Agent Design Patterns: Elements of Agent Application Design

Yariv Aridor
IBM Tokyo Research Laboratory
Yamato, Kanagawa, Japan
tayariv@trl.ibm.co.jp

Danny B. Lange
General Magic Inc.
Sunnyvale, California, U.S.A.
danny@acm.org

1. ABSTRACT

Agent technology is an emerging field and agent-based application design is still a pioneering discipline. We are all pioneers, inventing and re-inventing sometimes smart but perhaps more not-so-smart solutions to recurrent problems. It is here that agent design patterns can help by capturing good solutions to common problems in agent design. No special skills, language features, or other tricks are required for you to benefit from these patterns. Simply speaking, agent design patterns can make your applications more flexible, understandable, and reusable, which is probably why you were interested in agent technology in the first place. In this paper we report on several design patterns we have found in mobile agent applications.

1.1 Keywords

Agent design pattern, agent application, reuse, mobile agent

2. INTRODUCTION

Mobile agents are an emerging technology that makes it very much easier to design, implement, and maintain distributed systems. A mobile agent is not bound to the system in which it begins execution. It has the unique ability to transport itself from one system in a network to another. This ability to travel allows a mobile agent to move to a system that contains an object with which the agent wants to interact, and then to take advantage of being in the same host or network as that object.

You will find that mobile agents reduce network traffic, provide an effective means of overcoming network latency, and perhaps most importantly, through their ability to operate asynchronously and autonomously of the process that created them, help you to construct more robust and fault-tolerant applications.

However, as in other emerging technologies, we are all pioneers, repeatedly inventing and re-inventing sometimes smart but perhaps more often not-so-smart solutions to recurrent problems. During the early work on Aglets [3], the developers recognized a number of recurrent patterns in the design of mobile agent applications. Several of these patterns were given intuitive meaningful names such as *Master-Slave*, *Messenger*, and *Notifier*. They were implemented in Java [2] and included in the first release of the Aglets Workbench [5]. These early patterns were found to be highly successfully for jump-starting users who were new to Aglets and the mobile agent paradigm.

This experience tells us that it is important to identify the elements of good and reusable designs for mobile agent applications and to start formalizing people's experience with these designs. This is the role of *design patterns* [1]. The concept originated with software engineers and researchers in the object-oriented community, and has been recognized as one of the most significant innovations in the object-oriented field.

The focus of this paper is a new set of *agent design patterns* for creating mobile agent applications. We present a catalog of agent designs and describe representative patterns. Our patterns have an object-oriented flavor: that is, we describe them by using the notions of classes, objects, inheritance, and composition. Although they can be efficiently implemented in object-oriented languages such as Java, for most of the patterns an object-oriented language is not essential.

This paper is structured as follows: in Section 3 we present a classification scheme for agent design patterns; Section 4 describes three selected patterns in greater detail; Section 5 reports our experience in applying agent design patterns; and Section 6 concludes the paper.

In Proceedings of Autonomous Agents '98,
ACM Press, 1998.

3. CLASSIFICATION OF AGENT DESIGN PATTERNS

The patterns we have discovered so far can conceptually divided into three classes: *traveling*, *task*, and *interaction*. This classification scheme makes it easier to understand the domain and application of each pattern, to distinguish different patterns, and to discover new patterns. Below, we describe the three classes of patterns as well as the patterns in each class. The space limitations of a conference paper allow us to provide only a very brief description of each pattern. We hope, however, that the more detailed description of three of the patterns in Section 4 will make up for the brevity of the present section.

Itinerary. Objectifies agents' itineraries and routing among destinations.
Forwarding. Provides a way for a host to forward newly arrived agents automatically to another host.
Ticket. Objectifies a destination address and encapsulates the quality of service and permissions needed to dispatch an agent to a host address and to execute it there.

Figure 1 Travelling Patterns

3.1 Traveling Patterns

Traveling is the essence of mobile agents. The traveling patterns listed in Figure 1 deal with various aspects of managing the movements of mobile agents, such as routing and quality of service. These patterns allow us to enforce encapsulation of mobility management that enhances reuse and simplifies agent design.

