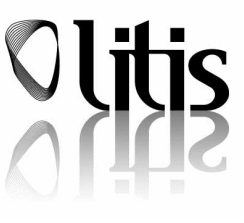

Modélisation, Interaction et Usage

Stage de Spécialité

Intégration de Composants à un Agent Conversationnel Animé

Auteur :
William Boisseleau

Maîtres de stage :
Alexandre Pauchet
Ovidiu Şerban

10 septembre 2013

Table des matières

Remerciements **3**

Introduction **4**

1 Environnement de stage et contexte **5**
1.1 Un laboratoire de recherche : le LITIS . . . 5
1.2 L'équipe Modélisation, Interaction et Usage . . . 5
1.3 Le projet ACAMODIA . . . 6
1.4 Objectifs du stage . . . 7

2 Le système interactif AgentSlang **8**
2.1 MyBlock : communication par composants . . . 8
2.1.1 Les données . . . 8
2.1.2 Les composants . . . 9
2.1.3 Scheduler et notions de Service/Client . . . 10
2.1.4 Exemple type et analyse de code . . . 10
2.2 La plateforme AgentSlang . . . 14
2.2.1 Composants système (MyBlock) . . . 14
2.2.2 Composants entrée/sortie . . . 14
2.2.3 Composants de traitement de langage . . . 16
2.2.4 Composant de traitement de rétroaction affective . . . 16
2.2.5 Emplacements des composants et topics . . . 16
2.2.6 Les paramètres additionnels . . . 17
2.3 Le module Syn !bad . . . 17
2.4 Un exemple d'utilisation des composants AgentSlang et Syn !Bad . . . 19
2.4.1 Présentation et objectifs . . . 19
2.4.2 Modèle et diagrammes . . . 19
2.4.3 Test du Chatbot . . . 22

3 Intégration d'AgentSlang à ACAMODIA **24**
3.1 Cadre du projet . . . 24
3.1.1 Rappel du contexte . . . 24
3.1.2 Objet et domaine d'application . . . 24
3.1.3 Cadre technique . . . 24
3.2 Description générale . . . 25
3.2.1 Spécifications fonctionnelles . . . 25
3.2.2 Cas d'utilisation . . . 25
3.2.3 Interface utilisateur . . . 26
3.2.4 Diagramme d'activité . . . 26
3.2.5 Modèle par composants proposé . . . 27
3.2.6 Perspectives . . . 29
3.3 Conception . . . 30
3.3.1 Organisation du projet . . . 30
3.3.2 Paquetage data . . . 31
3.3.3 Paquetage inout . . . 33
3.3.4 Paquetage out . . . 33
3.3.5 Paquetage process . . . 34
3.3.6 Tests unitaires . . . 34
3.4 Saisie des données . . . 35
3.4.1 Initialisation des fichiers de données, version 1 . . . 35
3.4.2 Initialisation des fichiers de données, version 2 . . . 35
3.5 Produit et améliorations . . . 36

Conclusion **37**

Table des figures **38**

Liste des tableaux **38**

Listings **38**

Références **39**

Je tiens tout d'abord à remercier l'équipe pédagogique de l'INSA de Rouen ainsi que l'équipe Modélisation, Interaction et Usage pour m'avoir permis d'effectuer ce stage au LITIS.

Je tiens à remercier M. Laurent Vercouter, coordinateur de l'équipe MIU, pour m'avoir permis d'intégrer l'équipe.

Je tiens également à témoigner ma reconnaissance envers M. Alexandre Pauchet et M. Ovidiu Şerban pour le suivi, l'aide en terme d'organisation, de modélisation et de développement, le temps qu'ils ont pu me consacrer, les conseils apportés et plus globalement l'expérience très enrichissante qu'ils m'ont fait vivre durant ces deux mois de stage.

Enfin, je souhaite remercier M. Jean-Philippe Kotowicz pour m'avoir mis en contact avec le groupe MIU et permis d'effectuer ce stage au sein du laboratoire.

Notre société est aujourd'hui de plus en plus dépendante des machines physiques et immatérielles dans de nombreux domaines, tels que l'industrie, la recherche, et même les services. Par extension, l'évolution technologique et humaine est étroitement liée aux algorithmes faisant tourner ces-dites machines, et leur étude approfondie depuis seulement une cinquantaine d'années a permis de grandes avancées. Il ne s'agit pas d'améliorer seulement les composants internes, propres à ces algorithmes, mais également d'implémenter une interaction homme/machine la plus optimale, afin d'en parfaire les résultats. Ainsi, c'est sous le thème des interactions que le stage de spécialité ici rapporté s'est déroulé.

Une des premières intelligence informatique programmée informatiquement fut ELIZA, écrit au MIT entre 1964 et 1966. Elle implémente de façon relativement précoce un traitement automatique du langage naturel, en utilisant un filtrage par motif -essentiellement des séquences de mots. C'est historiquement le premier agent conversationnel recensé (*Chatbot* en anglais), et dont les principes sont encore de nos jours les bases de nombreux travaux.

De ce fait et dans un tel contexte, ce rapport porte sur l'étude d'une plateforme interactive distribuée appelée AgentSlang et développée dans le cadre d'une thèse. La plateforme a été conçue pour reconnaître des motifs et des émotions humaines, permettant une post-gestion logicielle davantage accomplie. De plus, un des objectifs du stage est d'intégrer ce modèle à un autre projet du nom d'ACAMODIA. Ce dernier est un logiciel assisté de lecture d'histoire à un enfant, dans lequel il n'y a pas d'intelligence artificielle - l'expérience étant de type Magicien d'Oz. Le finalité de cette intégration est d'obtenir un système complètement autonome.

Pour ce faire, nous décrirons dans une première partie l'environnement de stage et le contexte du sujet de stage proposé. Dans une seconde partie, nous expliciterons les composantes du système interactif AgentSlang et leur fonctionnement global. Enfin, et avant de conclure, nous développerons en détail notre proposition de modèle d'intégration d'AgentSlang à ACAMODIA, de la modélisation jusqu'au tests finaux.

1 Environnement de stage et contexte

1.1 Un laboratoire de recherche : le LITIS

Le Laboratoire d'Informatique, de Traitement de l'Information et des Systèmes (LITIS) est un laboratoire de recherche public en Sciences et Technologies de l'Information et de la Communication (STIC) fondé en 2006 et réparti sur trois établissements d'enseignement supérieur de Haute-Normandie : l'Université de Rouen, l'Université du Havre et l'Institut National des Sciences Appliquées (INSA) de Rouen. C'est une unité de recherche EA4108 labellisée par le Ministère de l'Enseignement Supérieur et de la Recherche dont le directeur est Thierry Paquet.

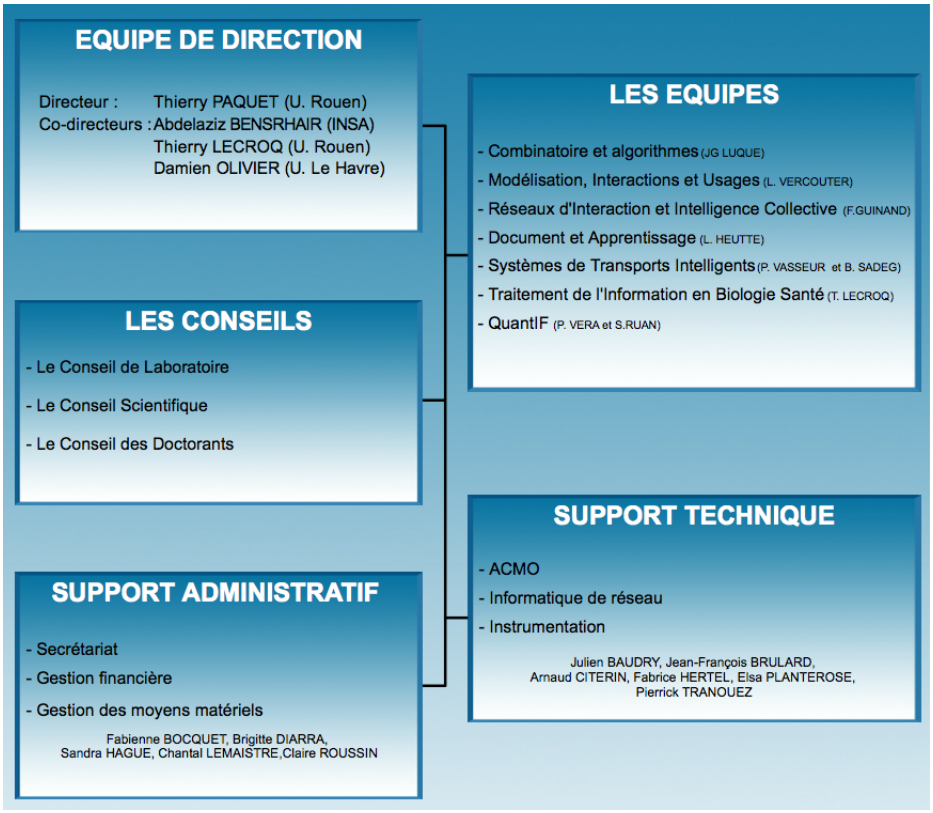

FIGURE 1 – Organigramme du LITIS

Le LITIS regroupe de nombreuses disciplines étant abordées de façon théorique ou pratique, que ce soit dans des domaines liés à l'informatique, reconnaissance des formes, des mathématiques et du traitement du signal et images de la médecine.

Il est structuré autour de trois axes regroupant sept équipes de recherche (cf. Figure 1) :
- l'axe "Combinatoire et algorithmes",
- l'axe "Traitement des masses de données",
  - Équipe "Documents et Apprentissages",
  - Équipe "Traitement de l'information en Biologie Santé ",
  - Équipe "Quantif",
  - Équipe "Systèmes de transport intelligents",
- l'axe "Interaction et systèmes complexes",
  - Équipe "Modélisation, Interaction et Usage",
  - Équipe "Réseaux d'Interactions et Intelligence Collective".

1.2 L'équipe Modélisation, Interaction et Usage

L'équipe Modélisation, Interaction et Usage (MIU) est composée de 11 chercheurs, de 10 doctorants et de 9 associés (juin 2013). Elle concentre ses recherches autour de l'accès personnalisé à

l'information. Dans un contexte de besoin de gestion de *Big Data*, il est important de réfléchir à des plate-formes spécifiques permettant la distribution et la gestion de ses données, relativement à un utilisateur lambda.

Suivant les situations, une personne pourra en effet accéder ou non à des informations. Elle peut, par exemple, faire partie d'un groupe n'autorisant qu'un échange interne des données, et de plus avoir accès à des données publiques. A l'intérieur même de ce groupe, elle peut ne pas avoir accès à certaines données suivant son statut. Dans le même sens, elle ne pourra pas non plus accéder à des informations d'autres groupes de recherche. Dans de tels cas, il est nécessaire de structurer un accès personnalisé à l'information, un ensemble de solutions caractérisées par des règles applicables à chaque individu.

De ce fait, l'équipe divise sa recherche sur plusieurs niveaux :
- La **modélisation** de structures permettant le stockage et la localisation des données, tels que des travaux d'indexation automatique, ou d'évaluation de la qualité des informations ;
- Les **interactions** entre utilisateurs, machines, et utilisateur-machine. Ceci peut par exemple concerner de la formation via des environnements virtuels ou des agents conversationnels (comme abordés dans ce stage) épaulant voire remplaçant l'homme suivant les situations ;
- Les **usages**, dont l'étude permet de cerner les besoins des utilisateurs et en implémenter des modèles y répondant. Il s'agit par exemple d'exploiter de manière automatique ou manuelle des traces des utilisateurs relatives à leurs tâches, dont la finalité serait d'en proposer un outil ayant une efficacité optimale.

Ainsi, les domaines scientifiques abordés par l'équipe MIU sont les systèmes multi-agents et agents autonomes et l'extraction et la gestion des connaissances. Les domaines d'applications s'inscrivent dans l'e-learning, l'e-teaching, les communautés virtuelles et l'intelligence ambiante.

1.3 Le projet ACAMODIA

Le stage ici rapporté s'inscrit dans le cadre du projet ACAMODIA lancé en septembre 2011. Il fut développé par le LITIS en partenariat avec le Groupe de Recherche en Informatique, Image, Automatique et Instrumentation de Caen (GREYC), le laboratoire Psychologie des Actions Langagières et Motrices (PALM) et le laboratoire de Psychologie et Neurosciences de la Cognition et de l'Affectivité (PSY-NCA). Il a été lancé grâce à un financement Projet Exploratoires Pluridisciplinaires (PEPS) des instituts INS2I et INHS. Le projet se situe dans la partie interaction du groupe de recherche MIU du LITIS, et a pour problématique l'adaptation et le perfectionnement des interactions Homme-Machine. Son objectif est de mettre en place une plate-forme d'analyse de dialogues, ainsi qu'un agent conversationnel affectif pour la narration d'histoires enfantines. Il s'agit également d'étudier les interactions entre l'enfant et l'agent durant la narration afin d'améliorer le système.

La partie LITIS de ce projet est développée suivant deux thèses dont celle d'Ovidiu Şerban et celle de Zacharie Ales, et s'est échelonnée ainsi :

1. La numérisation de corpus de dialogues lors de la narration Parents-Enfants. 120 dialogues, représentant 20h d'enregistrements et 9584 énoncés ont été retranscrits et annotés, suivant un codage relatif à la nature de l'énoncé (question, affirmation), sa référenciation (les personnes mises en œuvre), l'état mental (si l'énoncé est une marque d'émotion par exemple) et les justifications (par cause/conséquence, par opposition).
2. La mise en place d'une expérimentation de type magicien d'Oz, dans lequel les enfants interagissent avec un système informatique qu'ils croient autonome, qui est en réalité contrôlé par un humain, en l'occurrence un psychologue. L'expérimentation se déroule en deux parties. L'agent conversationnel est d'abord représenté par un avatar pour les 8 premières questions, puis par la vidéoconférence du psychologue contrôlant le système pour les 8 dernières.

3. La création d'une annotation automatique permettant notamment la détection d'émotion. Cette étude se base d'une part sur la classification d'Ekman, selon laquelle il est possible d'identifier certaines expressions d'une personne sur une image, émotions soit biologiques soit universelles, telles que la tristesse, la joie, la colère, la peur, le dégoût et la surprise. Une autre classification fut rendue possible grâce à l'analyse sémantique latente (LSA : Latent Sementic Analysis), qui est une technique d'analyse de relations entre un ensemble de documents (ici nos énoncés) et les termes qu'ils contiennent (les mots), produisant un ensemble de concepts. Cette analyse suppose que les mots qui sont proches de sens réapparaîtront avec un même contexte dans d'autres textes.
4. L'extraction des motifs dialogiques, permettant la construction de banques de motifs, via une heuristique de motifs dialogiques et de clustering de ces motifs.
5. La gestion du dialogue de l'agent. Les automates sont notamment utilisés pour générer des messages (le dialogue narratif, entre autres) et interpréter les messages reçus. Ceci peut être du texte, du son ou encore de la vidéo. Un composant d'écoute sensorielle a été utilisé pour ce dernier élément, à savoir le projet Semaine [2] qui permet l'interaction avec un humain via un personnage virtuel.

