Design Patterns for Managing Product Lifecycle Information

Kary Främling, Timo Ala-Risku, Mikko Kärkkäinen, Jan Holmström

Introduction

The increasing demands on *product lifecycle management* means that information about products has to be easily accessible during the product's entire lifetime. At the same time, the increasingly complex supplier networks (the "virtual enterprise") increase the need to exchange product information between organisations [9]. Increasing product customisations make it necessary to handle information on the product item level rather than on the product type level, which greatly increases the amount of product information [7]. Therefore vast amounts of product and component information are currently pushed forward in the supplier network so that all information can be associated with the final product. This easily leads to an information overflow in the downstream supply chain [10].

Transferring product information between supplier network members is technically challenging [9]. As shown by current EDI implementations, setting up such information links is costly and time-consuming for all participating companies. Even though information links exist, handling changes in products and in the information about them is not an easy task, especially when done on a product item level. The challenge is to know where the information should be updated if there are multiple copies of it in different companies.

In software engineering, *object-oriented programming* (OOP) was developed as a solution to similar problems that initially occurred in simulation systems and graphical user interfaces [2], which both require managing great amounts of structured data. In this paper, we show how an agent-based architecture can fulfil the same role as objects in OOP, where *product agents* benefit from OOP methods with only simple amendments to present IT services. The main technical difference is that agents need to be able to communicate over company bounds, while objects usually communicate inside a program running on a single computer.

Now that Internet is becoming universally accessible (at least in the industrial context), product information can be made available anywhere without copying it through a supplier network. Many companies already have existing web services, where product information is accessible at least on the product-type level. The challenge is how to easily know where that data is, how to access it, how to distribute the data to all parties needing it, and how to update the data in all places of storage. A simple solution is that the manufacturer of a product attaches product- or item-specific identities on all products manufactured. If the identities are globally unique, they can be used as references to where the product (or product item) information is accessible. Such an identity corresponds to an *object reference* in OOP, while message passing between agents corresponds to method calls in OOP. With these basic elements, it is possible to use standard OOP data structures and practices, e.g. design patterns, for managing product information in a way that is already tested.

From Object Oriented Programming to Product Agents

During the 1980's, the old procedural programming paradigm changed into an object-oriented paradigm. OOP has since then become the dominant paradigm in software engineering. A main reason for this was that OOP makes it easier to manage data and functionality of a program by concentrating them around the object-concept. This means that anyone (usually another object) that has a reference to the object can access information about the object through the object's methods. The available methods are declared in the class' *public interface*.

One-to-many relationships between objects are implemented by using *object containers* (or actually collections of references to objects) that exist in OOP at least since the Smalltalk programming language [4]. Object containers have obvious counterparts in the real world, e.g. containers, packages, sub-assemblies etc. More complex structures, e.g. hierarchies, can be constructed using object containers in standard ways called *Design Patterns* [3]. Design Patterns are reference solutions to many information management tasks, which have been designed by experienced programmers and tested in multiple applications.

Where OOP offers "object-centric information management", product agents offer "Product centric information management" [5][6][7], where information regarding a product is retrieved over information networks when needed using unique product identities as references. Similar efficiency gains as those obtained by moving to OOP can be expected by moving to agent-based product information management.

A common definition of an agent says that an agent should be autonomous, social, reactive and pro-active [11]. In practice there is no universal agreement on what an agent is and the agent concept is used in many different ways. A product agent is implemented as a software component that is accessible over Internet using some standardized communication protocol. The reference to an agent has to be globally unique and indicate the Internet address where the agent can be accessed. For the agent reference, we have proposed using an ID@URI format, where the URI part identifies a resource, whose uniqueness is guaranteed by definition [1]. The ID part can use some existing identification standard as long as it remains unique inside the address space of the URI. Standards for globally unique identifiers have also been developed, e.g. GTIN (Global Trade Item Number), GLN (Global Location Number) and EPC (Electronic Product Code).

In OOP, the communication between objects is based on method calls, but in the product agent concept communication is performed through messages between the product agents, typically based on XML. The association between methods and messages is not new, for instance Goldberg and Robson [4] call communication between objects "message passing", not "method calling". Interfaces have their direct correspondence in common communication protocols, e.g. SOAP, Corba. Object containers are implemented by using database tables. Table 1 summarises the relationships between these basic concepts of OOP and product agent systems.

OOP concept	**Product agent concept**
Object	Agent: Internet-accessible software component
Object reference	ID@URI or similar
Method	Message
Interface	Interface defining messages and message formats
Object Container	Database table for object relations, stored as ID@URI references

Table 1. Correspondence between basic concepts in OOP and Agent systems.

Using these basic parallels between OOP and agent systems, we will next study how OOP collections and Design Patterns can be used for handling product information in two typical product lifecycle management contexts.

