

Object-Relational Mapping The Fake

Speak of Relational Model In Your Favorite OO Languages

Compl Yue Still
Ableverse Platform

Note: This is a draft paper for reviews, last updated 2007/04/07

Copyright © 2007 Ableverse Platform.

All rights reserved. No part of this paper may be reproduced, stored in a retrieval system or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior written permission of the copyright holder.

Abstract:

What is today's Object-Relational Mapping REALLY doing? The answer will be given by this paper is: Mapping the Network Model and SQL interface. In this paper, we shall see why and how this is the truth. And your favorite object oriented programming languages, with sufficient modern features, such as Java™, are quite possible to be effectively consolidated with the power of relational model, without defective mappings.

Ableverse™ TheObjectBase as the DBMS and WebOfWeb as the blueprint project are showing a feasible solution in the Java™ programming language as realworld practices, on the basis of a new relational data model called the Object-Relation-Kin model.

Table of Contents:

- [1. The Object-Relational Impedance Mismatch](#)
 - [2. The Object-Relational Mapping](#)
 - [2.1. Problems Solved](#)
 - [2.1.1. Perform CRUD on Objects](#)
 - [2.1.2. Query Objects Expressively and Accelerated](#)
 - [2.1.3. Reduce Data Traffic by Cache](#)
 - [2.2. New Problems Created](#)
 - [2.2.1. Multi-Source of Schema/MetaData](#)
 - [2.2.2. Transaction Serializability](#)
 - [2.2.3. Confusing Property Writers](#)
 - [2.2.4. Discomfort for Purists](#)
 - [2.3. Problems Not \(Effectively\) Solved](#)
 - [2.3.1. Relationship Manipulation](#)
 - [2.3.2. Overall Complexity/Productivity](#)
- [3. Thank Again, What Is What](#)

[3.1. The Network Model Called Object Model](#)

[3.2. True Nature of the Relational Model](#)

[3.3. How ORM's Usage of SQL Harms Consistency/Isolation](#)

[4. Make a Match Instead of Defective Mapping](#)

[4.1. A Simple Job Hard To Do](#)

[4.2. What versus How](#)

[4.3. Implement Concurrency/Transaction Control Based On Objects](#)

[4.4. Server Side Object Graph and Hosting Based Interfacing](#)

[4.5. SQL Does Query](#)

[5. Seeking Fresh Meat](#)

1. The Object-Relational Impedance Mismatch

It is clear that object technologies and relational technologies are in common use in most organizations, that both are here to stay for quite awhile, and that both are being used together to build complex software-based systems. It is also clear that the fit between the two technologies isn't perfect, that there is an "impedance mismatch" between the two.

...

In the early 1990s the differences between the two approaches was labeled the "object-relational impedance mismatch", or simply "impedance mismatch" for short, labels that are still in common use today.

- Scott W. Ambler, The Object-Relational Impedance Mismatch

This problem is pretty well described in [[3, Ambler, ORIM](#)], together with some practical strategies worked out so far.

2. The Object-Relational Mapping

There is a good description of the general process of mapping objects to relational databases in [[4, Ambler, ORM](#)]. And there are also a number of mature frameworks/products promisingly doing the tricks, such as the popular [[5, Hibernate ORM](#)] framework. ORM is also a formal part of [[6, Java Persistence API](#)], which defines a mandatory requirement for Enterprise JavaBeans 3.0 implementations.

In this paper, we shall prefer [[6, Java Persistence API](#)] terms for O-R Mapping concepts since JPA is actually a unification of mainstream ORM paradigms.

2.1. Problems Solved

2.1.1. Perform CRUD on Objects

Thanks to ORM, we can now Create/Read/Update/Delete data records as objects.

2.1.2. Query Objects Expressively and Accelerated

We can now use SQL alike query languages to query objects by criteria. At the same time, queries tend to be accelerated by indices, and optimized by database engines.

2.1.3. Reduce Data Traffic by Cache

ORM mechanisms normally have built-in cache supports. When cache is enabled, no data needs to be loaded for an object once an up-to-date copy is already in cache.

2.2. New Problems Created

But ORM creates new problems while solving others.

2.2.1. Multi-Source of Schema/MetaData

Now both the underlying relational database and the code of application classes contains schema. The worse is mapping configurations tell exactly the same metadata. Much care is needed to keep these 3 sources accord at all.

