Managing Product Information in Supplier Networks by Object Oriented Programming Concepts

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

*) *Department of Computer Science and Engineering, Helsinki University of Technology, P.O. Box 5400, FIN-02015 HUT, Finland*

**) *Department of Industrial Management and Engineering, Helsinki University of Technology, PO BOX 9555, FIN-02015 HUT, Finland*

Abstract

As the number of companies participating in the manufacturing of products increases, the challenges on managing product related information over its life cycle also increase. A major issue is how to manage product-related information when it is created and stored on computer systems of multiple companies. *Object-oriented programming* (OOP) is a well-tested framework for managing information in computer programming. In this paper, we emphasize the similarities between OOP and *product agent* based information management models for handling product information. The well-known OOP concept of *design patterns* is shown to be applicable also to managing product information in supplier networks. The *composite* and *observer* design patterns are applied to two typical information management tasks using product agents. Their implementation as middleware software components is outlined, together with results obtained from pilot installations in a multi-company environment.

Keywords

Product agent, object-oriented programming, design pattern, product identity, middleware

1 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 intensifying technical sophistication of products and the increasingly complex supplier networks make managing the information more challenging than ever [Kärkkäinen, Ala-Risku, Främling, 2003]. This is why 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 [Främling, Holmström, Ala-Risku, Kärkkäinen, 2003]. However, in complex products this can lead to information overflow in the downstream supply chain, when the amount and sophistication of product components increases [van Dorp, 2002].

Transferring product information between supplier network members is technically challenging. As shown by current EDI implementations, implementing 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 mainly in simulation systems and in graphical user interfaces. Both these tasks require managing great amounts of structured data, which is now the challenge also with managing product information. In this paper, we propose an agent-based architecture where agents fulfil the same role as objects in OOP. The main technical difference is the fact 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. 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 for managing product information in a tested way that is already tested.

Further in this paper we show in more detail how *product agents* can benefit from OOP methods with only simple amendments to the present IT services. Section 2 gives an overview of OOP and section 3 describes the relation between OOP and product agents. Section 4 shows how OOP containers and design patterns can be applied to handling product information of complex products. Section 5 explains how these methods have been implemented and tested in industrial pilot installations, followed by conclusions.

2 Object Oriented Programming

During the 1980's, the old procedural programming paradigm changed into an object-oriented paradigm. A main reason for this was that object-oriented programming [Dahl, Nygaard, 1966] 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. In software engineering, OOP has become the dominant paradigm.

A general, but at least universally acceptable, definition of an object is that "an object is a collection of structured information". Objects store information both directly through basic data types, e.g. text and numbers, and through references to other objects. Contents of objects can only be accessed through their public *interface*, which in OOP is generally called *encapsulation*. An interface only declares methods that implementing classes have to implement, but it does not provide any method implementations. Method declarations contain 1) a method name, 2) a list of parameters and their type and 3) the type of the value returned by the method. The returned value can also be a reference to an object.

Inheritance is another key feature of OOP. Inheritance signifies the ability to derive new classes that inherit much of their code and functionality from the classes on which they are based. Inheritance can be extended to the concept of implementing one or more interfaces. Implementing interfaces is the way that multiple inheritance is done in the Java programming language [Arnold, Gosling, 1996], for instance. Inheritance also allows for *polymorphism*, which signifies that methods declared in the interface may be called for objects of any class implementing the interface.

Standard *object containers* (or actually collections of references to objects) exist in OOP at least since the Smalltalk programming language [Goldberg, Robson, 1983]. Object containers enable one-to-many relationships. 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* [Gamma, Helm, Johnson, Vlissides, 1995]. Design Patterns are reference solutions to many information management tasks, which have been designed by experienced programmers and tested in multiple applications.

3 Product Agents

Where OOP offers “object-centric information management”, product agents offer “Product centric information management” [Kärkkäinen, Holmström, Främling, Artto, 2003], [Kärkkäinen, Främling, Ala-Risku, 2003], [Kärkkäinen, Ala-Risku, Främling, 2003], 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.

