# Communication as a Means to Differentiate Objects, Components and Agents

Dwight Deugo, Franz Oppacher, Bruce Ashfield, Michael Weiss
*School of Computer Science, Carleton University*
*1125 Colonel By Drive, Ottawa, Ontario, Canada, K1S 5B6*
*deugo@scs.carleton.ca, oppacher@scs.carleton.ca, ashfield@computer.org, michael_weiss@Mitel.COM*

***Abstract***

*Choosing the right abstractions is important for managing the complexity of your system. Three important abstractions used today are object, component and agent. Many similarities exist between these abstractions, but to make proper use of each, one should have a good understanding of their differences. Too often, we hear people discussing their agent-based systems when they have simply used the object abstraction. In this conceptual paper, we use communication as a means to differentiate the three abstractions. We describe communications patterns for each abstraction using an abbreviated pattern format, identifying the contexts, forces and solutions to different problems that present themselves for each abstraction. Our objective is to help developers identify the abstractions they are working with so they can make better use of them.*

## 1. Introduction

When you create compound elements, name them, and manipulate them as units, you are forming abstractions. Abstraction helps you separate the details of how an element is implemented from how it is used. This separation makes it easier to replace one element for another since abstraction users are depended only on the abstraction's interface, not on its implementation. Moreover, by hiding the details of the implementation, abstraction users do not have to deal with the complexity associated with the implementation, making their development tasks easier.

You can layer abstractions on top of one another, where any abstraction may use only the services of the abstractions in the layer immediately below, and so on. For example, you can develop interval arithmetic out of the ability to add and subtract rational numbers. You can add and subtract rational numbers by being able to create them and by having access to their numerators and denominators. Finally, you can represent the numerator and denominator with a pair structure and with operations to get at the head and tail of the pair.

The division between an abstraction layer is called an abstraction barrier. One main advantage to using abstraction barriers is that each layer is easier to maintain and modify. Since the layer above depends only on the interface of the layer below, the lower layer can be replaced with new elements or implementations without having to inform the higher layer. You can defeat this advantage by bypassing abstraction barriers in your software, permitting one layer to use abstractions beyond the layer below. This jump results in a tight coupling between the implementations of the software layers, making your software more brittle.

To avoid jumping abstraction barriers, it is important for you to understand your abstractions. Abstractions can take many forms, including procedural, data and language [1]. More recently, object, component and agent have become increasingly important forms of abstractions. Do you know the differences between procedural, data and language abstractions? Most do. Do you know the differences between object, component, and agent abstractions? Most don't, and this is a problem. You cannot use an abstraction properly unless you know what it is.

Bradshaw noted that some consider procedure, object and agent abstractions as identical:

> *One person's "intelligent agent" is another person's "smart object'; and today's "smart object" is tomorrow's dumb program [2].*

Others have expressed the view that some algorithms can be expressed better using objects rather than with procedures [10]. While others have gone further, stating that it may be easier for developers to represent their programs in terms of agents rather than with objects [5].

We view all of these abstractions as being important and relevant, but for different contexts. Unfortunately, we find that there is much confusion between the object, component and agent abstractions. Given that you must understand an abstraction, before you can use it properly and successfully, we find that it is very important to clarify the distinction between the three. This a highly relevant task today because many developers are making claims about building object, component or agent-based systems without knowing what this implies or which abstractions they are actually using. For example, many are building agent-based systems out of objects [11]. Is this situation possible, or have they mixed abstractions?

We agree that there are many similarities between the object, component and agent abstractions. Some researchers have already tried to differentiate objects from agents [9, 12]. However, we have not found anyone that has attempted to provide a comparison of all three. We do not provide an exhaustive comparison here, but have found that we can use communication as one means to differentiate the three.

