A Modularized and Reusable Architecture for an Embedded MAS IDE

Gabriel Ramos Alves Lima[1][a], Elaine Maria Pereira Siqueira[1][b], Thácito R. Costa Medeiros[1][c], Carlos Eduardo Pantoja[1][d], Nilson Mori Lazarin[1,2][e] and José Viterbo[2][f]

[1]*Federal Center for Technological Education Celso Suckow da Fonseca (Cefet/RJ), Rio de Janeiro, Brazil*

[2]*Institute of Computing, Fluminense Federal University (UFF), Niterói - RJ, Brazil*

{*gabriel.ramos, elaine.siqueira, thacito.medeiros*}*@aluno.cefet-rj.br,*

Keywords: Multi-Agent Systems, Software Architecture, Software Engineering, IDE, ChonIDE.

Abstract: The rise of the Embedded Multi-Agent Systems (MAS) field brings challenges regarding technologies specifically tailored for development in this area. Among these, chonIDE, as a new Integrated Development Environment (IDE) dedicated to this newly explored scenario, has gaps in its code that can be improved to meet market demands. In this regard, this paper outlines a new architecture through the restructuring of the components of this IDE to make it more scalable and, consequently, mitigate challenges in the Embedded MAS scenario. Additionally, changes in file structure are suggested to promote code versioning and enhance interoperability with other IDEs at the project reuse level, thus making it better adapted to market demands.

1 INTRODUCTION

Changes in the characteristics and expectations of software systems have led to the emergence of a series of new software engineering challenges (Tennenhouse, 2000; Zambonelli and Van Dyke Parunak, 2003). In this new scenario, the use of Multi-Agent Systems (MAS) is advocated as a software engineering paradigm for designing and developing software systems (Wooldridge and Jennings, 1995; Jennings, 2001; Wooldridge, 2009). In MAS, applications are designed and developed in terms of autonomous software entities (agents) that can achieve their goals with flexibility by interacting with each other through high-level protocols and languages. These characteristics are well-suited for addressing emerging complexities (Zambonelli et al., 2003).

Integrated Development Environments (IDEs) widespread in the market, although sufficiently meeting the user experience aspect by offering features such as code versioning, indentation, syntax highlighting, among others, do not address the decentralized nature of distributed systems with agents (Tennenhouse, 2000). This is because these systems have unique contextual needs, such as remote embedding of projects, Agent-Oriented Programming (AOP), and embedding of devices on single-board computers, monitoring agent interactions and perceptions, among others.

The success and general implementation of complex software systems based on Multi-Agent Systems (MAS) require new models and technologies (Zambonelli et al., 2003). Among these technologies, chonIDE (Souza de Jesus et al., 2023) stands out in supporting Embedded MAS compared to other IDEs due to its specialized nature, which takes into account the specific demands of this scenario. However, the architecture of chonIDE is not flexible or extensible to changes, due to its tightly coupled nature, which leads to low scalability potential regarding features. It also does not ensure interoperability, as the file structure of its projects is not compatible with other IDEs, preventing the reuse of its projects.

Therefore, this paper proposes a new architecture for chonIDE through the restructuring of its components, aiming to make it less tightly coupled and, consequently, more scalable to enable the addition of new functionalities via third parties and facilitate unit testing, extensibility, and flexibility, among others. Additionally, a new folder and file structure has been implemented to provide code versioning and ensure interoperability between projects.

[a] https://orcid.org/0009-0009-3233-9408
[b] https://orcid.org/0009-0006-1879-4891
[c] https://orcid.org/0009-0009-3310-714X
[d] https://orcid.org/0000-0002-7099-4974
[e] https://orcid.org/0000-0002-4240-3997
[f] https://orcid.org/0000-0002-0339-6624

Lima, G. R. A., Siqueira, E. M. P., Medeiros, T. R. C., Pantoja, C. E., Lazarin, N. M. and Viterbo, J.
A Modularized and Reusable Architecture for an Embedded MAS IDE.
DOI: 10.5220/0013439600003929

In *Proceedings of the 27th International Conference on Enterprise Information Systems (ICEIS 2025) - Volume 1*, pages 1034-1045
ISBN: 978-989-758-749-8; ISSN: 2184-4992

This paper is organized as follows: Section 2 presents an extended comparative table of related work; Section 3 explains the components of the toolkit, their respective functions, and relationships; Section 4 introduces the new modular and scalable architecture, including the modeling and implementation that enable the folder and file structure for projects; and finally, Section 5 discusses the final considerations and expectations for future work.

2 RELATED WORK

A comparative analysis of IDEs for Embedded MAS Development (Siqueira et al., 2024) considered various software development factors by comparing popular IDEs on the market with the availability of IDEs for Embedded MAS development. In this case, the comparison took into account Eclipse, VSCode, and IntelliJ IDEs based on the following characteristics: installation, platforms, usage license, user manual, user community, developer community, interface, compilers or interpreters, debugger, editor, syntax checking, syntax highlighting, autocompletion, quick help, and indenting.

Considering the context of Embedded MAS, it is relevant to add functionalities such as remote project deployment, Agent-Oriented Programming, deployment on single-board computers, and monitoring of agent interactions and perceptions to the analysis criteria of the table. These characteristics increase the level of abstraction between the developer and the resources used in software engineering processes, allowing them to focus primarily on designing the application they are creating without worrying about, for example, manual integration configurations that are necessary when using general-purpose tools.

Furthermore, the installation of the IDE should be easy and quick, allowing the user to start using it as soon as possible. It is important for the IDE to work on the most common hardware/software platforms, including Windows, Linux, and macOS. IDEs distributed under open-source licenses are preferable due to their accessibility. It is desirable that the user manual be as complete as possible so that the user can resolve initial doubts without having to search in forums. A large community is a positive aspect, as it facilitates the resolution of any issues that may arise. Typically, IDEs developed by groups receive updates more quickly than those developed by a single person. The interface should be intuitive, clean, and user-friendly (Siqueira et al., 2024). Table 1 shows the comparison of desired functionalities in an IDE.