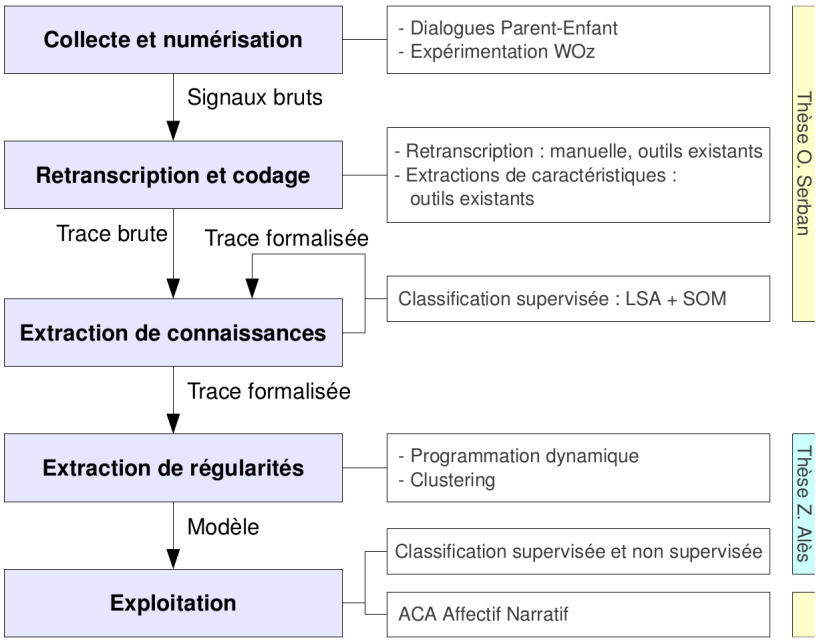

FIGURE 2 – Développement d'ACAMODIA au LITIS

1.4 Objectifs du stage

Le stage de spécialité ici rapporté a pour objectifs les points suivant :
- D'une part, apprécier et analyser les composants de la plateforme *AgentSlang* (ci-après explicitée), en comprendre son contenu et ses composants. Ceci inclue l'étude des modules *Syn !bad*, *MyBlock* et nécessite la compréhension de la communication entre composants utilisée pour ce projet.
- D'autre part, implémenter un agent conversationnel de type Chatbot utilisant des éléments du projet *AgentSlang*.
- Enfin, soumettre une proposition de modèle de Dialogue pour le projet ACAMODIA. Ce modèle a pour but de "remplacer le psychologue", en utilisant les éléments du projet *AgentSlang* et est développé en JAVA.

2 Le système interactif AgentSlang

AgentSlang est un projet conçu et développé en Java par Ovidiu Şerban dans le cadre de sa thèse [3] défendue en septembre 2013.

2.1 MyBlock : communication par composants

AgentSlang utilise des méthodes de communication spécifiques qu'il est nécessaire d'introduire avant d'expliciter son contenu. Le module *MyBlock* décrit des règles de communication génériques étant utilisées par *AgentSlang*.

2.1.1 Les données

MyBlock propose une implémentation objet pour la gestion des données entre composants. Au niveau des classes, lorsque l'on travaille directement sur le code du système, on considère les entrées comme des données de type chaîne de caractères (*String Data*). On peut avec ce *String* sortir les données sous forme de *Annotation Data*, ou encore sous forme de *Valence Data* (explicités ci-après). Ces 3 types de données sont ensuite caractérisées par un identifiant distinct (en binaire) permettant une optimisation des échanges de données (cf. figure 3).

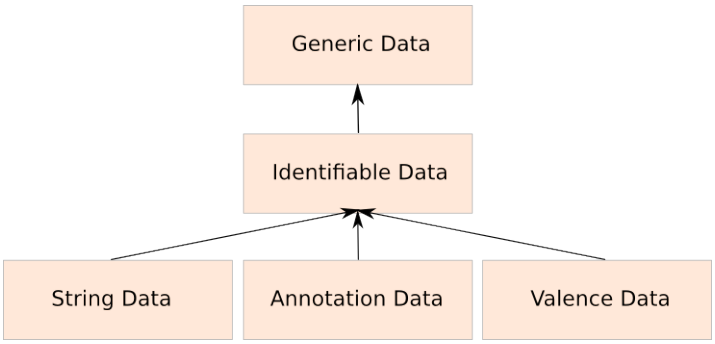

FIGURE 3 – Diagramme simplifié des objets Data de MyBlock

Les Annotations. Pour éviter l'utilisation d'un *String*, il est possible de travailler avec des annotations. Ceci consiste tout d'abord à séparer les éléments d'une chaîne de caractères suivant des règles précises. Deux méthodes sont utilisées pour la séparation de la chaîne en entrée. Premièrement, la séparation **text**, qui consiste à séparer la chaîne par mots ou groupes de mots. Une liste d'identifiants/groupes de mots (ou mot) est renvoyée : *list (index, token)*. Deuxièmement, une séparation par **phonèmes**, par l'algorithme *Double Metaphone* et qui renvoie de la même façon une *list(index, token)*. Enfin, pour chaque *token*, les composants sont annotés de différentes façons :

- **POS** (Part Of Speech), correspondant à une catégorisation des éléments suivant leur étiquette grammaticale (TAG), comme les verbes (VB), adjectifs (JJ), noms (NN), adverbes (RB).
- **NER** (Name Entity Recognition), la reconnaissance des entités nommées, qui consiste à rechercher un/des mot(s), ou un groupe de mots catégorisables dans des classes telles que noms de personnes, noms d'organisations/d'entreprises, noms de pays, quantités, valeurs, dates, etc. C'est typiquement ce que propose Wikipedia, avec son indexation de documents.
- **CHK** (Chunking), qui consiste à séparer des concepts en petits morceaux, rendant la lecture d'un même contenu à la fois plus rapide et plus facile.
- **SRL** (Semantic Role Labelling), qui permet de séparer des phrases par groupes de rôles sémantiques, tel que le verbe, l'agent principal, le complément, etc.
- **PSG** (Syntactic parsing), qui est une décomposition analytique identifiant des jetons significatifs, les analyse, puis en construit un arbre d'analyse.

Ces annotations sont effectués grâce au logiciel *Senna* [4].

Enfin, on considère les données de type *Valance Data* des informations liées à des détections affectives, telles que des comportements détectés par capture de texte, voix, et vidéo grâce au logiciel *Semaine* [2].

2.1.2 Les composants

FIGURE 4 – Composant défini dans MyBlock

On définit un composant comme une boite traitant un certain nombre de fonctions, boite explicitement déterminée par un hôte (*host*) et un port (cf. figure 4). Chaque composant autorise un nombre précis d'entrées (*inBound)* et de sorties (*outBound*), que l'on qualifiera de données ou informations. Il existe 4 types de composants :

- Le composant ***Source*** qui publie uniquement des informations via la méthode *publish*. La source peut toutefois recevoir des informations telles que des *topics*, des battements de cœur (*heartbeat*) [1], ou des messages système comme, par exemple, une entrée microphone. La méthode *act()* du composant source permet de lancer une action suivant un temps mort (*timeout*) pré-défini, et ce même s'il n'y a pas d'entrée activatrice. Ce processus est développé dans la section suivante.
- Le composant ***Sink*** est un composant recevant uniquement des informations, qui sont ensuite gérées via la méthode *handle(data)*.
- Le composant ***Mixed*** correspond à l'union entre un composant Source et Sink, sans la méthode *act()*.
- Le composant ***Living*** correspond à l'union entre un composant Source et Sink. Ce composant pourrait par exemple envoyer un message après une longue inactivité.

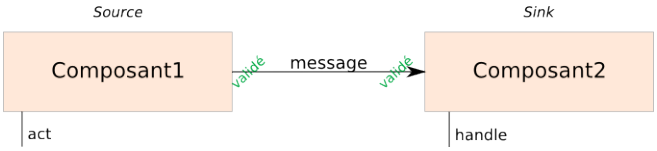

FIGURE 5 – Diffusion d'un message entre 2 composants

On comprend alors qu'il est possible d'implémenter nos composants de relation type clients-serveurs, qu'un serveur soit également un client, etc. Pour échanger des données entre deux composants, on utilise un canal sur lequel on peut publier des données, le *topic*. Il existe un topic interne qui est défini dans la méthode *publish* du composant et un topic externe qui est utilisé par le fichier de configuration *config.xml*. La liaison entre ces deux topics est effectuée par la méthode *publish*.

MyBlock gère automatiquement les embranchements de canaux, tel qu'un composant *Source* puisse envoyer un *topic* à deux composants *Sink* (au même titre que le *tee* en bash), mais aussi dans l'autre sens, de sorte que que deux composants *Source* puissent envoyer à un même composant *Sink* (*join* en bash).

1. On considère ici un Heartbeat comme une donnée envoyée de façon périodique par le Scheduler à ses abonnés

2.1.3 Scheduler et notions de Service/Client

Les composants ci-avant explicités ne s'exécutent pas d'eux-même ; ils demandent une entrée spécifique permettant de déclencher une action spécifique. Un composant *Sink* recevra par exemple un topic envoyé par un composant *Source* auquel il est abonné. Un composant source sera lui activé par un *heartbeat* envoyé par un ordonnanceur, le **Scheduler**. Le *Scheduler* envoie un *heartbeat* à tous les composants Source qui y sont abonnés, et ce à chaque fois qu'une période en millisecondes est terminée (le *timeout*). *MyBlock* permet de gérer un ou plusieurs *Scheduler* sur le même projet. Il peut effectivement être intéressant d'utiliser deux *schedulers* lorsque des composants se trouvent sur deux machines différentes pour éviter le phénomène de *Lag* dû à la distance et le chemin parcouru par le *heartbeat*, comme c'est le cas sur la figure 6.

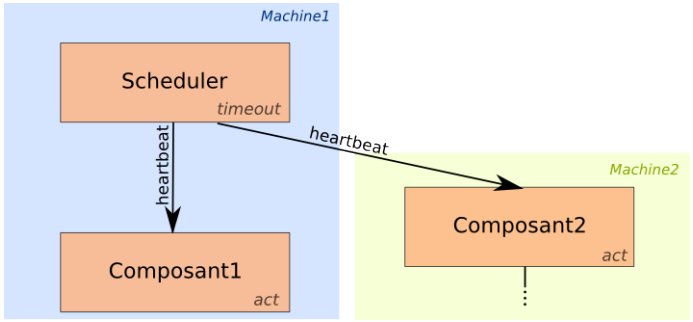

FIGURE 6 – Scheduler ayant 2 composants abonnés sur différentes machines

On considère un service comme une ligne d'information de base de données que l'on peut interroger à distance. Deux services permettant les échanges "serveur/client" sont définis dans *MyBlock* :

- Le **Computer Name Service** qui se charge de la résolution d'adresse. Les machines sont identifiées par des noms et ce service se charge de faire la correspondance entre leur *IP* et leur *hostname*.
- Le **Topic Service** qui se charge de la résolution des *topics* encodés, et qui met à jour une base de données de tous les *topics* tournants sur le réseau. En effet, un *topic* prend la forme d'un string sur le fichier de configuration. Il est ensuite encodé en un identifiant numérique, le rendant plus facile à manipuler.

2.1.4 Exemple type et analyse de code

On souhaite implémenter deux composants dont le premier "s'auto-abonne" et s'abonne à un deuxième composant. Le deuxième composant s'abonne également au premier. La figure ci-dessous représente cette modélisation.

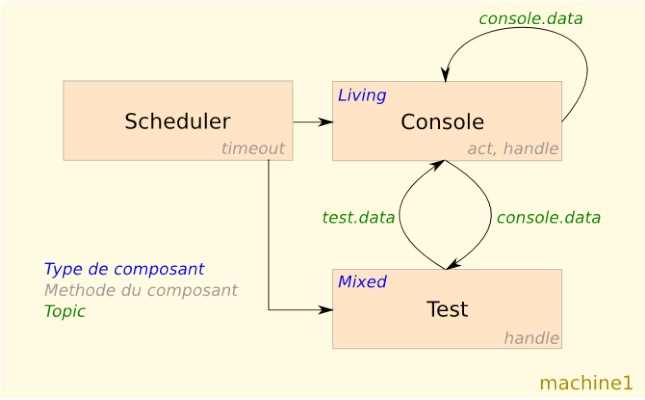

FIGURE 7 – Modélisation de l'exemple

Analysons tout d'abord le code source du **composant *Console*** (cf. Listing 1).

```
// Console.java
public class Console extends LivingComponent {
    private int counter;

    public Console(String port, ComponentConfig config) {
        super(port, config);
    }

    protected void setupComponent(ComponentConfig config) {
        this.counter = 0;
    }

    protected void handleData(GenericData message) {
        System.out.println("Printing on the Console! : "+message);
    }

    @Override
    public boolean act() {
        publishData("Console.data", new StringData(Integer.toString(this.counter
            ++)));
        return true;
    }

    public void defineReceivedData() {
        addInboundTypeChecker(StringData.class);
    }

    public void definePublishedData() {
        addOutboundTypeChecker("Console.data", StringData.class);

    }
}
```

Listing 1 – Console.java (exemple d'utilisation des composants)

Explicitons chacun des éléments.

- On constate tout d'abord que la classe *Console* a bien pour classe parente un *LivingComponent* (l.2) [2]
- La classe possède une variable d'instance entière, le compteur *counter* (l.3)
- Le constructeur est un appel du constructeur de la classe parente, avec le *port* et la *config* (l.6)
- La variable d'instance est initialisée dans la méthode *setupComponent*.
- La méthode *handleData* (l.13) définit les instructions qui seront effectuées une fois que des données valides (ou une seule) entrent dans le composant. Ici, on affiche simplement un message dans le terminal ainsi que le message inBound arrivé (dans notre cas de type String).
- La méthode *act* (l.18) est appelée à chaque fois qu'un *heartbeat* parvient au composant. A chaque *heartbeat*, on publie le topic interne identifié par la chaîne de caractères *Console.data*. Le topic en question est un string, comme le notifie le deuxième argument de la méthode *publish*. On retourne *VRAI* pour montrer qu'on est bien passé par la méthode *act()*.
- Enfin, on assigne les types de données que l'on peut recevoir en ajoutant des *inBound* dans la méthode *defineReceivedData* (l.24). On détermine de la même façon le type de données que l'on publie dans la méthode *definePublishedData*, en ajoutant aux *outBound* un topic (premier argument : *Console.data*) et le type de donnée que l'on envoie (deuxième argument : *StringData.class*).

2. (l.2) := ligne 2.

