 Координація з використанням середовища.

Артефакти реалізовані на Java. Вихідний код (у файлі **Blackboard.java**) простого артефакту дошки виглядає наступним чином:

```
package tools;
import cartago.*;
public class Blackboard extends Artifact {
  void init() {
    defineObsProperty("lastMsg","");
  }
  @OPERATION void writeMsg(String msg) {
    System.out.println("[BLACKBOARD] " + msg);
    getObsProperty("lastMsg").updateValue(msg);
  }
}
```

Класи Java використовуються як шаблони для визначення артефактів, з використанням коментованих методів для визначення операцій з артефактами та заздалегідь визначених методів, успадкованих **API** **Artifact** для роботи з властивостями, що спостерігаються, та іншими механізмами артефактів.

Вихідним кодом для bob в цьому випадку стає

```
+!say(M) <- writeMsg(M).
{ include("$jacamoJar/templates/common-cartago.asl") }
```

Тобто агент використовує дію **writeMsg**, надану артефактом, для запису повідомлення на дошці. Інструкція включення завантажує деякі корисні плани в бібліотеку планів bob.

Вихідний код Alice – це

```
+lastMsg("Hello") <- writeMsg("World!").
{ include("$jacamoJar/templates/common-cartago.asl") }
```

Агент (який спостерігає за дошкою) пише повідомлення "`World`", як тільки він переконається, що останнє повідомлення, написане на дошці (зроблене для спостереження за допомогою властивості `lastMsg`) – "`Hello`".

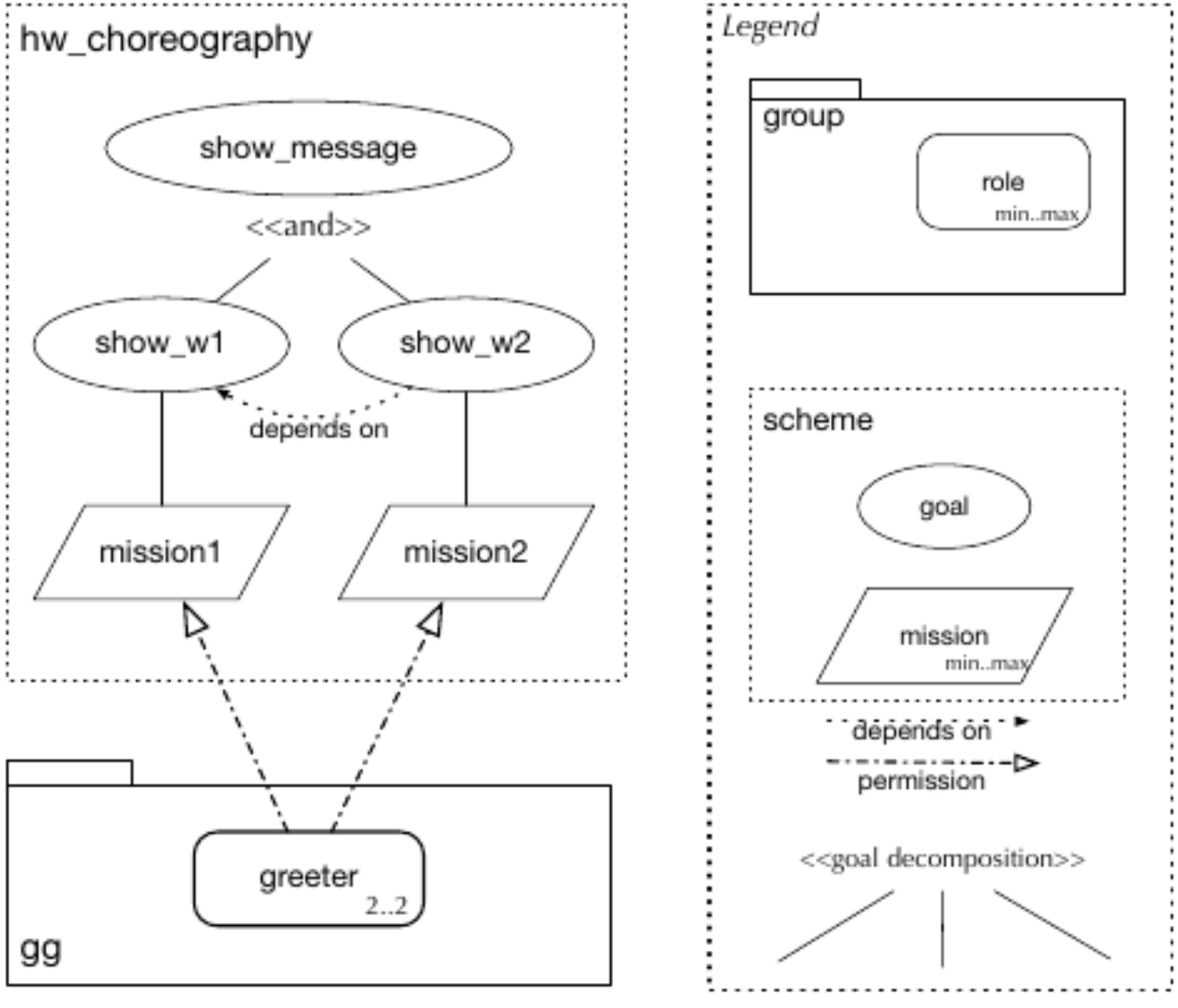
```
[alice] joined workspace room
[alice] focusing on artifact board (at workspace room)
using namespace default
[bob] joined workspace room
[BLACKBOARD] Hello
[BLACKBOARD] World!
```