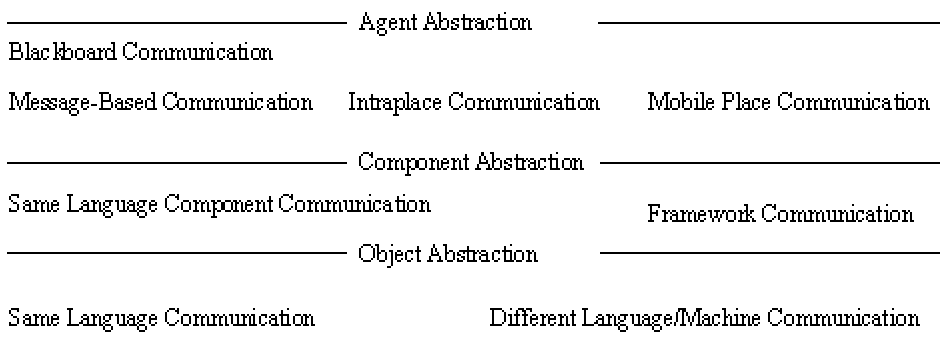
Agent Abstraction
Blackboard Communication
Message-Based Communication
Intraplace Communication
Mobile Place Communication
Component Abstraction
Same Language Component Communication
Framework Communication
Object Abstraction
Same Language Communication
Different Language/Machine Communication

**Figure 1. Abstraction Barriers and Patterns**

In this conceptual paper, we describe each abstraction's main communication patterns. The patterns identify the contexts, forces and solutions to different problems that present themselves for each abstraction. One may be familiar with some of the communication patterns described here. However, the main contribution of the paper is to associate each pattern with a specific abstraction, as shown in Figure 1. The consequence of providing these patterns is that developers will have something to compare their abstractions against, enabling them to determine which one they are using, and, thereby, enabling them to make better use of the abstraction.

For brevity and simplicity, we have used an abbreviated pattern notation. What follows are sections that begin with brief introductions to objects, components and agents and then continue with the main communication patterns of each abstraction. The final section provides a summary and our final thoughts.

## 2. Object Communication Patterns

In order to discuss patterns of object communication, we begin with a working definition of an object. We agree with the usual definition that an object is an entity that exists, is able to save state (information) and is able to offer a number of operations (behavior) to either examine or change the state [8]. In addition, a common class defines the structure and behavior of similar objects [3]. We also agree that instance variables and methods implement an object's state and behavior [4]. Implicit with this definition is that object-oriented applications are implemented using languages like C++, Smalltalk or Java. Consequently, applications are composed of language specific objects with development teams in control of their implementation.

This object-oriented approach to application development with specific focus on language results in two patterns of communication between objects. In short, either you leave it up to the language to manage the communication between objects or you develop your own add-hoc technique. The point we are trying to make is that once you use an object-oriented approach, rather than a component-based approach or agent-based approach, you form an expectation as to how your objects communicate because of the early commitment to a language.

In the following sections, we describe two communication patterns applicable once you make this choice.

### 2.1. Same Language Communication Pattern

**Problem.** How does an object communicate with another object?

**Context.** You have developed objects for an application and implemented them in the same language. One or more of the objects needs to collaborate with another or to delegate responsibility for a task. You and your development team are in close proximity and can agree on what information must be passed between objects and on its format.

**Forces:**

- Objects need to collaborate quickly with one another.
- Collaboration and delegation between objects is needed often.
- The implementation language supports the ability for objects to send messages to one another.
- The receiving object must always respond to a message. The response may be null, but this too is considered a valid response.
- A sending object is always willing to wait for a response to its message.

- Custom messaging protocols and frameworks take time to develop and can impact performance.
- The mapping between messages and methods to execute can be determined a priori.

**Solution.** Use the built-in facilities of the implementation language for communication. The quickest and most efficient means for objects to communicate is by having them send one another predefined messages resulting in the execution of corresponding methods implemented by the receiving object. Let the language facilities work out the details of ensuring the delivery of messages and the returning of their results.

### 2.2. Different Language/Machine Communication Pattern

**Problem.** How does an object communicate with another object in a different application on the same machine, on another machine, or written in a different language?

**Context**. You have developed objects for different applications and implemented them using either the same language or different combinations of object-oriented languages. One or more objects in different applications needs to collaborate with another or needs to delegate responsibility for a task to it. You and your development team are in close proximity and can agree on what information must be passed between objects and on its format.