Analysons à présent le **composant *Test*** (cf. Listing 2).

```
// Test.java
public class Test extends MixedComponent {
    private int tCounter;

    public Test(String port, ComponentConfig config) {
        super(port, config);
    }

    protected void setupComponent(ComponentConfig config) {
        this.Tcounter = 0;
    }

    protected void handleData(GenericData message) {
        System.out.println("Printing received message on Test! : "+message);
    }

    protected void sendMessage(String message) {
        publishData("Test.data", new StringData(Integer.toString(this.tCounter++))
            );
    }

    public void defineReceivedData() {
        addInboundTypeChecker(StringData.class);
    }

    public void definePublishedData() {
        addOutboundTypeChecker("Test.data", StringData.class);
    }
}
```

Listing 2 – Test.java (exemple d'utilisation des composants)

On constate à nouveau que :

- La classe *Test* a pour classe parente un *MixedComponent* (l.2).
- On a choisi un entier *tCounter* comme variable d'instance (l.3).
- Le constructeur appelle le constructeur de la classe parente (l.6).
- La méthode *setupComponent* permet d'initialiser les variables d'instance, ici le compteur.
- Lorsqu'une donnée entre dans le composant, la méthode *handleData* est appelée, et traite un certain nombre de tâches (l.13). Dans notre cas, il s'agit simplement d'une sortie sur le terminal.
- Le composant *mixed* renvoie des données via la méthode *sendMessage* (l.17). Cette méthode publie le topic *Test.data*, qui est de type *StringData*.
- Enfin, on définit les types d'entrées et de sorties grâce aux méthodes *defineReceivedData* et *definePublishedData* (l.21 et l.25).

La communication entre ces composants est gérée via deux fichiers *XML*. Le premier ***cnsService.xml*** (cf. Listing 3) correspond au service *Computer Name Service* de résolution de nom.

```
<!-- cnsService.xml -->
<dns>
    <machine>machine1@localhost</machine>
</dns>
```

Listing 3 – cnsService.xml (exemple d'utilisation des composants)

La *machine1* est localisée sur le localhost, correspondant en *IPv4* à l'adresse *127.0.0.1*. Il est possible d'ajouter autant de machines que nécessaire dans ce fichier.

Le deuxième fichier ***config.xml*** (cf. Listing 4) décrit la façon dont est lancé le projet exemple par MyBlock, dont l'exécution du *LivingComponent* et du *MixedComponent*.

```
<!-- config.xml -->
<project>
  <profile name="profile1" hostname="machine1">
    <scheduler>
      <port>1222</port>
      <timeout>100</timeout>
    </scheduler>

    <services>
      <service name="org.ib.service.cns.CNService">
        <port>1221</port>
        <config>cnsService.xml</config>
      </service>
      <service name="org.ib.service.topic.TopicService">
        <port>1220</port>
      </service>
    </services>

    <clients>
      <client name="org.ib.service.cns.CNClient">
        <host>127.0.0.1</host>
        <port>1221</port>
      </client>
      <client name="org.ib.service.topic.TopicClient">
        <host>machine1</host>
        <port>1220</port>
      </client>
    </clients>

    <components>
      <component name="org.ib.gui.monitor.MonitorComponent">
        <port>1233</port>
        <scheduler>machine1:1222</scheduler>
        <subscribe>Connexion.Element.Console.debug@machine1:1234</subscribe>
        <subscribe>Connexion.Element.Console.heartbeat@machine1:1234</
            subscribe>
        <subscribe>Connexion.Element.Test.debug@machine1:1235</subscribe>
        <subscribe>Connexion.Element.Test.heartbeat@machine1:1235</subscribe
            >
      </component>

      <component name="Connexion.Element.Console">
        <port>1234</port>
        <scheduler>machine1:1222</scheduler>
        <publish>StringData.Text@Console.data</publish>
        <subscribe>StringData.Text@machine1:1234</subscribe>
        <subscribe>StringData.Text@machine1:1235</subscribe>
      </component>

      <component name="Connexion.Element.Test">
        <port>1235</port>
        <scheduler>machine1:1222</scheduler>
        <publish>StringData.Text@Test.data</publish>
        <subscribe>StringData.Text@machine1:1234</subscribe>
      </component>
    </components>
  </profile>
</project>
```

Listing 4 – config.xml (exemple d'utilisation des composants)

Analysons chacun de ses composants.

- On constate tout d'abord qu'on a déterminé un profil *profile1* sur la *machine1*. On compte au moins un profil par machine. Il peut y avoir plusieurs machines de citées dans ce fichier de configuration, pourvu qu'elles soient ajoutées au service de résolution de nom.

- Un *scheduler* est défini sur le port 1222. On peut régler la fréquence des *heartbeat* via le paramètre *timeout*. Dans notre cas, un heartbeat est envoyé toutes les 100 millisecondes.
- Les deux services système (ci-avant explicités) sont déclarés, le *CNClient* et le *TopicClient*, respectivement sur le port 1221 et 1220. Concernant le premier service, la configuration la résolution d'adresse est chargée en paramètre.
- On assigne ensuite à chaque service un *String* de connexion, à savoir un nom référent à une machine et un port. L'adresse *localhost* 127.0.0.1 a été choisie pour le service *CNS* et la machine pour le service de topic. Les ports correspondent à ceux indiqués à la création du service.
- On liste ensuite les composants. On détermine d'abord sur quel port on le déclare, sur quelle machine et quel port se trouve le *scheduler* étant lié à lui (et par conséquent sur quelle machine), et enfin quels sont ses abonnements et ses publications. Concernant la définition de ces deux derniers points, une convention facilitant la lecture est demandée.
  - pour la publication d'abord :
    TypeDeDonnées.nomDuComposant@nomDuTopicInterne.data, ou encore
    topicExterne@topicInterne
  - pour l'abonnement ensuite :
    TypedeDonnées.nomDuComposant@nomDeMachine : port

Ainsi, *MyBlock* rend facile une réutilisation de composants créés, pourvu que les formats d'entrée et sortie soient respectés. AgentSlang a été développé suivant ce concept. Il est donc possible de réutiliser certains de ses composants pour d'autres projets, comme nous allons le décrire dans les parties suivantes.

2.2 La plateforme AgentSlang

Dans la partie précédente, nous avons compris que la structure MyBlock et, par extension, AgentSlang permettait une large réutilisation des composants, pourvu qu'on en connaisse les entrées, sorties. Nous détaillerons dans cette partie la fonction de chaque composant, les types de données requis en entrée et en sortie. La table 1 de la page suivante répertorie les composants suivant leurs types, les données publiées et reçues.

2.2.1 Composants système (MyBlock)

Tout d'abord, on répertorie 3 composants gérés MyBlock étant utilisés dans AgentSlang sans pour autant être spécifiques à celui-ci.

- Le composant **SystemMonitorComponent** est la plateforme permettant de connaître les statuts de tous les composants auxquels il est abonné. Les statuts sont renvoyés à tous les composants abonnés.
- Le composant **LogComponent** fourni trois niveau de debug à tous les composants, à savoir *critical, debug* et *uniform*. Les messages reçus par le *LogComponent* sont renvoyés directement à la console par défaut, mais il est possible de les faire suivre à un élément.
- Le composant **ComponentMonitor** étend le *LogComponent*, il permet d'afficher des liaisons entre les composants sous la forme d'un graphe mais aussi de filtrer les messages de debug.

2.2.2 Composants entrée/sortie

AgentSlang gère différents types d'entrée/sortie via des composants, comme le montre la liste suivante :

- Le composant **TextComponent** étant *Mixed*, il peut soit envoyer une chaîne de caractères dans le cas ou il se comporte comme un *Source*, ou afficher un message reçu s'il est considéré comme un *Sink*.
- Le composant **VoiceProxyComponent** permet de transcrire une voix en texte. Il est actuellement basé sur une application Android, en particulier l'API Google Speech.

Composant	**Fonction**	**Type**	**Données reçues**	**Données publiées**
Log Component	Debug/Logging	Sink	(in) DebugD	-
System Monitor Component	Event Monitoring	Mixed	(out) *system.monitor.data*, SystemEvent	(in) SystemEvent
Monitor Component	Platform Monitoring	Sink	(in) DebugD	-
Text Component	Text Input/Output	Mixed	(in) GenericD	(out) *text.data*, StringD
Voice Proxy Component	Speech Input	Source	-	(out) *voice.data*, StringD
Mary Component	Speech Synthesis	Mixed	(in) StringD (in) GenericTextAnnotation	(out) *voice.data*, AudioD
iSpeechTTS Component	Speech Synthesis	Mixed	(in) StringD (in) GenericTextAnnotation	(out) *voice.data*, AudioD
Senna Component	POS Tagging, Chunking, NER	Mixed	(in) StringD	(out) *senna.data*, GenericTextAnnotation
Metaphone Encoding Component	Phonetic Encoding	Mixed	(in) GenericTextAnnotation (in) StringD	(out) *metaphone.data*, GenericTextAnnotation
Template Extractor	Knowledge Extraction	Mixed	(in) GenericTextAnnotation	(out) *templateExc.command.data*, TemplateD (out) *templateExc.dialogue.data*, TemplateD
Dialogue Interpreter	Dialogue Generation	Mixed	(in) TemplateD	(out) *dialogue.data*, StringD
Command Interpreter	Dialogue and Command Generation	Mixed	(in) TemplateD	(out) *command.data* StringD
Valence Extractor Component	Valence Extraction	Mixed	(in) GenericTextAnnotation	(out) *valence.data* ValenceD

TABLE 1 – Résumé des composants AgentSlang

– Le composant **MaryComponent** prend en entrée un *StringData* et un *AnnotatedData* et génère une voix. La voix est générée par le toolkit marytts, qui fut par ailleurs utilisé pour le projet SEMAINE [2].
– Le composant **iSpeechTTSComponent** est un composant fournissant une voix naturelle parmi une liste assez complète de langues. L'API iSpeech TTS (Text To Speech) [5] est ici appelée.

2.2.3 Composants de traitement de langage

Une fois les données récupérées, elles sont traitées par certains composants.
– Le composant **SennaAnnotator** permet une segmentation et annotation de chacun des composants comme explicité dans la partie 2.1.1.
– Le composant **MetaphoneEncodingComponent** autorise la découpe par phonèmes. L'API ici utilisée est Metaphone3.
– Le composant **TemplateComposantExtracter** réfère au projet Syn !bad que nous expliciterons par la suite. Syn !bad permet l'identification de motifs dans des phrases, motifs listés à l'avance par le concepteur et suivant lesquels il est possible de réagir de façon pré-définie, notamment grâce aux composants suivants.
– Le composant **DialogueInterpreter** peut, suivant un motif identifié, répondre explicitement soit par une phrase soit par une commande via le **CommandInterpreter**.

2.2.4 Composant de traitement de rétroaction affective

Le composant **ValenceExtractorCompenent** permet d'extraire de la *valence* à partir de n'importe quel type de données. Cette notion de valence est explicitée dans la section (2.1.1.) .

2.2.5 Emplacements des composants et topics

Voici un arbre répertoriant les emplacements et les topics mis à disposition par les composants, pour le module MyBlock (figure 8) et la plateforme AgentSlang (figure 9).

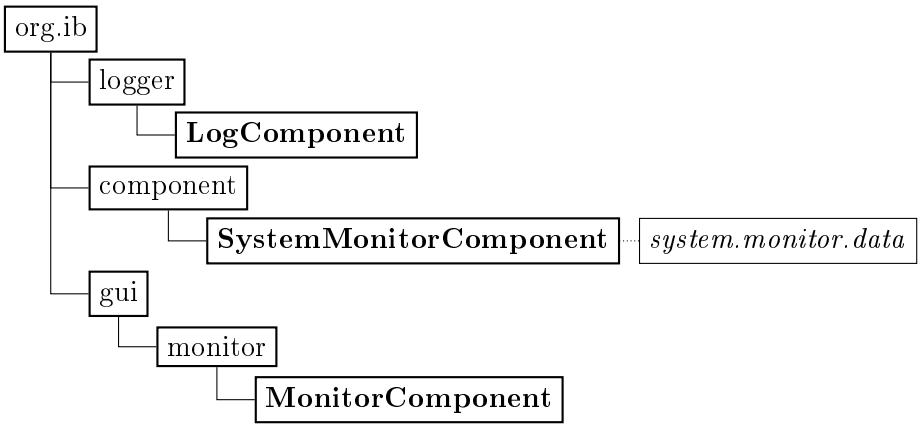

FIGURE 8 – Emplacements Composants et Channels sur le Module MyBlock

Dans ces figures, les nœuds simples représentent les paquetages, les nœuds en gras les composants et les nœuds en italique les différents channels publiés par les composants. Par exemple, le composant *TextComponent* se trouve dans le paquetage *org.agent.slang.inout* et publie sur le channel *text.data*. Les composants autorisant la réception de ce type de données peuvent s'inscrire à ce channel, et agir en conséquence dès qu'un élément passe par le channel.

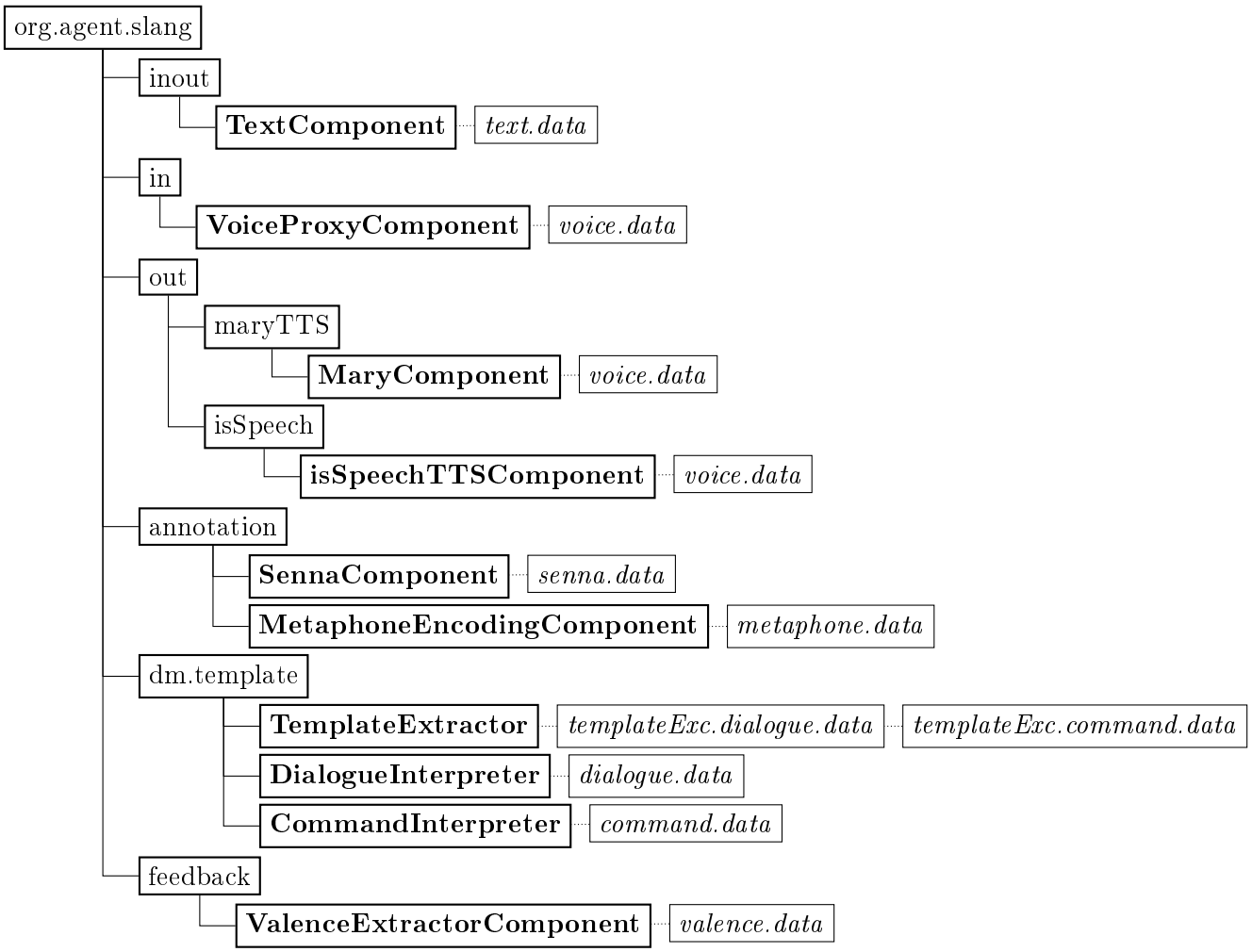

FIGURE 9 – Emplacements composants et channels sur la plateforme AgentSlang

2.2.6 Les paramètres additionnels

Le système propose également la possibilité d'ajouter des paramètres pour chacun des composants (autres que de l'affectation) sur un *port*, de l'inscription sur le *Scheduler* ou sur un autre composant (*subscribe*), de la publication (*publish*), et qui permet d'ajouter de l'information complémentaire sur le composant.