2.2.2. Transaction Serializability

ORM mechanisms normally shift ACID (Atomic, Consistent, Isolated, Durable) requirements off onto the underlying relational database, as well as **Read Committed** isolation level being strongly recommended. Higher isolation levels of transactions like **Serializable** are barely supported if not totally denied.

2.2.3. Confusing Property Writers

An ORM layer typically suggests application classes export property writers for it to push data into persistent objects, even for collection fields those hold references to related entity objects. This has almost become a standard paradigm.

ORM will advocate that application classes are doing this just to follow the [[8. JavaBeans](#)] specification, or just to encapsulate their fields as standard C# properties. But a simple examination on the JavaBeans specification will find it was designed really for components with unpredictable boundary interfaces those can be assembled dynamically, especially for GUI components to be used by visual UI designers. Persistent classes by applications barely fall into this category, they are statically linked by other portions of the applications for most cases.

These property writers, are not necessarily used from code other than the ORM layer, especially for the relationship collections. But application programmers will always see such public methods prompted when they type a persistent object variable in their IDEs, they have to filter the list once again using their brains, for they should really NOT invoke some of these from their code.

Another negative influence of this paradigm is it encourages the Anemic Domain Model ([[9. Fowler, ADM](#)]) to some degree.

2.2.4. Discomfort for Purists

Query By Language is a necessary feature for ORM mechanisms to support. However, the mixture of SQL alike syntax with original object oriented syntax for applications ever uncomfortable certain people. Some feel their Java (or the like) code are polluted by SQL (or the like), others feel their DML languages are polluted by foreign syntax.

2.3. Problems Not (Effectively) Solved

2.3.1. Relationship Manipulation

Note that it is the application that bears responsibility for maintaining the consistency of runtime relationships—for example, for insuring that the “one” and the “many” sides of a bidirectional relationship are consistent with one another when the application updates the relationship at runtime.

- JSR 220: Enterprise JavaBeans™, Version 3.0 - Java Persistence API

Obviously applications are burdened with unreasonable responsibilities.

2.3.2. Overall Complexity/Productivity

While some people found ORM helpful for simplicity and productivity, others doubt it at all.

See [[11](#), [PolePosition DB Benchmark](#)] .

3. Thank Again, What Is What

3.1. The Network Model Called Object Model

Many people believe their POJOs (or POCOs - Plain Old Java Objects or Plain Old CLR Objects) are so structured to form an *Object Model* , and it is the only possible form of data an object oriented programming language can represent. But this is not true, the *Object Model* are in fact data structures in the **Network Model** , just with behaviors added doesn't change this substance. Some people already figured it out.

It still holds the same nature from **structs** in **C** programming language. As long as a struct (or an object) maintains pointers (or references) to its relatives, a tree or web structure is resulted from such individual nodes. The **Hierarchy Model** describes such a tree, while the **Network Model** describes such a web.

3.2. True Nature of the Relational Model

The relational model equals to **tables + SQL** in many ones' mind, but it is actually not. **The Third Manifesto** ([[10](#), [C. J. Date and Hugh Darwen, TTM](#)]) seeks for clarification of such misunderstandings, and doubts existing *relational databases* about their claims.

For an easy to understand example, in Java code:

```
public class Customer
{
    Collection<Address> addresses;
    ...
}

public class Address
{
    Collection<Customer> customers;
    ...
}
```

Above forms a network model. While below qualifies for a relational model:

```

public class Customer
{
    ...
}

public class Address
{
    ...
}

public class CustomerLiveInAddress
{
    Customer customer;
    Address address;
    ...
}

```

*Note here we made an extension to the relational theory to regard every object can have its individual **ID** identifier as an implicit attribute, and the **ID** attribute should be allowed to act as the **primary key**. With this extension, a class field referencing another object can thus be considered a **foreign key**.*

But it is obvious the later is insufficient for the sake of application development. Because you have no way to get a customer's addresses even you've got a **Customer** instance. And creating a **CustomerLiveInAddress** seems meaningless.