It is highly desirable for the debugger to be integrated, as it facilitates the identification of logical issues. A powerful editor with various features can be helpful, such as line numbering for locating reported errors, multiple tabs for quick viewing of different files, and current line highlighting for easier navigation within the file, among other features considered below. Syntax checking is a very useful feature, as it notifies the user when there is something wrong with the code before the compiler or interpreter is run. Syntax highlighting is important, as it displays the code in a way that makes it easy to visualize and locate functions, variables, constants, strings, operators, and more. Autocompletion aids in typing the code, reducing errors. Quick help provides tips when the mouse hovers over certain parts of the code. The indenting feature is responsible for formatting the code, helping with its organization, and improving readability.

Table 1: Functionalities Comparison between IDEs.

	chonIDE	Eclipse	VSCode	IntelliJ
Easy to Install	●●●○○	●●●●●	●●●●●	●●●●○
Multiplatform	✗	✓	✓	✓
Multilingual	✗	✓	✓	✓
Free to Use	✓	✓	✓	✓
Clean Interface	✓	✓	✓	✓
Debugger	✗	✓	✓	✓
Syntax Check	✗	✓	✓	✓
Syntax Highlighting	✗	✓	✓	✓
Autocomplete	✗	✓	✓	✓
Quick Help	✗	✓	✓	✓
Multiple Tabs	✗	✓	✓	✓
User Manual	✗	✓	✓	✓
File Viewer	✓	✓	✓	✓
Line Counter	✓	✓	✓	✓
Highlighted Current Line	✗	✓	✓	✓
Indentation	✗	✓	✓	✓
Remote Deployment of Projects	✓	✗	✗	✗
Single Interface to Embedded MAS Programming Paradigms	✓	✗	✗	✗
Monitoring agents interactions and perceptions	✓	✗	✗	✗

3 FRAMEWORK

The chonIDE (Jesus et al., 2022; Souza de Jesus et al., 2023) is an environment that integrates, within a single-screen experience, both a set of technologies enabling the development of Embedded MAS and typical features of a modern IDE that assist during software design and allows designers to access and use these development resources effectively. This section explores the software components of chonIDE and their relationships with other parts of the toolkit for developing Embedded MAS, presented in Figure 1, to describe the IDE's functionality.

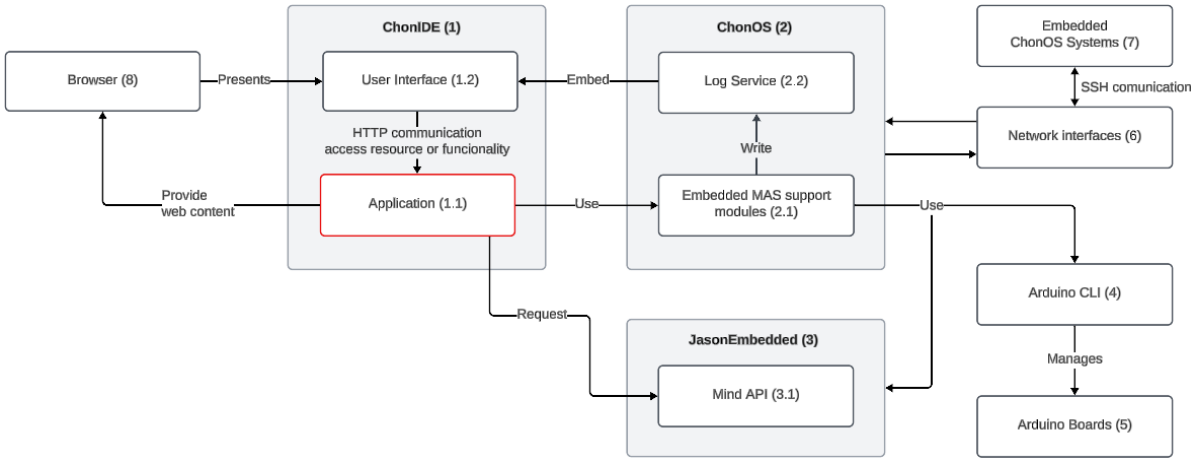

Figure 1: ChonIDE System Flow.

The interfacing between chonIDE (1), the implementation, and the management of the system designed by the developer is performed by *chonOS*: *Cognitive Hardware on Network - Operating System*[1] (2), a GNU/Linux distribution with the specific purpose of facilitating the conception and use of Embedded MAS This solution introduces a new layer into the development architecture (Lazarin et al., 2023). Alongside the four existing layers— Hardware, Firmware, Interface, and Reasoning (Pantoja et al., 2016) — a fifth layer, Management, is added. This layer enables remote control between the developer's computer and single-board embedded hardware (e.g., Raspberry Pi), achieved through *Shell Script-based services and modules* (2.1 and 2.2) that abstract different technological dependencies, providing access to several real-time resources. This capability spans the entire embedding experience through *SSH communication within the same Wi-Fi network* (2, 6, and 7).

Thus, remote management of the *Firmware* and *Reasoning* layers of the embedded hardware by the designer is made possible, with emphasis on:

- Sending source code to the *Arduino boards* (5) using the *Arduino CLI* (4).
- Implementing the Embedded MAS through the BDI frameworks supported by the distribution — Jason (Bordini et al., 2007), JaCaMo (Boissier et al., 2020), and *Jason Embedded* (Pantoja et al., 2023) (3).
- Visualization of agent logs — event and message records — during Embedded SMA activity and other system information through *log service* (2.2), a web page incorporated directly into the graphical interface (1.2).

Among the BDI frameworks previously mentioned for the implementation of MAS, the IDE uses, via ChonOS (2), the *Jason Embedded* (3), which is an extended version of Jason (Bordini et al., 2005) that supports the development of Embedded MAS to provide autonomy and communicability to IoT devices and mobility and adaptability to agents (Pantoja et al., 2023). Additionally, this component includes a feature called the *Mind API* (3.1), a REST API that allows inspection of the agents' minds—beliefs, rules, intentions, and plans—at a given cycle. Through the *application* (1.1), it is possible to query the most recent state of all agents or a specific agent within a particular cycle directly via the Mind Inspector graphical component, integrated into the right sidebar of the chonIDE interface (1.2).

Finally, the *application* (1.1) should serve as the integration and provision platform for the functionalities of the components that enable the development of Embedded MAS (2 and 3), as well as resources characteristic of an enhanced programming experience — such as the system management dynamics of the IDE project files, which will be structured, and the source code versioning through Git, which will be enabled — all in a uniform and centralized manner, specified through an API for consumption by the *user interface* (1.2), which is also provided by this component for the developer's *Browser* (8) (e.g., Mozilla Firefox, Google Chrome).