- Composant **VoiceProxyComponent** : *voiceBTuuid* pour l'identifiant bluetooth du périphérique et *voiceBTmac* pour l'adresse matérielle du périphérique (*Bluetooth Device Address*).
- Composant **SennaComponent** : *sennaPath* pour l'emplacement physique de Senna et *sennaParams* pour le type de tokenisation choisie
- Composant **TemplateExtractor** : *commandModel* sur lequel on y trouve les réponses relatives aux motifs, listés suivant un identifiant et *dialogueTemplate* listant les motifs.
- Composant **CommandInterpreter** : *commandModel*
- Composant **DialogueInterpreter** : *dialogueModel*

2.3 Le module Syn !bad

Syn !bad (acronyme de *Synonyms [are] not bad*) ou *Synnbad* est un module créé dans l'optique de simplifier et d'améliorer la reconnaissance de mots et de motifs. Il est intégré à AgentSlang, mais il est également possible de l'utiliser seul. La reconnaissance de motifs se base sur l'exacte similitude entre des caractères ou des séquences de caractères (ici des mots), mais aussi la qualité grammaticale des mots ou groupes de mots dans une phrase et leurs synonymes.

Le cœur de l'algorithme se base sur l'utilisation d'expressions rationnelles (*POSIX Regular Expression*). Une expression rationnelle est une suite de caractères typographiques aussi appelée motif (*pattern*), chargée de décrire une chaîne de caractères pour la trouver dans un bloc de texte. On peut assimiler les motifs à des automates de mots ou encore de types de mots. Une syntaxe spécifique est définie pour Syn !bad. Elle permet de définir des classes de caractères, à savoir des caractères ou des séquences de caractères qu'il est possible de rechercher dans des chaînes. Ces classes sont notamment définies grâce à des opérateurs standards, comme listés dans la table 2.

Opérateur	**Description**	**Exemple**
`|`	Opérateur de restriction par étiquette grammaticale	`[word|NN*]` permet de reconnaître le mot *word* ou un nom synonyme de *word*.
`$ ou #`	Notion de variable pour post-utilisation	`$nom` permet de pouvoir récupérer la valeur de *nom* dans un motif
`[ ]`	Prend en compte tous les synonymes de la chaîne entre crochets	`[manger]` sera reconnu avec les mots *avaler* ou *dévorer*
{}	Prend en compte l'élément précédent un nombre de fois défini sur un intervalle	$a\{2,4\}$ permet de reconnaître *aa*, *aaa* et *aaaa*
`< >`	Permet de gérer des structures définissant des types de mots	< <POS>[1] ('#' <variable>[2]) >
`?`	Signifie "zéro ou un" du précédent élément	`x y? z` permet de reconnaître *x z* et *x y z*.
`*`	Signifie "zéro ou plus" du précédent élément	`x y* z` permet de reconnaître *x z*, *x y z*, *x y y z*, *x y y y z*, etc.
`+`	Signifie "un ou plus" du précédent élément	`x y+` permet de reconnaître *x y*, *x y y*, *x y y*, etc.

TABLE 2 – Liste des opérateurs proposés par Syn !Bad

Prenons un exemple pour expliciter son fonctionnement. Considérons la phrase *Ovidiu eats good candies*. Elle peut être décrite en termes grammaticaux tels que <*nom*> <*verbe*> <*adjectif*> <*nom*>. Si l'on souhaite généraliser la phrase tout en gardant son sens (ou un sens proche) sans pour autant utiliser les mêmes mots, il est possible de créer un motif. On peut par exemple définir le motif suivant : `$name <VB*>* [good|JJ*] candies <#*>?`

Explicitons chacun des termes de ce motif :
- `$name` est une variable libre de contexte, représentant tout mot (non conditionné) et ayant pour contenu le premier mot de la phrase reconnue par le motif,
- `<VB*>*` correspond à un "zéro ou plusieurs" verbes, `VB*` référant à la catégorie des verbes,
- `[good|RB*]` réfère à un synonyme de *good* conditionné par `JJ*`, à savoir uniquement les adjectifs synonymes de *good*,
- `candies` doit apparaître explicitement dans la phrase,
- `<#*>?` est un marqueur de ponctuation, `#*` représentant le groupe générique de ponctuation du POS. Le `?` signifie "un ou aucun", ce token est donc optionnel, il peut ne pas y avoir de signe de ponctuation.

Ainsi, une phrase sera dite reconnue (*matched*) si elle respecte séquentiellement chacune des

1. <POS> := <PennPOS> | <GenericPOS>
avec <PennPOS> := 'JJ' | 'RB' | 'DT' | 'TO' | 'RP' | 'RBR'| 'RBS' | 'LS' | 'JJS' | 'JJR' | 'FW' | 'NN' | 'NNPS' | 'VBN' | 'VB' | 'VBP' | 'PDT' | 'WP$' | 'PRP' | 'MD' | 'SYM' | 'WDT' | 'VBZ' | '"' | '#' | 'WP' | '’' | 'IN' | '$' | 'VGB' | 'EX' | 'POS' | '(' | 'VBD' | ')' | '.' | ',' | 'UH' | 'NNS' | 'CC' | 'CD' | 'NNP' | 'PP$' | ':' | 'WRB'
et <GenericPOS> := '#*' | 'VB*' | 'RB*' | 'NN*' | 'JJ*'
2. <variable> := [a-z0-9]

caractéristiques des nœuds de ce motif, tel un automate.

Par exemple, la phrase *Ovidiu likes great candies !* sera reconnue par le motif, tel que le `$name`←"*Ovidiu*", `<VB*>*`↔"*likes*" en tant que verbe, [good|JJ*]↔"*great*" en tant que synonyme de good et adjectif, "*candies*" est présent explicitement et enfin `<#*>?`↔" *!*".

Lorsqu'un motif est reconnu, l'identifiant de celui-ci est retourné, ainsi que toutes les variables sauvegardées. On distingue deux types de variables, les variables locales comme `$name` et les variables globales comme `#name` :

- Le motif `$name is nice` autorise premièrement la reconnaissance de n'importe quel premier mot ou ponctuation (sans conditionnement) qui est stocké dans la variable `$name` puis `is nice` en brut.
- Le motif `<NN*#name>* is nice` autorise tout d'abord la reconnaissance d'un nom uniquement, qui est stocké dans la variable `#name` puis de `is nice` en brut.

Enfin, il est possible d'assigner aux motifs des étiquettes pouvant caractériser le motif, appelées *style*. Un *style* est une liste de paires étiquette/valeur assignée à chaque motif. Ceci permet d'ajouter une information supplémentaire au motif, et suivant celle-ci proposer des actions spécifiques. Un motif ayant un style *relation* (étiquette) qualifié de *familier* (valeur) provoquera par exemple un choix de réponses et/ou commandes particulières, qui n'auraient pas été choisies dans le cas ou l'étiquette *relation* eut été *soutenu*.

2.4 Un exemple d'utilisation des composants AgentSlang et Syn !Bad

2.4.1 Présentation et objectifs

Afin de se familiariser avec la plateforme AgentSlang et les méthodes de Syn !Bad, nous proposons de mettre en place un agent conversationnel type Chatbot répondant à quelques unes des réactions de l'utilisateur, et rendant possible le lancement d'actions sur notre ordinateur.

Le programme doit notamment répondre aux principales spécifications suivantes :

- Le Chatbot doit être testable sur le système Linux, et est développé en Java.
- Le Chatbot doit utiliser quelques composants d'AgentSlang et doit en proposer un nouveau. Il doit également appeler en direct des composants du module Syn !Bad afin d'en apprécier sa logique. Le tout doit rester le plus simple possible.
- Les composants MyBlock permettant la communication entre composants doivent être utilisés.
- Le Chatbot doit pouvoir récupérer du texte entré au clavier, éventuellement de la voix, et être capable de reconnaître son contenu grâce à l'appel des méthodes Syn !Bad.
- Les motifs du Chatbot sont stockés dans un fichier *XML*. Le fichier contient une liste d'éléments, dont chacun est décrit par l'identifiant du motif, le motif lui-même, et une liste de réponses liées à celui-ci.
- Le Chatbot doit pouvoir lancer une action suivant l'identifiant du motif. On considérera une action comme une commande texte exécutable dans un terminal. Les actions sont stockées dans un fichier *XML*, et sont déterminables par un identifiant.
- Suivant les réactions de l'utilisateur, les réponses lui seront envoyées de façon textuelle (affichage à l'écran) et vocale. De plus, elles sont choisies aléatoirement parmi la liste de réponses relatives à un motif. Il doit être possible de choisir une réponse spécifique, pour permettre sur une version plus évoluée (non demandée) de répondre suivant l'humeur de l'utilisateur.

2.4.2 Modèle et diagrammes

La figure 10 montre une proposition de modèle que nous avons pu développer. On compte trois composants tous abonnés au **Scheduler** (Monitor) : Le **TextComponent** (Text) qui publie un

String.data sur le port 1234, le composant **Jarvis** qui publie aussi un *String.data* sur le port 1236, le **MaryComponent** (Mary) qui publie un *Audio.data* sur le port 1235. Le TextComponent est abonné au *String.data* du composant Jarvis et à l'*Audio.data* du MaryComponent, le composant Jarvis est abonné au *String.data* du TextComponent, et enfin le MaryComponent est abonné au *String.data* du composant Jarvis.

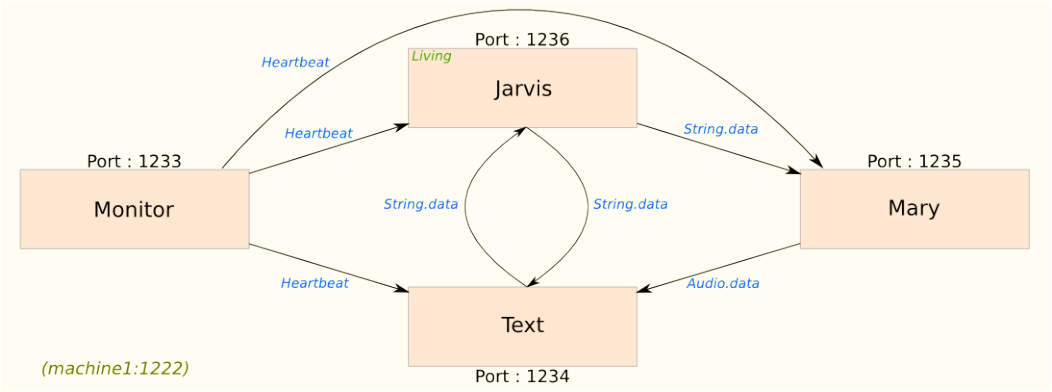

FIGURE 10 – Modèle d'interaction des composants du chatbot

Concrètement, le texte envoyé par l'utilisateur est tout d'abord récupéré grâce au composant **TextComponent**. Ce texte est ensuite traité en interne par le composant **Jarvis** en appelant des méthodes du module Syn !Bad. Une fois le procédé terminé, la réponse choisie par le composant Jarvis est envoyée au TextComponent pour un affichage sur la fenêtre, mais aussi **MaryComponent** qui va créer une donnée audio correspondant à la chaîne de caractères reçue. Cette donnée audio est adressée au TextComponent qui va la lire.

Ainsi, seul le composant Jarvis est à créer, les autres étant fournis par la plateforme AgentSlang. Pour alléger le composant, nous avons développé trois classes en parallèle qui décrivent différentes étapes du traitement de la chaîne de caractères en entrée. La figure 11 définit ces classes à travers un diagramme de classes.