**Forces:**

- Objects need to collaborate quickly with one another.
- Collaboration and delegation between objects is needed often.
- The implementation languages do not support the ability for objects to send messages between objects across languages, applications or machines.
- The receiving object may need to respond to a message, but not always.
- The sending does not need to wait for a response to a message.
- Sophisticated messaging protocols and frameworks take time to develop and can impact performance.

**Solution.** The sending object writes its message to a blackboard that the receiving object looks at periodically. After noticing and processing the message, if necessary, the receiving object writes its response back to the blackboard for the original sending object to read. You can consider shared memory, a file, or even a socket as physical examples of a blackboard.

## 3. Component Communication Patterns

In order to discuss component communication patterns, we begin with a working definition of a component. We agree with the definition formulated at the 1996 European Conference on Object-Oriented Programming [13]:

> *'A software component is a unit of composition with contractually specified interfaces and explicit context dependencies. A software component can be deployed independently and is subject to composition by their parties.'*

The belief is that a component is one abstraction level above an object. For example, we naturally think about methods when we think of object behaviors, since methods are the embodiment of the behaviors. However, this is a result of an early focus of language and implementation. Components change our focus from behaviors to interfaces, contracts, semantics and composition. When working with components, you should not care about how they conduct their operations or their implementation. Rather you should focus on what they can do for your application. In other words, components are the true notion of black box. Therefore, when your application communicates with a component, the language you are using and the one used to implement the component do not matter. Moreover, where objects are dependent on their execution environments and this dependency is yours to manage, components are meant to be independently deployable. This means that components must specify their own needs and any environment can use them provided those needs are met.

In summary, components manage their needs and hide their implementation details, including the implementation language. If you don't know which language was used to implement a component, how do you communicate with it? In the next sections, we describe two communication patterns applicable once you make the choice to work with components, not objects.

### 3.1. Same Language Component Communication Pattern

**Problem.** How does a component or an object communicate with other components when they are implemented in the same language?

**Context.** You decided to use the component abstraction for your building your application. You have either developed your components from scratch or have purchased them from a third party, but in both cases, they are implemented in the same language as other objects in your application. The result is that your application is composed of objects and components implemented in the same language.

**Forces:**

- Components need to collaborate quickly with one another and with other objects.
- Objects need to collaborate with components.
- Collaboration and delegation between objects and components is needed often.
- The receiving component must always respond to a message. The response may be null, but this too is considered a valid response.
- A sending component is always willing to wait for a response to its message.
- Custom messaging protocols and frameworks take time to develop and can impact performance.
- The mapping between messages and the methods to execute can be determined a priori.

**Solution.** Since your components are implemented in the same language as your objects, you can apply the same solution as in the Same Language Communication Pattern and for the same reasons. An example of this type of interaction is Java objects interacting with Java Beans.

### 3.2. Framework Communication Pattern

**Problem.** How does a component or an object communicate with other components when it is not implemented in the same language, not physically located on the same machine or not in the same application?

**Context.** You decided to use the component abstraction for your building your application. You have developed or purchased components implemented using a different language from the one used to develop your application's components. Your application's components may need to message to components contained in another application. The result is that your application is composed of objects and components implemented in different languages, and possible on different machines and in different applications.

**Forces:**

- Components need to collaborate quickly with one another and with other objects.
- Objects need to collaborate with components
- Collaboration and delegation between objects and components is needed often.
- The implementation language supports the ability for objects to send messages to one another.
- The receiving component must always respond to a message. The response may be null, but this too is considered a valid response.
- A sending component is always willing to wait for a response to its message.
- There is no way for you to determine a component's implementation language.
- The mapping between messages and the methods to execute can be determined a priori.

**Solution.** Since your application is composed of objects and components implemented in different languages, or you are not sure of a component's implementation language, you cannot rely on the communication services provided by a specific language. Therefore, your application will need to work with a framework that can 'install' the components and then use its communication services to communicate with the components. Examples of this type of framework include The Object Management Group's (OMG) Common Object Request Broker Architecture (CORBA) and Microsoft's Component Object Model (COM).

