

Reusable Security for Segmented Data Domains

John C. Dale

November 18, 2005

MS MIS December 2005

The Eller College of Management

The University of Arizona

Tucson, Arizona

Abstract

For firms providing software development outsourcing services, the practice of *software reuse* can reduce overhead and increase margins. At the current time, an alarming number of enterprise software development projects are over budget, delivered late, or both. As software development organizations mature, so too should their ability to deliver increasingly complex software solutions on time and on budget. One strategy for achieving this is to identify opportunities for software reuse. In traditional manufacturing nomenclature, this process would be expressed as 'manufacturing efficiency' or 'economy of scale'. This paper explores one way in which open source Java 2 Enterprise Edition Security Realms can be used to facilitate code reuse – and thus manufacturing efficiency – into the enterprise software manufacturing process. Subsequently, enterprise software development firms who employ this methodology should expect to deliver software with greater efficiency and predictability at a lower cost.

The Importance of Software Reuse and Cohesive Domains

Cohesion is "a measure of how strongly related and focused the responsibilities of a class are" [relative to the package and name of the class that is used to identify the class] [3].

Computers do what we tell them, but how do we describe what they do? The answer to that question depends upon to which service layer one is referring when asking the question. Services layers can be thought of as 'layers of indirection' or 'packages'.

For example, consider the case where a web browser on your computer requests a web page. When the user clicks a button, the user's action is translated into a series of software procedure invocations, each one of which represents a layer of indirection that accomplishes a specific task computing. The tasks are defined at various layers of abstraction. Some examples from a high level of abstraction might be "handle event" and "resolve internet protocol address". Some examples from a lower level of abstraction are "acknowledge receipt of packet" and "write byte to hard disk".

At certain points in the chain of indirect routine invocations, there are logical segmentation points, usually occurring where like functionality begins and ends. The grouping of like functionality is called 'packaging'. Packages are developed to facilitate reuse at various points on the chain of indirect routine invocations for any number of potential system inputs.

The proper distillation of software components within a given problem domain is accomplished through a process of normalization. The output of effective normalization is a generally agreed upon contract or standard of operation. Additionally, there is a polymorphic relationship between the contract and implementation.

Consider the case of network interface hardware. Each vendor provides an interface through which the operating system passes commands from user applications. The drivers adhere to the TCP/IP [4] standard for sending and receiving packets over the network, but the code across network interface cards is drastically different. The operating system interface is a normalized contract between operating system vendors and network interface cards (see [2] for more information).

To reiterate, the grouping of like functionality in this way is referred to as cohesion, and is central to software development efficiencies that resulted in the pervasiveness of the desktop computer. Cohesive system functionality is grouped together in packages, and made available for reuse.

To realize the full impact that library and package development have had in the computing world, imagine the impact if every software program written was

required to have its own operating system embedded within it. The operating system controls access to hardware devices, and implements security. The following table illustrates the size of the Operating System in terms of 'lines of code' [1]. Reproduction of a windows-like operating system for every software program on the market would be prohibitive, to say the least.

Year	Operating System	SLOC (Million)
1990	Windows 3.1	3
1995	Windows NT	4
1997	Windows 95	15
1998	Windows NT 4.0	16
1999	Windows 98	18
2000	Windows NT 5.0	20
2001	Windows 2000	35
2002	Windows XP	40

Outsourcing, Scalability, and National Competitive Advantage

A crucial factor in achieving competitive advantage rests with an organization's ability to scale to meet the demand of its customers.

Scalability can be achieved through the specialization of competency from an inter/intra organizational perspective.

Additionally, it comes from the optimization of internal processes such that more units can be produced and sold at a profit while reducing the sales price.

Intra-organization Specialization

Within a corporation, there are several departments with specialized expertise in given areas - finance, operations, technology, customer support, continuous quality improvement, compliance, accounting, and so on. Each of these departments is able to focus on their narrowed area of responsibility. As a result, they are more likely to discover opportunities to optimize efficiency. Narrow focus reduces the number of variables involved in an efficiency calculus, reducing the potential for error, and reducing the size of the domain in which a specialized labor force searches for efficiency gains. They operate under the motto 'do one thing and do it very well'.

In essence, the corporation has outsourced well defined, highly specialized responsibilities to internal organizations that are created along the lines a superimposed boundaries, based on a cohesive segmentation of responsibility. This delineation of departments within a corporation allows the entity to accommodate growth, more so than if there were no clear assignment of responsibility within the organization.