That being said, after specifying the parts that make up the IDE system, its resources, and their respective relationships, it is necessary to define a restructuring and establish an architecture for the application (1.1) that is capable of integrating the different mechanics and resources of the toolset in a scalable, modular, and decoupled manner. This new ar-

[1]https://os.chon.group

chitecture should facilitate implementing and scaling new functionalities, such as project file management within the IDE and versioning through Git.

4 THE ARCHITECTURE

The IDE application consists of a *web layer*, which provides the pages for the developer's browser and specifies an API responsible for managing communication with the user interface, and an *implementation layer*, which is responsible for implementing the integration, access methods, and manipulation of the *development resources* provided by the solution.

In this context, the *implementation layer* consequently centralizes elements that handle data persistence, communication with external services, utility mechanics, domain dynamics, and business logic, which makes the structure tightly coupled. As a result, it is not feasible to ensure aspects such as scalability, the extensibility of current and future functionalities, and the creation of unit tests — key aspects for solutions that encompass a variety of dynamics within the same context.

Based on the presented scenario, an architecture modularizing the *implementation layer* should be defined. Considering that the source code is implemented in Java, object-oriented concepts such as *inheritance*, *composition*, *interfaces*, *polymorphism*, and *dependency injections* (Fowler, 2004) are explored. Additionally, structures based on *design patterns* (Gamma, 1995) and *layer patterns* (Fowler, 2002) are employed.

4.1 Layers

It is proposed to divide the implementation layer into three other layers for the architectural process: *infrastructure*, *application*, and *domain* (Fowler, 2002; Evans, 2004). This approach organizes the structure into hierarchical levels to separate responsibilities and simplify system maintenance and evolution. Furthermore, abstractions will be defined to promote modularization and low coupling as they establish clear responsibilities for the application's elements.

Initially, it is necessary to understand the nature of access to the development resources to define the elements responsible for interfacing with these external means in a decoupled manner.

Based on the dynamics shown in Figure 2, it is possible to determine the interfaces and implementations that define the access methods to the *development resources*. For persistence in the file system:

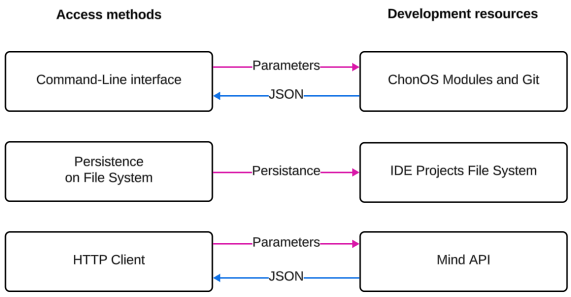

Figure 2: Access methods to development resources.

- *FileRepository*: Interface that defines file manipulation operations such as reading, creating, updating , and deleting (Figure 3).
  - *LocalFileRepository*: Implements file manipulation on the machine system on which the application operates (locally) using native Java libraries.

For executing commands via the prompt, also considering aspects such as remote access via SSH, the following are defined:

- *CommandPromptRunner*: Interface that defines the execution of commands via terminal (Figure 4).
  - *LocalCommandPromptRunner*: Implements the execution of commands on the machine's local terminal on which the application operates (local).
- *SSHHandler*: Interface that defines access to SSH resources (e.g., SFTP) and the remote execution of commands via terminal (extends *CommandPromptRunner*) (Figure 4).
  - *JschSshHandler*: The default implementation of this abstraction, using the *JSCH* library [2].

For access to Web APIs, the following is defined:

- *HttpClient*: Interface that defines communication operations via the HTTP protocol, considering aspects such as request (HttpRequest) and response (HttpResponse): data body (HttpBody); request parameters (HttpQueryParameter); request method (HttpMethod); and the response return code (Figure 5).
  - *StandardHttpClient*: The default implementation of this abstraction, using native Java libraries.

The interfaces presented follow the *ports pattern* (Fowler, 2002) — *CommandPromptRunner* and *HttpClient* — and the *repositories pattern* (Evans, 2004)

[2]Available at http://www.jcraft.com/jsch/

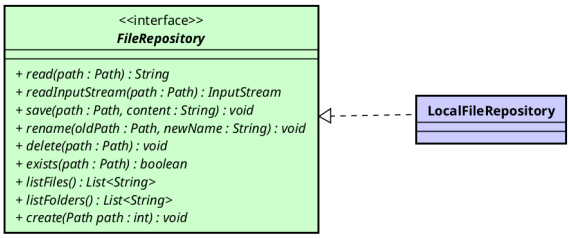

Figure 3: File persistence interface and classes.

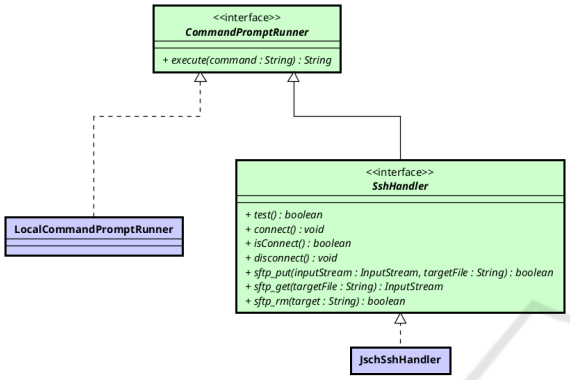

Figure 4: SSH and Prompt Executor interface and classes.

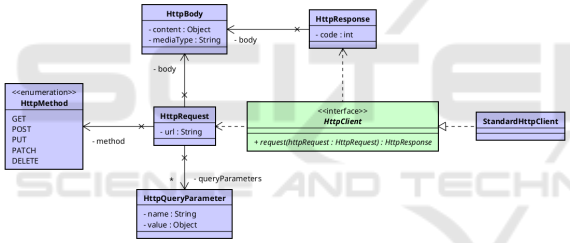

Figure 5: HTTP client interface and classes.

— *FileRepository*. Both abstract technical details and encapsulate the communication that enables access and manipulation of an external resource, providing a single entry point for requests and abstracting the complexity of an underlying system. At the same time, the implementations serve as *adapters* (Fowler, 2002) , determining real behavior within the system influenced by a specific technology or the desired context.