## 4. Agent Communication Patterns

In order to discuss agent communication patterns, we begin with a working definition of an agent. It is difficult to define an agent since many different definitions exist. For example, Franklin and Graesser [7] describe ten different agent definitions. However, we agree that their definition is best at providing an overall view of an agent:

> *"An autonomous agent is a system situated within and a part of an environment that senses that environment and acts on it, over time, in pursuit of its own agenda and so as to affect what it senses in the future."*

Our belief is that an agent is a level of abstraction above a component. Like a component, an agent is situated in an environment and is capable of autonomous action in this environment. However, unlike an object or component, which have no control over the execution of their methods, agents have precisely this type of control [9]. If an object or component sends a message to another object or component, the receiver has no control over whether the corresponding method executes; the method always executes. This control is provided by the underlying language or framework, not the object or component. With agents, we view communication as a request by one agent to another. Whether the receiving agent performs any action upon receiving the request, is entirely under its control, not with the underlying language or framework.

Another important aspect of agency, is mobility. In the past, we have built distributed systems from objects and components that, once distributed, stay in one place. The only difficulty in having the distributed entities communicate with one another is in finding their locations. Though, once located, you can apply the Different Language/Machine Communication Pattern or the Framework Communication Pattern to provide for their communication. Mobile agents offer another powerful technique for developing distributed applications at a higher-level of abstraction [14]. However, mobility presents a number of new problems for agent communications. For example, it is difficult to message to an agent that never stays in one place long enough to receive its messages.

The agent abstraction creates new demands on communication. The agents rather than the language or framework are in control of receiving messages, and due to their mobility, agent messages will need to locate the agents before being transported in a secure manner. In the following sections, we describe a few patterns that address these problems.

### 4.1. Message-Based Communication Pattern

**Problem.** What mechanism should agents use to communicate when security, portability and addressing are primary concerns?

**Context.** You are developing a multi-agent system in which communication is required. The communication and exchange of data must be encoded, controlled and managed. Regardless of the decisions made later in the design process, message based communication will change very little.

**Forces:**

- Agent communication (knowledge sharing) must be encapsulated as it cannot be part of a language.
- Messaging must be secure because it may occur across the network.
- Messaging must be portable across platforms, agents and execution engines.
- A single type of message must be understood by all types of agents (i.e. a standard way of communicating)
- Messages must be of minimal size to reduce network load.
- Different vendors will want to define different message formats; therefore, additional work may be required for interworking

**Solution.** Define a message class independently of how it is to be used and the information that it may contain. This class hides the platform and agent type (e.g., BDI or Reactive) dependent nature of the information sent between agents. Message objects can be grouped into protocols, or sequences, but always remain the basic building block of agent communication.

Next, develop a messaging layer that sits above the agents as part of the execution environment and which performs all access and manipulation of the messages through well-defined API methods. This messaging layer is responsible for providing the security and portability of the information encoded in the message. Use a suitable encoding for communication, such as KQML [6], embeded into the API of the message objects to allow effective collaboration and communication. Additionally, the communication (messaging) layer should employ established technologies such as Corba or RMI to take care of the low-level details of sending messages.

The platform and agent independent nature of the message allows security and portability to be encapsulated. For example, message objects can calculate check sums and verification codes automatically.

Creating a dedicated messaging layer allows agents of different vendors to be interworked, since information can be converted upon receipt to a format that the target agent can understand. This solution allows the agents that make up a system to concentrate on other issues, remain simple and still take advantage of complex messaging features.

### 4.2. Intraplace Communication Pattern

**Problem.** How does an agent communicate with other agents located in the same execution engine (machine)?

**Context.** You are developing a multi-agent system and agent collaboration is required. To complete tasks, agents must communicate with other agents physically located in the same agent execution engine.