The **Itinerary** pattern is an example of a traveling pattern that is concerned with routing among multiple destinations. An itinerary maintains a list of destinations, defines a routing scheme, handles special cases such as what to do if a destination does not exist, and always knows where to go next. Objectifying the itinerary allows one to save it and reuse it later, in much the same way as one saves URLs as bookmarks. With a graphical user interface, itinerary objects can be represented as icons that can be dragged and dropped onto icons of agents to dispatch these agents with these itineraries.

Another very fundamental travelling pattern is the **Forwarding** pattern. This simple pattern allows a given host to mechanically forward all or specific agents to another host.

The **Ticket** pattern was first described by White [4]. Conceptually, a ticket is an enriched version of a URL that embodies requirements concerning quality of service, permissions, and other data. For example, it may include time-out information for a dispatching an agent to a remote host. Thus, instead of naively trying to dispatch to a disconnected host forever, the agent now has the necessary information to make reasonable decisions while travelling.

Master-Slave. Defines a scheme whereby a master agent can delegate a task to a slave agent.
Plan. Provides a way of defining the coordination of multiple tasks to be performed on multiple hosts.

Figure 2 Task Patterns.

3.2 Task Patterns

Task patterns (see Figure 2) are concerned with the breakdown of tasks and how these tasks are delegated to one or more agents. In general, tasks may be dynamically assigned to general-purpose agents. Furthermore, a given task can be accomplished either by a single agent or by multiple agents working in parallel and cooperating to accomplish it (e.g., in the case of parallel search).

A very fundamental task pattern is the **Master-Slave** pattern, which allows a master agent to delegate a task to a slave agent. The slave agent will move to a destination host, perform the assigned task, and return with the possible result of that task.

The more complex **Plan** pattern adopts a workflow concept to organize multiple tasks to be performed in sequence or in parallel by multiple agents. The plan encapsulates the task flow, which is then hidden from the agent. The agent merely provides the mobility capabilities needed to perform tasks at specific destinations. The plan promotes reusability of tasks, dynamic assignment of tasks to agents, and even composition of tasks.

Meeting. Provides a way for two or more agents to initiate local interaction at a given host.
Locker. Defines a private storage space for data left by an agent before it is temporarily dispatched (send) to another destination.
Messenger. Defines a surrogate agent to carry a remote message from one agent to another.
Facilitator. Defines an agent that provides services for naming and locating agents with specific capabilities.
Organized Group. Composes agents into groups in which all members of a group travel together.

Figure 3 Interaction Patterns.

3.3 Interaction Patterns

The ability of agents to communicate with each other is vital for cooperation among agents. The interaction patterns in Figure 3 are concerned with locating agents and facilitating their interactions.

The **Meeting** pattern was first described by White [4], is an interaction pattern that provides a way for two or more agents to initiate local interaction at a given host. It abstracts the synchronization in time and place that is required for local interactions. Agents can dispatch themselves to a

specific destination, called a meeting place, where they are notified of the arrival of their counterparts and able to engage in local interaction.

Agents can exploit the **Locker** pattern to temporarily store data in private. In this way, they can avoid bringing along data that for the moment are not needed. On a later occasion, agents can return and retrieve the private data stored in the locker. For example, in an agent-based purchasing system, an agent may visit a dealer's host outside the company network. In this case, it can store its sensitive data in a locker before leaving the company network. The result is a reduction of network traffic and improved data integrity.

Agents can establish remote communication by using the **Messenger** pattern, which objectifies messages in the form of agents that carry and deliver messages between agents. For example, if a slave agent wishes to report a possibly intermediate result back to the master agent, it can send the result by a messenger agent while continuing with its current task.

The **Facilitator** pattern describes a naming and locating service for agents. It is often convenient to assign a symbolic (i.e., meaningful) name to an agent in order to locate it on a later occasion. For instance, an information-gathering agent may continuously move in the network and other agents may from time to time wish to retrieve updates from the information-gathering agent without actually knowing its present location.

Use the **Organized Group** pattern to compose multiple agents into a group in which all members of the group travel together (we also call this the group tour pattern). This pattern can be considered as a fundamental element of collaboration among multiple mobile agents.