So ORM kindly allows application code written like this:

```

@Entity
public class Customer
{
    private Collection<Address> addresses;

    @ManyToMany
    public Collection<Address> getAddresses()
    {
        return this.addresses;
    }
    public void setAddresses(Collection<Address> addresses)
    {
        this.addresses = addresses;
    }
    ...
}

@Entity
public class Address
{
    private Collection<Customer> customers;

    @ManyToMany(mappedBy="addresses")
    public Collection<Customer> getCustomers()
    {
        return this.customers;
    }
    public void setCustomers(Collection<Customer> customers)
    {
        this.customers = customers;
    }
    ...
}

```

Where the ORM layer will maintain a **CustomerLiveInAddress** equivalent table under the ground, leaving the application defines its class schema in the comfort *object model*, although you have to bear the responsibility for

maintaining the peered reference collections like:

```
customer.getAddresses().add(address);
address.getCustomers().add(customer);
```

Okay, but what will happen if comes we need to allow a customer to cite a condition for his preference of an address?

For a simple solution with the network model, without structure change, you may decide to add a **prefCondition** attribute to the **Address** class. But it effectively forbids the sharing of addresses between customers, because of potential conflicts between different customers reside at a same address. If you choose to accept this side effect, and write extra code to prevent addresses from being shared, the model silently changes in semantics: It effectively becomes a OneToMany relationship instead of the original ManyToMany one, regardless of whether you modify the structure or not.

Okay, it is not your only choice, the network model CAN just handle such requirements correctly. Chen also agreed with this, he had even given out a way to derive the network model from the entity-relationship model in his famous paper [[2, Chen, 1976](#)]. With the intermediate process skipped, a new class schema regarding the **prefCondition** as a relationship attribute will look like this:

```
@Entity
public class Customer
{
    private Collection<CustomerPreferAddress> addresses;

    @OneToMany(mappedBy="customer")
    public Collection<CustomerPreferAddress> getPreferredAddresses()
    {
        return this.addresses;
    }
    public void
setPreferredAddresses(Collection<CustomerPreferAddress> addresses)
    {
        this.addresses = addresses;
    }
    ...
}

@Entity
public class Address
{
    private Collection<CustomerPreferAddress> customers;

    @OneToMany(mappedBy="address")
    public Collection<CustomerPreferAddress> getPreferringCustomers()
    {
        return this.customers;
    }
    public void
setPreferringCustomers(Collection<CustomerPreferAddress> customers)
    {
        this.customers = customers;
    }
    ...
}

@Entity
public class CustomerPreferAddress
{
    private Customer customer;

    @ManyToOne
    public Customer getCustomer()
```

```

    {
        return this.customer;
    }
    public void setCustomer(Customer customer)
    {
        this.customer = customer;
    }

    private Address address;

    @ManyToOne
    public Address getAddress()
    {
        return this.address;
    }
    public void setAddress(Address address)
    {
        this.address = address;
    }

    String prefCondition;
    ...
}

```

Let's see how you do create a relationship between a customer and an address now:

```

CustomerPreferAddress pref = new CustomerPreferAddress(customer,
address);
pref.setPreferenceCondition(prefCondition);
entityManager.persist(pref);
customer.getPreferredAddresses().add(pref);
address.getPreferringCustomers().add(pref);

```

Moreover, you are aware or not - there will be 2 extra underground tables created by ORM, one associates **Customer** with **CustomerPreferAddress** , and another associates **Address** with **CustomerPreferAddress** .

And you see similar structures? Right, **CustomerPreferAddress** looks just like **CustomerLiveInAddress** in our previous *relational* class schema. However if with the relational schema, the requirement can be simply fulfilled by adding the **prefCondition** attribute to class **CustomerLiveInAddress** , without any structure or semantic changes. And in fact the ORM schema simulated the relational schema at last, but at the cost of extra tables than necessary. It may be finally correct in semantics, but far behind optimal, both for storage efficiency and straightness in relationship manipulations (which translate to runtime performance and development productivity).

But! The *relational schema* there is just useless at all, how does it make any sense when does NOT even work? Don't worry, this paper (and sophisticated details in [[1, Compl, ORKM](#)]) is just to make it work in simple ways.

To this point, you just need to make it clear in mind, that when relationships are represented by individual objects instead of (possibly peered) references, the schema is in the **relational model** instead of the **network model** . So in a nutshell,

```

public class Worker
{
    Collection<Task> assignedTasks;
}

public class Task
{
}

```

is network model while

```

public class Worker
{
}

public class Task
{
    Worker assignee;
}

```

is relational model. Being able to distinguish the 2 schemata is essential to understand how a relational schema can be constructed with objects.

