Designing Modular Agent Systems

Diego Bonura[2], Leonardo Mariani[1], and Emanuela Merelli[2]

[1] DISCO - Università degli Studi di Milano Bicocca
via Bicocca degli Arcimboldi, 8, I-20126 Milano, Italy
mariani@disco.unimib.it
[2] Dipartimento di Matematica e Informatica, Università di Camerino
via Madonna delle Carceri 9, I-62032 Camerino, Italy
{diego.bonura, emanuela.merelli}@unicam.it

Abstract. The paper contributes to research on component and multi-agent systems by presenting a practical approach to the development of a modular and reusable middleware. In particular, we address the problem of the construction of the *core* of a middleware for MAS. Then we introduce two case studies for two different application domains: biological data integration and quality assurance in manufacturing. Our experience proves that the component-based approach provides several benefits, such as the facilitation of refactoring and reusability, but also introduces some pitfalls, such as excessive reuse. We acknowledge that reuse exploits its potential in the lower layers of a system, because components are quasi-free of business level concepts.

Keywords: Software Architectures, Component-Based Systems, Component Composition, Multi-Agent Systems

1 Introduction

The agent technology has been successfully applied in several contexts i.e., manufacturing [1], information retrieval [2], and e-commerce [3], but today it is still difficult to develop a Multi-Agent System (MAS) due to its complexity. Several methodologies supporting the developer exist [4–6], and several middleware and implemented MAS are available [7, 8], but none of these is tailored to directly support management of updates and to fulfil of new requirements. These systems are often described by an architectural style and a more or less accurate object-oriented design, but do not explicitly support configurability and reusability.

Because MASs are so complex, it is preferable and more convenient to reuse existing artefacts rather than to develop a system from scratch. Reuse and composition can be efficiently addressed by suitable technologies, and we propose adopting the well known and widely used component-based approach [9]. By composing components, we create several MASs each addressing a very different set of requirements. Requirements drive the components selection and composition processes. In particular, requirements are used to arrange suitable

components at different abstraction levels. The developer designs component integration and implements additional components so as any necessary glue code. The goal is to obtain a modular system with several implemented reusable components organized in different abstraction levels.

We focus on the middleware construction and we propose a design and an implementation plan to serve as a preliminary approach to the construction of modular and easily extendible systems. Effectiveness of the approach is evaluated by altering some requirements on the implemented MAS, so as to provoke the addition and the substitution of some components.

In this paper, we concern with the middleware level implications of the application domain and we do not take into account the organization of the agents. For a more readily comprehensible understanding of the middleware we remark that at the application level the problem of the heterogeneity is addressed by providing each host with a set of service agents that publish data in a comprehensible format using technologies like wrappers [10] and ontologies [11].

Section 2 presents basic aspects of data integration in the Biological domain and introduces some of its implications on middleware construction. The complete design and implementation experience is detailed in Section 3. The evaluation of the obtained system is presented in Section 4. To provide evidence of the possibility of reusing components in different contexts, we illustrate, in Section 5 the possible results of developing a middleware for MAS in a manufacturing context (a very different domain). Finally, in Section 6 we present similar experience and we conclude in Section 7.

2 The biological domain - 1° scenario

In the present post-genomic era, biological information sources are crammed with information gathered from results of experiments performed in laboratories around the world, i.e., sequence alignments, hybridization data analysis or proteins interrelations. The amount of available information is constantly increasing, and it is difficult to exploit available data from all sources [12]. Usually, information must be collected and integrated to gain evidence of the correlations between cause and effect. The amount of available information, its wide distribution and the heterogeneity of the sources make it impossible for bioscientists to manually perform these operations and also make it difficult for automatic tools to access all these information sources. Moreover, we might have users who possess their own proprietary algorithm, but have no data to apply it. On the other hand, a data provider might have data, but not the algorithm. Existing services offer no easy or efficient solution to this problem. The scenario we outline provides a natural application for the mobile code. Even if several services integrating multiple data sources have been proposed [13, 14], we wish to explore a different approach: Multi-(Mobile)Agent Systems.

The goal of the application is to implement a retrieval system capable of specifying complex selection criteria, possibly by combining results from different heterogeneous sources (e.g. PDF papers and HTML pages) or by performing

different workflows of activities that depend upon intermediate results. In this scenario, mobile agents provide the technological solution that crosses enterprise boundaries and arrives nearest resources which provide data. For this reason, security, data integrity and reliability are particularly important. Security prevents agents from performing dangerous operations; data integrity assures that interactions leave data free from faults; and finally, reliability assures that services accessed from agents behave correctly and that local system failures lowly affect user agents. Mobile agents have been chosen to exploit the computational power of servers storing data and to enhance performances by providing agents with the ability to filter data and to decide their actions based upon intermediate results. Thus, the MAS uses all resources of the network in an approximately optimized way. Preferably, pools of mobile agents coordinate their actions, so that each agent can interact with other agents of the same pool. Finally, the system must implement an account management service enabling the user to be authenticated locally to each data provider.

In following sections, we will focus on components located at the lower layers of the system to illustrate the advantages of implementing the core of the application as a component-based system.

3 System design and implementation

MASs are complexes [15], often involving distribution, mobility, communication and security.

To master this complexity we suggest the adoption of a layered software architecture. The use of a layered architecture style facilitates mastering of complexity and enhances security because interactions occurring among different layers can be monitored and filtered. In order to support also to flexibility, we decided to adopt a layers plus components strategy where each layer is designed by component composition.

