RC 21597 (log 97395) 15NOV1999 **Computer Science/Mathematics**

Research Report

Outerfaces: How to Make Less Fragile Software Than Interfaces Allow

William Harrison
IBM Research Division
T.J. Watson Research Center
P.O. Box 704
Yorktown Heights, NY 10598

IBM Research Division
Almaden – Austin – Beijing – Haifa – T.J. Watson – Tokyo – Zurich

ABSTRACT
Outerfaces are a generalization of the concept of interfaces. Unlike interfaces, which characterize a set of functions supported by a provider through a port or reference, outerfaces characterize a set of functions provided to holders of a reference within a community, without identifying a particular provider. This paper introduces the concept and illustrates the problem they solve.

Keywords
Interface, Object-oriented, packaging, flexibility, fragility

Introduction
Conventional Object-Oriented programming produces programs that are extremely fragile with respect to the repackaging of function. Software suppliers often find they need to restructure implementation hierarchies, as evidenced by the number of solutions for the "class hierarchy reorganization problem" [e.g. 4, 7]. But in many strongly-typed languages this reorganization breaks client code as well as supplier code. This "component-structure fragility" is intimately connected with the idea of "interfaces" which intrinsically commit the code in a function consumer to a particular package as the supplier. But, in strongly-typed systems, interfaces play a role in type-checking, to prevent a different sort of breakage. "Missing-component" fragility results from failing to detect that necessary components are missing. This paper describes how such fragility arises, and how a broader interface-like concept called "outerfaces" can be used to eliminate both sorts of fragility, rather than just the one sort that the concept of "interface" deals with.

Models For Function Consumers

The Common Model for OO
In Object-Oriented languages a request for function is made by "sending a message to an object" or "calling a method of an object". So, assuming that "item, store, and customer" are variables that refer to objects of those types and that a sale of an item to a customer of a store is registered by calling the registerSale method of the item being sold, a Java programmer might write:

"item.registerSale(store, customer);"

To make this request for function, the consumer-programmer examined the published ***interfaces*** for item, store, and customer to find the definition of "registerSale", and wrote the logic to call "registerSale" on the item, passing it the store and customer, as defined in the interface to item.
To enable this request to be satisfied, the supplier-programmer registered the implementation by providing code for the class of object referenced by "item" to the name-binding system (generally the linker or loader), and published the interface describing the functions supported by items.
To deliver the request, the delivery infrastructure simply goes to the object identified by "item" to determine where to find its implementation of "registerSale".
The common model for OO prevents missing-component fragility. In the absence of a network connectivity failure or of a language without referential integrity (like C++), delivery can be assured. The "registerSale" request is defined in the "item" interface that the supplier committed to support when the item was created and a reference to it was returned. Type-checking assures that the consumer can only issue "registerSale" requests using references to objects that support "item", and that the other parameters have the appropriate types. Finally, the name-binding system ensures that all of the code elements required to carry out the supplier's commitment are present in the system.

The Generic Function Model for OO
Early in the development of Object-Oriented Programming Models, another alternative also contended for consideration by language designers. This "Generic Function Model", was adopted primarily by LISP-based programming languages, notably, the Common Lisp Object System (CLOS)[1], and, more recently, the Cecil

language[2]. CLOS used LISP's sophisticated management of name spaces to allow the registration of many different implementations of a method, one or more of which would be applied when different combinations of parameter types were used. Common combinations included "at all times" or "if a certain parameter is an object of a certain class". To reflect the most similarity to the above case, the circumstance here would be "if the 'item' parameter has object class 'item'".

To accomplish the task discussed above, the CLOS programmer would write the LISPish equivalent of:

"registerSale(item, store, customer)"

To make this request for service, the programmer examined the space of published generic functions to find a definition for registering sales, and wrote the logic to call registerSale, passing it the item, store and customer, as defined in the interface to registerSale.