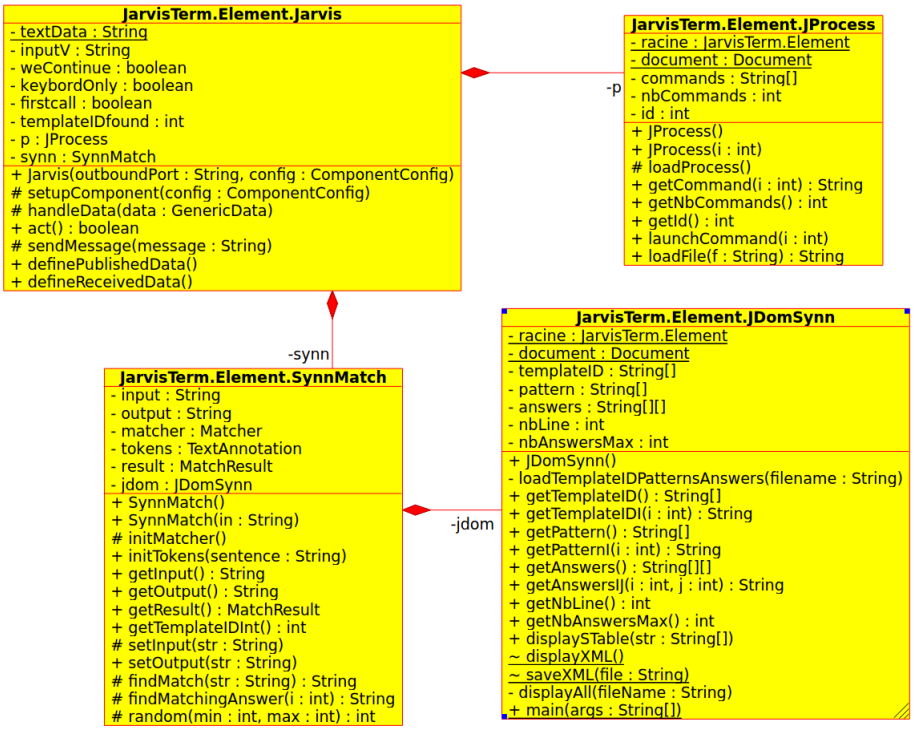

FIGURE 11 – Diagramme de classes développé pour le Chatbot

- La classe *SynnMatch* permet d'utiliser et d'appeler les variables d'instance et méthodes Syn !bad pour un traitement en interne. A l'initialisation, on charge les différents motifs (*matcher*). Suivant l'entrée (*input*), on la sépare en *tokens*, qui sont ensuite testés dans chacun des motifs. Si un des motifs est reconnu, on assigne la réponse requise à la variable *output*
- La classe *JDomSynn* permet le traitement et la récupération des données *XML* de part l'utilisation de la bibliothèque JDOM [6], spécifiquement aux besoins de tests sur les motifs et récupération de réponses dans le fichier *patterns.xml*. Il s'agit principalement de getters sur le fichier à traiter.
- La classe *JProcess* gère l'accès et le lancement de commandes, en l'occurrence commandes envoyées à l'interpréteur de commande Unix *Shell*. Les commandes sont également stockées dans un fichier *XML*, *commands.xml*.
- Enfin, la classe *Jarvis* est définie comme un composant qui traite en entrée une chaîne de caractères (texte envoyé par le composant TextComponent) et publie une chaîne de caractères correspondant à la réponse du Chatbot. Nous expliciterons 3 méthodes de la classe, les autres étant codées suivant le même principe que décrit en 2.1.4 .

1. La méthode *setupComponent* (cf. Listing 5) permet configurer le composant en initialisant notamment les booléens, l'entrée (*input*), la construction par défaut du *JProcess* et du *SynnMatch*.

```
// Extrait de la classe Jarvis.java : methode setupComponent
    protected void setupComponent(ComponentConfig config) {
        this.inputV="";
        this.weContinue=false;
        this.keybordOnly=false;
        this.p = new JProcess();
        this.firstcall=true;
        this.synn= new SynnMatch();
    }
```

Listing 5 – Jarvis.java (extrait 1 - composant Jarvis)

2. La méthode *act* (cf. Listing 6) est appelée à chaque fois qu'un *heartbeat* est reçu. On a choisi de l'appeler uniquement lorsqu'une chaîne de caractères (même vide) est envoyée par l'intermédiaire du TextComponent. Au premier *heartbeat* reçu, on averti simplement l'utilisateur qu'il peut soit appuyer sur la touche *Entrée* pour parler au microphone, soit taper le texte de son choix pour clavarder(l.15). Il eut également été possible de le notifier à l'initialisation. Dans le cas où le texte envoyé est vide (l.5), on lance la commande 0 correspondant tout d'abord à l'enregistrement du son via un microphone puis à un traitement de ce son par l'intermédiaire l'API google de reconnaissance vocale (l.7). Le texte reconnu est ensuite chargé en tant que chaîne de caractères dans *inputV* (l.8). On génère le retour grâce à la classe SynnMatch, que l'on publie avec la méthode *sendMessage*. Enfin, à chaque fin d'*act*, on assigne *FAUX* à *weContinue* (l.12). En effet, le *heartbeat* sera envoyé de façon périodique et sans arrêt, et l'on souhaite pouvoir appeler les précédentes commandes uniquement lorsqu'un texte (même vide) est envoyé par le TextComponent.

```
// Extrait de la classe Jarvis.java : methode act
    @Override
    public boolean act() {
        if (weContinue)     {
                if (!keybordOnly){
                    sendMessage(" (  ( (Recording) )  ) ");
                    p.launchCommand(0);
                    this.inputV=p.loadFile("output.txt");
                }
                this.synn = new SynnMatch(inputV);
                sendMessage(this.synn.getOutput());
            this.weContinue=false;
```

```
        }

        if(firstcall){
            sendMessage("[NOTE: press enter to talk, or write in the
                chat]");
            this.firstcall=false;
        }
        return true;
    }
```

Listing 6 – Jarvis.java (extrait 2 - composant Jarvis)

3. La méthode *handleData* (cf. Listing 6) est appelée à chaque fois qu'un élément est publié sur un ou plusieurs channels dont le composant est abonné. A chaque donnée reçue, dans notre cas une chaîne de caractères publiée par le TextComponent, on vérifie tout d'abord qu'il s'agit bien d'une chaîne de caractères. Si c'est le cas, alors on vérifie si elle est vide ou non. Si elle est vide alors on assigne *FAUX* à *keybordOnly*, sinon on lui assigne *VRAI* (l.11), et on assigne à *inputV* la chaîne de caractères recueillie (l.6).

```
// Extrait de la classe Jarvis.java : methode handleData
    protected void handleData(GenericData data) {
        System.out.println(data.toString());
        if (data instanceof StringData) {
            if(!data.toString().isEmpty()){
                this.inputV=data.toString();
                this.keybordOnly=true;
            }
            else {
                this.keybordOnly=false;
            }
            this.weContinue=true;
        }
    }
```

Listing 7 – Jarvis.java (extrait 3 - composant Jarvis)

Le projet Chatbot n'étant pas l'objectif ultime de ce stage et considérant le reste des constituants comme non indispensables pour sa compréhension globale, nous ne développerons pas davantage l'explication de ses éléments.

2.4.3 Test du Chatbot

La figure 12 montre le moniteur de texte lors de l'exécution du Chatbot, et notamment un échange entre un utilisateur (entrées "<<<") et l'agent conversationnel (sorties ">>>").

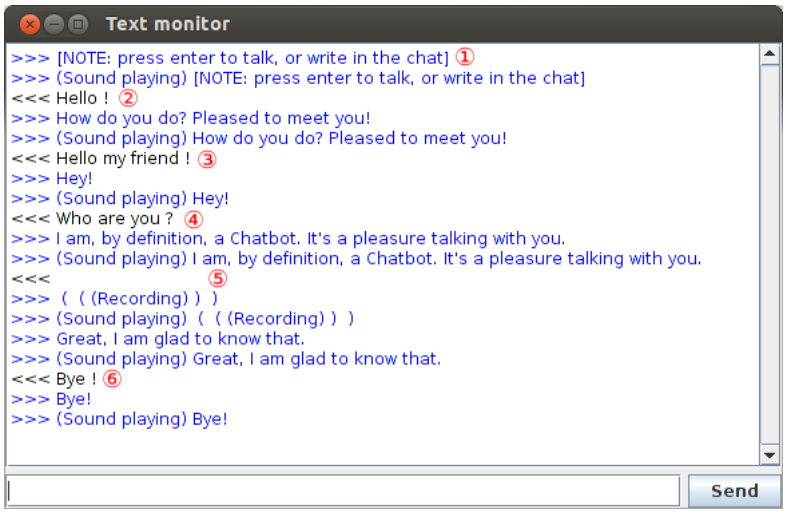

FIGURE 12 – Clavardage sur le moniteur de texte du Chatbot

Ci-dessous une description des éléments de clavardage de la figure 12 :

1. Le Chat est initialisé. *(Sound playing)* réfère aux éléments effectivement lus par la synthèse vocale.
2. L'utilisateur envoie au clavier "*Hello !*". Il est reconnu par le motif `[hello]*`, et une réponse liée à ce motif et choisie aléatoirement est donnée, et lue par la synthèse vocale.
3. L'utilisateur envoie au clavier "*Hello my friend !*". Il est à nouveau reconnu par le motif `[hello]*`, même procédure.
4. L'utilisateur envoie au clavier "*Who are you ?*". Cette chaîne de caractères est reconnue par le motif `who are you ?`, et une réponse liée est renvoyée par le chatbot puis lue par la synthèse vocale.
5. L'utilisateur appuie sur entrée, sans envoyer de texte. Un système d'enregistrement est lancé. L'utilisateur dit "*I am feeling better*". Le son est envoyé à google, reconnu puis retourné au Chatbot. Le motif `i am [feeling] [better]` est reconnu, et une réponse liée est renvoyée et lue par la synthèse vocale.
6. L'utilisateur envoie au clavier "*Bye*". Le motif `[goodbye]*` est reconnu, une réponse liée est renvoyée et lue par la synthèse vocale.

3 Intégration d'AgentSlang à ACAMODIA

3.1 Cadre du projet

3.1.1 Rappel du contexte

L'expérience effectuée dans le cadre du projet ACAMODIA était, pour rappel, la narration d'une histoire à un enfant, ceci en déroulant un certain nombre de diapositives. Pour chaque étape du scénario, le conteur virtuel raconte un morceau de l'histoire, et, pour passer à la suivante, soit pose une question à l'enfant relative à l'histoire, soit attend une réaction corporelle quelconque de l'enfant. Jusqu'alors, c'est un psychologue qui sélectionnait les réponses appropriées suite aux réactions des enfants. L'objectif de ce stage est d'utiliser certains composants d'AgentSlang et d'en créer de nouveaux pour remplacer les actions du psychologue sur l'interface graphique.

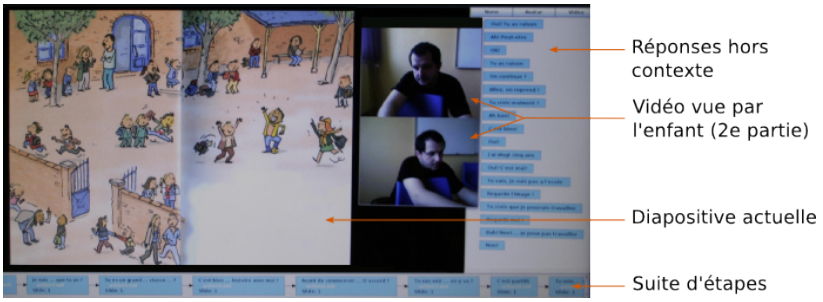

FIGURE 13 – Interface graphique sur l'ordinateur du psychologue

Comme montré sur la Figure 13, à chaque étape de l'histoire, le psychologue peut interagir de trois façons différentes avec l'interface graphique :

- Lors de la narration de l'histoire, le psychologue peut insister sur des éléments importants (lieux, personnages) en pointant la souris sur ceux-ci.
- Si l'enfant donne la réponse attendue ou une réaction corporelle relative à la situation, le psychologue passe à l'étape suivante par un clic sur l'étape consécutive.
- Sinon, et dans tous les autres cas, le psychologue peut envoyer une réponse hors contexte parmi une liste de réponses pré-définies. Par exemple, ceci comprend les cas où l'enfant n'est pas attentif et ou lorsqu'il n'a pas répondu correctement. Il est également possible de relancer une étape par un clic sur celle-ci.

3.1.2 Objet et domaine d'application

La finalité à long terme de ce projet est d'automatiser les actions du psychologue dans le projet ACAMODIA par une intelligence artificielle. L'objet de ce rapport est tout d'abord de définir les spécifications fonctionnelles détaillées du projet d'intégration de la plateforme AgentSlang au projet ACAMODIA. Il décrit l'ensemble des fonctionnalités de l'application, les éléments créés (leurs principes et utilités) et l'interface utilisateur. Toutes les fonctionnalités prévues pour ce projet d'intégration sont décrites ici. La maquette a été travaillée avec les commanditaires et approuvée, et c'est sur celle-ci que nous nous baserons lors de la phase de conception.

Ce rapport est applicable lors de la phase de développement de l'application. Ses fonctions sont conformes aux éléments et spécificités détaillées ci-après.

3.1.3 Cadre technique

L'application est exécutable sur un système d'exploitation de type Linux (démonstration sur Ubuntu 13.04), tournant sur un ordinateur possédant des entrées et sorties son. La version de la plateforme d'exécution et de compilation Java utilisée est la 1.7.0_21. Les bibliothèques JDom [6] (v. 2.0.5) et ZeroMQ [7] (v. 3.2.3) doivent être également installées. Enfin, la dernière version de la plateforme AgentSlang est nécessaire pour un accès aux composants déjà implémentés.

3.2 Description générale

3.2.1 Spécifications fonctionnelles

L'application devra être capable de dérouler les différentes étapes du scénario suivant les réactions de l'utilisateur, à savoir être capable de :

- Lancer une étape. On définit une étape par la lecture d'une partie de l'histoire par synthèse vocale, accompagnée d'un diaporama. Il ne peut y avoir qu'un et un seul type d'histoire à lire par étape, mais une même diapositive peut servir à plusieurs étapes.
- Afficher le numéro de l'étape courante à l'écran.
- Gérer la transition entre les étapes. On considère que l'on peut passer à l'étape suivante à partir du moment où l'utilisateur réagit. On appelle réaction une entrée texte par l'utilisateur. Si l'utilisateur réagit, alors l'application :
  1. Récupère le texte de réaction de l'utilisateur
  2. Vérifie si cette réaction est reconnue parmi les motif stockés dans un fichier *XML*
     - Si un des motifs est reconnu, alors la synthèse vocale lit la réponse qui s'y réfère. Dans le cas où le motif correspond à celui permettant de passer à l'étape suivante, on envoie la diapositive suivante si nécessaire.
     - Si aucun des motifs n'est reconnu, alors l'application avertit l'utilisateur qu'elle n'a pas compris, et attend à nouveau une réponse.
- Relancer, au bout d'un certain temps, l'utilisateur en cas de non réaction de sa part, par l'intermédiaire de réponses prédéfinies l'y encourageant. Suivant les situations, il doit être possible de passer à une nouvelle étape.
- Avoir la possibilité de passer à l'étape suivante sans pour autant avoir de réaction.
- Gérer des groupes de motifs ayant des groupes de réponses prédéfinies.
- Avoir la possibilité de considérer de la voix comme réaction en entrée.

De plus, la modélisation devra prévoir (implémentation toutefois non demandée) les points suivants, pour un développement ultérieur :

- Avoir la possibilité de remplacer la synthèse vocale par un agent conversationnel virtuel personnifié, au même titre que GRETA [8] du projet SEMAINE.
- Avoir la possibilité de considérer en entrée la visioconférence pour la récupération des réactions.
- Animer des éléments du scénario, notamment le pointeur pour rendre le projet plus interactif.

3.2.2 Cas d'utilisation

Comme on peut le constater sur la figure 14, l'utilisateur n'aura que très peu d'interactions avec l'application si ce n'est une expression textuelle au clavier, puis audiovisuelle pour une version ultérieure (non incluse dans la version 1). Cette spécificité est conforme à notre projet, étant donné que l'application est développée pour être la plus autonome possible.