We think that our point of view is a natural and effective approach to middleware construction and, more generally, to the development of complex systems. In the following paragraphs we report a detailed design showing the feasibility and the effectiveness of our approach. We have chosen UML as architecture description language because is widely accepted in both the academic and industrial worlds as a reference language for system design, and the numerous tools and technologies based upon it constitute a background adequate to support a component-based approach to middleware construction.

3.1 The Layered View

We propose the three-layered architecture in Figure 1 based on the general architecture introduced in [16].

The *Core layer* role is similar to the kernel of an operating system, and it implements basic features, such as communication protocols, agent traceability and security. The *Core layer* is essentially free of any system strategy. The

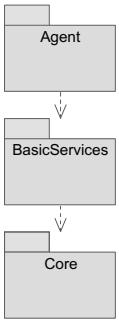

Fig. 1. Layered Architecture

BasicServices layer extends low level features providing services that directly support the agent activities, e.g., mobility and agent communication is implemented on top of inter-place communication. The *BasicServices layer* contains system strategies, but does not implement any application level feature. The *Agent layer* is the container of all service and user agents of the business application. The *BasicServices layer* is always present in any node, so that minimum support to agent execution is guaranteed.

3.2 Core Layer

The *Core* layer is the lowest layer of the architecture (Figure 2) and contains base functions of the system, such as the implementation of the inter-platform communication protocols and agent management functions. This layer is composed of four components: *ID*, *SendReceive*, *Starter* and *Security*.

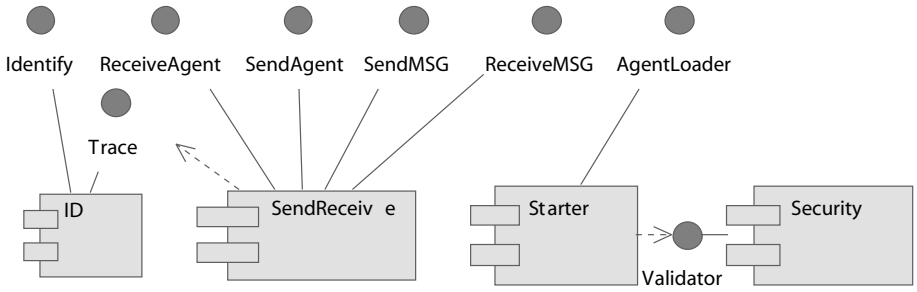

Fig. 2. The Core Layer

The *ID* component implements general identity management functions by managing a repository containing information about locally generated agents

(Figure 3). This repository is accessed whenever we want to know the current position of an agent.

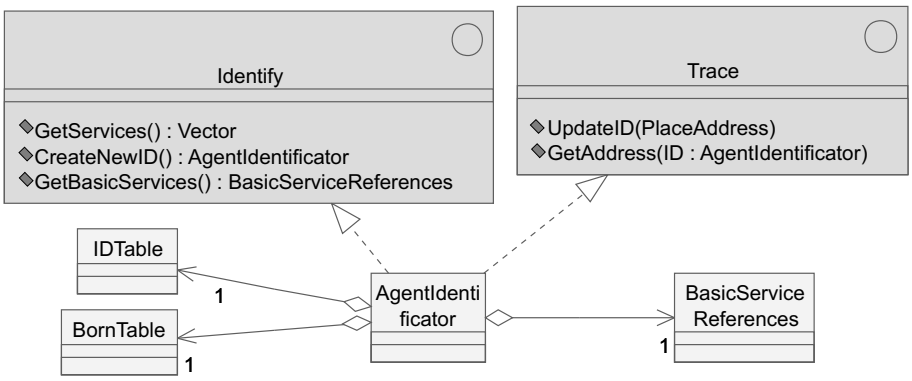

Fig. 3. ID Component

The *ID* component is also responsible for the creation of the identifiers to be associated to new agents. These identifiers contain information about the birthplace, date and time of the agent's creation. Agent localization is simplified by information contained directly in the "ID", such as the birth place. In fact, the birth place of an agent hosts information about the agent's current location.

A second important feature of the *Core* is the *SendReceive* component (Figure 4). This component implements low level inter-platform communication by

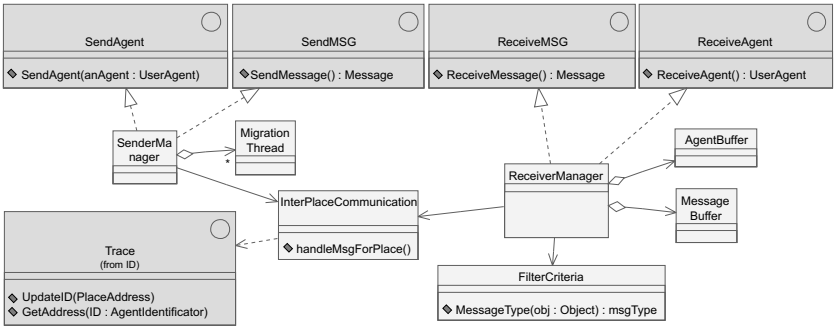

Fig. 4. SendReceive Component

sending and receiving messages and agents. By using the traceability services of-

fered by the *ID* component, *SendReceive* can easily update or retrieve the exact position of a specific *user agent*.

It is important to note that every change in the communication protocol is hidden within the *BasicService layer*. The *SendReceive* component can also send and receive agent instances. This feature is reused by the upper layer to implement agent migration.

The *Starter* component processes any request for agent creation. This particular component, in fact, takes an inactive agent (just created or migrated), and checks it for the absence of malicious or manipulated code. These agents, before activation, are dynamically linked to all basic services of the platform. During execution the agent is isolated from the *Core layer* by the *Basic Service layer*.