The structure is such that specialized departments, if run efficiently, provide economies of scale and therefore competitive advantage for the corporation. This is a microcosm of what occurs on a more grand scale in industries.

Inter-organization Specialization

The presence of specialized services firms within an industry segment increase the likelihood that that industry will be more competitive in a global environment [6]. When players in a given industry attempt to take their products and/or services to a global market, the scale of demand can increase exponentially. Therefore, specialization of tasks within an industry can increase the ability of an industry segment to more efficiently meet global demand. Some examples are Information Technology services, Financial Services, and Business Process Optimization Services. The availability of these supporting entities - assuming that because of specialization they are able to offer their services at a lower rate than would be required for firms to bring these operations in house - free up capital to increase product performance, offer more customer service, and allocate more funds to shareholders. These benefits make the firms more competitive in a global marketplace. Inasmuch as being more competitive is a virtuous goal, the availability of specialized, scalable outsourcing firms is also virtuous.

Dot Com, Inc. (DCI) - A Case Study

DCI is a company that develops Internet-based software. DCI is interested in writing custom software for niche markets. The company specializes in understanding a wide variety of business processes, and in producing software that models those software processes quickly. DCI provides services that cross-cut industries, but nonetheless provide significant value to their customers in a way that is not cost prohibitive.

In line with Porter's model for national competitive advantage [6], DCI provides software development services for its clients that result in greater specialization and scalability since they (DCI clients) do not have to invest in the development of internal software development capacities.

So, the DCI team set out to design an application deployment model that would utilize their available hardware to its fullest possible capacity, and reduce development costs. These two things DCI overhead, making it possible to pass

some of the savings on to their clients. As a result, DCI, its clients, and the industries in which its clients compete are all more competitive.

Special Consideration about the DCI Business Model

DCI is not interested in relinquishing ownership of the software that they manufacture for clients. On the contrary, the company was willing to offer the software to their clients on a contractual basis, allowing DCI to resell the software to other firms, potentially in the same industries as their clients. The price that is negotiated with their clients is dependent upon the software's market potential. Higher potential results in a lower rate for the client. Lower potential results in a higher rate for the client.

This business model inherently adds some constraints upon the type of software that DCI manufactures. First, the software must be relatively generic, and must codify aspects of a client's business that are transferable to its competition. Next, because the development processes are specialized to facilitate economies of scale for DCI, the technology used to develop the software, as well as the general architecture must be relatively static.

There are some risks inherent in this business model. DCI has considered them and taken measures - both technological and policy in nature - to mitigate these risks. An in depth analysis of the risks and contingencies in the DCI business model, however, is beyond the scope of this paper.

Reusing a Security Library

The DCI development staff was experienced in writing Internet applications. The developers realized the value of using libraries to accomplish repetitive tasks. In addition, many members of the team had experience writing custom security frameworks for Internet applications. There were many commonalities that cross cut their respective implementations, and the team saw a great advantage to extracting these commonalities into a reusable library. If they didn't have to write the security for their applications over and over, they would save time and money, and would reduce errors associated with the development, testing, and deployment of security software, a critical aspect of Internet applications that provided great value to their customers.

Supporting Tools for the Job

There is an n-level hierarchy of services that make up the software development value chain. DCI uses many tools available from standards based open source projects [8]. The adherence to standards like Java 2 Enterprise Edition (J2EE) [9] lowered maintenance and switching costs for DCI, once again contributing to its competitive advantage in its industry.

The team settled on Java and Tomcat [7]. Tomcat is an open-source, proven operating environment for Java-based Internet applications, and could be

extended if needed. The detailed analysis for this selection is beyond the scope of this paper. The criterion used to make the decision were 1) level of platform independence, 2) support for object oriented development, 3) performance, and 4) quality of available development tools, and 5) total cost of ownership.

Moreover, Tomcat supports the deployment of multiple contexts for the same application. Each application is given its own class loader [10], and can be configured to access its own data source connection pool [11]. This meant that the DCI development group could partially meet the requirement of deploying many instances of their software onto the same machine, allowing them to fully utilize their hardware.

Tomcat allowed the team to easily segment application code deployments, but did not have functionality for segmenting customer data in a way that was secure, efficient at runtime, easy to maintain operationally, and not cost effective build and maintain.