Design patterns for communicating product agents

For products with parts made by different companies, product information may have to be fetched from several sources. The same applies to information updates, where the updated information has to be forwarded to the information systems of all companies that are concerned by the update. Several projects conducted with industrial partners at Helsinki University of Technology have revealed that this is a major issue for information management. Two common industrial requirements and solution models for them are discussed in this section:

1. *Composite products*. These are typically products that contain sub-assemblies made by different companies. Shipment units, e.g. containers, pallets etc., are also treated under this title.
2. *Observers*. Companies that have not manufactured sub-assemblies may also need to receive information updates, even though they do not provide any information about the product. A transportation company, the recipient of a product in transport, or the users of recalled products are examples of these kinds of companies.

Design patterns used in OOP exist under similar names for both of these requirements.

Composite

One of the most important design patterns presented in OOP is the "Composite" design pattern [3]. The intent of this design pattern is to compose objects into tree structures to represent part-whole hierarchies, where individual objects and compositions can be treated uniformly. One of the most common uses of this design pattern is in drawing programs, where graphical objects may be grouped together to form new objects, which can then be grouped together with others etc. A set of operations is then applicable both to groups and objects.

The same is true for both products and shipment units used in transport, which usually contain parts that come from many different companies. This signifies that physical product items become parts of each other, so the information related to them becomes interconnected. The construction of composite products usually does not change too much during the life cycle of most products, but it is a vital piece of information to manage when changes occur. Such changes occur when a shipment is transferred from one transport container to another, or when a part of a product is replaced with another during maintenance or re-furbishing. Often the product individual forms a multi-level hierarchy, in which a product individual

consists of a set of other product individuals as illustrated in Figure 1 where every part of the product has a list of references to the parts they contain in ID@URI format.

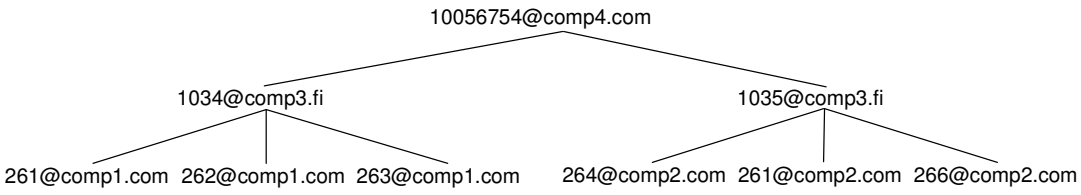

Figure 1. Example of composite product hierarchy.

Figure 2 illustrates the propagation of an information update, e.g. location update, through the containment hierarchy of a composite product. In many cases, only the identifier of the "outermost" part is accessible for reading. The outermost part is also the one at the top of the containment hierarchy, which makes it easy to propagate the information update to all the parts of the containment hierarchy. Fetching information about composite products works in a similar way. All companies in Figure 2 have product agents that receive the messages and autonomously decide on what information should be forwarded. This is important because it allows every company to control exactly what information is sent where.

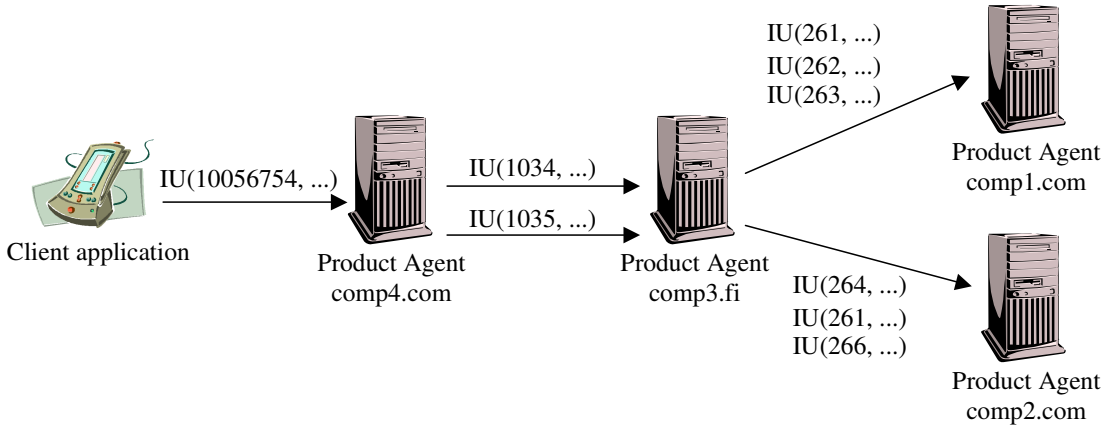

Figure 2. Propagation of information updates (IU) for the composite hierarchy of Figure 1.