4. EXAMPLE PATTERNS

We have selected a pattern from each group for closer examination. We will start with the **Master-Slave** pattern from the group of task patterns, followed by the **Meeting** pattern from the group of interaction patterns. The last pattern is the **Itinerary** pattern from the group of traveling patterns.

The pattern descriptions follow a common template that covers *intent*, *motivation*, *applicability*, *participants*, *collaboration*, and *consequences*. Associated diagrams all follow common object-oriented notation [7].

4.1 Master-Slave

4.1.1 Intent

The Master-Slave pattern defines a scheme whereby a master agent can delegate a task to a slave agent.

4.1.2 Motivation

There are several reasons why agents (termed masters) would like to create other agents (termed slaves) and delegate tasks to them. One is performance. A master agent can continue to perform other tasks in parallel with the slave agent. Another reason is illustrated via the following example. Consider an agent-based application that provides a GUI for inputting data and displaying the intermediate results of a specific task to be performed remotely. With a single agent to provide the GUI and perform that task, it will not be possible to maintain the GUI (e.g., to run the windows) after the agent has traveled from its origin to a remote destination. Alternatively, a stationary (immobile) master agent can provide and maintain a GUI while a slave agent moves to another destination, performs the assigned task, sends intermediate results, and finally returns and deliver the task's result to the master agent, which displays it to the client.

The key idea of the master-slave pattern is to use abstract classes, `Master` and `Slave`, to localize the invariant parts of delegating a task between master and slave agents: dispatching a slave back and forth to other destinations, initiating the task's execution, and handling exceptions while performing the task. Master and slave agents are defined as subclasses of `Master` and `Slave`, in which only varying parts such as what task to perform and how the master agent should handle the task's result are implemented.

In practice, the `Master` class has a `getResult` abstract method (i.e., one that should be overridden in subclasses) to define how to handle the task's result. The Slave class has two abstract methods, `initializeJob` and `doJob`, which define the initialization steps to be performed before the agent travels to a new destination and the concrete task, respectively. Both classes are defined in terms of these methods. Figure 4 shows the `Slave` class implemented as an aglet.

```
public abstract class Slave extends Aglet {
   Object  result = null

   public void onCreation(Object obj) {
      // Called when the slave is created.Gets the
      // remote destination, a reference to the master
      // agent, and other specific parameters.
   }

   public void run () {
      // At the origin:
      initializeJob();
      dispatch(destination);  // Goes to destination
      // At the remote destination:
      doJob();   // Starts on the task
      result=...;
      // Returns to the origin.
      // Back at the origin.
      // Delivers the result to the master and dies.
      dispose();
}
```

Figure 4 The Slave Class

4.1.3 Applicability

Use the Master-Slave pattern in the following cases:

- When an agent needs to perform a task in parallel with other tasks for which it is responsible.
- When a stationary agent wants to perform a task at a remote destination.

Both of these cases concern tasks to be executed at a single destination.

4.1.4 Participants

Four classes participate in the Master-Slave pattern. See Figure 5 for their structural relationships.

- `Master`. Defines a skeleton of a master agent, using abstract methods to be overridden in the `Concrete-Master` class.
- `Slave`. Defines a skeleton of a slave agent, using abstract methods to be overridden in the `ConcreteSlave` class.
- `ConcreteMaster`. Implements abstract methods of the `Master` class.
- `ConcreteSlave`. Implements abstract methods of the `Slave` class.