The Virtual Private Database (VPD)

One alternative available to the development team for segmenting customer data was the Virtual Private Database (VPD) [5]. VPD allows for the logical segmenting of data across different domains (a domain is defined as a single customer's inventory data), while physically storing all data in the same database.

Although VPD doesn't require that the exact same code be written over and over, it does require that the same type of code be written over and over. It is perhaps best to understand VPD by example.

A Sample Database

Each customer of DCI software would have the ability to maintain customer information. Below is the database definition that DCI started with.

```
create table Customer
(
  customerId char(36),
  firstName varchar(75),
  lastName varchar(75)
);

create table
CustomerAddress
(
  customerAddressId char
(36),
  street varchar(100),
  city varchar(100),
  state char(2),
  zip varchar(10)
  customerId char(36)
);
```

Sample Database Query

The user's information is selected by joining the Customer and CustomerAddress tables on the customerId.

```
select
c.firstName, c.lastName,
ca.street, ca.city, ca.state,
ca.zip
from
Customer c, CustomerAddress
ca
where
c.customerId = ?
and
c.customerId = ca.customerId
```

Applying VPD to the DCI Domain

For DCI, VPD is a way to segment the data that is in the database by organization. The organizations that will be segmented are their e-commerce customers, one organization for each. With VPD, each customer dataset would exist in the same database schema, side by side. The data would then be delimited by adding an extra column to each database table, then including this column in every query to the database to make sure that the correct data was being read and updated.

Sample database after VPD

Note that there was an additional field added to each table (highlighted in **bold**). This field would be added to every table that contains data that would require segmentation.

```
create table Customer
(
  customerId char(36),
  firstName varchar(75),
  lastName varchar(75),
  organizationId char(36)
);

create table CustomerAddress
(
  customerAddressId char(36),
  street varchar(100),
  city varchar(100),
  state char(2),
  zip varchar(10),
  customerId char(36),
  organizationId char(36)
);
```

Sample Database Query after VPD

Note that there is an additional join that must be performed to read the same data. This join effectively segments the customer data, making it possible to store multiple customer datasets in the same database instantiation. Accompanying the added join is another test for the organizationId that is populated into the statement from the data access Java code.

```
select
c.firstName, c.lastName,
ca.street, ca.city, ca.state, ca.zip
from
Customer c, CustomerAddress ca
where
c.customerId = ?
and
c.organizationId = ?
and
c.customerId = ca.customerId
and
ca.organizationId =
c.organizationId
```

The Effects on Performance

Since the VPD approach introduces more table joins for the Relational Database Management System to perform when issuing queries to the system, and since joins are 'expensive' in database performance terms, the implementation of VPD degrades system performance.

Propagating Complexity

The complexity involved in the VPD approach is not confined to the database schema. In addition, all Structured Queries must be written to accommodate the additional joins. Furthermore, Java code that issues queries to the system and parses database result sets must take this additional complexity into account. It is obvious that the cost of taking this approach is high since the complexity leaks to other areas of system development.

Additional Database Complexity

As you can see, the database design becomes more complex when implementing VPD. Therefore, the cost to maintain a VPD database will also go up, assuming that cost and complexity are proportional. I make a strong appeal to common sense in support of this assertion.

Conclusions about VPD

As you can see from the example, the VPD approach to solving the problem of operational efficiency has serious issues. Although it does allow the team to segment the data in the required fashion, it appears to increase development overhead by adding complexity to the database structure and queries.

Additionally, there are performance issues at the database level that will degrade database performance, requiring the operations team to upgrade the equipment faster than if the extra joins were not required. There are development complexities that propagate to the Java database interaction code.

Operating under the assumption that the application will become wildly successful, and that user customer demand will meet or exceed expectations, the applications and network topology will significantly change to meet the performance needs of customers.

If It Weren't Bad Enough

One way to accommodate the increase in demand is to move customer databases (virtual or otherwise) to different physical machines. Consider the VPD case. Special queries would be required to extract customer data by organization. Then, routines would be needed to load the data into another schema. Both of these activities would have some output that would require maintenance over time.