Additionally, a reusable instance is required to convert *json* into object instances (and vice versa), enabling the manipulation of response contents from the Mind API, chonOS modules, and, eventually, Git when applicable. For this purpose, is defined a *strategy* (Gamma, 1995) that provides access to this implementation, and a singleton (Gamma, 1995) that ensures a unique instance for the system:

- *JsonConverter*: Interface that defines the operations for serializing and deserializing JSON content to instances of application classes (Figure 6).

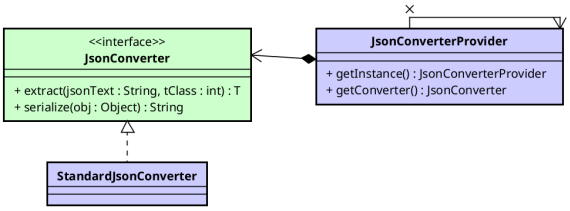

Figure 6: Json classes.

  - *StandardJsonConverter*: The default implementation of this abstraction, using native Java libraries (Corporation, 2024).
- *JsonConverterProvider*: Singleton that provides a unique instance of *JsonConverter* for the application (Figure 6).

Considering that the implementations defined — adapters and strategies — are purely technical and disconnected from any business logic, they should be directed to the *infrastructure layer*, which is responsible for providing the necessary technical and utility services for the other layers of the architecture (application and domain), such as communication with external services, data persistence, and utility elements. Thus, it is possible to decouple the technical implementation for external access from the rest of the application.

Moreover, an *application layer* is established, which, through *application services*, is responsible for orchestrating use cases and coordinating activities between elements that provide access to the *development resources*. Thus, by merely determining these two layers (infrastructure and application) and their elements, it is possible to provide access to the Mind API, chonOS, and, in the future, Git, as external services define their operating mechanics, in other words, they are not part of the IDE application domain, so do not require extensive manipulation of returned content or internal logic.

However, in the case of managing project file systems, business rules need to be added since its mechanics will be defined by the application, given that it is a development resource within the IDE domain. For this case, a *domain layer* is defined, with its respective *domain services*, which encapsulate a set of operations containing system logic for managing multiple entities or data sources, and *domain models* (Evans, 2004), which are abstractions designed to reflect business concepts and behaviors and serve as a cohesive core for organizing and expressing system knowledge.

In this way, a decoupled architecture between its elements and modularized layers is achieved, allowing the system's scalability, as demonstrated in Figure 7. From this general definition, the next subsections

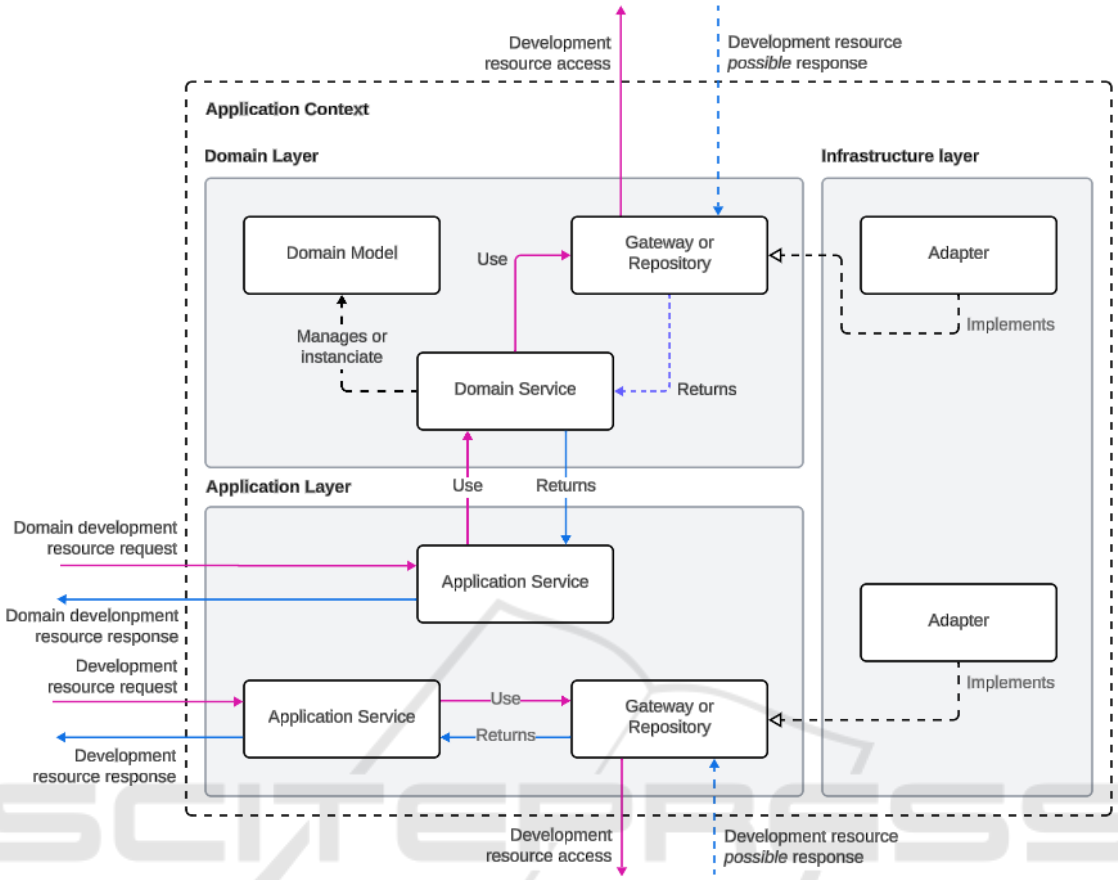

Figure 7: Application architecture.

will detail the specific abstractions and their use cases for accessing the development resources.

4.2 chonOS

The chonOS distribution provides functionalities that streamline the deployment and management process of Embedded MAS by introducing a set of modules accessed via *Shell Script*, which are as follows:

- *chonosEmbeddedMAS*[3] promotes the management of BDI-based Embedded MAS at the reasoning layer of the system through the use of interpreters supported by the solution— *Jason* (Bordini et al., 2007), *JaCaMo* (Boissier et al., 2020), or *Jason Embedded* (Pantoja et al., 2023), allowing the project to be deployed, started, and stopped.
- *chonosFirmwareManager*[4] enables communication with the *interface* and *firmware* layers, thus facilitating the deployment and updating of source code for Arduino boards, the detailed display of information about boards connected to the hardware, and the import, removal, and listing of libraries.
- *chonosDDNSManager*[5] allows the definition and updating of the *Fully Qualified Domain Name* (FQDN) for the device, thereby eliminating the need for the designer to know the embedded hardware's IP address on the current network.
- *chonosWifiConn and chonosWifiConf*[6] enable the management of embedded hardware connectivity. They allow connection to a network in client mode, the creation of a private network in AP mode (with or without encryption), and the detailed display of the connected network as well as other nearby networks.
- *chonosNeighbors*[7] allows the identification of all embedded hardware devices on current network.