To enable this request to be satisfied, the supplier-programmer registered the implementation by providing code for an implementation of "registerSale" with a description of the appropriate circumstances to the name-binding system (the LISP name-management system), and published the function and the circumstances which the supplied implementation supports.

To deliver the request, the delivery infrastructure simply goes to the function definition for "registerSale" to determine where to find its implementation alternatives, and then identify and execute the one that applies when the "item" parameter is an object of class "item".

Assurance of delivery in the generic function model is problematic. In the common model of objects, the interface provides a contractual statement of needs or support. But the generic functions support in CLOS was implemented with Lisp's general lack of type-checking. So, in CLOS, neither consumer nor supplier asserts such a contract in many realizations of the generic function model. And while CECIL tightens up the type checking to be able to assure delivery, it does so at the cost of characterizing parameters in terms of their implementation classes rather than in terms of the interfaces they support..

By avoiding the common model's view of interfaces for single-delivery, the generic function model overcomes the problem of fragility that is causing many developers of large commercial software systems to use a paradigm other than that of Objects. However, the generic function model had generally been embedded in particular programming languages that do not have the accessibility of C++ or Java.

The Brokered Message Model

Many large enterprises are deploying message brokering and message delivery solutions as an alternative to distributed object solutions [11, 12, 13, 14, 15]. With these solutions, the consumer programmer brings the function name and the parameters of interest together by constructing a ***message***. The message is then sent to have the function performed. Message sending follows an event flow paradigm, and so the consumer is not expected to wait for a response. In order to realize the request/response paradigm commonly employed in object systems, the message sender may wait for delivery of a related message that contains the response.

Brokered message systems have all of the same technical disadvantages as the generic function model: lack of strong typing and of assurance of delivery to an appropriate implementation. In fact they often have more disadvantages, because message construction is usually done outside the normal call paradigm as data structure initialization, followed by a special "send" operation. In spite of these disadvantages, they are rapidly gaining in use and popularity as an alternative to distributed object systems. Over and over, developers cite two primary reasons for using message brokering rather than distributed objects:

- the consumer's code is independent of the specification of how to locate the service supplier(s) because the message model is symmetric, and
- the consumer and the supplier can function independently in time, perhaps even to the point of never co-existing, because of the use of an event model rather than a request/response model.

The rules by which a service supplier is found are supplied independently to a *message broker*, which takes the responsibility for reliably delivering the message. The message broker takes on the role that had been played by the

specialized language support in CLOS or Cecil systems. This allows it to take on the same kind of role which the OMG's CORBA ORB or Microsoft's DCOM ORB take in a distributed object system - facilitating communication among applications written in different languages and running in different environments in a network, and makes the generic function model widely accessible.

Delocalizing References and Interfaces

The problem, then, is to introduce the advantages given by an interface-like construct without making it depend on the reference to which it is attached. This problem has two intertwined roots: the fact that references to concepts like "person" must be coerced to refer to a particular object before they can be used to call for function and the fact that the set of functions that may be called is tied to the object in which that function is implemented.

Object References and Keys

The divorce of references from computational data is one of the root causes of component-structure fragility. In languages like Java, interfaces are attached to object references, special non-computational data used to name objects, and not to the computational data that really identifies the concept.

Consider the example of the Java use of registerSale used earlier. It is most likely that what was read from the bar-code reader of the purchased item was not, in fact, an object reference to an "item". More likely it was a piece of computational data like an integer or string. The string is only a reference to the item. Java, like other OO languages, divides the space of references up into those that may have interfaces and those that may not. The latter must be used to find a reference to an object, and are called "keys" instead of "references".

But this means that not only does a function-consumer use a syntactic device to specially denote where the function is packaged, but this device proliferates into antecedent code, making changes to the structure much more difficult than a simple syntactic redirection. The actual code used to register a sale in the Java code is more like:

"item = ItemRegistry.find(itemBarCode); item.registerSale(store, customer);"

Extra code has been written to convert the simple call fro function as it might have been written in a generic function system into a classic one with interfaces. The fact that this code exists causes the function requestor to be even more strongly sensitive than the simple syntactic device.