The *Security* component, as mentioned above, checks for the presence of malicious code or manipulations within the agent code. Note that at this abstraction level permissions are not an issue. The code inspection concerns only dangerous agents that attempt to perform illegal operations, such as viruses.

3.3 The BasicService Layer

BasicServices layer (Figure 5) has five main components: *Discovery*, *Mobility*, *Genesis*, *Communication* and *Security Politics*.

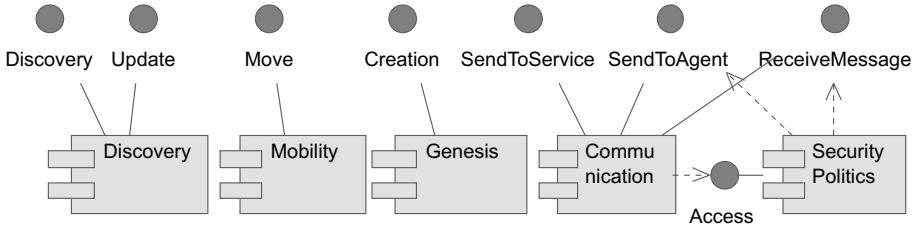

Fig. 5. BasicServices Layer

The *Discovery* component searches and detects service agents. When a user agent wants to communicate with a service, it will ask the *Discovery* for the right identifier to use as the message's receiver. The service detection strategy can be implemented in different ways; for example by a fixed taxonomy or by an UDDI [17], commonly used in the Web Services application domain.

The *Mobility* component enables the movement of code across platforms [18], it implements the interface used by the *UserAgent* and it accesses to components of the *Core layer* to send, receive and load agents. It is important to note that real communication between different locations can be achieved only through *Core*'s *SendReceive* component, and then migration is independent of the type of used transport.

The *Communication* component (Figure 6) makes possible to send and receive agent-directed messages both in an intra- and inter-platform context, it can implement the Agent Communication Language (ACL) [19]. Intra-platform messages are messages sent between agents and services residing in the same platform. Inter-platform messages are messages sent to agents residing in different platforms (our system does not allow for remote communication between user agents and service agents).

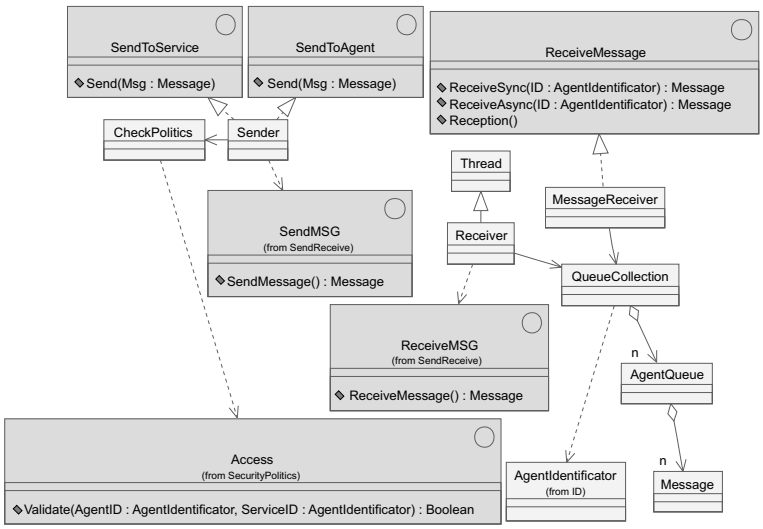

Fig. 6. Communication Component

The agent requesting the dispatch of a message does not need to know, effectively, where the target agent is; in fact, the ID is sufficient to post correctly a message. The *Communication* component uses one of the *Security Policy*'s interfaces to check whether the specific *UserAgent* or *ServiceAgent* has the right privileges for communication, if an *Agent* is not authorized to use a service, the message is destroyed.

Before accessing resources and services, an agent must authenticate itself. The identification is performed by sending a login message to a specific *ServiceAgent*, as consequence the *SecurityPolitics* component (Figure 7) jointly with the *Communication* component intercept the message and unlock the communication. The *SecurityPolitics* component centralizes control of permissions, protects services and resources from the user agents, and provides the administrator with an easy way to manage all permissions.

The last component of the service layer is the *Genesis* component that enables agent creation. A special case of agent creation is cloning that is performed

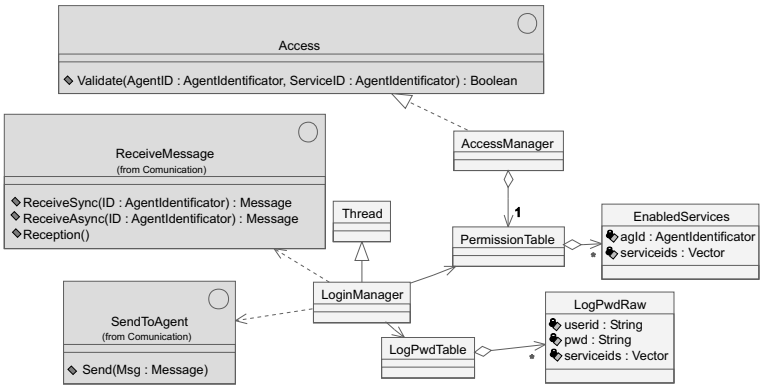

Fig. 7. SecurityPolitics Component

when it is necessary to create a copy of an existing agent. The two copies differ only for the agent identifier.

3.4 The Agent Layer

The last layer of the platform, the *Agent Layer*, contains all service and user agents. This layer implements features like workflows management [20], ontology management [11] or integration of heterogeneous resources [21]. Generally, the business logic of the application is implemented at the *Agent layer*.