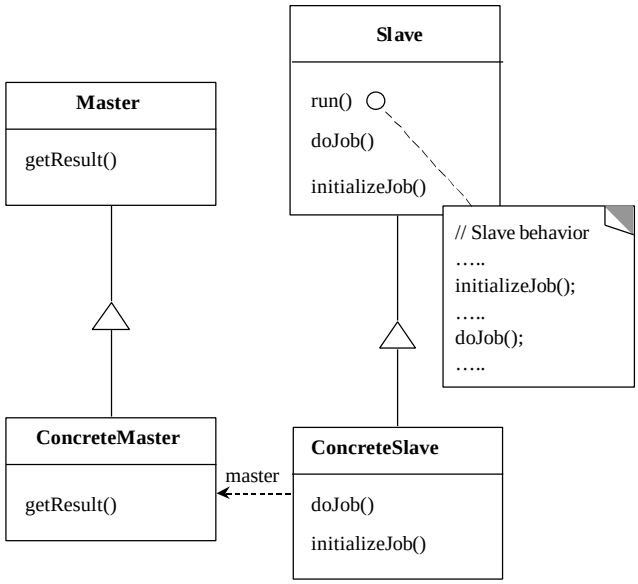

Figure 5 Participants in the Master-Slave Pattern

4.1.5 Collaboration

The collaboration between the participants in the Master-Slave pattern is as follows (see also Figure 6):

- A master agent creates a slave agent.
- The slave agent moves to a remote host and performs its task.
- The slave agent returns with the result of the task to the master.

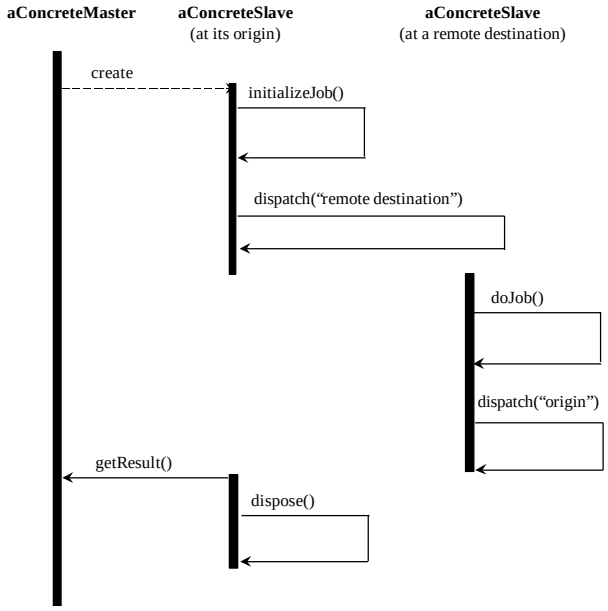

Figure 6 Collaboration in the Master-Slave Pattern

4.1.6 Consequences

The Master-Slave pattern provides a fundamental way to reuse code among agent classes. In practice, the process of agent design and implementation is simplified by letting developers implement only the variable aspects of already defined agents (provided by libraries or agent builder tools).

One drawback of an inheritance-based pattern is that the behavior of a slave agent is fixed at the design time. For example, an agent cannot be transformed into a slave at runtime nor can a slave agent easily be assigned to perform new tasks. A more sophisticated version of this pattern can use a delegation-based model. In this model, the task will be objectified and a slave agent can be assigned any task object during its lifetime.

4.2 Meeting

4.2.1 Intent

The Meeting pattern provides a way for agents to establish local interactions on specific hosts.

4.2.2 Motivation

Agents in different destinations may need to interact locally between them. Consider, for example, commerce agents created by different clients (i.e., at different destinations) to search for, buy, and sell particular goods on behalf of their clients. To do so, buyer agents and seller agents need to locate each other and interact extensively to try and make deals. Having all these commerce agents dispatched from

their origins to a central destination (termed the virtual marketplace) where they interact locally among themselves has two major advantages. First, once these agents leave their origins, they can carry on interactions even if their origins (e.g., clients' machines) are disconnected from the network or located inside firewalls (since agents inside firewalls cannot receive messages). Second, their local interactions will incur low communication overheads, compared with remote interactions.

The key problem is how to synchronize these commerce agents, which are initially at different hosts, so that they can visit the virtual marketplace and find each other. The Meeting pattern provides a solution to problems of this type, using the notion of a meeting. It uses a `Meeting` class that encapsulates a specific destination (meeting place) and a unique identifier. In general, agents that need to interact locally with each other will be equipped with a meeting object. Each agent will dispatch itself independently to the meeting place, where it will use the unique identifier to locate a specific local meeting manager object to register itself (i.e., add itself to a list of agents that already have arrived at that host). The meeting manager object will then notify the already registered agents about the newly arrived agent (they get a local reference to it) and vice versa, so that interactions involving the new agent can start. Agents should unregister themselves before leaving the meeting place. Through the use of unique identifiers, multiple meetings can take place simultaneously at a single host. Meeting objects can be distributed by messages or located in central directories