An agent is implemented as a software component that is accessible over Internet using some standardized protocol, e.g. http, SOAP. 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 [Främling, 2002], [Kärkkäinen, Främling, Ala-Risku, 2003], where the URI part identifies a resource, whose uniqueness is guaranteed by definition [Berners-Lee, Fielding, Irvine, Masinter, 1998]. The ID part can use some existing identification standard as long as it remains unique inside the address space of the URI [Huvio, Grönvall, Främling, 2002]. The Auto-ID Centre at MIT has also identified the need for globally unique identifiers. The Electronic Product Code (EPC) [Brock, 2001] has been developed for this purpose, but it still needs to become widely accepted and integrated into companies before becoming universally usable. An EPC also makes it possible to access the URI part through the Object Name Service (ONS) [Auto-ID Center, 2003a]. ONS specifications are currently on a draft level, but using EPC as the ID part in ID@URI enables developing Auto-ID compatible applications before the ONS infrastructure is operational.

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, [Goldberg, Robson, 1983] for instance call communication between objects “message passing”, not “method calling”. Interfaces have their direct correspondence in common communication protocols, e.g. SOAP. Object containers are implemented by using database tables as explained in section 5.2. Table 1 summarises the relationships between these basic concepts of OOP and agent-based systems.

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

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 information in two typical industrial contexts.

4 Design patterns for communicating product agents

In supplier networks, 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 [Aerts, Szirbik, Goossenaerts, 2002], [Främling, 2002], [Främling, Holmström, Ala-Risku, Kärkkäinen, 2003]. 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.

4.1 Composite

One of the most important design patterns presented in OOP is the "Composite" design pattern [Gamma, Helm, Johnson, Vlissides, 1995]. 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. 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.

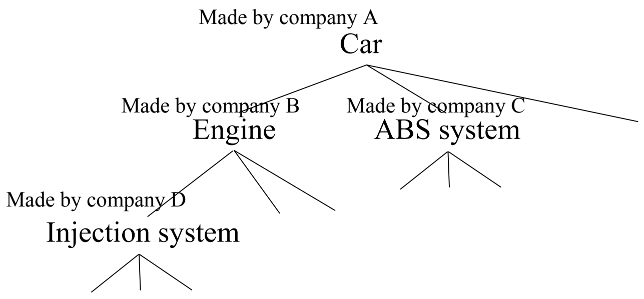

Figure 1. Example of composite product hierarchy.

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.

Figure 2 illustrates a composite product, where every part of the product has a list of references to the parts they contain in ID@URI format, and the propagation of an information update, e.g. location update, through the containment hierarchy. 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.

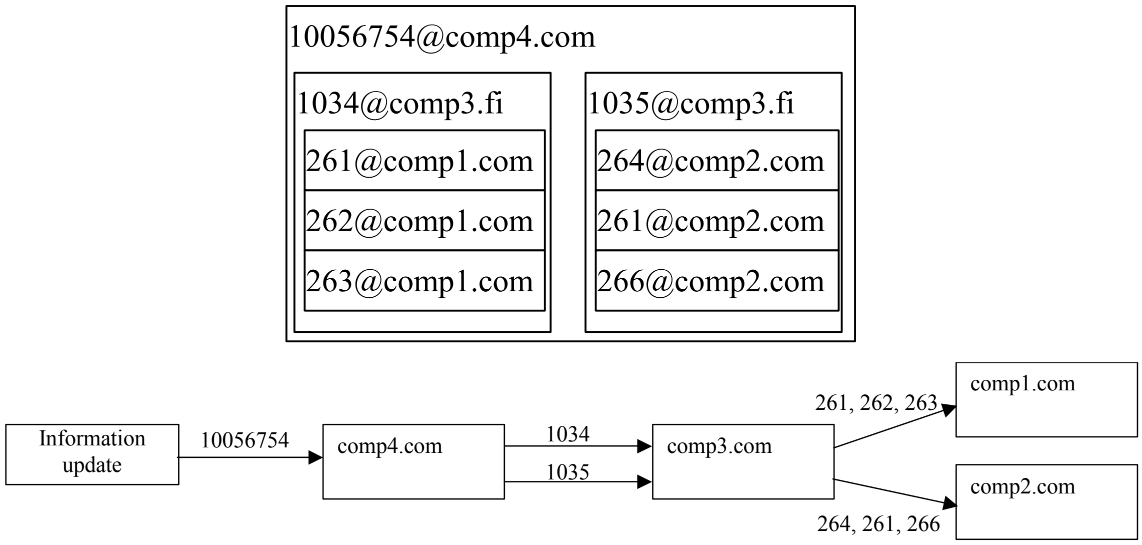

Figure 2. The construction of a composite product and the propagation of information updates through the composite hierarchy.