The only component in this layer is the *Agent* component (Figure 8). This

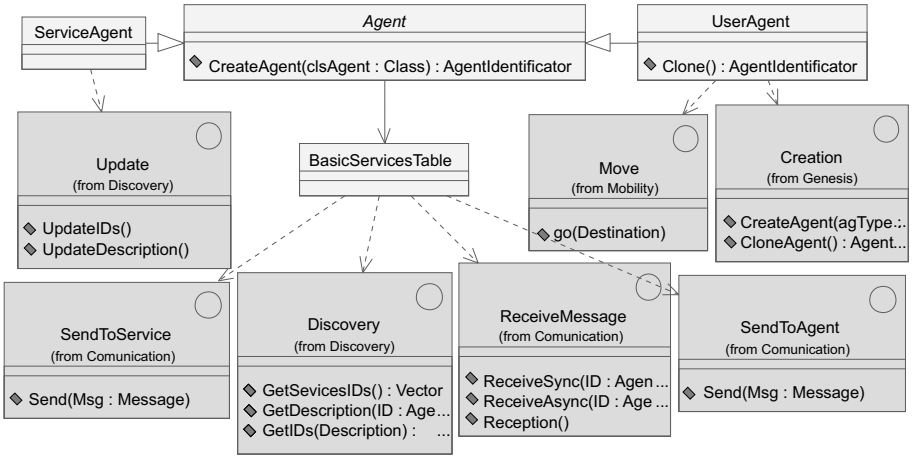

Fig. 8. Agent Component

component has not any interface, but is has only several dependencies upon the *BasicService layer*. The Agent component contains a general abstract agent class and two inherited classes. *ServiceAgent* consists of agents enabling access to biological databases or providing algorithm. *UserAgent* represents agents created by biologists. User agents execute complex tasks and implement part of the logic of the application.

The "component view" of the final design of the system is shown in Figure 9.

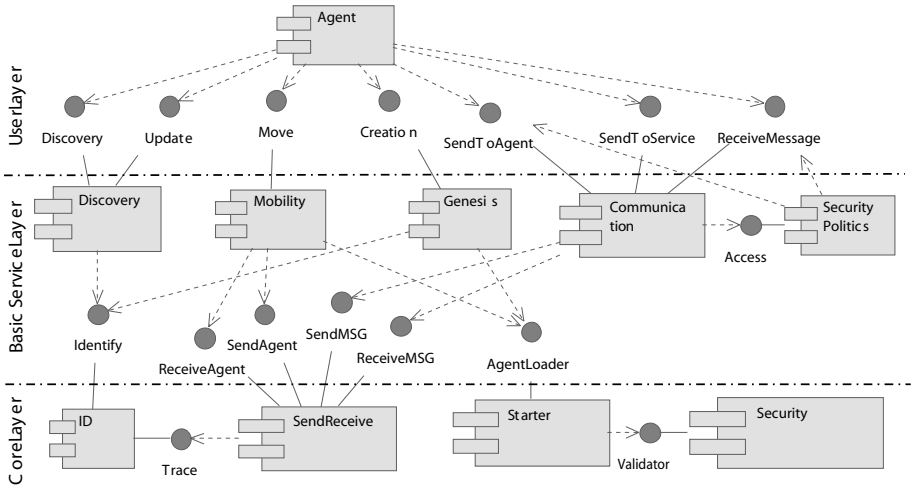

Fig. 9. Component Interfaces and Dependencies

We implemented a prototype of the system in Java. The resulting system yields a high degree of freedom to change the internal composition of components, but as a cornerstone, any update affecting the interface rapidly propagates to other components leading to a difficult implementation of the glue code. In such cases, we implement new families of components that support the new set of interfaces. More details about the implemented system and issues concerning reuse are addressed in the following two Sections.

4 System Evaluation

The middleware we implemented is separated into several functional units (components) with mutual dependencies explicitly documented by UML diagrams. In general, the design is produced by a developer that evaluates the validity of results using intuitive and subjective criteria. To confirm any criteria, the developer can apply any of several software metrics to obtain a "measure" of one or more characteristics. In instances, where a requirement is not satisfied or a measure

is not sound, the final result can be improved by applying a disciplined (automatic or not) refactoring technique [22]. In the case of white box components the restyling of the system can be rendered even more effective by changing not only the arrangement of the components, but also their interior composition. In this case, we have white box object-oriented components that enable both intra- and inter-component level refactoring. For example, during the development of the MAS for the biological domain we needed to reshape the *Agent* component. In the first instance, two components existed: *User Agent* and *Service Agent*. These two components are strongly coupled with the *BasicService* layer and are also mutually hard-coupled. In fact, several inheritance links cross the component's boundary, and most of the interfaces are shared. Consequently, we decided to merge these two components into one cleaner and more simple component (the *Agent* component).

Each time the layered approach was violated, such as when a dependence starting from one layer ended in a higher rather then in a lower layer, we restyled the design in one of the two following ways. Where the component was placed in the wrong layer, we simply moved the component to another layer. Where there was a design error, so that the component contained features characteristic of two different layers, we split the component into two subcomponents and we arranged one component for each layer. For example, the two components that implement communication features were originally a unique component which was then separated into two subcomponents, successively arranged in the *Core* and *BasicService* layers.