4.2.3 Applicability

This pattern is applicable whenever there is a need for agents on different hosts to interact locally. Here are three common situations in which this pattern is applicable:

- When agents need to interact, and the overhead of travelling to a central place and interact locally is less than that associated with remote communication.
- When agents cannot interact remotely, since they are located behind firewalls or on hosts with unreliable and low-bandwidth network connections, e.g., laptops and handheld computers. One solution is for these agents to dispatch themselves to a remote host where they can interact more efficiently.
- When agents need to access local services on a given host. In this case the agents need to interact locally with the service provided by a given host.

4.2.4 Participants

The structural relationships of the participants in this pattern are shown in Figure 7.

- `Agent`. A base class of a mobile agent.
- `ConcreteAgent`. A subclass of the `Agent` class that maintains meeting objects.
- `Meeting`. Object that stores the address of the meeting place, a unique identifier, and miscellaneous information. It also notifies the `MeetingManager` of the arrival or departure of agents.
- `MeetingManager`. This object knows all the agents currently participating in the meeting. It notifies agents that have already arrived at the meeting of the arrival of new agents and vice versa.

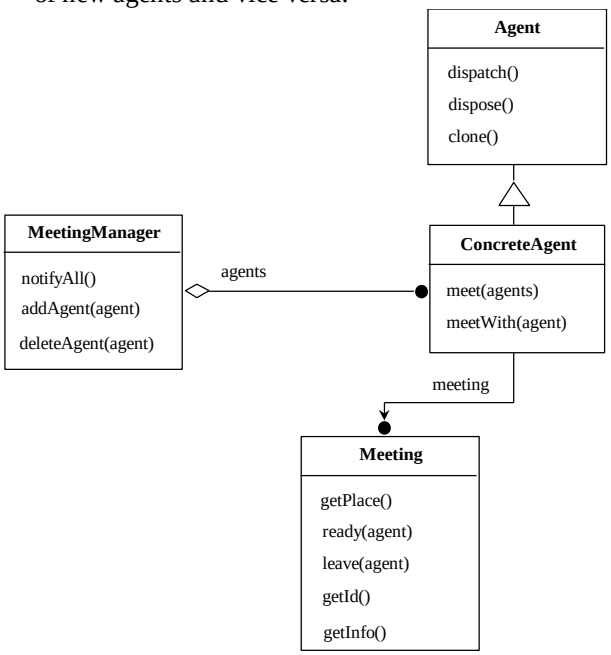

Figure 7 Participants in the Meeting Pattern

4.2.5 Collaboration

Collaboration among the participants in the Meeting pattern is also sketched in Figure 8.

- A `Meeting` object is created, with a unique identifier and a meeting place.
- A `ConcreteAgent` object is dispatched to the meeting place. Upon arrival, it notifies its `Meeting` object of its arrival through the `ready` method.
- Whenever a `Meeting` object is informed of the newly arrived `ConcreteAgent`, it locates the `MeetingManager` object, which registers the agent by the `addAgent` method.
- Upon registration, `ConcreteAgent` is notified by `meet()` of all the agents that have already arrived at the meeting (denoted by `arrivedAgents` in the interaction diagram ahead). The latter are notified of the newly arrived agent by `meetWith()`.
- Whenever a `Meeting` object is informed of an agent's leaving a meeting, it locates the `MeetingManager`

object, which unregisters the agent by the `deleteAgent` method.