Виконання файлу програми дає результат, подібний до попередніх, за винятком того, що зараз це артефакт дошки, що роздруковує повідомлення, і ніякого зв'язку між Бобом та Алісою не потрібно.

Організація Hello-World

Тепер ми організовуємо набір агентів для створення повідомлення `"Hello World"`. Як було представлено в попередньому розділі, організація може бути використана для регулювання та координації діяльності агентів. Хоча приклад простий, використання організації полегшує зміну певної схеми координації та регулювання. У нашому прикладі шаблон координації використовується для досягнення мети `show_message`, яка повинна бути досягнута спільною роботою двох агентів і, таким чином, є спільною метою. Щоб відрізнити таку мету від мети агента, ми називаємо її *організаційною метою*.

Щоб відрізнити таку мету від мети агента, ми називаємо її організаційною метою. Ми використовуємо соціальну схему для програмування того, як організаційна мета `show_message` розкладається на підцілі, які призначаються агентам (як показано на малюнку 2.4). Для декомпозиції мета `show_message` має одну підціль для кожного слова повідомлення. Для їх призначення агентам ми створюємо місії, в даному випадку по одній для кожної підцілі. Для того, щоб брати участь у виконанні схеми, агенти повинні взяти участь у місії та досягти відповідних цілей цієї місії. Здійснення до місії - це форма обіцянки групі агентів, які спільно працюють за схемою: «Я обіцяю, що, коли буде потрібно, я виконаю свою частину завдання». Коли агенти виконують усі завдання, схему можна виконати з гарантією того, що, принаймні в принципі, у нас достатньо агентів для роботи над усіма необхідними підцілями.



2.4 Координація за організацією: специфікація Hello-World.

Цей приклад організації також визначає єдину роль, яку будуть грати всі агенти: роль привітання відіграється у типі групи, ідентифікованій `gg` (для «привітальної групи»). Агенти, які виконують цю роль (і тільки вони), мають право виконувати завдання місії.