The component based approach makes it possible to obtain different systems by configuring and composing families of components released in several versions. It is difficult to check the accuracy of all configurations; extensive testing is necessary. In the case of a disciplined development, the component-based approach limits the amount of testing that must be repeated. In fact, once a new component has been located, it is possible to check which components are influenced by this new unit and which components may influence the new component. The Chaining approach [23] is an example of a technique supporting this type of analysis; it consists of the creation of opportune matrixes of dependencies that are used to predict consequences of updates. Looking at Figure 9, we note that by altering the *ID* component we affect the *Discovery*, *Genesis* and *Agent* components. Thus, by altering the method of assigning identifiers, we alter the method of creating, discovering and managing agents. Likewise, by updating the *SendReceive* component, we alter the low level communication and must therefore retest the *Mobility*, *Communication*, *Security Politics* and *Agent* components. This is absolutely justified because these components depend upon how the two local places interact. Updates are more localized in the case of the *Genesis* and *Communication* components. These observations are mainly a consequence of the mixed layered and components architectural style. Figure 10 shows the effect of an update on the system. Altering a high level component produces numerous dependencies that must be rechecked. On the other hand, by altering a low level component, we must monitor for possible side effects.

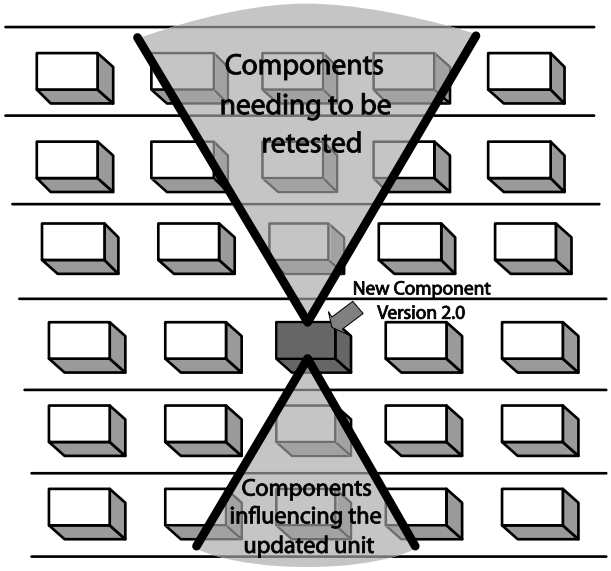

Fig. 10. Consequences of Updates

The same information useful for testing is useful for the management of the system's evolution. Whenever there are "acceptable" changes in the system's requirements, we must modify the application, which, in the case of the component-based approach, means releasing new versions of some components. The question is, "What components must be modified and what are possible side effects of such modifications?". By and large, the developer must locate components responsible for the requirement that has been modified and then develop new versions of these components. Side effects can be produced on components which depend upon newer components, and compatibility must be ascertained by checking all dependencies of the new subsystem. The operation is complicated because the developer might fail to locate components responsible for certain particular requirements, or the transitive closure of the dependencies might impact upon the whole system so that any update would influence all components. In our experience, we have found it most useful to keep a mapping from requirements to the components implementing them, so as to facilitate the prediction of complexity and the cost of an update. In our case, we attempted to alter the requirement that messages be sent on top of the TCP/IP connection, replacing it so that the requirement to send message is carried out by the SOAP protocol. We release a new version of the same component and test it with the same test suites used in the previous version. Finally, we perform integration testing by retesting components transitively dependent upon the *SendReceive* component. In particular we retest the *Mobility*, the *Communication*, the *Secu-*

rityPolitics and the *Agents* components. Where tests are successfully passed, we assess compatibility between the two versions.

The component-based approach also has some drawbacks. To date, black box components are still not reliable, because specifications may be incomplete or incorrect. Moreover, in the cases where components were not tested properly, they may contain critical faults. To obtain more reliable results from a component, the component's assembler tests it within the context of the system where it is used. However this may not be sufficient. Therefore the design-for-testability [24] approaches are strongly recommended to facilitate integration testing. For example, if the *Starter* component is provided as a black box component we may not know, unless it is explicitly stated in the specification, whether it is possible to use the *Starter* component for both new and migrated agents. If the *Starter* component is designed only for newly created agents, we perform an error by connecting the *Mobility* and the *Starter* components. A complete framework to "safely" compose and evaluate components does not exist as yet, so integration of third-part components in the core of the application is not a realistic scenario, unless intensive testing is performed or formal specifications are required. In the case we want to reuse software artefacts at any cost, we can easily produce misbehaving systems. In fact, it is possible to use components for different purposes respect to the developer's intention [25] impacting negatively also on the relevance of any test that has been performed on the component in isolation.

5 Components Recycling

In this Section we focus on the manufacturing case study, in particular the domain quality control aspect. We study the new requirements of the system aimed at reusing components of the biological case study. A large part of our design is focused on the middleware and low level components, affording ample opportunities to reuse some software units, because different high level requirements can produce different business level, but similar low level components. At the end of the paragraph we illustrate the design of the new middleware, discussing successful and unsuccessful.

5.1 The manufacturing Case Study - 2° scenario

The production processes of a manufacturing company are usually performed by a set of distinct activities, sequential or not. A complex set of activities leads to the definition of often complex paths where activities are associated to multiple agents, each responsible for a particular task. In the specific case of a supply chain, the actors are the suppliers and the production plans; the former usually provide both raw and semi-manufactured materials while the latter assemble the various input components to produce a final, more complex assembled product.

Over the years, many solutions have been proposed to solve the problems inherent in the complex production process. These solutions, known as supply

chain management, contributed to the improvement of production process management by using both planning and communication techniques, and by supporting information and data exchange within a company or between companies [26]. Unfortunately, to date these solutions offered no solutions for the traceability of the different components and semi-manufactured products in terms of quality.