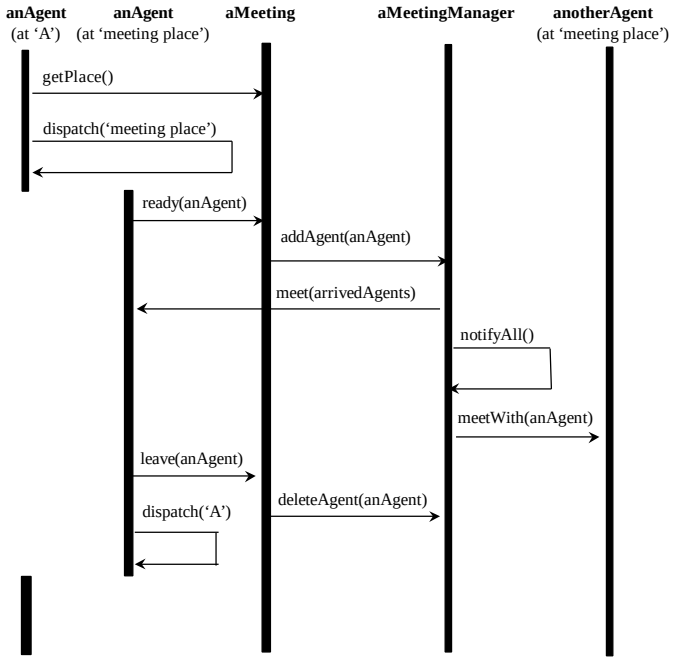

Figure 8 Collaboration in the Meeting Pattern

4.2.6 Consequences

The Meeting pattern has the following benefits and drawbacks:

- It provides a means for inter-agent communication that complies with the mobile nature of agents; unlike distributed static objects, mobile agents cannot maintain private references to directly locate each other and interact. This pattern enables agents to interact locally without having references to each other in advance.
- It enables an agent to interact locally with an unlimited number of agents.
- It can simplify inter-agent communication. The `MeetingManager` object can be implemented as a kind of mediator to transfer or multicast messages between agents. Consequently, many-to many interactions between agents will be replaced by many-to-one interactions between agents and the `MeetingManager` object. Many-to-one interactions are easier to maintain. From a different perspective, it can allow the `MeetingManager` object to maintain data shared by multiple agents and automatically notify them of any change in those data.
- It may strike a tradeoff between interactions with low overhead, and agents being idle, waiting for the arrival of their counterparts of the meeting place. Thus, in some situations, remote interactions may be preferable.

4.3 Itinerary

4.3.1 Intent

The Itinerary pattern objectifies agents' itineraries and their navigation among multiple destinations.

4.3.2 Motivation

Being an autonomous mobile entity, an agent is capable of navigating itself independently to multiple hosts. Specifically, it should be able to handle exceptions such as unknown hosts while trying to dispatch itself to new destinations or make a composite tour (e.g., return to destinations it has already visited). It might even need to modify its itinerary dynamically. For example, it might dispatch itself to inquire about a local Yellow Pages service, extract relevant addresses, and add them to its itinerary.

Consequently, it is probably preferable to separate the handling of navigation from the agent's behavior and message handling, thus promoting modularity of every part. The Itinerary pattern lets you do so. The key idea is to shift the responsibility for navigation from the agent object to an `Itinerary` object. The itinerary class will provide an interface to maintain modify the agent's itinerary and to dispatch it to new destinations. An agent object and an `Itinerary` object will be connected as follows:

The agent will create the `Itinerary` object and initialize it with (1) a list of destinations to be visited sequentially and (2) a reference to the agent. Then, the agent will use the `go` method to dispatch itself to the next available destination in its itinerary or back to its origin, respectively. To support the above, it is necessary that the Itinerary object be transferred together with the agent, and that their references to each other be maintained at every destination.

4.3.3 Applicability

Use this pattern when you wish to:

- Hide the specifics of an agent's tour from its behavior in order to promote modularity of both parts.
- Provide a uniform interface for sequential traveling of agents to multiple hosts.
- Define tours that can be shared by agents.

4.3.4 Participants

The structural relationships of the participants in this pattern are shown in Figure 9.

- `Itinerary`. Defines an interface for navigating with an agent.

- `ConcreteItinerary`. Implements the `Itinerary` interface and keeps track of the current destination of the agent.
- `Agent`. A base class of a mobile agent.
- `ConcreteAgent`. A subclass of the Agent class that maintains a reference to a `ConcreteItinerary` object.