For example, it is unlikely that the interface definer would have required all objects keys to be converted to object references, making to code look like:

"item = ItemRegistry.find(itemBarCode); store=StoreRegistry(storeCode);"
customer = FrequentCust.findID(customer);
item.registerSale(store, customer);"

After all, this would mean deciding to which use the key is being put in order to use the correct look-up.

Even further, if the function of registering sales *is* moved from item to store, a natural suggestion is that the registerSale method in *item* be rewritten to invoke the registerSale method in *store* instead. Even aside from efficiency issues, there is a problem. As illustrated here, the *store*'s registerSale probably expects the item to be referred to by its key and not its reference. This means that the delegation method in item must be able to re-derive the item's bar-code. Although it may seem that this is clearly possible, consider the fact that customers might be denoted by any of a variety of keys from a frequent-customer id, to a home address, or to a default of "none". Not all implementations of items may be able to retrieve the particular key of interest.

This all bespeaks the need to write the invocation in terms of the original keys, treating them as the references, as well as the need to treat them all symmetrically from a syntactic point-of-view. The task of examining the classes or values to deliver the request to the appropriate implementation is given to the *dispatcher* or *message broker*. The dispatcher employs efficient multiple-dispatch algorithms like those described in [3], and may even use one or more of the keys to locate instance data in an object or data base.

But if references may be any sort of data, and not just special system-generated data, why are not all data references? The distinction lies in the fact that a reference also carries a statement of the functions to be accessed using the underlying datum. The means of expressing this statement of functionality without relation to its packaging is the ***outerface***.

Outerfaces, Signatures, and Required Support

Common object-oriented systems employ the concept of interface to specify a set of messages that can be sent to an object by its client. They are all messages that take a reference to the object in a syntactically distinguished position, in addition to other specified parameters. The interface is "the thing between" the client and the object referred to. Passing a reference to an object characterized by an interface passes the guarantee that the messages in the interface may be sent to that object. To retain the desirable avoidance of missing-component fragility, we need to employ an interface-like concept that enables us to locally check that a guarantee that a message sent will be received. But, to realize the objective of eliminating component-structure fragility, we need to extend the concept to remove the special role of "the object to which a message is sent" in its definition. Therefore, an ***outerface*** is a specification for a reference type of a set of messages, all of which have at least one parameter of the reference type. The outerface specifies a set of messages exchanged between a component and the entire collection of components that make up a larger system, without reference to which component or object will service them. Passing a reference characterized by an outerface passes the guarantee that some component the system will respond to those messages when that reference is used.

The local type-checking rule for a operation call involving references characterized by outerfaces is: **the operation call being made must be included in the outerface of at least one of the parameters of the operation**, with the usual rules of subtyping applied.

A better understanding of outerfaces and the way they eliminate component structure fragility can be obtained by looking at a program written in a "little outerface language".

A Little Outerface Language Illustration

Perhaps the easiest way to illustrate the differences between interfaces and outerfaces is to present a computationally vacuous programming language that we can call LOLI. LOLI has seven constructs: Class Definition, Type Definition, Outerface Definition, Message Definition, Message Implementation, Message Call, and Local Object Definition. LOLI is computationally vacuous in that it has no constructs for doing computations on primitives, allocating storage, assignment, or anything else although, in most cases, it is clear how a similar real language would be created. Simply presenting the BNF for LOLI would be confusing in its choices, so we present a small LOLI program that allows us to define the registerSale message used at the beginning of this paper, choosing to use charge cards to represent the customers who use them.