3.3. How ORM's Usage of SQL Harms Consistency/Isolation

Almost without exception, ORM mechanisms generate SQL commands to interact with the *relational* side of its company. There can be just too many reasons for SQL's domination, but we'd better not complain about accomplished fact but to face it now.

An ORM layer normally runs at the *client* side of a SQL database server, and doesn't deprecate concurrent access from other ORM instances or database clients. Unfortunately concurrency/transaction control by the SQL DBMS is only responsible for reads and writes of SQL commands. While it's unaffordable regarding performance for an ORM layer to map every object reads/writes to SQL reads/writes, the ORM layer just doesn't do so. The result makes the DBMS no longer in acknowledgment and control of what the ORM-wrapped application is really doing, thus is put down from correct judgments and reactions according to consistency/isolation requirements.

While ORM providers advocate they provide cache *for free* , it will actually cost your right to operate on canonical data in certified ways.

4. Make a Match Instead of Defective Mapping

You see the so called *Object Model* is actually **Network Model** , however it has good reasons for such confusion. There are real difficulties for an in-memory object to get a related object when the relationship is represented by a separated object (aka relation object) without an access path from the object in question. Quite the same situation as for in-memory **structs** in the C programming language, in-memory objects are still data structures in substance.

4.1. A Simple Job Hard To Do

Recall the previous relational class schema:

*And please keep in mind that we have made an extension to the relational theory that each object has an implicit **ID** attribute associated with it, and **ID** is allowed to act as the **primary key** . So a class field referencing another object is considered a **foreign key** attribute holding the referenced object's **ID** value.*

```

public class Customer
{
    ...
}

public class Address
{
    ...
}

public class CustomerLiveInAddress
{
    Customer customer;
}

```

```

    Address address;
    ...
}

```

How do we list all of a customer object's addresses and vice versa? We know perfectly how SQL does this, just like:

```

SELECT a.* FROM Address a
LEFT JOIN CustomerLiveInAddress l ON l.address = a.ID
WHERE l.customer = ${customer.ID}

```

Obviously we don't mean to simulate such an approach by enumerating all **CustomerLiveInAddress** objects and **Address** objects then returning all matched ones as a reference collection. We just want a directly available set of references to eligible **Address** objects. However, if we store such reference sets by our classes, the schema falls back to the network model. Then how do we?

4.2. What versus How

There is a subtle difference between the ways we think of a solution in Java and SQL.

We think **How** in Java:

Define a collection for a **customer**, to contain all his **address** objects. Add a new **address** object to this collection when he gets it. And when all his addresses are needed, just enumerate over the collection.

We think **What** in SQL:

Get all **addresses** those associated with the **customer** by the **CustomerLiveInAddress** relation.

This is no surprise since Java is a [[12](#), [4GL](#)] while SQL is a [[13](#), [5GL](#)]. Different thinking patterns had been purposed by designs of the languages.

However, a little improvement could be made after we think of **declarative programming** in Java (or the like). **Annotation** has been introduced and **Annotation Processor** can be used to generate code according to application classes. With such necessary infrastructures, we will be able to add some **what** semantics to Java classes where **how** is mainly concerned.

Traditionally, a class is coded by either an application module or a system module, doing totally different things at different layers. Even the application module runs atop the system module in intensive cooperations, they won't be jointly implementing a single class. But with the new infrastructures and ideas, we will let the DBMS implement some specialized functionalities for application defined classes those create persistent objects. Such as to permanently store an object's states upon transaction commits. It is possible (and already done in [[14](#), [TheObjectBase](#)] by means of [[1](#), [Compl. ORKM](#)]).

And finally we will get a class schema like:

```

public class Customer
{
    Iterable<Kin<CustomerLiveInAddress, Address>> addresses;

    ...
}

public class Address
{
    Iterable<Kin<CustomerLiveInAddress, Customer>> customers;

    ...
}

```

```

public class CustomerLiveInAddress extends TheRelation
{
    Tie<Customer> customer;
    Tie<Address> address;
    public CustomerLiveInAddress(Customer customer, Address address)
    {
        this.customer = new Tie<Customer>(customer);
        this.address = new Tie<Address>(address);
    }

    ...
}

```

There are some new elements introduced, they are defined by the DBMS like:

```

public interface Kin<R, T>
{
    R getSourceRelationObject();

    T getRelatedObject();
}

public class TheRelation
{
    // Tie is an inner class which only allows
    // instantiation from subclasses of TheRelation
    public class Tie<T>
    {
        final public targetObject;

        protected Tie(T targetObject)
        {
            assert targetObject != null;
            this.targetObject = targetObject;
        }
    }

    ...
}

```

These new elements are introduced to aid application classes in expressing their relational semantics.