[3]https://os.chon.group/masmng/
[4]https://os.chon.group/firmwaremng/
[5]https://os.chon.group/ddnsmng/
[6]https://os.chon.group/networkmng/
[7]https://os.chon.group/neighborsmng/

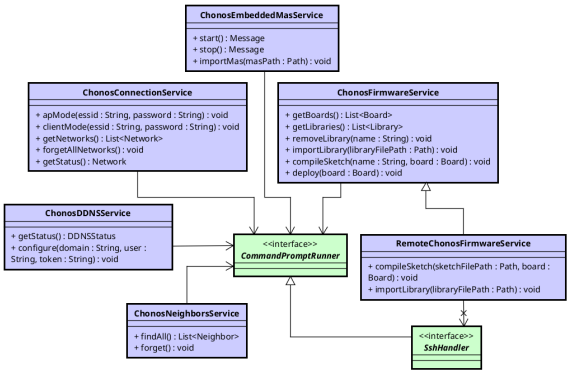

Figure 8: ChonOS services.

Based on this, each of the mentioned modules is considered a *development resource* and must have corresponding access abstractions. This will be reflected in services within the *application layer* that receive dependency injections from *CommandPromptRunner* and *JsonConverter*. These services allow module execution via command prompt — either locally (*LocalCommandPromptRunner*) on the designer's computer or remotely (*JschSshHandler*) on the embedded hardware — and serialize the response into object instances for straightforward manipulations, as illustrated in Figure 8.

4.3 Mind API

The Mind API is a REST API that provides information about the current state or a specific cycle of the agents' minds in the MAS executed by the *Jason Embedded* interpreter. To enable its access as a *development resource*, a defined *service* within the *application layer* receives dependency injections from *HttpClient* to consume the API's *endpoints* and *JsonConverter* for response deserialization, as illustrated in Figure 9.

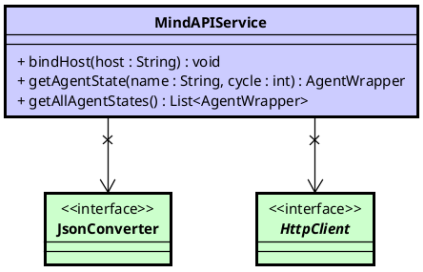

Figure 9: Mind API Service.

4.4 IDE Project File System

The modeling was designed to support the use of the *Jason Embedded* language and enable the extensibility needed for integration with the *JaCaMo* platform, providing a solid foundation for projects that require modularity, scalability, and alignment with agent-oriented programming paradigms.

The new structure emerged from the need to reorganize folders and files in the IDE to improve project control and management. In the previous chonIDE, the entire project was stored in a single *.json* file, hindering modularity and the integration of new functionalities. The proposed structure was developed to focus on future compatibility with the *JaCaMo* platform, an open-source technology for MAS that organizes projects across three dimensions: agents, environments, and organization. These dimensions are implemented using *Jason*, for programming autonomous agents; *CArtAgO*, which models and manages artifacts in the environment; and *MOISE*, which specifies and regulates multi-agent organizations (Boissier et al., 2020). Figure 10 illustrates the planned folder structure, including directories such as *agent*, *environment* (subdivided into *endogenous* and *exogenous*), and *organization*, alongside example files such as *agent.asl*, *artifact.java*, *file.ino*, *lib.zip*, and project configuration files (*project.jcm* or *project.mas2j*).

The folder structure modeling within the IDE domain organizes Embedded MAS projects across the three dimensions and includes a project configuration file. Figure 11 illustrates this architecture, highlighting the relationship between the *Project* and *EmbeddedMAS* classes, representing a project in IDE and an Embedded Multi-Agent System, respectively. The *EmbeddedMAS* class directly connects the three main dimensions, represented by the *Agency*, *Environment*, and *Organization* classes, to the project configuration file. The *MasConfiguration* interface abstracts various configuration file formats, ensuring flexibility to support both *Jason Embedded* and *JaCaMo* in the future. Additionally, the *ProjectFile* interface pro-

Figure 10: Proposed new folder structure.

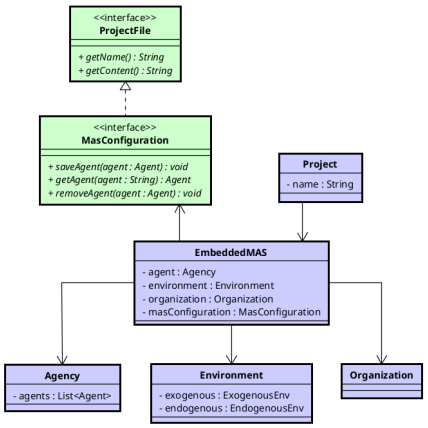

Figure 11: Class diagram of the project's main structure.

vides the necessary abstraction for managing the diverse files comprising a project. The project's concept classes are:

- **Project.** Represents a project in IDE, organizing projects by providing a name to identify them and linking them to an *EmbeddedMAS*.
- **EmbeddedMAS.** Represents an Embedded MAS, connecting the dimensions of agents, environments, and organizations, along with the project configuration file. This class centralizes the logical structure of an Embedded MAS, allowing the dimensions to interact cohesively and adapt to the system's specificities.
- **ProjectFile.** An interface that abstracts the different types of files making up the project. This abstraction simplifies the integration of various files, such as agents, artifacts, firmware, and project configuration files, ensuring flexibility and modularity in IDE's structure.

The dimensions of agents, environment, and organization, as well as configuration files, were structured independently to reflect the characteristics of an Embedded MAS. This approach allows each dimension to be represented by specific classes that capture their responsibilities and interactions, while the configuration layer centralizes the information necessary for system initialization and operation. Subsequent paragraphs will detail each of these parts, accompanied by diagrams illustrating their relationships.