LOLI is described with an object-oriented flavor, in its use of the concept of class, but does not depend on the concept of "object" except as an abstraction for the things that can be done with a reference to that class. In a sense, LOLI provides for programming with classes, references, and interfaces, without actually insisting on using objects. Classes also provide a simple way of registering implementations. As with the subjective views of objects discussed in [5], a Class does not provide a single point of definition for state and behavior. Many different definitions of "Card" may be provided which may yield a realization of the conceptual class as one or many objects. The LOLI statements in this example are each followed by an explanation of the interpretation and characteristics of the statement. The BNF for LOLI can be found in the Appendix. In this example, Integer and String are taken to be predefined classes, and actual statements that would perform computations in some more real language (but not actually in LOLI) are italicized.

1. ***Class Card;***
   This class definition simply defines that there is a class called "Card". A definition like this can be used to form, for example, an abstract class intended to generalize a variety of particular "Cards". Unlike most OO

languages, LOLI's classes do not carry interfaces (or outerfaces). Outerfaces are carried not by the objects, but by the references to the objects.

2. ***Type CardRef = referenceTo Card as string;***
   This type definition defines a reference type called CardRef. A CardRef us used to identify things of class Card. It is represented as a string.

3. ***Type CardId = referenceTo Card as string supporting CardLike;***
   This type definition defines a reference type called CardId. A CardId is expected to refer to a "Card" or to one of its specializations. It carries the assurance that the messages described in the CardLike outerface are supported.

4. ***Outerface CardLike {***
   getBalance(card:CardRef):(balance:integer);
   setBalance(card:CardRef, balance:integer):()
   };
   This outerface definition defines an outerface called CardLike. CardLike specifies support for two messages: "getBalance" and "setBalance", whose message definitions it contains. The message signature for getBalance has an element called "card", which is a CardRef. The response signature for getBalance has an element called balance, which is an integer. The message signature for setBalance has an element called "card", which is a CardRef and an element called balance, which is an integer. The response signature for setBalance is null.

5. ***Class Store;***
   This class definition defines a class called "Store" as described above for "Card".

6. ***Type StoreRef = referenceTo Store as string;***
   This type definition defines a reference type called StoreRef as described above for CardRef.

7. ***Type StoreId = referenceTo Store as string supporting StoreLike;***
   This type definition defines a reference type called StoreId as described above for CardId.

8. ***Outerface StoreLike {***
   sell(item:ItemRef, store:StoreRef):()
   };
   This outerface definition defines an outerface called StoreLike. StoreLike specifies support for the "sell" message, whose message definition it contains. The message signature for "sell" has an element called "item", which is an ItemRef and an element called "store", which is a StoreRef. The response signature for "sell" is null.

9. ***Class Item;***
   This class definition defines a class called "Item" as described above for "Card".

10. ***Type ItemRef = referenceTo Item as string;***
    This type definition defines a reference type called ItemRef as described above for CardRef.

11. ***Type ItemId = referenceTo Item as string supporting ItemLike;***
    This type definition defines a reference type called ItemId as described above for CardId.

12. ***Outerface ItemLike {***
    sell(item:ItemRef, store:StoreRef):();
    getPrice(item:ItemRef):(amount:integer);
    charge(card:CardId, item:ItemRef):()
    };

Ok, now things start to get interesting. This defines an outerface called ItemLike, containing three message definitions. Like StoreLike, ItemLike also specifies support for the same "sell" message, even though there is no "inheritance" relationship among the outerfaces. Because support for "sell" is specified by outerfaces associated with both the ItemId and StoreId reference types, a message implementation for some other message can use "sell" with either an ItemRef and a StoreId or with an ItemId and a StoreRef. It cannot use "sell" with an ItemRef and StoreRef because neither carries the assurance of "support" for "sell" (or any other operation). And since the syntax for sending messages is symmetric, that message implementation is robust with respect to any sense of "which" reference the support is actually coming from.