The "Composite" design pattern recommends using bi-directional links in the hierarchy, which means that information can be fetched and updated by reading the identifier of any part of the composite. Messages currently implemented from the "Composite" design pattern are "GetComposite" (returns list of parts), "add" and "remove" ("add" and "remove" actually use only one message with a parameter indicating the operation to perform). Removing a part from a composite product can be done either explicitly by sending a message or implicitly by

receiving a location update message that indicates that the part has been removed from the composite. We assume that the best way to handle removal of parts will turn out to be rather application-specific.

Observer

The intent of the "Observer" design pattern is to define one-to-many dependencies between objects so that when one object changes state, all its dependents are notified and updated automatically [3]. One of the most common uses of this design pattern is in graphical user interfaces (GUI), where user actions on one GUI element also affect other GUI elements. Standard GUI classes of the Java language are an example of this, where the observer pattern is used in numerous "listener interfaces". This can be used for propagating information updates in cases where the "Composite" pattern is not applicable, e.g. when both the sender and the receiver need to track a shipment, or when break-downs of product components need to be communicated to a logistics company handling spares replenishments in addition to the producers of the product.

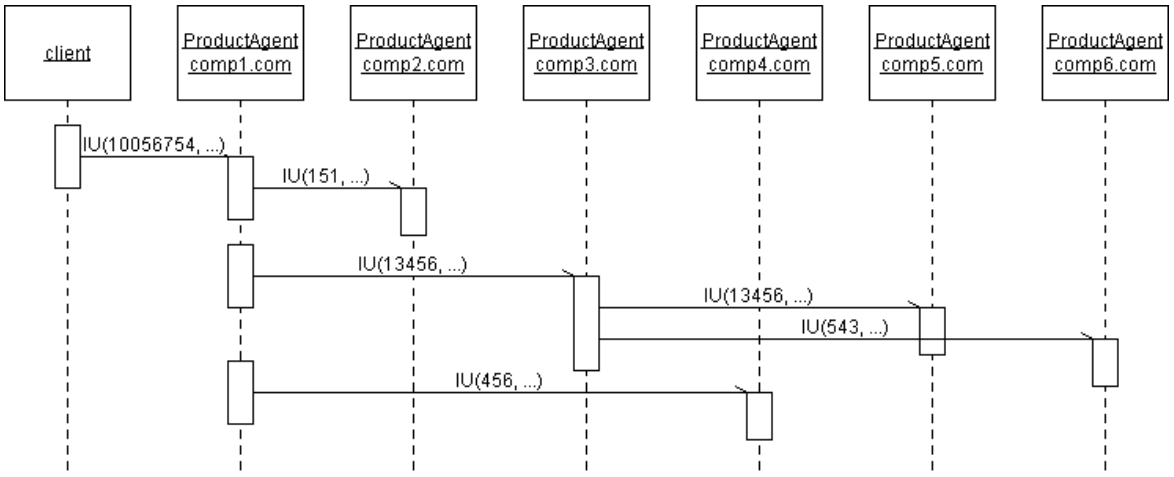

Figure 3. Propagation of information update (IU) through "Observer" references as for "Composite" in Figure 2, here represented as UML sequence diagram.

Figure 3 illustrates how an information update could be propagated through "Observer" references. Items 13456@comp3.com, 151@comp2.com and 456@comp4.com observe item 10056754@comp1.com. Items 13456@comp5.com and 543@comp6.com observe item 13456@comp3.com. Therefore the information update message shown in the figure will be sent to the corresponding product agents.

Implementing the "Observer" design pattern means that messages have to exist for adding observers and removing them. Observers are also usually interested only in specific

information updates, so there may be several different add and remove messages for different kinds of information updates.

Implementation

In OOP, objects live and evolve inside the working memory of the computer that the program is running on. If the object data needs to be persistent, then it has to be written to a file or stored into a database. The same is true for product agents; persistent data is usually stored in databases. Therefore, product agents are a bridge between external access to product information over Internet and access to information stored in company databases. Such software components are usually named by the term *middleware* because they enable otherwise isolated information systems to communicate with each other.

At Helsinki University of Technology we have developed a pilot product agent system that is implemented according to the above specifications. The software is available at "http://dialog.hut.fi" and is distributed under the terms of the GNU Lesser General Public License. We will first study the implementation from a messaging perspective, and then explain how "object containers" are implemented using database tables, followed by practical experiences from industrial pilots.

Messaging protocols

The current implementation uses three alternative communication protocols for message passing, i.e. SOAP, HTML form and Java RMI. These different protocols are supported because they all provide different trade-offs between ease of installation and maintenance, firewall configuration, communication speed etc. In order to make the system as open as possible, a major challenge is to standardise the messages used independently of the communication protocol itself so that any software producer could implement them and have their applications communicate with others successfully.