One way to Mitigate the Downside

One way to mitigate the software maintenance issue is to write generic stored procedures at the database level that post process all queries coming into the System. This approach has three downsides that made the approach untenable. First, it will require development effort to implement and test the procedures. Additionally, this approach does not address the issue of excessive joining at the RDBMS level. Lastly, since stored procedure syntax changes from one RDBMS vendor to another, unless the development team undertakes the task of rewriting (and maintaining) the same stored procedures for every database vendor, the company's RDBMS deployment options would be limited to a single RDBMS system. From a strategic standpoint, remaining database agnostic lowers switching costs if a new RDBMS serves the needs of the business better.

A Reusable Approach

The development team at DCI recognized that the VPD approach would be difficult to build and difficult to maintain. As a result, they did some research and came up with a more useful solution that didn't require the instrumentation of each database table with an extra field. Their solution did not require the extra SQL joins. In addition, it outlined a clear path for the migration of customer data as the popularity of their application would grow. Lastly, the solution that they came up with would work with any application with similar requirements to their ecommerce application. This gave the business a strategic advantage inasmuch as new market opportunities that were identified would not require the re-development of security, and could scale to meet user demand in the same way as their retail management system.

The Notion of a Data 'Domain'

A single database structure can be instantiated multiple times within the same database. Applications talk to databases over the network using a database driver. In the case of Tomcat, Java Database Connectivity (JDBC) is used to connect web applications running within the Tomcat server process with any number of different RDBM systems. In the case of DCI, they decided to instantiate one instance of the database per customer. Since Tomcat provides a way to segment customers by application using the 'context' file, they needed a way to connect each customers' application with its respective Data Domain. This was accomplished using Java Naming and Directory Interface (JNDI) connection pools, which are configured for each context. The customers' applications are tied to their respective Data Domains using this method, but lacking was a way to tie the customers' users to the application domains when they logged into the system. This required a little tweaking of the default tomcat security realm.

The Security Database

The security database is used to store user authentication and authorization information. The database stores information about the users that are available on the system, what roles they play within a given application, and which domains in which they are granted privileges to access.

Security Database DDL

The UserInfo table is used to store personal information about the User including the authentication information (username and password).

The Domain table stores the available data domains on the system. The dataSource field in the Domain table matches the catalog name that is returned by the JDBC Driver implementation.

If a user is attempting to access a particular data domain, the user must have a record in the UserInfoRole table that correlates the domain being requested with the data domain catalog name, and that specifies any roles that the user has within that domain.

```
CREATE TABLE UserInfo
(
  id varchar(36) NOT NULL,
  email varchar(100) NOT NULL,
  password varchar(36),
  CONSTRAINT PRIMARY KEY (id),
  UNIQUE(email)
);

CREATE TABLE UserInfoRole
(
  role varchar(36) NOT NULL,
  userId varchar(36) NOT NULL,
  domainId varchar(250) NOT NULL,
  UNIQUE(role, userId, domainId)
);

CREATE TABLE Domain
(
  id varchar(36) NOT NULL,
  name varchar(250) NOT NULL,
  dataSource varchar(250) NOT
NULL,
  CONSTRAINT PRIMARY KEY(id)
);
```

J2EE Realm Security

When a user makes a request to access a DCI application over the web, the Tomcat J2EE container intercepts the user's request and introspects upon it. Tomcat then looks at the security configuration for the web application to determine if the user must login. If the web application security configuration states that the URI that the user is attempting to access is protected, then it will prompt the user with an authentication request by displaying a username-password prompt.

Once the user fills-out the authentication information and submits the authentication request, Tomcat receives the request and relinquishes control to a security Realm. The Realm must implement a method that takes in the username-password combination, and returns a Principal object. The Principal is a server-side representation of the user's access rights on the system.

Example Application Security Configuration

Below is an example of the security configuration that is packaged with a web application, and deployed into Tomcat. Multiple security-constraint elements can be specified in the web application metadata (a.k.a. web.xml) to correlate user roles with web application services by URL.

```
<security-constraint>
  <display-name>Administrative Security Configuration</display-
name>
  <web-resource-collection>
    <web-resource-name>
      Administrative Security Configuration
    </web-resource-name>
    <url-pattern>/admin</url-pattern>
    <http-method>DELETE</http-method>
    <http-method>GET</http-method>
    <http-method>POST</http-method>
    <http-method>PUT</http-method>
  </web-resource-collection>
  <auth-constraint>
    <role-name>myapplication.admin</role-name>
  </auth-constraint>
</security-constraint>
```

Example of Custom Realm Implementation

The custom realm overrides a single method that delegates to a custom Data Access Object (DAO). The DAO has the responsibility of reading the user's authentication and authorization information from the database, also known as a Principal.