**Forces:**

- Other agents may not know an agent’s location or if it exists before communication occurs.
- Agents may not be of the same type; therefore, issues such as capabilities determination are required
- Security is a concern, i.e., Can other agents be trusted. What are their intentions?
- Agents cannot move from engine to engine.
- Agents have limited knowledge of other agents outside of their execution engine.
- Agent communication must be controlled to keep the system in a consistent state (i.e., lost messages need to be managed)
- Agent complexity must be kept to a minimum to avoid creating a high maintenance overhead.

**Solution.** Use an agent "supervisor" to provide the means for agents to locate and communicate with others. Upon its creation, force an agent to register with a supervising agent (layer). Upon its destruction, force an agent to deregister. When an agent initially registers with the supervisor, have it communicate information such as its goals, intentions and capabilities. This allows the supervisor to match agent queries (clients) with providers (servers).

The centralization of communication in the supervisor allows security, reliability and agent location to be transparently provided for agents. Agents never directly communicate but instead construct messages and pass them to their targets via the supervisor. Although this design can cause a message bottleneck, it increases message flexibility and security. Additionally, the supervisor can implement advanced features such as flow control or caching of messages and queries. Another benefit is that individual agents remain relatively simple and do not have to implement methods to locate and communicate with other agents.

### 4.3. Mobile Place Communication Pattern

**Problem.** How do mobile agents communicate and collaborate with other mobile agents?

**Context.** You are developing a multi-agent system where agents must collaborate with others located in the current execution engine or on other remote engine. Agents are free to move about the network and are not constrained to remain at a single execution engine. Agents may actively move about the network seeking collaborators or wait until collaborators are found and move to their location.

**Forces:**

- Agents are free to move throughout the network and are aware of the network topology. However, the more they move, the greater the demands placed on the execution engines for transport.
- Not all agents have the same capacities or "type".
- An agent's location and capabilities must be accessible by other agents in the community.
- Network load should be reduced through local processing whenever possible.
- Agents must be portable between execution engines.
- An agent can be assigned a unique identity and home location that do not change during or after it moves.

**Solution.** Similar to the Intraplace Communication Pattern, upon an agent's creation, force it to register with a supervising agent at its 'home' location. Upon its destruction, force it to deregister. When an agent moves, it informs its home, supervising agent where it will be located. It also informs the remote, supervising agent that it has arrived at a location. Anytime a home, supervising agent is asked to deliver a message to an agent, it forwards the message to the remote, supervising agent where it believes the agent is currently located. The remote, supervising agent finally delivers the message. However, if the agent has moved again and the remote, supervising agent does not know where to forward the message, it informs the home, supervising agent of the situation. The home, supervising agent can send the message again once the agent identifies its new location.

### 4.4. Blackboard Communication Pattern

**Problem.** How can agents communicate with others without having to know their locations?

**Context.** You are developing a multi-agent system in which agents need to share information not knowing the exact location of each other.

**Forces:**

- Agents cannot directly message between one another. This may be because of performance concerns, or because agents move so frequently that a supervising agent is not practical.
- Communication between specific agents is required.
- Communication may also need to be multi-cast.
- An agent can be assigned a unique identity
- A single consistent messaging interface is required because add-hoc ones do not scale well.

**Solution.** Create a new system object called a Blackboard. The blackboard is a centralized place where agents can post and retrieve messages in a particular execution engine.

The blackboard handles all communication in a transaction-based manner. This means that agents can post messages or events for other agents and the blackboard ensures their consistency and validity through a transaction model. This model includes support for things such as commit events, retracting messages and sequences of actions. Any agent can poll the blackboard to see if they have any messages.

Agents can post messages directed to specific agents by including the identifier of the target agent in the message, or they can post higher-level, more abstract messages, such as goals, for all agents to read. Permit goals to be seen by the entire agent community and let any agent with the proper capabilities and intentions to post a response.

The blackboard is a passive system object and consequently consumes very few system resources. The flexibility, performance and consistent messaging interface it provides are difficult to match using directed messages. The blackboard can also provide functionality such as persistent messages and the ability to transfer messages to other blackboard, all without any additional agent complexity.

You can extend the blackboard concept with a facilitator agent. Agents register with the facilitator and the blackboard. The blackboard provided the passive services and the facilitator takes an active role by notifying any registered agents if a message was posted for them.