The definitions of "charge" and "getPrice" follow the general pattern established in the earlier outerface definitions. But the message signature for "charge" defines "card" as a CardId. **Using CardId rather than CardRef in defining "charge" means that any implementation of charge can use the operations in CardLike on the card.** This addresses the needs of implementations and places constraints on all consumers of the "charge" function, so we would expect such usages in describing message implementations rather than in describing message invocations. The problem of determining whether the invocations are supported by the available implementations is part of the work of the broker. It sees all of the definitions of requirements and of available support.

13. ***Class Card {***
    This class definition reopens the definition of the "Card" class to give a home to the implementations of getBalance and setBalance. I'm not sure the implementations really need to go with classes, but it provides a naming scope that may be useful later when doing subclass dispatching. But it also raises the whole question of object and reference creation and constructors.

14. ***ImplementationOf getBalance (card:CardRef):(balance:integer) {***
    look up card (as string) in vector of card/balance structures
    return balance = corresponding balance
    };
    This message implementation is for getBalance whose syntactic details are irrelevant. You shouldn't be put off by the notion of looking up the card as a string. Had "cardId" been represented by a pointer, the lookup would have been trivial. The conversion of string to pointer has to take place somewhere, and putting it here simplifies this initial presentation. issues surrounding conversion of references, caching conversions, etc. must all be addressed in implementing efficient dispatch structures, but are beyond the scope of this paper.

15. ***ImplementationOf setBalance (card:CardRef,balance:integer):() {***
    look up card (as string) in vector of card/balance structures
    set corresponding balance
    };
    This message implementation is for setBalance whose syntactic details are irrelevant.

    };

16. ***Class Item {***
    This class definition reopens the definition of the "Item" class to give a home to the implementations of "sell" and "charge". In view of their parameters, the "sell" message might have been implemented in "Store", and the "charge" in "Card", without any of their callers being affected. This is the advantage that outerfaces confer over interfaces. In LOLI, they are the places an ItemRef or StoreRef (CardRef) can be created. But putting them here simplifies the exposition and allows the discussion of some interesting issues to be isolated better when we define the "Store" class later.

17. ***ImplementationOf getPrice(item:ItemRef):(amount:integer) {***

return amount = $.05
};

This message implementation is for getPrice. GetPrice uses a price of $.05 for everything.. Maybe that will change someday.

18. ***ImplementationOf***
sell (item:ItemRef, store:StoreRef):() {
look up item and store (as strings) in array of availability structures
set inventory counter
};

This message implementation is for "sell". "sell" updates the inventory records for the "item", tracking how many are available at each store.

19. ***ImplementationOf charge (card:CardId, item:ItemRef):() {***
Integer balance = ***getBalance(card);***
Integer price = ***getPrice(item);***
Integer newBalance = balance – price;
setBalance(card, newbalance)*;*
};

This message implementation is for "charge" that subtracts the price of the "item" from the card balance. Each of the messages calls made by this implementation (getPrice, getBalance, and setBalance) can be legally sent because the message is in the outerface for at least one of its parameters.

};

20. ***Class Store {***

This class definition reopens the definition of the "Store" class to give a home to the implementation of registerSale.

21. ***ImplementationOf registerSale (card:CardId, store:StoreId, item:ItemId) {***
charge(card, item);
sell (item, store)
};

This message implementation is for registerSale that charges the item's cost to the card and updates the inventory.

};

SOME OBSERVATIONS

There are a number of subtle differences between outerfaces and interfaces that should be brought into focus:

- We speak of *an* outerface for a reference type rather than *the* outerface for the reference type because the reference type may key access to a variety of functions and services from different sources even for the same referenced entity. Subject-oriented programming [8, 10] extends the concept of object so that methods are not supported by a class, but by a subject's model of the class. Extending this further, we note that the functionality available through a general reference is embodied not in some object it refers to, but in the semantics of the reference itself.