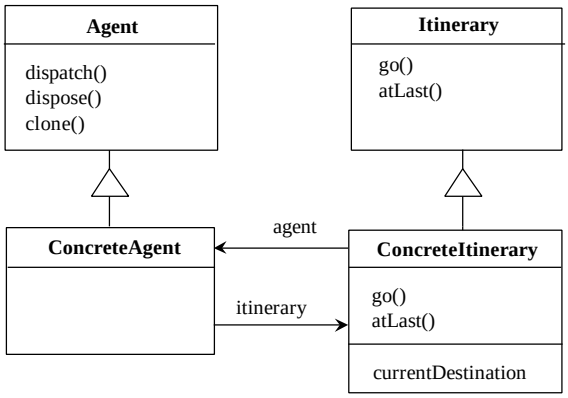

Figure 9 Participants in the Itinerary Pattern

4.3.5 Collaboration

Figure 10 shows collaboration according to this pattern.

- The `ConcreteItinerary` object keeps track of the current destination of the `ConcreteAgent` and can dispatch it to new destinations.
- Whenever the `ConcreteAgent` is dispatched to a new destination, the `ConcreteItinerary` is also transformed, and their references to each other are restored at the target destination.

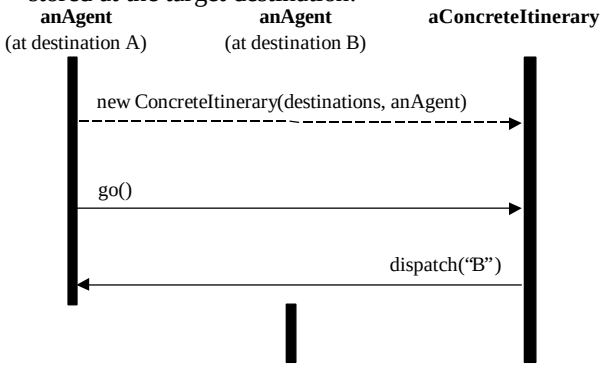

Figure 10 Collaboration in the Itinerary Pattern

4.3.6 Consequences

This pattern has three main consequences:

- It supports variations in navigation. For example, a different exception handling routine can be defined if an agent fails to dispatch itself to a new destination: cancel the tour and return to the origin, try to go to another destination and later try again. This pattern makes it easy to provide such variations by simply replacing one Itinerary object with another or by defining Itinerary subclasses. The agent class is not modified.
- It facilitates sharing of tours by different agents. For example, two agents may use the same tour to multiple users' desktops, one to schedule a meeting between all users and the other to deliver them notification messages. This pattern enables agents to share tours by sharing itinerary objects, although not simultaneous.
- It simplifies the implementation of sequential tasks. Tasks can be encapsulated in special Task objects while an Itinerary class is extended with an interface to associate Task objects with destinations. The itinerary object keeps track of the current tasks to perform. Whenever the agent is dispatched to a new destination, it simply triggers the execution of the current task saved by its itinerary object. In Java-based agent systems such as Aglets and Odyssey [6] in which agents are transported with only their code and data, and not with their entire execution state, the Itinerary pattern obviates the need to manually keep track of the execution state of an agent (i.e. what the agent should do) when it travels.

5. EXPERIENCE

In this section we briefly describe two agents, showing how the above patterns were combined to simplify their design and implementation. Both agents are implemented as aglets.

```
public final class FileSearcher extends Slave {
   protected void doJob() throws Exception {
      // Performs a local file search
      Result = …;
   }

   protected void InitializeJob() throws Exception {
      // Performs necessary initialization.
   }
}

public final class FileSearcherMaster extends Master {
   public onCreation( Object obj) {
      // Creates the window.
   }
   public getResult (Object result) {
      // Updates the content of the window.
   }
}
```

Figure 11 Master and Slave Agents

5.1 File Searcher

The *File Searcher* is an agent-based application that searches for files in a remote file archive. Modeled on the Master-Slave pattern, it consists of two agents. A stationary agent, `FileSearcherMaster`, that manages the GUI and a mobile agent, `FileSearcher`, that can move to a remote archive, search for files with a specific substring in

their filenames, and return to the master with the search result.

Figure 11 shows the implementation of the slave agent, `FileSearcher`, and its master agent, `FileSearcherMaster`. Notice that only the abstract methods of `FileSearcher` need to be implemented. As the example indicates, no particular skills in mobile agent programming are needed to create these two agents.