The agent dimension organizes agents modularly, allowing each to be treated as an independent unit within the project. This structure enables the integration of different types of agents into the system, reflecting the specific demands of an Embedded MAS. Figure 12 presents the class diagram for this dimension, highlighting agents and their respective classifications.

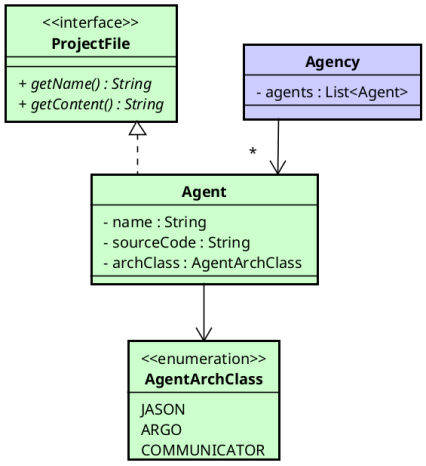

Figure 12: Agent dimension.

The agent dimension is represented by:

- **Agency.** Represents the layer responsible for grouping the agents of an Embedded MAS. It centralizes agent management and allows new agents to be easily added to the system.
- **Agent.** Defines an individual agent within the project and is modeled as an extension of the *ProjectFile* interface, representing **.asl** files in the system's folder structure. This approach ensures each agent is stored independently, simplifying the organization and management of agents directly at the file level. Furthermore, the class encapsulates basic agent information, such as identity and source code, and establishes the relationship between the agent and its architecture, enabling the coexistence of different agent types within the same system.
- **AgentArchClass.** An enumeration that categorizes possible agent architectures. This enumeration includes Jason representing standard agents based on the BDI (Belief-Desire-Intention) paradigm, widely used in MAS; Argo denoting agents with specific capabilities for interacting with hardware or physical components, such as sensors and actuators (Pantoja et al., 2016); and Communicator representing agents specialized in communication between different MAS, ensuring interoperability and efficient information exchange (Pantoja et al., 2018).

The environment dimension organizes the elements constituting the physical and logical context in which the agents of an Embedded MAS operate. This organization reflects the need for modular separation between external aspects, related to the physical world, and internal ones, which are computational abstractions used by agents to logically and efficiently

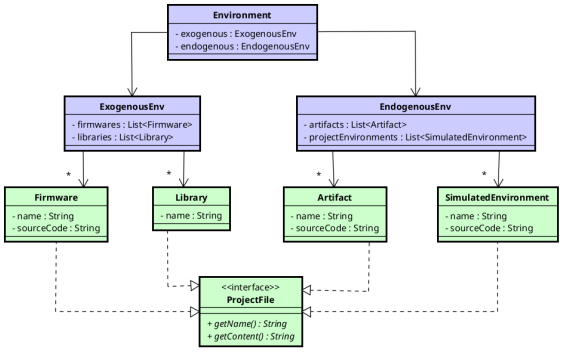

Figure 13: Environment dimension.

interact with the environment. Figure 13 presents the class diagram for this dimension, illustrating how the *Environment* class organizes the two types of environments — exogenous (*ExogenousEnv*) and endogenous (*EndogenousEnv*) — along with their respective components.

The agent environment is represented by:

- **Environment.** The main class representing the environment dimension in the project. It encapsulates references to the exogenous and endogenous environments, centralizing their management and allowing integrated manipulation.
- **ExogenousEnv.** Represents the exogenous environment, which includes external elements such as physical devices and libraries required for operation (Ricci et al., 2012). This implementation contains references to the *Firmware* and *Library* classes, which represent files in this environment.
- **Firmware.** Project files (*.ino*) that encapsulate the logic for controlling physical devices, such as sensors and actuators. These files implement the logic required to operate hardware components.
- **Library.** External library files used in the exogenous environment to extend firmware functionality and integrate additional resources, such as hardware communication support.
- **EndogenousEnv.** A computational abstraction providing resources and functionalities to support agent tasks within the system's context. It reflects the internal, logical context, enabling agents to interact with the environment directly (Ricci et al., 2012). This implementation includes references to the *SimulatedEnvironment* class and an extensible architecture to include *Artifacts* in the future, allowing integration with the *CArtAgO* platform.
- **Artifact.** Represents artifacts defined in the environment. These components abstract functionalities or resources from the exogenous environment, enabling agents to interact with them logically and in a controlled manner. This resource type will become available with the future implementation of the *CArtAgO* platform, which provides a foundation for modeling and managing artifacts in MAS environments.
- **SimulatedEnvironment.** Additional files that configure or customize the endogenous environment within a project. These files may include specific definitions for connecting agents to the environment or customizing their interaction.

The organization dimension (*Organization*) encompasses the elements responsible for structuring an Embedded MAS project's organization, centralizing definitions related to agent interactions, roles, and hierarchies within the system (Boissier et al., 2020). While the development and implementation of this dimension are planned for future work, its importance lies in supporting organizational frameworks, such as *MOISE*, which provide a formal foundation for modeling multi-agent organizations.

The Model of Organization for Multi-Agent Systems (*MOISE*) (Hübner et al., 2002) is a platform that offers a formal model for representing organizations in MAS. It is based on three main dimensions: structure, which defines roles and hierarchical relationships among agents; functionality, which specifies the goals and tasks to be achieved; and regulation, which describes the plans and norms governing agent actions. The use of *MOISE* in MAS development enhances their organization and governance (Bordini et al., 2007).

The project configuration layer organizes the essential elements required to define and initialize an Embedded MAS. It centralizes fundamental configuration properties and allows the inclusion of a single configuration file per project. Figure 14 illustrates the class diagram for this layer, highlighting the *MasConfiguration* interface and its possible implementations:

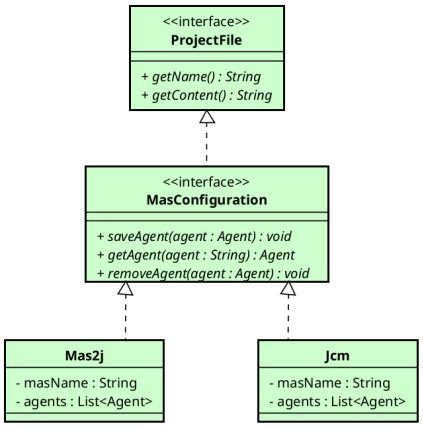

Figure 14: Project configuration layer.