## 5. Summary

Our goal was to help the reader understand the differences between the object, component, and agent abstractions. To achieve this, we used communication as a means to differentiate them. In short, if you use entities written in the same language and rely on the language for communication support, you are using the object abstraction, even when those entities are on different machines. If you use entities implemented in different languages, but still rely on a sender, a receiver and language support for messaging, you are using the component abstraction. Even with this abstraction, there is a one-to-one mapping between a message and a corresponding method or procedure. For example, when one entity sends a message, a corresponding method or procedure is executed in the receiver. If messages are first class objects with their processing involving the following steps, you are using the agent abstraction.

- locating the appropriate receiving entity(s),
- delivering the message,
- entities reacting with the appropriate behavior determined after the message is received

We based our choice of patterns and pattern format partly due to space limitations and on the fact that we did not intend our patters to be the complete set. This is especially true of the agent abstraction, as we are in the process of documenting an additional ten communication patterns. We attempted to choose those patterns that were important for distinguishing one abstraction from another. We believe the abbreviated pattern format provides enough of the details to help the reader understand the difference between the abstractions.

Are there other dimensions to compare the object, component and agent abstractions? The answer is obviously yes. However, when someone says they are developing an agent-based system in the future, we hope you will now understand what it means, at least from a communication perspective.

## 6. References

[1] Abelson, H., Sussman, G.J., and Sussman, J., “Structure and Interpretation of Computer Programs”, McGraw Hill, 1996.

[2] Bradshaw, J.M., “An Introduction to Software Agents”, In J.M. Bradshaw (ed.) Software Agents, AAAI Press/MIT Press, 3-49, 1997.

[3] Booch, G., “Object Oriented Design with Applications”, Benjamin/Cummings, 1991.

[4] Campione, M., and Walrath, K., “The Java Tutorial”, Second Edition, Addison-Wesley, 1998.

[5] Dennett, D.C,, “he Intentional Stance”, MIT Press, 1987.

[6] Finin, T., Labrou, Y., Mayfield, J., “KQML as an Agent Communication Language”, In J.M. Bradshaw (ed.) Software Agents, AAAI Press/MIT Press, 291-316, 1997.

[7] Franklin, S., and Graesser, A., “Is It an Agent, or Just a Program?: A Taxonomy for Autonomous Agents, In J.P. Müller, M. J. Wooldridge and N. R. Jennings (eds.) Intelligent Agents III Agent Theories, Architectures, and Languages – ECAI 96 Workshop, Springer-Verlag, Heidelberg, 21-35,1997.

[8] Jacobson, I., “Object-Oriented Software Engineering”, Addison-Wesley, 1992.

[9] Jennings, N.R., and Wooldridge, M., “Applications of Intelligent Agents”, In N. Jennings and M.J. Wooldridge (eds.) Agent Technology: Foundations, Applications, and Markets, Springer-Verlag, Heidelberg, 3-28, 1997.

[10] Kaehler, T., Patterson, D., “A Small Taste of Smalltalk”, BYTE, August, 145-159, 1986.

[11] Lange, D.B, and Oshima, M., “Programming and Deploying Java Mobile Agents with Aglets”, Addison Wesley, 1998.

[12] Shoham Y., “Agent-Oriented Programming”, Journal of Artificial Intelligence, 60(1), 51-92, 1993.

[13] Szyperski, C. and Pfister C., Workshop on Component-Oriented Programming, Summary, In M. Mühlhäuser (ed.) Special Issues on Object-Oriented Programming – ECOOP 96 Workshop Reader, Springer-Verlag, Heidelberg, 1997.

[14] Vogler, H, Moschgath, M., Kunkelmann, T, “Enhancing Mobile Agetns with Electronic Capabilities”, In M. Jlusch and G. Weiβ (eds.) Cooperative Information Agents II: Learning, Mobility, and Electronic Commerce for Infromation Discovery on the Internet – Second International Woekshop, CIA ’98, Springer-Verlag, Heidelberg, 148-159, 1998