Реалізація цієї організації написана в XML так:

```
1   <?xml version="1.0" encoding="UTF-8"?>
2
3   <organisational-specification
4     id="hello_world"
5     os-version="0.8"
6
7     xmlns='http://moise.sourceforge.net/os'
8     xmlns:xsi='http://www.w3.org/2001/XMLSchema-instance'
9     xsi:schemaLocation='http://moise.sourceforge.net/os
10                        http://moise.sourceforge.net/xml/os.xsd' >
11
12  <structural-specification>
13    <group-specification id="gg">
14      <roles>
15        <role id="greeter" max="2"/>
16      </roles>
17    </group-specification>
18  </structural-specification>
19
```

```
<functional-specification>
  <scheme id="hw_choreography">
    <goal id="show_message">
      <plan operator="sequence">
        <goal id="show_w1"/>
        <goal id="show_w2"/>
      </plan>
    </goal>

    <mission id="mission1"  min="1" max="1"> <goal id="show_w1"/> </mission>
    <mission id="mission2"  min="1" max="1"> <goal id="show_w2"/> </mission>
  </scheme>
</functional-specification>

<normative-specification>
  <norm id="norm1"  type="permission" role="greeter" mission="mission1"/>
  <norm id="norm2"  type="permission" role="greeter" mission="mission2"/>
</normative-specification>

</organisational-specification>
```

Агенти відіграють роль привітання та беруть участь у місіях, щоб показати свої слова (боб показує «Hello», а Аліса - «World»). У кожного агента своя місія/мета/- слово, яке слід показати. Як показано на малюнку 2.4, привітальнику дозволено виконувати будь -яку місію, але ми не хочемо, щоб усі агенти брали участь у всіх місіях, які вони можуть. Щоб вирішити це, кожен агент має віру в місію, яку він повинен взяти на себе. Ці переконання є `my_mission` (місія1) для боба та `my_mission` (місія2) для Аліси.

Рішення взяти участь у місії реалізується за таким планом:

```
1  // when the organization gives me permission to
2  // commit to a mission M in scheme S,
3  // do that if it matches the belief my_mission
4  +permission(A,_,committed(A,M,S),_)
5     :  .my_name(A) &  // the permission is for me
6        my_mission(M)  // my mission is M
7     <- commitMission(M).
```

Символ + у рядку 4 означає "у разі віри ..."; код після `:` це умови щодо того, що агент вважає поточною ситуацією, які необхідні для використання плану; а код після `<--` це «дії» (наприклад, дії, які необхідно виконати, і цілі, яких необхідно досягти). Таким чином, цей план ініціюється додаванням переконання, що агент `A` має дозвіл на виконання місії `M` у схемі `S`. Якщо значення змінної `M` в переконанні агента `my_mission(M)` відповідає дозволеній місії `M`, план застосовується для події та агент виконує дії, спрямовані на виконання місії `M`.

Листові цілі соціальної схеми повинні бути досягнуті агентами, і тому вони мають для цього плани:

```
9   // when I have goal show_w1, create subgoal say(...)
10  +!show_w1  <- !say("Hello").
11  +!show_w2  <- !say("World").

12

13  +!say(M) <- writeMsg(M).
```

Символи `+!` у рядку 10 можна прочитати як "у разі досягнення нової мети ...". Код `!say (...)` в тому ж рядку створює нову підціль. Щодо віри в дозвіл, цілі `show_w` ... виходять від організації. Організація інформує агентів про цілі, які вони мають переслідувати, враховуючи поточний стан виконання схеми та зобов'язання агента. У цьому прикладі всі агенти, які беруть участь в організації, мають плани щодо всіх цілей `show_w`; агенти володіють ноу-хау, щоб показати обидва слова, і те, яке слово вони показують, залежить від місії, яку вони взяли на себе.

Коротко кажучи, агенти мають плани реагувати на події, вироблені організацією (нові дозволи та нові цілі), і їм не потрібно чітко координувати між собою за допомогою спілкування; тобто Бобу більше не потрібно надсилати повідомлення Алісі. Для підтримки координації також не потрібне середовище.