Mas2j and *Jcm*.

- **MasConfiguration.** An interface extending *ProjectFile* that abstracts project configuration files. It establishes a unified standard for structuring the necessary information, facilitating the management of specific configurations.
- **Mas2j.** Represents project configuration files in the *.mas2j* format, used in projects developed with *Jason Embedded*. This type of file specifies agents and their basic configurations, offering an organized structure for defining and initializing system components.
- **Jcm.** Represents project configuration files in the *.jcm* format, applicable to projects that will use the *JaCaMo* platform. This file enables the integration of agents defined in *Jason Embedded* with environmental artifacts and organizational structures, allowing synchronization across the system's different levels.

The new file structure implemented in IDE organizes the elements of Embedded MAS into independent dimensions of agents, environment, and organization. This modular approach allows each dimension to be represented with specific files, facilitating project management and expansion. The introduction of a centralized project configuration layer abstracts the *.mas2j* and *.jcm* formats, offering support for both *Jason Embedded* and, in the future, *JaCaMo*.

This reorganization enables a clearer separation between the system's various components, improving internal structure and project clarity. Additionally, dividing elements into distinct files simplifies the implementation of new functionalities and integration with additional tools. With this modular structure, the development and application of a *Git-based version control system* become feasible, allowing for more specific and organized tracking of changes. This restructured file base prepares IDE to support future functionalities, promoting an organization that reflects the requirements of Embedded MAS while facilitating project maintenance and evolution.

The *Domain Service* module in ChonIDE implements file, project, and path management services modularly. This chapter details the *ProjectFileService* interface and its implementations, highlighting the *Decorator* design pattern, which enables flexible functionality extension, and the *Singleton* pattern, which ensures a single controlled instance of a service. It also covers the *ProjectService* and *ProjectPathHandler* interfaces and their implementations, which collaborate to maintain consistency and extensibility in project element handling. Figure 15 presents the class diagram of these interfaces, their implementations, and their relationships.

The *ProjectFileService* interface defines basic operations such as loading, saving, deleting, and renaming project files. *StandardProjectFileService* implements this interface using *ProjectPathHandler* to obtain the correct file paths. For specific operations with agent files, *AgentFileService* also implements *ProjectFileService*, ensuring that agent files are handled differently, interacting not only with the agent file itself but also with the project configuration layer.

In the file management service implementation, the *IntercepterProjectFileService* acts as a *decorator* (Gamma, 1995) by delegating operations to *StandardProjectFileService* or *AgentFileService* based on the type of file being managed. This approach provides flexibility in handling different file types without directly modifying existing services. In IDE, *ProjectPathHandler* uses *Singleton pattern* (Gamma, 1995) through the *ProjectPathHandlerManager* class to centralize the definition of file and project paths,

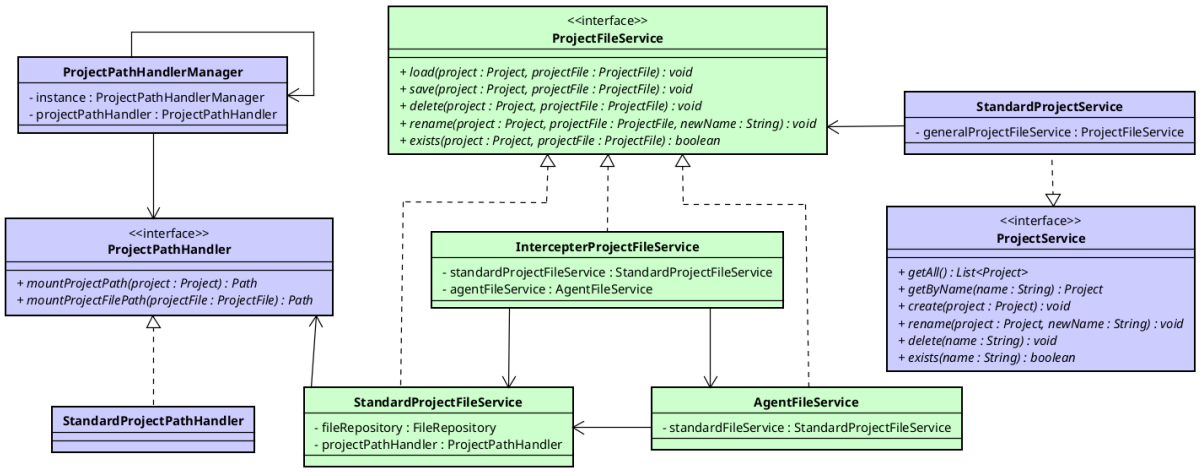

Figure 15: File management service module.

ensuring all operations dependent on these paths are consistent and synchronized.

The *Service* module structure in IDE organizes path, project, and file management modularly. The *ProjectFileService*, *ProjectService*, and *ProjectPathHandler* interfaces, along with their implementations, provide extensibility and consistency in project element handling. This approach simplifies IDE maintenance and evolution, enabling the integration of new features without compromising the existing structure, thus ensuring a solid foundation for IDE's continuous development.

5 REPRODUCIBILITY

Aiming to ensure the reproducibility of this work, the source code and an implementation of the new architecture are available on the paper's page[8].

6 CONCLUSIONS

This work established a modular architecture for the chonIDE codebase, enhancing its flexibility and extensibility to enable software scalability through the reorganization of its components. Furthermore, a folder and file structure was implemented to replace the previous single-file project structure, promoting interoperability with other IDEs and enabling code versioning—features previously unattainable due to the proprietary project file format being manipulable only within the IDE itself. Thus, it is understood that with the proposed changes, chonIDE is better equipped to meet market demands, thereby reducing software engineering challenges in the context of Embedded MAS.

6.1 Future Works

As future works, the goal is to explore another aspects of chonIDE's adaptability to market demands: integration with Language Servers (LSP) to enable syntax checking and highlighting, code autocompletion, and quick assistance; the evolution of the Mind Inspector into a comprehensive control tool for a running MAS, allowing developers to insert and modify elements of agent logic (beliefs, intentions, plans, etc.) through a more interactive interface; and the implementation of project code versioning through Git, integrated into the IDE.