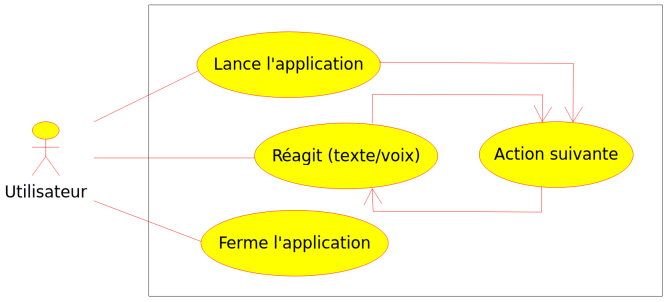

FIGURE 14 – Diagramme de cas d'utilisation de l'application

3.2.3 Interface utilisateur

En définitive, l'interface utilisateur à développer se présentera de la même façon que la figure 15. La diapositive courante est affichée. Le titre du scénario et le numéro de l'étape courante sont affichés en haut à gauche de l'écran. Une souris peut également être affichée pour pointer des parties de la diapositive à l'enfant.

FIGURE 15 – Interface utilisateur proposée

La taille de la fenêtre est celle de la diapositive, en l'occurrence de 965x721 pixels, et la largeur de l'agent conversationnel qui sera ajouté dans les versions suivantes. Il n'y a pas de plein écran de prévu ni de défilement horizontal/vertical. Les données utilisateurs seront entrées sur le moniteur du composant TextComponent de la plateforme AgentSlang. Par la suite, un composant de reconnaissance vocale et visuelle pourra remplacer le TextComponent.

3.2.4 Diagramme d'activité

La figure 16 décrit par étapes explicites et commentées les activités de l'application.

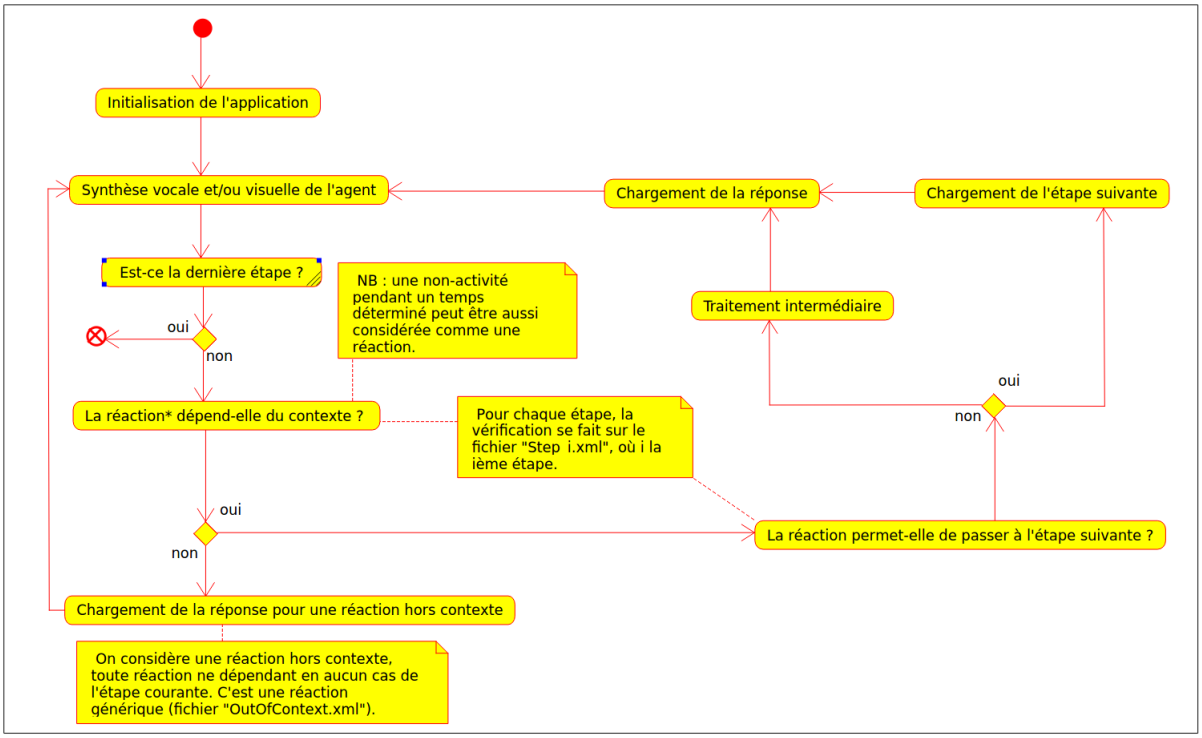

FIGURE 16 – Diagramme d'activité de l'application

L'intelligence artificielle de l'application se base, à chaque étape, sur la réaction de l'utilisateur [3]. On catégorise et stocke les réactions dans deux types de fichiers : d'une part les réactions que l'on qualifiera de hors contexte, à savoir les réactions qui ne dépendent pas de l'étape dans laquelle nous nous trouvons [4] (fichier *OutOfContext.xml*), et d'autres part les réactions dépendant du contexte (à l'étape i, fichier *Step_ i.xml*).

Ainsi, le fichier *Step_ i.xml* comprend, pour chaque étape i, une liste de réactions spécifiques pour lesquels on définit des réponses spécifiques et une commande. Les motifs considérés peuvent être tels que :

- les réactions amenant à l'étape suivante,
- les réactions amenant la répétition de l'étape courante,
- les réactions relatives à des réponses fausses,
- les réactions vides, qui peuvent parfois permettre de passer à l'étape suivante.

3.2.5 Modèle par composants proposé

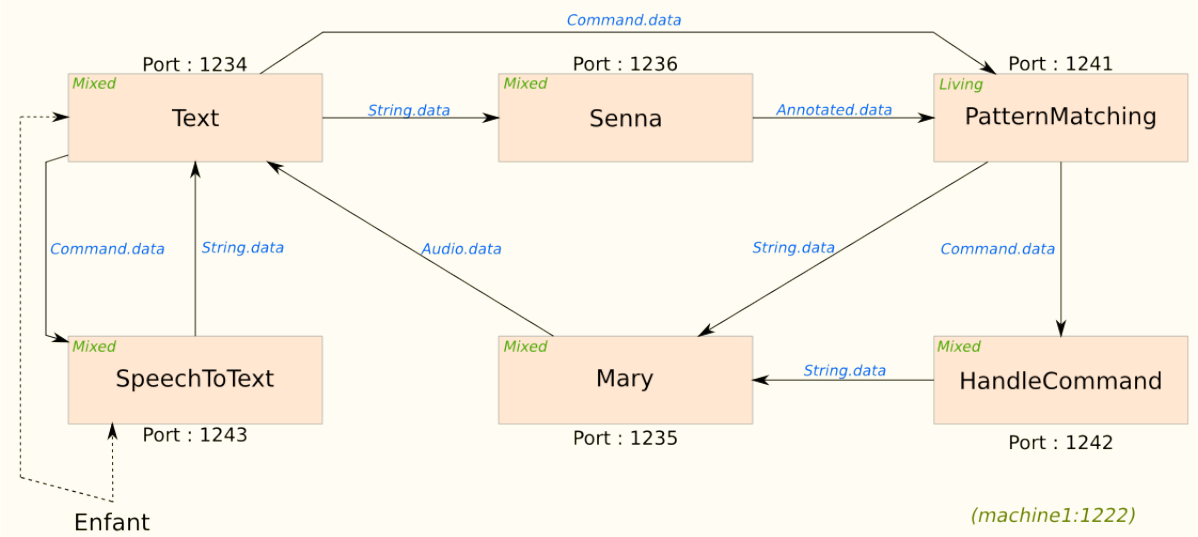

FIGURE 17 – Proposition de modèle d'intégration d'AgentSlang à ACAMODIA

Nous avons pu identifier différentes étapes pour modéliser les interactions du psychologue, et nous avons tout d'abord proposé un *Minimum Working Example* (MWE) dont les composants sont décrits sur la Figure 17.

1. On considère se trouver à une étape i du scénario. La partie de l'histoire relative à l'étape i a été lue, et l'enfant réagit, dans un premier temps au clavier, grâce au composant **TextComponent** (Text) au clavier d'AgentSlang, et/ou oralement grâce au composant **SpeechToTextComponent** (SpeechToText), qui gère la capture audio et la transcription de la voix en texte. Le composant TextComponent pourra être remplacé par n'importe quel autre composant de capture et lecture audio, et/ou de vidéo conférence (développement ici non demandé).
2. Comme sur AgentSlang, le composant **TextComponent** publie un *String.data*. Celui-ci est reçu par le composant **SennaComponent** (Senna) qui l'annote puis publie un *Annotated.data*.

3. NB : après quelques secondes, une réaction peut être considérée comme effective ou comme vide, telle que, quelque soit les moyens d'acquisition (textuel/vocal/visuel), l'utilisateur ne laisse transparaître d'altération textuelle, orale ou visuelle.
4. Une réaction non comprise sera considérée comme une réaction hors contexte également.

3. L'*Annotated.data* est reçu par un nouveau composant : **PatternMatchingComponent** (PatternMatching). Il s'agit d'un composant de type *Living*. Il va traiter l'annotation de la façon suivante :
   – Il vérifie tout d'abord si l'annotation est reconnue parmi une liste de motifs attendus. Ces motifs sont classés suivant des catégories (<*state*>) (cf. listing 8). Chaque identifiant est lié à un ou plusieurs motifs (<*lPatterns*>), une ou plusieurs réponses (<*lAnswers*>), une commande (<*command*>) et un statut explicite déterminant vers quelle étape on se dirige (<*next*>).

```
<!-- Step_i.xml -->
<state>
  <lPatterns>
    <pattern>oui*</pattern>
    <pattern>je peux le [voir]</pattern>
  </lPatterns>
  <lAnswers>
    <answer>Bravo !</answer>
    <answer>Tres bien</answer>
    <answer>Je te felicite</answer>
  </lAnswers>
  <command>command1</command>
  <next>2<next>
</state>
...
```

Listing 8 – Exemple de stockage XML des motifs attendus, reponses et commandes

   – Dans le cas où l'annotation est reconnue par un des motifs du fichier *Step_ i.xml*, le composant :
     -> Soit appelle une étape suivante, le composant publie la réponse relative correspondante à l'étape i, et publie une donnée *Command.data*, commande relative au motif, bien souvent pour passer la diapositive suivante si c'est nécessaire. Il est potentiellement possible de passer à n'importe quelle étape pourvu que ceci soit notifié par le <*next*>.
     -> Soit reste sur la même étape, et dans ce cas on ne publie qu'une réponse correspondante.
     -> En cas de non réaction, le silence sera considéré comme un motif effectif (ie. comme une réaction) suivant une période prédéfinie grâce à la méthode *act* du composant.
   – S'il n'a pas trouvé de motif correspondant dans ce premier fichier, alors ceci signifie que la réaction est hors contexte. On compare alors notre *Annotated.data* avec les motifs d'un deuxième fichier correspondant aux réponses "hors contexte". Ce fichier s'organise de la même façon que le listing 8. Le créateur du scénario pourra ainsi exceptionnellement autoriser le passage à l'étape suivante. Dans tous les autres cas, il pourra répéter le scénario, proposer des pistes à l'utilisateur, lui demander de se concentrer, etc. sans passer au scénario suivant.
   – Quelque soit le type de réaction, le composant PatternMatchingComponent renvoie un *String.data* correspondant à la réponse proposée par le système.

4. Dans le cas ou une donnée *Command.data* est publiée, la méthode *handleData()* du composant **HandleCommandComponent** (HandleCommand) est lancée. Celui-ci coordonne les actions telles que les changements de diapositives, la lecture du script dialogique relatif à l'histoire et tous autres éléments interactifs suivant les besoins. Il est lié à une classe gérant l'interface graphique (uniquement affichage des diapositives dans un premier temps). Les commandes sont listées dans un fichier et sont spécifiques à un identifiant, donné par l'entrée *Command.data*, comme le montre le listing 9. On note également que le processus *tellStory* permet l'appel d'un autre fichier *XML* répertoriant le script global de l'histoire, script lu dans tous les cas lors des étapes *i* du scénario, suite à la réponse de l'agent aux réactions des utilisateurs.

```
<!-- Commands.xml -->
<state>
  <command>command 1</command>
  <lLaunches>
    <launch>tellStory 1</launch>
    <launch>changeTitle Le ballon perche</launch>
    <launch>changeStepName Etape 1</launch>
    <launch>displaySlide 1</launch>
  </lLaunches>
</state>
...
```

Listing 9 – Exemple de stockage XML des commandes et de leurs actions

5. Le composant **MaryComponent** récupère la réponse publiée par **PatternMatchingComponent** et/ou par le **HandleCommandComponent** et en crée un son, qui sera enfin lu par le **TextComponent**. Le composant MaryComponent peut être remplacé par un autre lisant des sons pré-enregistrés, ou encore par un agent conversationnel virtuel, comme ce fut le cas initialement avec le projet ACAMODIA.
6. Enfin, on peut remarquer la publication de Command.data par le composant **TextComponent**. Cette commande permet d'une part d'activer le composant de reconnaissance vocale, et d'autre part d'initialiser et lancer le chronomètre de reconnaissance de "non-réaction" du **PatternMatchingComponent**.

Les données pouvant être maintenues par de tierces personnes seront donc stockées et structurées dans des fichiers *XML*. Ainsi, on compte au total n fichiers (où n nombre d'étapes du scénario) décrivant les motifs dépendant du contexte (type listing 8), 1 fichier décrivant les motifs hors contexte (type listing 8), un fichier de stockage de commandes (type listing 9), un fichier décrivant le script du scénario à raconter étape par étape. Cette organisation permet de proposer un outil évolutif et incrémental.

3.2.6 Perspectives

L'application pourra être pourvue de tous les éléments non développés mais cités dans la spécification fonctionnelle. De plus, il pourrait être intéressant de :

- Proposer une gestion et sauvegarde des entrées de l'enfant
- Proposer une interface graphique pour la création de scénarios suivant le modèle proposé ci-avant.

3.3 Conception

Nous allons à présent expliciter les différents éléments développés durant la phase de conception.

3.3.1 Organisation du projet

L'application utilise un certain nombre de données qui doivent être facilement accessibles par l'utilisateur. Ces données sont réparties comme le montre la figure 18.