After Tomcat delegates to the Realm to perform the authentication, it sends a message to the Principal object to authorize the user's request. Assuming that the user is authenticated, the Realm is no longer consulted for the remainder of the user's session. The Principal object that is returned from the DAO request is stored in the user's server-side session, and can be accessed for each subsequent request to authorize user access to protected resources.

Configuring the Realm in Tomcat

Tomcat has several configuration files. The primary system file is known as server.xml. This configuration file defines the host (Ex: www.mydomain.com), the Realm to be used, and the data source configuration for the security database.

Security Database Datasource Configuration

A connection pool is a collection of pre-initialized database connections. For applications that must support concurrency (multiple users accessing the application at the same time), the initialization and destruction of database connection resources can cause a bottleneck since both of these processes have a fixed overhead. Database connection pools were created to eliminate the connection initialization and destruction overhead. Below is a sample configuration for a connection pool to the centralized security database instance. Note that there is one authorization database for any number of data domain database instances.

```
<GlobalNamingResources>
  <Resource
    name="authdb"
    auth="Container"
    type="javax.sql.DataSource"/>
  <ResourceParams name="authdb">
    <parameter>
      <name>factory</name>
      <value>
        org.apache.commons.dbcp.BasicDataSourceFactory
      </value>
    </parameter>
    <parameter>
      <name>maxActive</name>
      <value>10</value>
    </parameter>
    <parameter>
      <name>maxIdle</name>
      <value>15</value>
    </parameter>
    <parameter>
      <name>maxWait</name>
      <value>10000</value>
    </parameter>
    <parameter>
      <name>username</name>
      <value>dbuser</value>
    </parameter>
    <parameter>
      <name>password</name>
      <value>12xc43</value>
    </parameter>
    <parameter>
      <name>driverClassName</name>
      <value>com.mysql.jdbc.Driver</value>
    </parameter>
    <parameter>
      <name>url</name>
      <value>jdbc:mysql://localhost:3306/authdb</value>
    </parameter>
    <parameter>
      <name>removeAbandoned</name>
      <value>true</value>
    </parameter>
    <parameter>
      <name>removeAbandonedTimeout</name>
      <value>60</value>
    </parameter>
    <parameter>
      <name>logAbandoned</name>
      <value>true</value>
    </parameter>
  </ResourceParams>
</GlobalNamingResources>
```

Realm Configuration

The Realm configuration is quite simple. It is a one-line XML entry in the Tomcat's primary configuration file, server.xml.

```
<Realm
  className="com.mycompany.MyCustomRealm
"
  debug="99"/>
```

Security Realm Polymorphism

The team used polymorphism by overriding the 'authenticate' method in the Realm super class. This method is called by the container after a user submits his or her username and password to be authenticated for access to the system, and subsequently authorized for access to a particular system resource.

```
public Principal authenticate(String userName,
    String password)
{
    DBConnection conn = null;
    try
    {
        conn = new DBConnection(DBConnection.AUTHDB);
        // Acquire a Principal object for this user
        Principal principal = AuthDAO.
            authUser(conn, userName,
                password);
        return principal;
    }
    catch (Exception e)
    {
        log("Unable to auth user [" +
            userName + "] password [" +
            password + "]", e);
        return null;
    }
    finally
    {
        if(conn != null)
        {
            conn.cleanup();
        }
    }
}
```

DAO Method

You may have noticed that the overridden Realm method makes a call to a component to authorize a user. This component is called a Data Access Object (DAO), and uses implementation hiding. The type of RDBMS, the type of driver, and any data access strategies are hidden from the calling code (in this case the Realm for the purposes of authentication). This provided the team with the option to reuse the authentication method for other purposes, or to reuse the code in future, more robust custom security implementations.

```
public static UserInfo authUser(DBConnection conn,
    String userName, String password) throws SQLException,
    EntityNotFoundException
{
    PreparedStatement ps = conn.getPSForSQL(sqlFactory.
        getSql(authUser));
    ps.setString(1, userName);
    ps.setString(2, password);
    ResultSet results = ps.executeQuery();
    conn.addResultSet(results);
    if(results.next())
    {
        UserInfo user = AuthDAO.parseUser(results);
        return user;
    }
    else
    {
        throw new EntityNotFoundException(
            "Unable to read user " +
            "userName [" + userName + "]");
    }
}
```