At first sight this context, geared towards quality, reflects problems with the integration of heterogeneous data. In fact, each single supplier uses his own quality control mechanisms and stores results of test in his own format. The goal is to integrate and rendered readily accessible all these data among manufacturers. It would be useful, once a defect or malfunction in the final product has been identified, to be able to trace and recover all information regarding quality that has been generated by the different tests and controls on components composing the faulty product.

A MAS can be the technology exploiting resources and services integration in the manufacturing applicative domanin, but several issues must be taken into account. Embedded systems that perform the various quality tests of the products are very heterogeneous, and data is stored in repository providing access services that differ significantly. The security issues, moreover, play a vital role all along the supply chain. In fact, both generated reports and embedded checking system must be protecting from malicious access.

5.2 System requirements in brief

Users of the manufacturing system are not computer scientist and they do not need to use proprietary algorithms, hence it is necessary to have a set of ready-to-use agents in contrast with in-house developed bioagents.

Security continues to be a key requirement, so it is absolutely necessary to protect the system from the execution of malicious code and from attacks to the node's services. Consequently code inspections and access control techniques must be implemented.

In our first solution to the manufacturing case study, there is no need for agents to communicate. Actually, at the moment of its creation every agent is instructed about its objectives and then works independently of other agents. When the job is accomplished, the agent will deliver any extracted data or the overall result of the operation to a central repository.

Mobility is a characteristic our manufacturing system. Agents move between the different nodes of the extended supply chain or between several suppliers in order to efficiently access resources. We decided to implement mobility because each agent performs significant computations and accesses to large portion of data, so it is convenient to run locally. Should we wish to address the application to static agents, we can develop a new system quite readily by substituting only a well-defined subset of components.

5.3 System design

The middleware we propose in Section 3 is made up of three layers: *Core* layer, *BasicServices* layer and *Agent* layer. This view can be maintained for the manufacturing case study as well. In fact, the division is essentially conceptual: a layer for low level features, a layer for supporting agent activities, and a layer for all agents.

Core layer: Using mobile agents, communication between hosts is still required; hence we can reuse the *SendReceive* component that is responsible for inter-platform communication. This component must be extended to support multiple communication protocols, because depending upon the type of the embedded system used in the production plant, it might be necessary to communicate with different platforms by different protocols. However, communication between agents is no longer necessary, so part of the component is no longer used.

The *Security* component is personalized to avoid the execution of dangerous code. The implemented checks are different from the previous case because in the manufacturing domain it is not possible to develop new agents, but only to configure existing ones.

Finally the *ID* and the *Starter* components can be fully reused.

BasicServices layer: The *BasicServices* layer contains all components implementing features which support agent activities. In the manufacturing case study, agents are very different from biological agents, but features that must be implemented to enable and support their execution are not much different. The *Genesis* component can be reused without any changes because it implements agent creation and cloning - two features that are quasi-free of context.

The *Discovery* component must be re-implemented because it is dependent upon the application domain. In fact the discovery component, required for the recording and the identification of services offered by the running node, must use new recording and search methods strongly oriented to the industrial domain.

The *Mobility* component, responsible for agent migration is reusable in this context as well. Mobility as we know plays a vital role in the process of search and integration of different quality reports.

The *Security politics* component must be updated to shift from the per-service identification (used in the biological domain) to the per-node identification (used in the manufacturing domain). In fact, every node will share a small number of services and the agents who transit between a supply chain's nodes can identify themselves once per node.

A new component could be inserted in this layer. In fact, it is necessary to monitor the agents received in the running node, to verify the state of the various services agents, and to generate statistics on the use of the resources. This component, called *Console*, would interact with the *Discovery* component and with the *ID* component. Thus, the new component would offer to the administrator of the node a useful tool to guarantee a high level of security and control of the middleware.

Several features of the *Communication* component are no longer necessary because we did not plan for the implementation of any communication among agents. The *Communication* component will be updated and simplified to cover only interactions between user agents and service agents and vice versa.

Agent layer: Reusing the previous experience in the biological domain, the agent layer contains only a single component. This component contains both the user agents and the service agents. The component could not be reused in any other context because it is highly dependent upon the application domain.

Observing Figure 11, it is evident which components have been re-used and which have been modified. For example, the *SendReceive* component clearly had nothing more to offer to the *BasicServices* layer than the send and the reception of inter-platform agent messages. As important consequence, the Communication component in the *BasicServices* layer will be modified to allow interaction between agent and services only. To allow the control and verification of the running node, a new component responsible for monitoring the agents received and the services present is introduced.

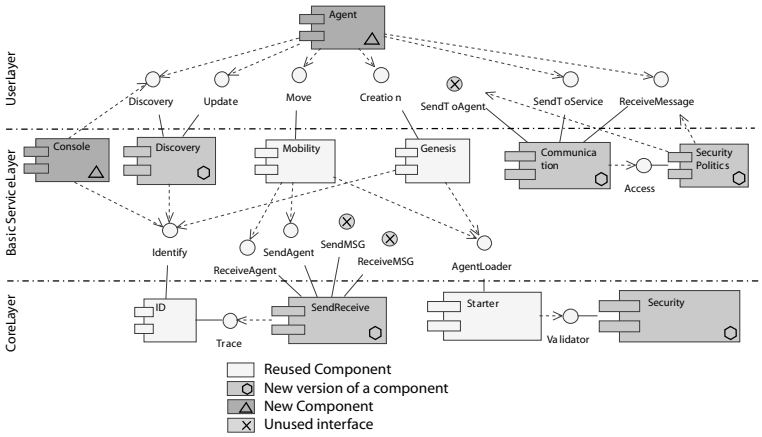

Fig. 11. Components overview in the industrial domain

Our experience in the manufacturing domain shows that components can be reused in different domains, especially at the core level of an application. The main low level building blocks of classes of homogeneous systems are also the most critical pieces of software, not only because they are accessed from almost all components of the system, but also because they represent a more reusable class of components that can be transported into different contexts. In fact, reusing components from the first two layers of our application was simple. It was, difficult to reuse high level components.