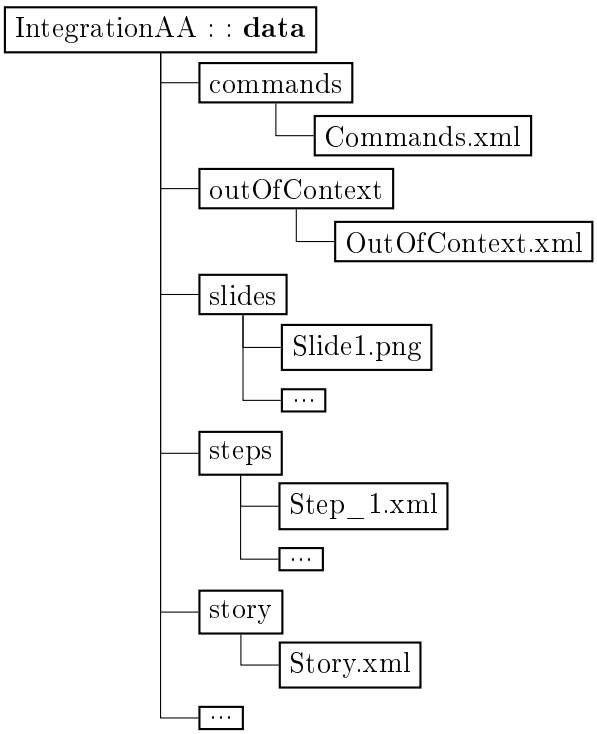

FIGURE 18 – Organisation des données éditables par l'utilisateur

On retrouve les fichiers *XML* définis précédemment, ainsi que les fichiers images étant affichés sur chaque diapositive (*slide*). A noter qu'il peut y avoir autant de fichiers *slide* et de *step* que nécessaire. On y trouve également les scripts permettant la génération automatique des données.

Le code source du projet se décompose quant à lui suivant les paquetages présentés en figure 19.

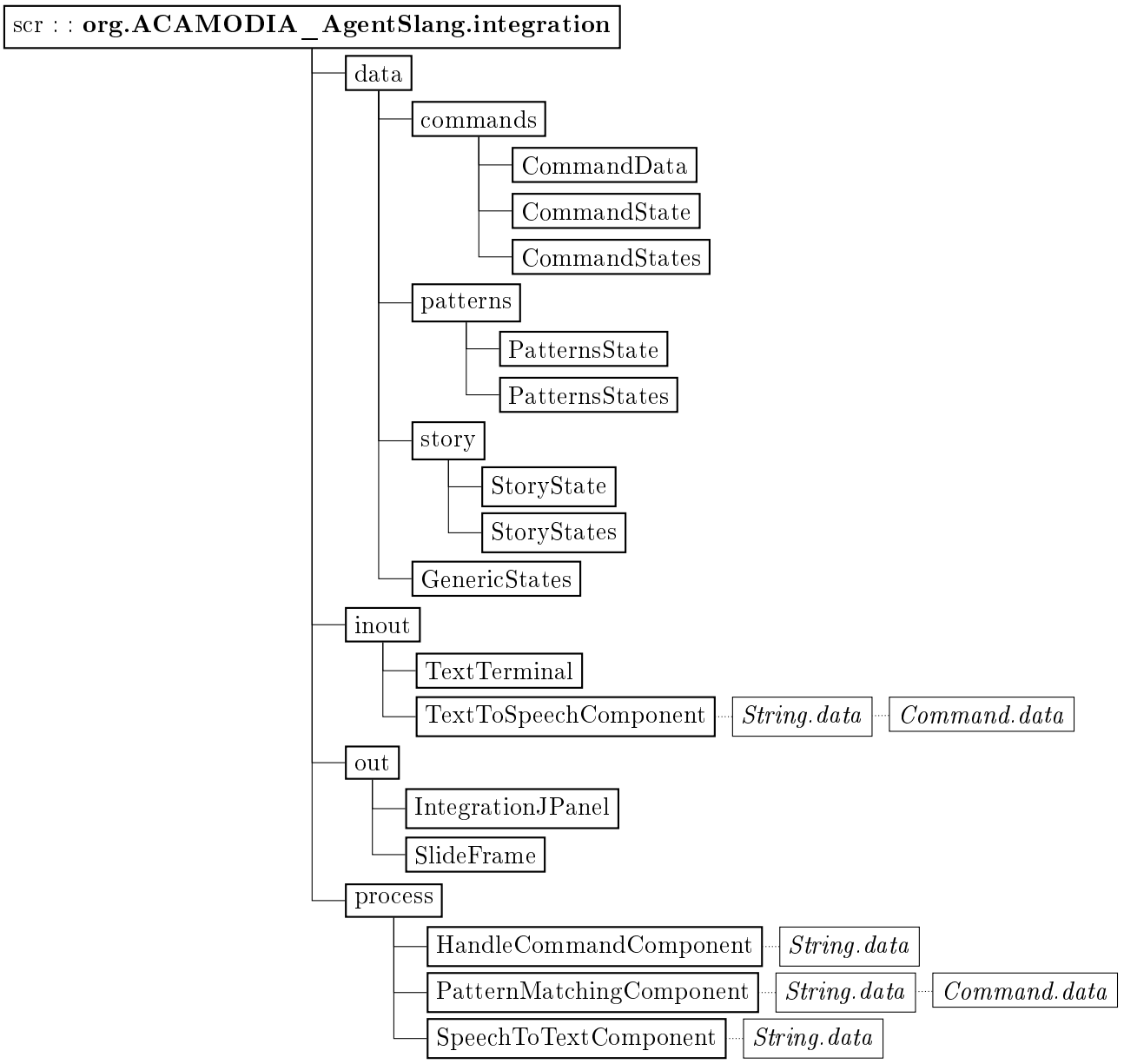

FIGURE 19 – Organisation des sources du projet InsertionAA.

3.3.2 Paquetage data

Le paquetage *data* permet, comme son nom l'indique, de regrouper la gestion des données en présence. La figure 20 en présente ses classes.

La classe **GenericData** tout d'abord, est une classe générique initialisant l'acquisition de données, à savoir l'ouverture des fichiers XLM. On récupère d'une part le nom du fichier, on créé par ce biais un nouveau Document (Jdom) et on initialise la racine *XML*. Le nombre d'états dans ce fichier est ensuite compté, et on initialise la liste des nœuds correspondant à ces états dans une collection. Les autres méthodes sont spécifiques au type de fichier, et sont implémentées dans les classes étendant la classe GenericData.

Dans le paquetage *commands*, la classe **CommandsState** regroupe les éléments d'un état listé du type de ceux de *Commands.xml* (cf. listing 9) : la commande faisant office d'identifiant et une liste d'éléments correspondant à des processus à lancer. Cette classe les charge puis permet leur récupération de façon optimale. La classe **CommandsStates** a pour variable d'instance une liste de CommandsState. Concernant les méthodes principales, une méthode permet de

retourner une CommandsState spécifique, une autre permet de retourner une liste de processus relativement à une commande. Ainsi, dans les cas où l'utilisateur a choisi de séparer une même commande (même identifiant) en deux listes de processus, ceux-ci seront tout de même tous chargés. On compte également la classe **CommandData** dans ce paquetage, nouveau type de données utile à la communication entre les composants PatternMatching et HandleCommand.

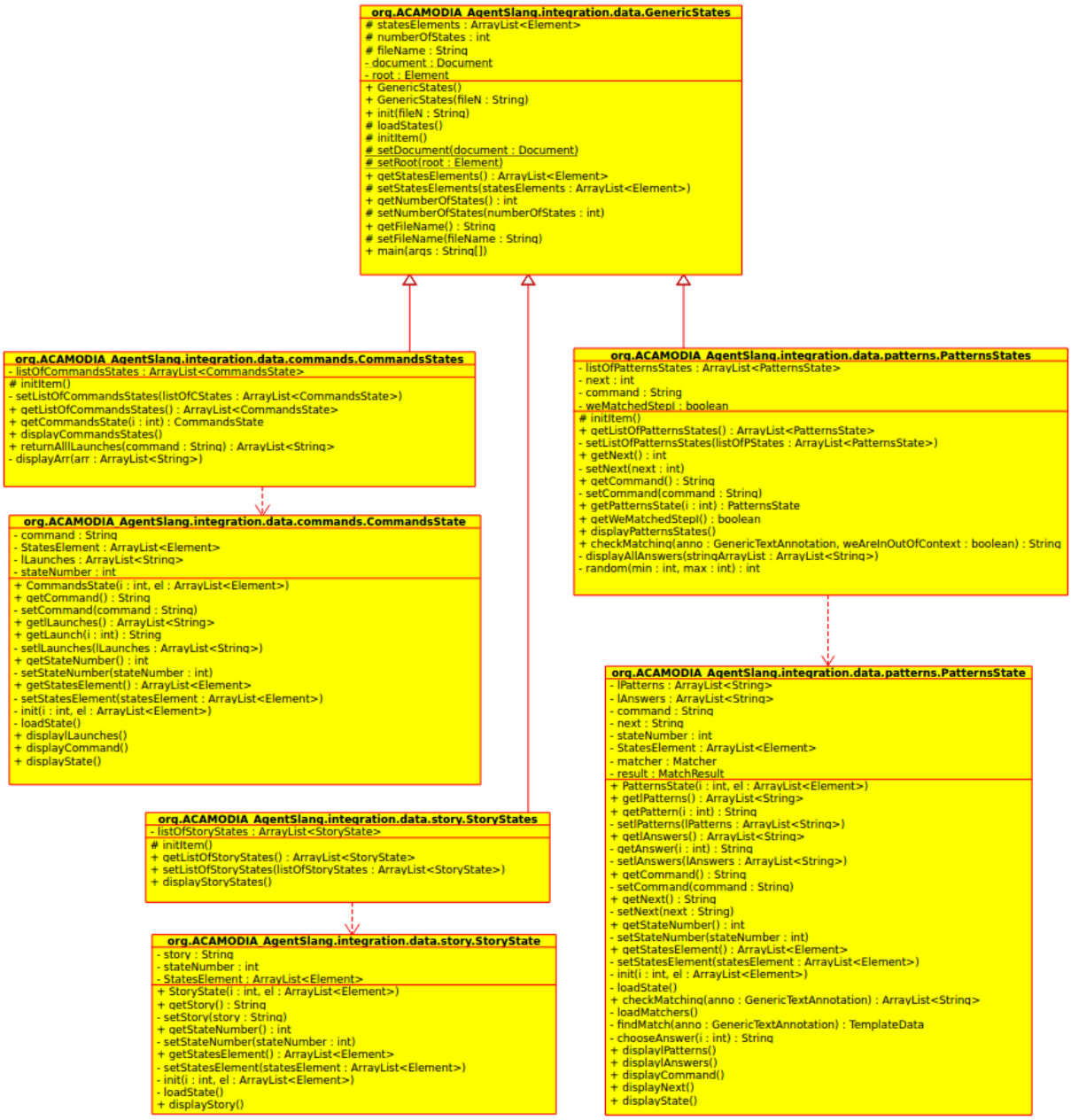

FIGURE 20 – Diagramme de classes du paquetage data

Le paquetage *patterns* regroupe les classes gérant des fichiers *XML* de type listing 8, à savoir des *Step_ i.xml* ou le *OutOfContext.xml*. La classe **PatternsState** implémente, relativement à chaque état, une liste de motifs, une liste de réponses, une commande et un entier correspondant à l'étape suivante. Cette classe permet leur initialisation, et leur récupération via des méthodes *get*. Une méthode permet également de vérifier si un des motifs est reconnu, et de retourner les réponses qui s'y réfèrent, notamment via les méthodes des classes Matcher et TemplateMatchResult de *Syn !bad*. La classe **PatternsStates** a pour variable d'instance une liste de PatternsState, qui rend ainsi exploitables les données entières du fichier. La méthode checkMatching effectue une vérification globale de ce qui est énoncé par l'utilisateur sur la base de données courante. Parmi la liste de réponses possibles retournées, une réponse est choisie aléatoirement.

Enfin, le paquetage *story* regroupe les classes **StoryState** et **StoryStates** permettant l'accès aux données de l'histoire stockées dans *Story.xml*. Ce sont de simples classes d'accès aux données, un scénario défini par des étapes numérotées, décrivant la base de l'histoire.

3.3.3 Paquetage inout

Le paquetage *inout* regroupe les classes gérant à la fois des entrées et sorties interfacées utilisateur (cf. figure 21).

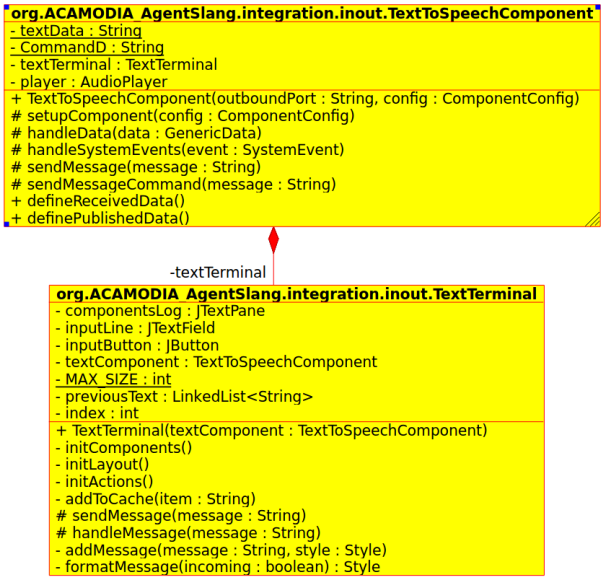

FIGURE 21 – Diagramme de classes du paquetage inout

Les classes **TextTerminal** et **TextToSpeechComponent** sont des copies des classes au même nom du projet AgentSlang, à quelques modifications près. D'une part elles permettent l'empilement des entrées audio et leur lecture progressive afin d'éviter l'écrasement d'une donnée à lire lors d'une nouvelle arrivée d'audio. De plus, le composant **TextToSpeechComponent** publie une commande dès réception d'une donnée du **SpeechToTextComponent**, afin que ce dernier relance la capture audio et que le **PatternMatchingComponent** commence le décompte de "non réaction".

3.3.4 Paquetage out

Le paquetage *out* montré figure 22 regroupe les classes gérant les sorties interfacées utilisateur.

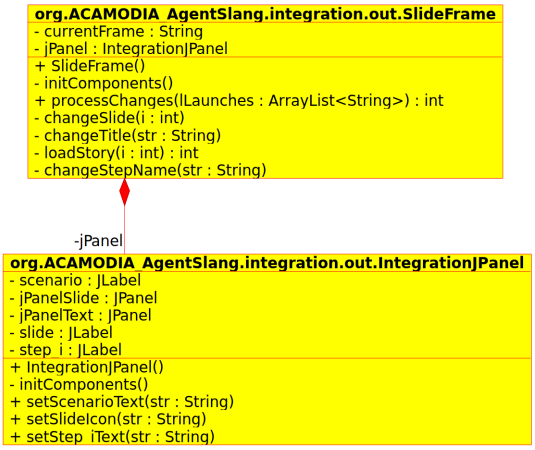

FIGURE 22 – Diagramme de classes du paquetage out

Tout d'abord, la classe **IntegrationJPanel** définit l'affichage global décrit en 3.2.3, initialisant un panneau sur la droite affichant la diapositive courante, un panneau sur la gauche déterminant l'étape courante, et un emplacement libre pour la future intégration d'un agent animé conversationnel. La classe **SlideFrame** permet le changement de certains de ses éléments suivant les commandes décrites par le scénario, telles que *displaySlide, changeTitle, tellStory, changeStepName, launch.* On pourrait par ailleurs implémenter tout nouveau type de commande dans cette classe, tel que le fait de pointer un emplacement de l'écran avec une souris.