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" (as optional parameter of existing "LocationUpdateMessage") and "remove". 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. A more thorough description of the information management challenges of complex products as well as a more detailed description of the proposed solution with product agents can be found in [Främling, Holmström, Ala-Risku, Kärkkäinen, 2003].

4.2 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 [Gamma, Helm, Johnson, Vlissides, 1995]. 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.

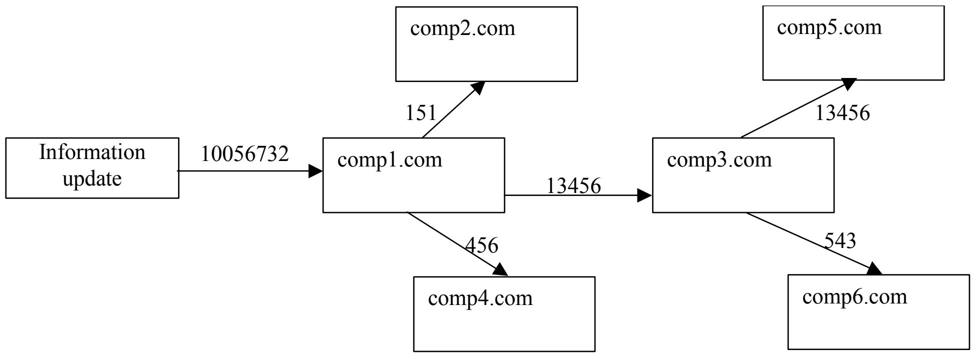

Figure 3. Propagation of information update through "Observer" references.

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 10056732@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.

5 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.

5.1 Messaging protocols

The current implementation uses three different communication protocols and data formats for message passing:

- SOAP messaging [W3C, 2000]. Programming language-independent protocol. Data is transferred as text using the XML notation.
- HTML form [W3C, 1999]. Programming language-independent protocol. Data is transferred as text using the HTML form format.
- Java RMI messaging [Sun Microsystems, 2002b]: Mainly used in development and in intra-company installations. RMI is flexible and easy to use, but it may be problematic for firewalls and version management when service interfaces change.

SOAP is a standard message exchange paradigm based on HTTP for exchanging information between peers in a decentralized and distributed environment. SOAP should be suitable for inter-company information exchange, but experiences from multi-company pilot installations indicate that firewalls may have to be re-configured also for this protocol. Using HTML forms avoids such firewall problems, but the underlying HTTP protocol may limit future agent communication due to its strict request/response model.

In order to make the system as open as possible, a major challenge is to standardise these communication protocols and messages so that any software producer could implement them and have their applications communicate with others successfully. This is also one of the main goals in the Savant middleware specifications under development at the Auto-ID center [Auto-ID center, 2003b].

5.2 Database tables as agent containers

The software components are programmed in Java, so the communication with databases is performed using the JDBC (Java Database Connectivity) protocol [Sun Microsystems, 2002a]. 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 some 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. "Composite" and "observer" use one database table each, where the needed agent relations are stored as ID@URI pairs. Since the middleware components manage these relations, both "composite" and "observer" functionality can be implemented without modifying existing information systems.

5.3 Experiences from pilot installations

Two pilot installations have been performed in a multi-company environment. The first pilot consisted in tracking project shipments in a global environment [ISI Industry Software, 2003]. Shipments were identified with RFID technology and tracked in several checkpoints. Location update messages were sent using the RMI protocol, which in this case worked despite multiple firewalls.