We did not try to reuse black-box components because it is still extremely difficult to safely reuse a third-party software artfact, but we plan to attempt the experience at some future time, possibly, for example, comparing results of white-box development and black-box development.

6 Related Experiences

Most of existing platforms that support Multi-Agent Systems have been designed by the object-oriented approach [27, 7] that enables class level modularity and extensibility. To fully exploit code reuse, it is necessary to address reuse at the proper level of abstraction. The class granularity is not suitable to reuse complete subsystems (i.e., aggregate of classes), because the developer cannot easily gain the understanding of the subsystem's purpose.

A first way to overcome to this problem is using design patterns. A design pattern [28] is a good design solution that can be reused in different scenarios. Several agent design patterns [29–31] have been proposed in the literature, but often they are not of general applicability or difficult to integrate each other.

A different approach consists on the reuse of binary pieces of code (components) instead of a design solution. This promising technology trend can lead to a component market made up of configurable software artefacts, each one available in several versions and ready to be integrated in new or existing systems. A component-based design [9] permits to implement quite-independent binary units fully exploiting a set of functionalities. This approach has been lowly addressed in the case of agent-based systems. The Java Agent Framework (JAF) [32] is an attempt to create an agent component framework where domain-dependent and domain-independent components are integrated. Some design conventions facilitate the developer during creation, integration and reuse of components. The framework highlights developer's difficulties during the learning stage. Composition is difficult also due to the lack of knowledge about the thread of control of each component and moreover is quite difficult to design agents that must comply with time requirements. The JAF approach addresses mainly to the development of a single agent and refers to component by an abstract description. We differ from JAF because we focus on middleware level components and we explicitly refer to UML design. In this case, components are formally described and the compositional approach enables formal reasoning about subsystems. There are several formal languages for component composition [33] and there exist also some approaches in the context of MAS, such as in [34]. In our case, we point to a pragmatically point of view and we do not address to formal verification. Anyway our approach is extendible by introducing a formal notation on UML diagram, such as OCL [35]. Software evolution and managements of concerns crossing software functionality are also addressed by the aspect-oriented technologies [36] that we are considering to increase discipline of our approach.

7 Conclusions and Future Work

We claim that the joint "components and layers" approach is an effective architectural style for the development of complex systems and we have demonstrated the great potential of reuse if exploited at the core of an application. We provided evidence for our assertions in two case studies, in the first case, we implemented a MAS for biological information retrieval, and in the second, we developed an MAS for tracing defects on artefacts. The component+layers approach affords several advantages during the design of the first system. We can easily recognize components that implement a logic spanning two layers; these components are redesigned into two different components. In similar fashion, we recognize components located at the wrong layer or components too closely linked to the rest of the system. The white-box approach allows for the application of refactoring techniques involving the interiors of components as well. Advantages of the component-based approach are evident during the testing activity as well, because we reuse test suites for the same version of the same component as well as for components used in the manufacturing system.

The joint approach is not applicable to all systems. In fact, in cases where it is not easy or feasible to distinguish several abstraction layers, the approach is inapplicable. In all other cases, it is possible to use the joint approach. In particular, it is useful in the design of complex systems, because the layered approach seems to be very effective in mastering complexity.

Reuse of components is not easily achieved, but we show how it is much simpler to reuse components located at lower layer of a system. This is because lower layers are conceptually far from the business logic, so they are more general and reusable. Very specific components can be reused too, but the more we approach to high layers, and the more difficult it is to accomplish successful reuse.

Using white-box UML components, we can easily modify the structure of the system. For example, we trace dependencies to predict consequences of updates. We hold that black-box components would increase the complexity of the management of the system; only disciplined approaches will achieve similar results.

We have presented an application of the layers plus components approach to the design and implementation phases of a middleware, however this approach could be successfully used also during the re-engineering process of existing middleware.

Future trends will surely consist of the definition of framework supporting the safe composition of black-box components to fully exploit the potential of reuse in the component environment. We plan to conduct a similar experiment with both black-box components and aspect-oriented techniques to compare results obtained in these cases. Then we propose to define a conceptual framework where components can be integrated with a high degree of confidentiality by the component composer. Future activities also concern the definition of a methodology for the development of systems using the joined approach of components and layers.