3.3.5 Paquetage process

Le paquetage *process* montré figure 23 contient les composants décrits dans la section 3.2.5 à savoir la classe **PatternMatchingComponent**, la classe **HandleCommandComponent** et la classe **SpeechToTextComponent**.

FIGURE 23 – Diagramme de classes du paquetage process

Noter que nous avons opté d'identifier le motif de "non réponse" [5] par *wxcvbn.* Ainsi, si le créateur du scénario souhaite, à partir d'une étape *i*, passer à une étape *i+k*, il devra ajouter dans le fichier *Step_ i.xml* un état dont le motif est *wxcvbn*, avec les commandes et les étapes choisies.

Enfin, la classe **SpeechToTextComponent** appelle deux composants externes pour l'acquisition et le traitement des données vocales, le logiciel Sox [10] et le projet Java-Speech-API [11].

3.3.6 Tests unitaires

La majorité des méthodes développées étant de l'accès et/ou du traitement de l'information, nous avons développé des méthodes d'affichage de données dans chacune de nos classes au cours du développement. Celles-ci se sont avérées fonctionnelles et peuvent être à tout moment testées via un *main* affichant ces données dans le terminal.

Le debugger du logiciel Intellij IDEA 12.1.4 s'est avéré également très utile et efficace dans les phases de débogage des classes, et la résolution de problèmes annexes.

5. silence appuyé de x secondes, ou le fait de taper sur "Entrée" dans le TextTerminal avec une chaîne de caractères vide.

3.4 Saisie des données

La campagne de saisie de données du stage ici présenté repose sur différentes étapes :
- La récupération de retranscriptions faites durant les expériences ACAMODIA avec les psychologues et les enfants,
- La création d'une table globale répertoriant explicitement ces données, étant ensuite validée par les maîtres de stages,
- L'entrée de ces données dans les fichiers *XML* pour un traitement par le logiciel développé.

3.4.1 Initialisation des fichiers de données, version 1

Nous avons développé un script *bash*, qui, suivant le scénario global de l'histoire classifiée dans un fichier ou chaque ligne correspond à une étape du scénario, créé et initialise les fichiers *Commands.xml*, *Story.xml* et toutes les *Steps_ i.xml*. Ces fichiers contiennent uniquement l'histoire, et non les motifs, commandes, etc. L'utilisateur doit rentrer "à la main" toutes ces informations. Nous avons ainsi décidé d'automatiser davantage cette procédure de génération, afin de simplifier l'accès aux données et leur gestion.

3.4.2 Initialisation des fichiers de données, version 2

Nous proposons la procédure suivante pour la génération automatique des données :

1. L'auteur doit tout d'abord définir les étapes du scénario dans un fichier type listing 10, pour lequel une ligne correspond à une étape.

```
Bonjour, je m'appelle Anne.
Et toi, comment t'appelles tu ?
Je suis très contente de faire ta connaissance
Quel âge est-ce que tu as?
```

Listing 10 – Scénario A

2. Grâce à un script Bash fourni et au scénario précédemment créé, l'auteur génère ensuite un fichier de type listing 11, correspondant à une base de données vide.

```
# ID ; Etape ; Motif ; Réponse ; Commandes ; Suivant ;
1;Bonjour, je m'appelle Anne.;;;tellStory 2|;2;
2;Et toi, comment t'appelles tu ?;;;tellStory 3|;3;
3;Je suis très contente de faire ta connaissance;;;tellStory 4|;4;
4;Quel âge est-ce que tu as?;;;tellStory 5|;5;
```

Listing 11 – Base de données générée du scénario A

Il est composé de 6 champs par ligne : un identifiant numéraire de l'étape, le texte devant être lu et correspondant à cette étape, le motif devant être reconnu ainsi que la réponse devant être donnée suivant ce motif, un champ de commande(s) devant être séparées par le caractère "|", et enfin l'étape suivante correspondante.

3. L'auteur complète ensuite ce fichier suivant ses besoins comme montré dans le listing 12. Pour une même étape (définie par son identifiant *id*), il est possible d'avoir différents motifs, différentes réponses, différentes commandes .

```
# ID ; Etape ; Motif ; Réponse ; Commandes ; Suivant ;
1;Bonjour, je m'appelle Anne.;;;tellStory 2|;2;
2;Et toi, comment t'appelles tu ?;$nom;Bonjour $nom !;tellStory 3|;3;
2;Et toi, comment t'appelles tu ?;$nom;Salut $nom !;tellStory 3|displaySlide
    2|;3;
3;Je suis très contente de faire ta connaissance;;;tellStory 4|;4;
3;Je suis très contente de faire ta connaissance;Moi aussi;;tellStory 4|;4;
4;Quel âge est-ce que tu as?;$age ans;;tellStory 5|;5;
```

Listing 12 – Base de données complétée par le créateur du scénario A

4. Enfin, l'auteur exécute un dernier script Bash fourni qui génère les données suivant la base de données complétée, en respectant les spécifications précédemment rapportées.

3.5 Produit et améliorations

Le logiciel a été développé et testé suivant le scénario d'ACAMODIA et les retranscriptions des réactions des enfants traduites en anglais. En effet, le module *TextToSpeech* et le dictionnaire *Wordnet* d'AgentSlang n'est pour l'instant disponible qu'en anglais[6]. Cependant, une base de données française retraçant le scénario *Le ballon perché* et comprenant l'ensemble des retranscriptions fournies par les psychologues a été préparée.

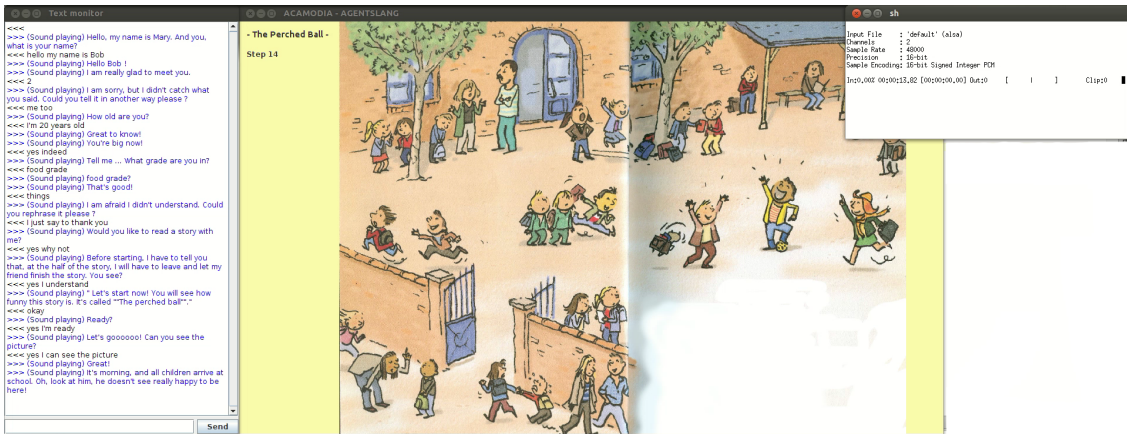

FIGURE 24 – Capture d'écran du logiciel en cours d'exécution - Interface

La figure 24 montre le programme en exécution. On y voit les logs du chat vocal sur la gauche, la fenêtre principale au centre et la fenêtre de capture son sur la droite.

Un didacticiel vidéo de génération de données est également fourni pour les utilisateurs. Il décrit comment créer, étape par étape, notre base de données suivant un scénario pré-défini. Une autre vidéo montrant une session d'interaction avec le logiciel est également à disposition. Elle souligne notamment plusieurs exceptions pouvant être rencontrées : la non-réaction, la non compréhension, l'utilisation de variables rendant l'échange plus interactif, et le fait de parler en même temps que la lecture virtuelle courante.

Le projet développé pourra être quant à lui amélioré sans difficulté (étant donné le caractère objet de sa conception) consécutivement aux propositions données en 3.2.1 et 3.2.6. L'intégration d'un agent virtuel tel que Marc [9], la mise à jour régulière des données, et l'ajout d'autres fonctionnalités interactives permettra de compléter l'objectif initial : reproduire l'expérience ACAMODIA de manière automatique, sans l'aide d'un contrôleur humain externe.

Enfin, il faut noter que le programme pourra être réutilisé suivant tout autre type de scénario, pourvu que l'utilisateur prévoit, répertorie et classifie les possibles réactions des utilisateurs. De ce fait, il pourrait être intéressant de rendre "intelligent" notre modèle en intégrant les réactions alors non répertoriées de façon automatique, à chaque fois où elles se présentent. De plus, on pourrait doter l'outil d'un module de gestion de "mémoire", tel que, suivant chaque utilisateur, l'agent puisse à nouveau accéder à certaines données précédemment abordées avec celui-ci, et répondre à ses réactions de manière optimale.

6. Un support français est en développement et est prévu pour une prochaine version.

Ce stage au sein du LITIS fut très enrichissant dans de multiples domaines. D'abord d'un point de vue humain, être intégré dans un laboratoire de recherche, avoir un suivi aussi complet et régulier est fructueux et permet d'être davantage productif. La grande disponibilité des maîtres de stage m'a notamment permis de clarifier certaines de mes méthodes de travail, fait découvrir de nouvelles pistes ou solutions en terme de modélisation. Ce fut également l'occasion de revoir les notions de génie logiciel, de développement, et de souligner l'importance du détail dans l'expression de rapport de projet. J'ai pu aussi découvrir l'environnement et le contenu d'une section de thèse, puis comprendre et en assimiler des méthodologies spécifiques s'y référant.

Concernant le thème de stage, le domaine des interactions homme/machine fut en grande partie une découverte pour ma part. Les applications possibles m'ont grandement intéressées, car celles-ci ne se bornent pas uniquement à un projet spécifique tel qu'ACAMODIA, mais peuvent dans de nombreux autres cas être réutilisées.

En terme de résultats opérationnels, nous avons pu d'une part étudier l'agencement du projet AgentSlang grâce à des méthodes de reverse engineering, d'implémentations de tests, et d'explications du concepteur, puis fourni une documentation en français de celui-ci. Nous avons ensuite développé un programme intégrant l'AgentSlang au projet ACAMODIA, suivant des spécifications précises définies avec les commanditaires. Tous les composants listés pour une interface ACAMODIA complète n'ont pas pu être développés dans notre logiciel. Je regrette par exemple de n'avoir pu étudier le projet MARC et l'intégrer au programme. Toutefois, la modélisation proposée permet de tous les intégrer, ceci étant une des lignes de conduite du stage.

Enfin, d'un point de vue plus personnel, ce stage m'a permis de prendre du recul sur mon projet professionnel, notamment d'apprécier mes forces et faiblesses en terme de réflexion ingénieur et application des méthodologies acquises, et qui plus est de renforcer ma volonté de centrer mes études et mon futur travail autour du domaine de l'informatique.

Table des figures

1 Organigramme du LITIS . . . 5
2 Développement d'ACAMODIA au LITIS . . . 7
3 Diagramme simplifié des objets Data de MyBlock . . . 8
4 Composant défini dans MyBlock . . . 9
5 Diffusion d'un message entre 2 composants . . . 9
6 Scheduler ayant 2 composants abonnés sur différentes machines . . . 10
7 Modélisation de l'exemple . . . 10
8 Emplacements Composants et Channels sur le Module MyBlock . . . 16
9 Emplacements composants et channels sur la plateforme AgentSlang . . . 17
10 Modèle d'interaction des composants du chatbot . . . 20
11 Diagramme de classes développé pour le Chatbot . . . 20
12 Clavardage sur le moniteur de texte du Chatbot . . . 23
13 Interface graphique sur l'ordinateur du psychologue . . . 24
14 Diagramme de cas d'utilisation de l'application . . . 25
15 Interface utilisateur proposée . . . 26
16 Diagramme d'activité de l'application . . . 26
17 Proposition de modèle d'intégration d'AgentSlang à ACAMODIA . . . 27
18 Organisation des données éditables par l'utilisateur . . . 30
19 Organisation des sources du projet InsertionAA. . . . 31
20 Diagramme de classes du paquetage data . . . 32
21 Diagramme de classes du paquetage inout . . . 33
22 Diagramme de classes du paquetage out . . . 33
23 Diagramme de classes du paquetage process . . . 34
24 Capture d'écran du logiciel en cours d'exécution - Interface . . . 36

Liste des tableaux

1 Résumé des composants AgentSlang . . . 15
2 Liste des opérateurs proposés par Syn !Bad . . . 18

Listings

1 Console.java (exemple d'utilisation des composants) . . . 11
2 Test.java (exemple d'utilisation des composants) . . . 12
3 cnsService.xml (exemple d'utilisation des composants) . . . 12
4 config.xml (exemple d'utilisation des composants) . . . 13
5 Jarvis.java (extrait 1 - composant Jarvis) . . . 21
6 Jarvis.java (extrait 2 - composant Jarvis) . . . 21
7 Jarvis.java (extrait 3 - composant Jarvis) . . . 22
8 Exemple de stockage XML des motifs attendus, reponses et commandes . . . 28
9 Exemple de stockage XML des commandes et de leurs actions . . . 29
10 Scénario A . . . 35
11 Base de données générée du scénario A . . . 35
12 Base de données complétée par le créateur du scénario A . . . 35

Références

[1] *Wordnet*, grande base de données lexicales de la langue anglaise (noms, verbes, adjectifs et adverbes)
`http://wordnet.princeton.edu/`.

[2] *Projet Semaine*, développement d'un agent conversationnel comprenant les signaux sociaux faciaux
`http://www.semaine-project.eu/`.

[3] Ovidiu Şerban, *Detection and Integration of affective feedback into distributed interactive systems.*
Institut National des Sciences Appliquées de Rouen, Laboratoire d'Informatique de Traitement de l'Information et des Systèmes.
Universitatea Babeș-Bolyai, Facultatea de Matematică și Informatică.

[4] *Projet SENNA*, logiciel non commercial de traitement automatique de langage
`http://ml.nec-labs.com/senna/`.

[5] *iSpeech Text To Speech*, logiciel non commercial de synthèse vocale
`http://www.ispeech.org`.

[6] *JDOM*, bibliothèque open source pour manipulation de fichiers XML en Java
`www.jdom.org`.

[7] *ZeroMQ*, bibliothèque pour les échanges de messages asynchrones
`http://www.zeromq.org/`.

[8] *GRETA*, agent conversationnel personnifié en 3 dimension
`http://perso.telecom-paristech.fr/~pelachau/Greta/`.

[9] *Multimodal Affective and Reactive Characters*, agent conversationnel de traitement et conception et d'interaction en temps réel avec des agents virtuels expressifs visuellement réalistes
`http://marc.limsi.fr`.

[10] *SoX*, logiciel de traitement de son en ligne de commande.
`http://sox.sourceforge.net/`.

[11] *Java-speech-api*, api java incluant capture, reconnaissance et synthèse vocale. Les services Google sont utilisés pour la synthèse et la reconnaissance vocale. Requiert une connexion internet.
`https://github.com/The-Shadow/java-speech-api`.