Database tables as agent containers

The software components are programmed in Java, so the communication with databases is performed using the Java Database Connectivity (JDBC) protocol. For now, databases seem to be the only universal way to communicate with most existing Enterprise Resource Planning (ERP) systems because ERP systems tend to have proprietary Application Programming Interfaces. A more direct integration with ERP systems could be necessary for

real-time applications with short reaction times, but for most product lifecycle management applications it should not be necessary.

Database tables used by the product agents are created by the middleware component as/when needed. Relations between product items are stored as three fields of a database table as "relation_name;subjectReference:ObjectReference" where references are stored as ID@URI. "Composite" and "observer" data structures use relation names such as "parent", "child", "observe" etc. Since the middleware components manage these relations, both "composite" and "observer" functionality can be implemented without modifying existing information systems.

Experiences from pilot installations

Two pilot installations have been performed in a multi-company environment [8]. The first pilot consisted in tracking project shipments in a global environment. Shipments were identified with RFID technology and tracked at several checkpoints. The second pilot consisted in tracking incoming and outgoing goods at warehouses of several third-party contractors. Goods were identified with barcodes. These pilot installations show that the middleware messaging architecture presented here is operational. "Composite" and "observer" have been implemented and tested by exchanging messages defined by the corresponding design patterns in a similar way as in the pilot installations. Therefore this kind of middleware can provide a common technical platform for distributed management of product information, independently of the application area. In what way these design patterns are used in practice (user interfaces, connections to existing software, …) can be expected to be highly application-dependent.

Conclusion

In this paper we have presented basic building blocks for using OOP concepts to manage product lifecycle information in a multi-supplier context. These building blocks make it possible to create distributed, agent-based information architectures that make product information accessible in controlled ways over the Internet. We have also presented how well known OOP practices can be applied to information management of composite products and for propagating information updates through the "observer" pattern.

The concepts have been implemented using several communication protocols and data formats, which all have different advantages and challenges as shown by experiences from two industrial pilot installations. Using different protocols does not have to be a problem

because the same product agent component can simultaneously support all communication protocols and data formats used. However, establishing universally accepted standards especially for message formats remains one of the biggest challenges of the future. We believe that by keeping our system open, i.e. open-source and general-purpose, it will be possible to implement new functionality for applications not yet identified while allowing the integration of evolving standards.

References

1. Berners-Lee, T., Fielding, R., Irvine, U.C., Masinter, L. (1998). *Uniform Resource Identifiers (URI): Generic Syntax*. Available online (March 5th 2004): "http://www.ietf.org/rfc/rfc2396.txt"
2. Dahl, O.-J., Nygaard, K. (1966). SIMULA – an ALGOL-Based Simulation Language. *Communications of the ACM*, Vol. 9, No. 9. pp. 671-678.
3. Gamma, E., Helm, R., Johnson, R., Vlissides, J. (1995). *Design Patterns: elements of reusable object-oriented software*. Addison-Wesley Publishing Company, Reading, Massachusetts.
4. Goldberg, A., Robson, D. (1983). *Smalltalk-80: The language and its Implementation*. Addison-Wesley Publishing Company, Reading, Massachusetts.
5. Kärkkäinen, M., Holmström, J., Främling, K., Artto, K. (2003). Intelligent products - a step towards a more effective project delivery chain. *Computers in Industry*, Vol. 50, No. 2. pp. 141-151.
6. Kärkkäinen, M., Främling, K., Ala-Risku, T. (2003). Integrating material and information flows using a distributed peer-to-peer information system. In: Jagdev H.S., Wortmann J.C., Pels H.J. (eds.) *Collaborative Systems for Production Management*, Kluwer Academic Publishers, Boston, USA. pp. 305-319.
7. Kärkkäinen, M., Ala-Risku, T., Främling, K. (2003). The product centric approach: a solution to supply network information management problems? *Computers in Industry*, Vol. 52, No. 2. pp. 147-159.
8. Kärkkäinen, M., Ala-Risku, T., Främling, K. (2004). Efficient Tracking for Short-Term Multi-Company Networks. *International Journal of Physical Distribution and Logistics Management*, Vol. 34, No. 7. pp. 545-564.

9. Petrie, C., Bussler, C. (2003). Service Agents and Virtual Enterprises: A Survey. *IEEE Internet Computing*, Vol. 7, No. 4. pp. 68-78.

10. van Dorp, K.J. (2002). Tracking and tracing: a structure for development and contemporary practices. *Logistics Information Management*, Vol. 15, No. 1. pp. 24-33.

11. Wooldridge, M., Jennings, N.R. (1995). Intelligent Agents: Theory and Practice. *Knowledge Engineering Review*, Vol. 10, No. 2. pp. 115-152.