- The **Kin** interface implies that the relationship to a **related object** is established through a **source relation object**, and they are always in pairs.
- Relation classes are supposed to derive from **TheRelation** base class, and
- Every participant objects in a relationship are supposed to be *tied* by a **Tie** reference as a field of the relation class.
- **TheRelation.Tie** roughly maps to **foreign key**.

But isn't it much the same as the previous **ORM** class schema in the **network model** ?

Okay, if you haven't figured it out, please accept that the following 2 fields:

```

Iterable<Kin<CustomerLiveInAddress, Address>> addresses;

```

at class **Customer**, and

```

Iterable<Kin<CustomerLiveInAddress, Customer>> customers;

```

at class **Address**, will be transparently maintained by the DBMS instead of class **Customer** and **Address** themselves. These application defined classes just define such fields, then use them directly when needed.

And more differences in how a new relationship is created, the new code looks like:

```
tob.birth(  
    new CustomerLiveInAddress(customer, address));
```

Compared to the relationship creation code for the **ORM** schema we've shown previously:

```
customer.getAddresses().add(address);  
address.getCustomers().add(customer);
```

The new algorithm got 2 significant traits:

1. Only operate on relation object: maintain a relationship only by manipulating the relation object representing it.
2. No consistency responsibility: peered references - from the **customer** to the **address** and vice versa - are charged by the DBMS.

If you feel it is still less than enough to be distinguished, we consider adding a **Preference Condition** attribute as described previously.

As you should still remember, in the **network model**, it needs 2 extra underground tables - totally 5 tables - to fulfill this requirement without semantic lose. Now let's see how does the **relational model** do.

```
public class CustomerLiveInAddress extends TheRelation  
{  
    private String prefCondition;  
    // getter and setter method for prefCondition omitted  
  
    Tie<Customer> customer;  
    Tie<Address> address;  
    public CustomerLiveInAddress(Customer customer, Address address)  
    {  
        this.customer = new Tie<Customer>(customer);  
        this.address = new Tie<Address>(address);  
    }  
  
    ...  
}
```

The addition of field **prefCondition** (and its getter+setter) to class **CustomerLiveInAddress** just solves. 3 tables for this 3 classes schema just fit.

As you may wondering how to list a customer's addresses matching a specific condition, here is the code:

```
public class Customer  
{  
    Iterable<Kin<CustomerLiveInAddress, Address>> addresses;  
  
    public List<Address> getMatchAddresses(String cond)  
    {  
        List<Address> list =  
            new ArrayList<Address>(this.addresses.size());  
        for(Kin<CustomerLiveInAddress, Address> akin : this.addresses)  
        {  
            if(cond.equals(  
                akin.getSourceRelationObject().getPrefCondition()))  
                list.add(akin.getRelatedObject());  
        }  
        return list;  
    }  
}
```

```

    ...
}

```

In real world, the **Iterable** fields will have a **KinSet** class type, so that sorting order of the kins could be specified with a **@UseKinComparator** annotation. Other utility methods shown below are also useful for realworld applications.

```

public final class KinSet<R, T> implements Iterable<Kin<R, T>>
{
    ...

    public Iterable<T> getDistinctRelatedObjects() {...}

    public boolean containsRelatedObject(TheObject p) {...}
}

```

Last to mention that in realworld programming, more modifiers like **private** or **protected** should really be added to some fields of the sample Java classes there.

4.3. Implement Concurrency/Transaction Control Based On Objects

Object based concurrency/transaction control is necessary, to correctly support the full range of transaction isolation levels, up to **SERIALIZABLE** .

Declarative Programming is useful again in automating transaction management. So object states reader methods can just be annotated with a **@Reading** annotation (or the like) to indicate its intent, then runtime invocations to such methods will be traced by the DBMS to perform correct pre and post operations regarding desired consistency requirements by configuration. Writer methods alike. And methods tend to change a relation object's **Tie** fields are considered to change the participants in a relationship, they should also be under transaction control.