The second pilot consisted in tracking incoming and outgoing goods at warehouses of several third-party contractors. Goods were identified with barcodes, while location update messages were initially sent using SOAP. Due to firewall problems that occurred with some contractors, messaging was later modified to use HTML forms. The conclusion of these pilot installations is that the middleware concept presented here is operational. This also provides a proof-of-concept for the technical aspects needed to implement "composite" and "observer" patterns, even though they were not used in these pilots.

6 Conclusion

In this paper we have presented basic building blocks for using OOP concepts to manage multi-supplier product information. These building blocks make it possible to create distributed, agent-based information architectures that make 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 is easy to implement new functionality for applications not yet identified while allowing easy integration of evolving standards.

Acknowledgement

This research has been supported mainly by Tekes, the National Technology Agency of Finland. We also thank the companies who have contributed in the financing and validation of the methods described here.

References

Aerts, A.T.M., Szirbik, N.B., Goossenaerts, J.B.M., (2002). A flexible, agent-based ICT architecture for virtual enterprises. *Computers in Industry*, Vol. 49, No. 3. pp. 311-327.

Arnold, K., Gosling, J. (1996). The Java Programming Language. Addison-Wesley Publishing Company, Reading, MA.

Auto-ID center (2003a). *Auto-ID Object Name Service (ONS) 1.0*. Available online (March 5th 2004): http://www.epcglobalinc.org/standards_technology/Secure/v1.0/WD-ons-1.0-20030930.pdf

Auto-ID center (2003b). *Auto-ID Savant Specification 1.0*. Available online (March 5th 2004): http://www.epcglobalinc.org/standards_technology/Secure/v1.0/WD-savant-1_0-20030911.doc

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"

Brock, D.L (2001). *The Electronic Product Code (EPC) - A Naming Scheme for Physical Objects*. MIT Auto-ID Center White Paper, January 2001. Available online (December 13th, 2002): http://www.autoidcenter.org/research/MIT-AUTOID-WH-002.pdf

Dahl, O.-J., Nygaard, K. (1966). SIMULA – an ALGOL-Based Simulation Language. *Communications of the ACM*, Vol. 9, No. 9. pp. 671-678.

Främling, K. (2002). Tracking of material flow by an Internet-based product data management system (in Finnish: Tavaravirran seuranta osana Internet-pohjaista tuotetiedon hallintaa), *Tieke EDISTY magazine*, No. 1, 2002 (Tieke: Finnish Information Society Development Centre, Finland).

Främling, K., Holmström, J., Ala-Risku, T., Kärkkäinen, M. (2003). *Product agents for handling information about physical objects*. Laboratory of Information Processing Science series B, TKO-B 153/03, Helsinki University of Technology. Available online: http://www.cs.hut.fi/Publications/Reports/B153.pdf

Gamma, E., Helm, R., Johnson, R., Vlissides, J. (1995). *Design Patterns: elements of reusable object-oriented software*. Addison-Wesley Publishing Company, Reading, Massachusetts.

Goldberg, A., Robson, D. (1983). *Smalltalk-80: The language and its Implementation*. Addison-Wesley Publishing Company, Reading, Massachusetts.

Huvio, E., Grönvall, J., Främling, K. (2002). Tracking and tracing parcels using a distributed computing approach. In: SOLEM, Olav (ed.) *Proceedings of the 14th Annual Conference for Nordic Researchers in Logistics (NOFOMA'2002)*, Trondheim, Norway, 12-14 June 2002. pp. 29-43.

ISI Industry Software (2003). *Consignment Tracking for Heavy Industry*. Available online (March 23rd, 2004): http://www.isiindustrysoftware.com/news/kvaerner.html

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.

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.

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.

Sun Microsystems (2002a). *JDBC™ Data Access API*. Available online (December 13th, 2002): http://java.sun.com/products/jdbc/

Sun Microsystems (2002b). *RMI Specification*. Available online (December 13th, 2002): http://java.sun.com/products/jdk/1.2/docs/guide/rmi/spec/rmiTOC.doc.html

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.

W3C (1999). *HTML 4.01 Specification*. Available online (March 24th, 2004): http://www.w3.org/TR/html401/

W3C (2000). *Simple Object Access Protocol (SOAP) 1.1*. Available online (October 14th, 2003): http://www.w3.org/TR/SOAP/