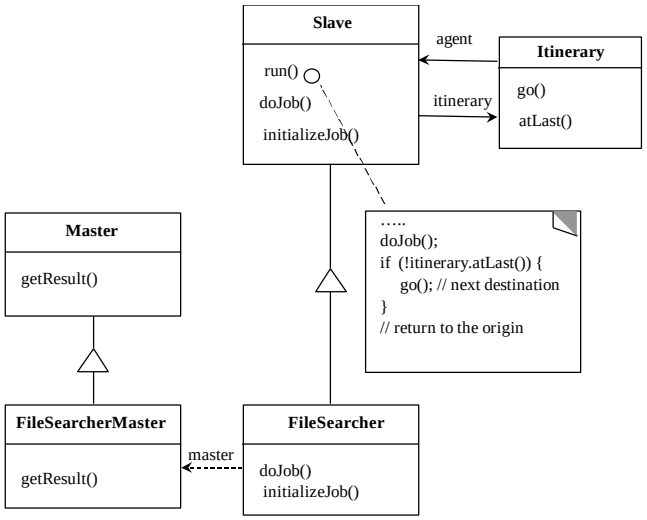

Figure 12 Enhanced File Searcher

5.2 Enhanced File Searcher

We extended the File Searcher to search in multiple remote file archives, using a combination of the Master-Slave and the Itinerary patterns, as shown in the class diagram in Figure 12.

The `Slave` class was modified to allow a slave agent to travel among multiple destinations and perform the task, as defined in the `doJob` method, at every destination. The modifications included (1) receiving an itinerary object, created by the master aglet, to control the tour, instead of a single destination and (2) changing its behavior, defined by the run method, so that it dispatches itself to the next available destination after performing the task at the current destination. The `FileSearcherMaster` class remains unchanged. In general, this modified version of the Slave class can be used to repeat a task (defined in `doJob()`) at multiple destinations, in the order defined by a separate itinerary object. The only required change to the `FileSearcher` class was modification of its `doJob` method to combine the result of a local search with previous results.

6. CONCLUSION

In this paper we have reported on several design patterns that we have found in mobile agent applications. Weary of inventing and re-inventing solutions to recurrent problems, we have found that agent design patterns can help by capturing solutions to common problems in agent design. We have also found that no special skills, language features, or other tricks are required to benefit from these patterns.

We expect agent patterns to pragmatically fill in the space between very high-level agent-specific languages and system-level programming languages such as Java. Patterns can also provide a sound foundation for visual agent development environments. We envision that the agent developer can select and combine multiple patterns in a graphical environment. Based on standard implementations of these patterns, the development environment can generate agents with the desired properties.

Design patterns have proved highly useful within the object-oriented field, and have helped to achieve good design of applications through reusability of validated components. We hope that the design patterns described in this paper and the catalog will serve this purpose in the context of mobile agent-based applications. Specifically, we hope that this paper will motivate others to continue and discover more patterns that make it easier for designers of distributed applications to learn and use of agent technology. We also believe that non-mobile agent systems will benefit from the pattern idea, and we strongly encourage the development of a pattern catalog for agents of this type as well.

7. ACKNOWLEDGEMENTS

We wish to thank Mitsuru Oshima for his feedback on the proposed patterns and in particular to acknowledge Kazuhiro Minami's creation of the Plan pattern. We are also grateful to Mike McDonald of IBM Japan for checking the wording of this paper.

8. REFERENCES

[1] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[2] Arnold, K. and Gosling, J. *The Java Programming Language*. Addison-Wesley, 1998.

[3] Lange, D. B. and Oshima, M. *Programming and Deploying Mobile Agents with Java*. Forthcoming book. Addison-Wesley, 1998.

[4] White, J., Telesrcipt Technology: Mobile Agents. In Bradshaw, J. (ed.) Software Agents. MIT Press, 1997.

[5] IBM. Aglets Workbench. <`http://www.trl.ibm.co.jp/aglets`>

[6] General Magic Inc. Odyssey. <http://www.genmagic.com/agents>

[7] Rumbaugh, J. et al. *Object-Oriented Modeling and Design*. Prentice Hall, 1991