4.4. Server Side Object Graph and Hosting Based Interfacing

Obviously so constructed in-memory graph of persistent objects is hard to synchronize between a database server and its clients.

In contrast to **ORM** mechanisms those designed for client side db access, an **Object Oriented Relational Data Model** best suits server side constructs or applications with exclusive write access to the permanent storages.

So how should access be provided to multiple client applications from distributed nodes? The answer is **Hosting Based Interfacing** .

HBI is yet another new discovery for interoperability between software components. Its application here can be shown as the analogy to a SQL server's client/server architecture and related instruments:

	Relational Object Server	SQL Server
Host Server Environment	Persistent Class	Table Definition
	Persistent Object Behavior	Stored Procedure
	Persistent Object Notification	Trigger
	Persistent Object Graph	Cached Table Data

	Relational Object Server	SQL Server
	Tabular Permanent Storage	
Client Access	Task Agent Object	SQL Command
	Task Agent Behavior	SQL Code
	Task Agent Result	SQL Result

So an HBI enabled system will allow **Task Agent** objects sent from clients to the server and get executed there, the execution result will be sent back to the originating client. The key difference from **Invocation Based Interfacing** (which is on the contrary of HBI) is: The code describing business logics that written by clients (task agent behaviors similar to SQL commands) assumes server side environment upon its execution, thus round trips of invocations are reduced and server side resources get more efficiently leveraged.

SQL commands are easier to be sent for remote execution, as just textual scripts. However, with dynamic deployment supports (and even code security mechanisms) of Java and similar OO programming languages, the **HBI** architecture is no hard to be implemented.

4.5. SQL Does Query

As its name implies, SQL's nature is query. So let it do what it's best at.

The easiest way to implement SQL supports is by delegating queries to an existing RDBMS against its table data, while updates of runtime object states all commit to tables of the RDBMS. But in this way, unless SQL's execution is against in-memory object data as well as committed table data, the isolation level of SQL queries will not go up over **READ_COMMITTED**. Applications get to tolerate it or seek better integrated SQL implementations over object data.

5. Seeking Fresh Meat

So far we just went over the conceptual process to speak the **relational model** in [[7, Java Programming Language](#)] (should apply to other similar OO languages). If interested in further details, you find:

- A sophisticated **Object Oriented Relational Data Model** described in detail by [[1, Compl, ORKM](#)]
- A DBMS implementation as [[14, TheObjectBase](#)]
- An open source blueprint project as [[15, WoW Project Home](#)] .

Acknowledgments:

References and Suggested Readings:

- 1 . Compl Yue Still, The Object-Relation-Kin Model: Toward Relational Analysis and Design in General Object Oriented Languages
<http://www.ableverse.org/articles/orkm.pdf>
- 2 . P. Chen. The entity-relationship model: Toward a unified view of data. ACM TODS 1 (1976)
<http://doi.acm.org/10.1145/320434.320440>
- 3 . Scott W. Ambler, The Object-Relational Impedance Mismatch
<http://www.agiledata.org/essays/impedanceMismatch.html>

- 4 . Scott W. Ambler, Mapping Objects to Relational Databases: O/R Mapping In Detail
<http://www.agiledata.org/essays/mappingObjects.html>
- 5 . The Hibernate ORM Framework for Java and .NET
<http://www.hibernate.org>
- 6 . Java Persistence API
<http://java.sun.com/javase/technologies/persistence.jsp>
- 7 . SUN Microsystems, Java Programming Language
<http://java.sun.com>
- 8 . JavaBeans
<http://java.sun.com/products/javabeans>
- 9 . Martin Fowler, Anemic Domain Model
<http://www.martinfowler.com/bliki/AnemicDomainModel.html>
- 10 . C. J. Date and Hugh Darwen, The Third Manifesto
<http://thethirdmanifesto.com/>
- 11 . The PolePosition Database Benchmark
<http://www.polepos.org>
- 12 . Fourth-Generation (programming) Language
http://en.wikipedia.org/wiki/Fourth-generation_programming_language
- 13 . Fifth-Generation (programming) Language
http://en.wikipedia.org/wiki/Fifth-generation_programming_language
- 14 . Ableverse TheObjectBase
<http://tob.ableverse.org>
- 15 . The WebOfWeb Project Site
<http://wow.dev.java.net>
- 16 . The Official WoW Site
<http://www.webofweb.net>