- A message's appearance in an outerface doesn't imply servicing commitment by the reference type. In the common object-oriented model of interfaces, a reference to an object typed by an interface both makes calls

on the functions syntactically legal and "guarantees" that the object will implement the functions. An outerface simply indicates a set of functions that are supported by some component of the system when presented with a reference typed by the outerface.

- A message's appearance in an outerface qualifying a reference type does imply a passed service commitment. That is it asserts that the component passing the reference has received similar commitments that all of the messages appearing in the outerface will be serviced. This characteristic permits language-level checking for component self-completeness.
- Messages appear in many outerfaces. This is because the same message may be defined in many outerfaces for a reference type, each of which presents a different set of functions. In fact a message may even appear in outerfaces for the different reference types of its several parameters.

Registering Support and Checking Completeness

LOLI is intended to illustrate the concept of outerfaces as a way of providing the symmetric characterization of support provided by passing generalized references. As such, its language for registering support for implementations is actually vacuous. Although the local checking to verify that each call in an implementation is justified by the types of its parameters is present, there is no way of creating a reference such as CardId, or even CardRef, and providing the *ab-initio* check that support is available. In the absence of defining how references are coined, a discussion of the rules for dispatch is impossible. Although it is not our intent to explore these intermodular issues in this paper, since a wide variety of embeddings of the outerface concept are possible, we would envision an underlying dispatcher capable of supporting dispatch on classes, arranged in hierarchies and on predicates over values. An excellent treatment of the issues involved in dispatch can be found in [3] and of those concerning the required checking in [5]. These approaches can be directly applied here.
However, while leaving it to later work to explore reference coining and dispatching rules, it is incumbent on us to exhibit that some reasonable definition is possible. To that end we suggest that each class definition (<class-def>, of which there are 6 in the above example) could contain one or more construction verifiers, boolean functions that verify their argument to be a reference to that class capable of supporting the implementations within that class definition. Downcasting a value into a reference type invokes the construction verifiers associated with the class definitions for the referenced class to verify that the operations the reference is declared to support are actually supported. Note that when references are represented as non-computational tokens (as in most common OO languages), the "new-object" constructor would use such a downcast to produce its return value from the pointer it constructs.

Implementation

While it is generally the case that non-linguistic factors slow the movement of software developers to better languages, it is possible to piggyback the capabilities of outerfaces on top of existing programming languages. We plan to provide three elements to do so:

An Overbroad Language Embedding

For any programming language to be extended for outerfaces, it should be possible to define an overbroad language embedding. For example, in Java, method calls can be modeled as static methods. This imposes naming restrictions on the methods (e.g., they must have a "." in them), but enables programs to be written that use computational references and are not fragile with respect to component structure.

A Restrictive Preprocessor

The overbroad embedding does not ensure that the compile-time (static) type-checking of the assurances of support that are conveyed by attaching an outerface specification to a reference datum. In fact, in Java, there is no way to attach an outerface declaration to a general reference. So the task of accepting the extended language, checking it, and transforming it to an overbroad Java

A Flexible Dispatcher Generator

Since the preprocessor must translate to an overbroad base which suffers from an inadequate (asymmetric) dispatch model, the support for registering implementations must generate the code appropriate for dispatching a call to the appropriate implementation.

The MessageCentral Project

The MessageCentral[9] project at the IBM T.J.Watson Research Center is concerned with generating inter-component connections from message definitions from a variety of surfaces, both programming languages and messaging systems. MessageCentral supports an extended model of messages that treats call/return and event styles of program structuring in a unified paradigm for their interoperation that employs a generalization of outerfaces. In addition to producing message transformers, the MessageCentral tool can also produce adapters for the various surfaces and the dispatching code needed to support languages like LOLI.

Summary

Although Object-oriented programming provides many useful constructs for creating more flexible and more reusable software, the classic model's use of interfaces and single-dispatch are not among them. Their use causes the creation of software that is more fragile with respect to component structure than was the case with earlier technologies. The use of outerfaces and multiple-dispatch restores the balance and allows the creation of software that is much more robust when the structures of implementations changes.