Файл програми для цієї реалізації **`Hello-World`** виглядає наступним чином:

```
1  mas hello_world {
2    agent bob : hwa.asl {
3      focus: room.board
4      roles: greeter in ghw   // initial role for bob
5      beliefs: my_mission(mission1) // initial belief
6    }
7    agent alice : hwa.asl {
8      focus: room.board
9      roles: greeter in ghw
10     beliefs: my_mission(mission2)
11   }
12   workspace room {
13     artifact board : tools.Blackboard
14   }
15   organisation greeting : org1.xml {
16     group ghw : gg {
17       responsible-for: shw
18     }
19     scheme shw : hw_choreography
20   }
21 }
```

Як і раніше, у цьому файлі є записи для агентів та робочих областей, але тепер додано організаційний блок. У рядку 19 організаційна сутність створюється на основі файлу XML, що описує тип груп і схем, доступних в організації. Одна сутність групи створюється у рядку 20 (ідентифіковано **ghw**), а одна сутність схеми створюється у рядку 23 (ідентифікується **shw**). У рядку 21 зазначено, що група **ghw** надає агентів для виконання схеми **shw**. Рядки 5 і 11 відводять роль, яка є більш привабливою для наших агентів у групі **ghw**. Рядки 6 та 12 додають переконання в агентах щодо місій, які вони повинні виконувати.

Виконання файлу програми (`.jcm`) відбувається таким чином:

1. Створюється кімната робочого простору та артефактна дошка.
2. Група **`ghw`** та схема **`shw`** створюються та пов'язуються (відповідальні за).
3. Створюються агенти Боб і Аліса, які приєднуються до кімнати робочого простору.
4. Агентам відводиться роль привітання.
5. Граючи цю роль, вони починають вірити

```
permission(bob,_,committed(bob,mission1,shw),_)
permission(bob,_,committed(bob,mission2,shw),_)
permission(alice,_,committed(alice,mission1,shw),_)
permission(alice,_,committed(alice,mission2,shw),_).
```

6. Додавання цих переконань ініціює їх перший план, і вони виконують свої місії. Зображення загального стану системи

показано на малюнку 2.5.

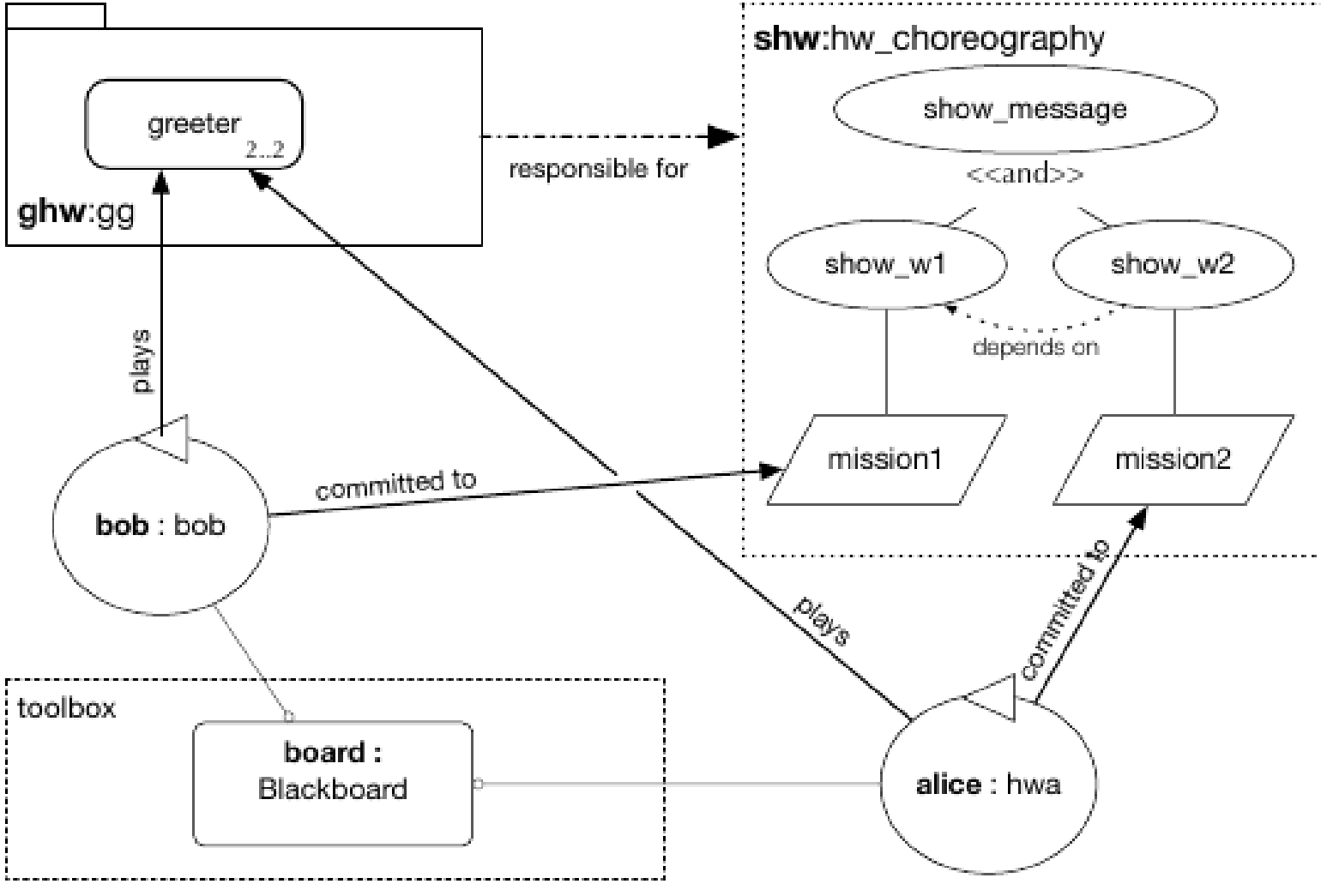
7. Коли агенти виконують свої місії, схема **shw** має достатньо агентів для її виконання, і мета **show_w1** може бути остаточно досягнута.