[8]https://papers.chon.group/ICEIS/2025/modularizedArchChonIDE/

REFERENCES

Boissier, O., Bordini, R. H., Hubner, J., and Ricci, A. (2020). *Multi-agent oriented programming: programming multi-agent systems using JaCaMo*. Mit Press.

Bordini, R., Hübner, J., and Wooldridge, M. (2007). *Programming Multi-Agent Systems in AgentSpeak using Jason*. Wiley Series in Agent Technology. Wiley.

Bordini, R. H., Hübner, J. F., and Vieira, R. (2005). Jason and the golden fleece of agent-oriented programming. In Bordini, R. H., Dastani, M., Dix, J., and El Fallah Seghrouchni, A., editors, *Multi-Agent Programming: Languages, Platforms and Applications*, pages 3–37, Boston, MA. Springer US. DOI: 10.1007/0-387-26350-0_1.

Corporation, O. (2024). Java Development Kit (JDK). https://www.oracle.com/java/technologies/javase-downloads.html.

Evans, E. (2004). *Domain-Driven Design: Tackling Complexity in the Heart of Software*. Addison-Wesley Professional, Boston, MA.

Fowler, M. (2002). *Patterns of Enterprise Application Architecture*. Addison-Wesley.

Fowler, M. (2004). Inversion of control containers and the dependency injection pattern. https://martinfowler.com/articles/injection.html.

Gamma, E. (1995). *Design patterns: elements of reusable object-oriented software*. Addison-Wesley.

Hübner, J. F., Sichman, J. S. a., and Boissier, O. (2002). Moise+: towards a structural, functional, and deontic model for mas organization. In *Proceedings of the First International Joint Conference on Autonomous Agents and Multiagent Systems: Part 1*, AAMAS '02, page 501–502, New York, NY, USA. Association for Computing Machinery. DOI: 10.1145/544741.544858.

Jennings, N. R. (2001). An agent-based approach for building complex software systems. *Commun. ACM*, 44(4):35–41. DOI: 10.1145/367211.367250.

Jesus, V., Lazarin, N., Pantoja, C., Manoel, F., Alves, G., Ramos, G., and Filho, J. V. (2022). Proposta de uma IDE para desenvolvimento de SMA Embarcados. In *Proceedings of the 16th Workshop-School on Agents, Environments, and Applications*, pages 49–60, Porto Alegre, RS, Brasil. SBC. https://sol.sbc.org.br/index.php/wesaac/article/view/33425.

Lazarin, N., Pantoja, C., and Viterbo, J. (2023). Towards a Toolkit for Teaching AI Supported by Robotic-agents: Proposal and First Impressions. In *Proceedings of the 31st Workshop on Computing Education*, pages 20–29, Porto Alegre, RS, Brasil. SBC. DOI: 10.5753/wei.2023.229753.

Pantoja, C., Soares, H. D., Viterbo, J., and Seghrouchni, A. E. F. (2018). An Architecture for the Development of Ambient Intelligence Systems Managed by Embedded Agents. In *30th International Conference on Software Engineering and Knowledge Engineering*, pages 215–220. DOI: 10.18293/SEKE2018-110.

Pantoja, C. E., Jesus, V. S. d., Lazarin, N. M., and Viterbo, J. (2023). A Spin-off Version of Jason for IoT and

Embedded Multi-Agent Systems. In Naldi, M. C. and Bianchi, R. A. C., editors, *Intelligent Systems*, pages 382–396, Cham. Springer Nature Switzerland. DOI: 10.1007/978-3-031-45368-7_25.

Pantoja, C. E., Stabile, M. F., Lazarin, N. M., and Sichman, J. S. (2016). ARGO: An Extended Jason Architecture that Facilitates Embedded Robotic Agents Programming. In Baldoni, M., Müller, J. P., Nunes, I., and Zalila-Wenkstern, R., editors, *Engineering Multi-Agent Systems*, pages 136–155, Cham. Springer. DOI 10.1007/978-3-319-50983-9_8.

Ricci, A., Santi, A., and Piunti, M. (2012). Action and Perception in Agent Programming Languages: From Exogenous to Endogenous Environments. In Collier, R., Dix, J., and Novák, P., editors, *Programming Multi-Agent Systems*, pages 119–138, Berlin, Heidelberg. Springer. DOI: 10.1007/978-3-642-28939-2_7.

Siqueira, E., Ramos, G., Raboni, T., Pantoja, C., and Lazarin, N. (2024). Análise Comparativa de um Protótipo de IDE para Desenvolvimento de SMA Embarcados. In *Proceedings of the 18th Workshop-School on Agents, Environments, and Applications*, pages 85–95, Porto Alegre, RS, Brasil. SBC. https://sol.sbc.org.br/index.php/wesaac/article/view/33458.

Souza de Jesus, V., Mori Lazarin, N., Pantoja, C. E., Vaz Alves, G., Ramos Alves de Lima, G., and Viterbo, J. (2023). An IDE to Support the Development of Embedded Multi-Agent Systems. In Mathieu, P., Dignum, F., Novais, P., and De la Prieta, F., editors, *Advances in Practical Applications of Agents, Multi-Agent Systems, and Cognitive Mimetics. The PAAMS Collection*, pages 346–358, Cham. Springer Nature Switzerland. DOI: 10.1007/978-3-031-37616-0_29.

Tennenhouse, D. (2000). Proactive computing. *Commun. ACM*, 43(5):43–50. DOI: 10.1145/332833.332837.

Wooldridge, M. (2009). *An Introduction to Multiagent Systems*. John Wiley & Sons, Chichester, U.K, 2nd edition.

Wooldridge, M. and Jennings, N. R. (1995). Intelligent agents: theory and practice. *The Knowledge Engineering Review*, 10(2):115–152. DOI: 10.1017/S0269888900008122.

Zambonelli, F., Jennings, N. R., and Wooldridge, M. (2003). Developing multiagent systems: The gaia methodology. *ACM Trans. Softw. Eng. Methodol.*, 12(3):317–370. DOI: 10.1145/958961.958963.

Zambonelli, F. and Van Dyke Parunak, H. (2003). Signs of a revolution in computer science and software engineering. In Petta, P., Tolksdorf, R., and Zambonelli, F., editors, *Engineering Societies in the Agents World III*, pages 13–28, Berlin, Heidelberg. Springer Berlin Heidelberg. DOI: 10.1007/3-540-39173-8_2.