APPENDIX: BNF for LOLI

Type Definition: <type-def> :=
  type <type-name> = <type-name> |
  <class-name> |
  referenceTo <type-name> [as <type-name>] [supporting <outerface-name>]
Message Definition: <message-def> :=
  <message-name> ([<element-name> : <type-name>]*)
Outerface Definition: <outerface-def> :=
  outerface <outerface-name>
  { <message-def> [; <message-def>]* }
Class Definition: <class-def> :=
  class <class-name> [[is <class-name>*] {
  <local-object-definition>*
  [<message-impl> [; <message-impl>]*]}]
Message Implementation: <message-impl> :=
  implementationOf
  <message-name> ([<element-name> : <type-name>]*){
   [<local-object-definition>[;<local-object-definition>* ;]
  <message-call> [; <message-call>]* }
Message Call: <message-call> :=
  <message-name> (<element-name>[;<element-name>]*)

REFERENCES

1. Daniel Bobrow, Linda DeMichiel, Richard Gabriel, Sonya Keene, Gregor Kiczales, David Moon, Common Lisp Object System Specification, *SIGPLAN Notices* Vol 23 September 1988, pp.1-11
2. Chambers, Craig, Object-Oriented Multi-Methods in Cecil, *Proceedings of the 1992 European Conference on Object-Oriented Programming, Lecture Notes in Computer Science #615*, Springer Verlag, Berlin, 1992
3. Chambers, Craig, Chen, Weimin, Efficient Multiple and Predicate Dispatching, *Proceedings of the 1999 Conference on Object-Oriented Programming, Systems, Languages and Applications*, Denver, November, 1999
4. Dicky H., Dony, C., Huchard, M., Libourel T., On Automatic Class Insertion with Overloading, *Proceedings of the 1996 Conference on Object-Oriented Programming, Systems, Languages and Applications*, San Jose, November, 1996
5. Harrison, William, Ossher, Harold, Subject-Oriented Programming (a critique of pure objects), *Proceedings of the 1993 Conference on Object-Oriented Programming, Systems, Languages and Applications*, Washington, D.C., October, 1993
6. Litvinov, Vassily, Constraint-Based Polymorphism in Cecil: Towards a Practical and Static Type System, *Proceedings of the 1998 Conference on Object-Oriented Programming, Systems, Languages and Applications*, Vancouver, October, 1998
7. Moore, Ivan, Automatic Inheritance Hierarchy Restructuring and Method Refactoring, *Proceedings of the 1996 Conference on Object-Oriented Programming, Systems, Languages and Applications*, San Jose, November, 1996
8. Ossher, H., Tarr, P., Harrison, W., Sutton, S., N Degrees of Separation: Multi-Dimension Separation of Concerns, *Proceedings of 1999 International Conference on Software Engineering*, May 1999
9. –, MessageCentral Home Page, IBM T.J. Watson Research Center, <http://www.research.ibm.com/messagecentral/>
10. –, Subject-Oriented Programming Home Page, IBM T.J. Watson Research Center <http://www.research.ibm.com/sop/>
11. –, Implementing e-Business with NEON, Neon, Inc., <http://www.neonsoft.com/products/impe-biz.html>
12. –, Case Studies, Telino, S.A., <http://www.telino.com/english/index2.htm>
13. –, Case Studies, TSI Enterprise, <http://www.tsisoft.com/enterprise/success.html>
14. –,White Paper on Message Brokers, VIE Systems, <http://www.viesystems.com/support/index.html>
15. –, "The Network is the Computer: Applications Integration, Message Broker, and Infrastructure, Strategic Positioning and Market Forecasts to 2001", Wintergreen Research Corp., Report #A8021197273, November, 1997, <http://www.wintergreenresearch.com/>