8. Агент Боб, будучи відданим **місії1**, повідомляється, що мета **show_w1** може бути прийнята, і він це робить; на дошці записується повідомлення **«Hello»**.

9. Потім агенту Алісі повідомляють, щоб він досяг **show_w2**, і він робить це; на дошці написано повідомлення «**World**».

10. Схема закінчена.

Ми можемо помітити узгоджену поведінку: слова завжди показуються у правильному порядку. Більш того, координація реалізується в організації, а не в агентах (в агентах немає коду для координації їх окремих дій, щоб загальна поведінка системи була такою, як очікується).



2.5 Суб’єкти організації Hello-World.

Джерела до лекції 2

Boissier, Olivier, Rafael Bordini, Jomi Fred Hübner, Alessandro Ricci, and Andrea Santi. 2013. Multi-agent oriented programming with JaCaMo. Science of Computer Programming 78 (6): 747–761. https://doi.org/10.1016/j.scico.2011.10.004 .

Boissier, Olivier, Rafael H. Bordini, Jomi F. Hübner, and Alessandro Ricci. 2019. Dimensions in programming multi-agent systems. The Knowledge Engineering Review 34. https://doi.org/10.1017/S026988891800005X .

Bordini, Rafael H., Jomi Fred Hübner, and Michael Wooldridge. 2007. Programming multi-agent systems in AgentSpeak using Jason. John Wiley & Sons.

Ricci, Alessandro, Michele Piunti, Mirko Viroli, and Andrea Omicini. 2009. Multi-agent programming: Languages, tools and applications, eds. Amal El Fallah Seghrouchni, Jürgen Dix, Mehdi Dastani, and Rafael Bordini, 259–288. Springer. https://doi.org/10.1007/978-0-387-89299-3_8 .

Hübner, Jomi Fred, Jaime Simão Sichman, and Olivier Boissier. 2007. Developing organised multiagent systems using the MOISE+ model: programming issues at the system and agent levels. International Journal of Agent-Oriented Software Engineering 1 (3-4): 370–395.

Наступна лекція буде присвячена детальному програмуванню виміру агентів на платформі JaCaMo.