References

1. Shen, W., Norrie, D.: Agent-based systems for intelligent manufacturing: A state-of-the-art survey. Knowledge and Information Systems, an International Journal **1** (1999) 129–156
2. Klusch, M.: Information agent technology for the internet: A survey. Journal on Data and Knowledge Engineering, Special Issue on Intelligent Information Integration **36** (2001)
3. Guttman, R., Moukas, A., Maes, P.: Agent-mediated electronic commerce: A survey. Knowledge Engineering Review **13** (1998)
4. Wooldridge, M., Jennings, N.R., Kinny, D.: The Gaia methodology for agent-oriented analysis and design. Journal of Autonomous Agents and Multi-Agent Systems **3** (2000) 285–312
5. Brazier, F.M.T., Dunin-Keplicz, B.M., Jennings, N.R., Treur, J.: DESIRE: Modelling multi-agent systems in a compositional formal framework. International Journal of Cooperative Information Systems **6** (1997) 67–94
6. Wood, M., DeLoach, S.: An overview of the multiagent systems engineering methodology. In Ciancarini, P., Wooldridge, M., eds.: Agent-Oriented Software Engineering - Proceedings of the First International Workshop (AOSE-2000), Springer-Verlag (2000)
7. Bellifemine, F., Poggi, A., Rimassa, G.: Jade - a FIPA-compliant agent framework. In: Proceedings of the Forth International Conference on the Practical Application of Intelligent Agents and Multi-Agent Technology (PAAM'99). (1999) 97–108
8. Cheyer, A., Martin, D.: The open agent architecture. Journal of Autonomous Agents and Multi-Agent Systems **4** (2001) 143–148
9. Szyperski, C.: Component Software: Beyond Object-Oriented Programming. Addison-Wesley (1998)
10. Bartocci, E., Mariani, L., Merelli, E.: An XML view of the "world". In: in proceedings of 5th International Conference on Enterprise Information Systems, Angers, France (2003) 19–27
11. Fensel, D.: Ontologies: A Silver Bullet for Knowledge Management and Electronic Commerce. Springer-Verlag (2001)
12. Frishman, D., Heumann, K., Lesk, A., Mewes, H.W.: Comprehensive, comprehensible, distributed and intelligent databases: current status. Bioinformatics **14** (1998) 551–561
13. Bioscience, L.: SRS. http://srs.ebi.ac.uk/ (2003)
14. Goble, C., Stevens, R., Ng, G., Bechhofer, S., Paton, N., Baker, P., Peim, M., Brass, A.: Transparent access to multiple bioinformatics information sources. IBM Systems Journal **40** (2001) 532–552
15. Jennings, N.R.: An agent-based approach for building complex software systems. Communications of the ACM **44** (2001) 35–41
16. Corradini, F., Mariani, L., Merelli, E.: A programming environment for global activity-based applications. In: WOA 2003 dagli Oggetti agli Agenti - Sistemi Intelligenti e Computazione Pervasiva. (2003)
17. Bellwood, T., Clément, L., Ehnebuske, D., Hately, A., Hondo, M., Husband, Y.L., Januszewski, K., Lee, S., McKee, B., Munter, J., von Riegen, C.: UDDI version 3.0. Published specification, Oasis (2002)
18. Fuggetta, A., Picco, G., Vigna, G.: Understanding code mobility. IEEE Transactions on Software Engineering **24** (1998) 352–361

19. FIPA-ACL: FIPA97 specification, part 2: Agent communication language. Specification, FIPA (1997)
20. Judge, D., Odgers, B., Shepherdson, J., Cui, Z.: Agent enhanced workflow. BT Technical Journal **16** (1998)
21. Nodine, M., Fowler, J., Ksiezyk, T., Perry, B., Taylor, M., Unruh, A.: Active information gathering in infosleuth. International Journal of Cooperative Information Systems **9** (2000) 3–28
22. GPCE'03: International conference on generative programming and component engineering, Erfurt, Germany, Springer-Verlag (2003)
23. Stafford, J., Richardson, D., Wolf, A.L.: Chaining: A software architecture dependence analysis technique. Technical Report CU-CS-845-97, University of Colorado (1997)
24. Binder, R.V.: Design for testability in object-oriented systems. Communications of the ACM **37** (1994) 87–101
25. Rosenblum, D.: Challenges in exploiting architectural models for software testing. In: Proceedings of the International Workshop on the Role of Software Architecture in Testing and Analysis (ROSATEA). (1998)
26. MESA: The benefits of MES: A report from the field. MESA International (1994)
27. Baumann, J., Hohl, F., Rothermel, K., Straβer, M.: Mole - concepts of a mobile agent system. World Wide Web **1** (1998) 123–137
28. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley (1995)
29. Aridor, Y., Lange, D.: Agent design patterns: Elements of agent application design. In: Proceedings of Second International Conference on Autonomous Agents (Agents '98), ACM Press (1998) 108–115
30. Cossentino, M., Burrafato, P., Lombardo, S., Sabatucci, L.: Introducing pattern reuse in the design of multi-agent systems. In: Agent Infrastructure, Tools and Applications Workshop at NODe 2002, Erfurt, Germany (2002)
31. Silva, O., Garcia, A., Lucena, C.: The reflective blackboard pattern: Architecting large-scale multi-agent systems. In Garcia, A., Lucena, C., Castro, J., Omicini, A., Zambonelli, F., eds.: Software Engineering for Large-Scale Multi-Agent Systems. Volume 2603 of LNCS., Springer-Verlag (2003) 76–97
32. Wagner, T., Horling, B., Lesser, V., Phelps, J., Guralnik, V.: The struggle for reuse: Pros and cons of generalization in TAEMS and its impact on technology transition. Proceedings of the ISCA 12th International Conference on Intelligent and Adaptive Systems and Software Engineering (IASSE-2003) (2003)
33. Medvidovic, N., Taylor, R.N.: A classification and comparison framework for software architecture description languages. IEEE Transactions on Software Engineering **26** (2000) 70–93
34. Hameurlain, N.: A formal framework for behavioural reuse of agent components: Application to interaction protocols. In: 10th European Workshop on Multi- Agent Systems. Modelling Autonomous Agents in a Multi-Agents Wold (MAAMAW'01). (2001)
35. OMG: Object constraint language specification version 1.1. Technical report, OMG (1997)
36. Kiczales, G., Lamping, J., Mendhekar, J., Maeda, C., Lopes, C.V., Loingtier, J., Irwin, J.: Aspect-oriented programming. In: proceeding of the European Conference on Object-Oriented Programming (ECOOP), Springer-Verlag (1997)