SQL Used by the DAO to Authenticate a User

The DAO above uses Java Database Connectivity (JDBC) to access and iterate over data maintained by the RDBMS. A Structured Query (SQ) is issued to read a user's authorization information. The SQ is below.

```
select
u.id as userId, u.email as userEmail,
u.password as userPassword,
u.disabled as userDisabled,
uir.role as userInfoRole,
d.id as domainId, d.name as domainName,
d.commerceEnabled as domainCommerceEnabled,
d.merchantUser as domainMerchantUser,
d.merchantPass as domainMerchantPass,
d.dataSource as domainDataSource
from
UserInfo u left join UserInfoRole uir on u.id = uir.userId
left join Domain d on uir.domainId = d.id
where
u.email = ?
and
u.password = ?
```

Principal Object Polymorphism

A principal is a server side representation of a system user's authentication and authorization information. For the custom security implementation, the team developed its own Principal object to encapsulate the authentication and authorization information for each user. Of particular interest is the 'getRoles' method, which is called by the Tomcat container to authorize a user's request for a particular system resource. Recall that the web application deployment descriptor (web.xml) contains information about which roles have access to which resources. The roles defined in the web application deployment descriptor must match the roles that have been assigned to the user, or the Tomcat container will deny access to a system resource.

```
public String[] getRoles()
{
    Collection result = new ArrayList();
    Iterator rolesIt = this.roles.iterator();
    UserInfoRole currentRole;
    while(rolesIt.hasNext())
    {
        currentRole = (UserInfoRole)rolesIt.next();
        result.add(currentRole.getRole());
    }
    return (String[])result.toArray();
}

public boolean authorize(String forDomain)
{
    return this.rolesByDomainHash.get(forDomain) != null;
}
```

The Tomcat Application Context File

The Context file contains reference to the data source that is used for tomcat, defines the document base where the web application files to be used are stored, and the context to use when accessing the application, i.e., www.mydomain.com/context. Different context files can deploy the same instance of a web application within Tomcat. This feature is critical for meeting the requirements of the DCI development team.

```
<Context
  path = "/contextname"
  docBase = "applicationdocumentbase"
  debug = "false"
  reloadable = "true"
  crossContext = "true">

  <Resource
    name="jdbc/domaindb"
    auth="Container"
    type="javax.sql.DataSource"/>

  <ResourceParams name="jdbc/domaindb">
    <parameter>
      <name>factory</name>
      <value>
org.apache.commons.dbcp.BasicDataSourceFactory</value>
    </parameter>
    <parameter>
      <name>maxActive</name>
      <value>10</value>
    </parameter>
    <parameter>
      <name>maxIdle</name>
      <value>15</value>
    </parameter>
    <parameter>
      <name>maxWait</name>
      <value>10000</value>
    </parameter>
    <parameter>
      <name>username</name>
      <value>dbuser</value>
    </parameter>
    <parameter>
      <name>password</name>
      <value>12xc43</value>
    </parameter>
    <parameter>
      <name>driverClassName</name>
      <value>com.mysql.jdbc.Driver</value>
    </parameter>
    <parameter>
      <name>url</name>
      <value>
jdbc:mysql://www.mydatabaseserver.com:3306/yourdb</value>
    </parameter>
    <parameter>
      <name>removeAbandoned</name>
      <value>true</value>
    </parameter>
    <parameter>
      <name>removeAbandonedTimeout</name>
      <value>60</value>
    </parameter>
    <parameter>
      <name>logAbandoned</name>
      <value>true</value>
    </parameter>
  </ResourceParams>

  <ResourceLink global="authdb" name="jdbc/authdb"/>
</Context>
```

Flexibility Through Polymorphism and Declarative Security

The Object Oriented Analysis and Design facilities in the Java programming language coupled with the declarative security and configuration capabilities of the Tomcat J2EE Internet Operating System make the implementation of the DCI security platform possible. Java and Tomcat themselves possess many layers of indirection that are hidden from their user base that provide great value via the encapsulation of complex operations, and the exposure of complex operations through a cohesive, normalized interface. As you have probably determined on your own - either before reading this paper, or as a result of reading it - reuse in the software development domain is everywhere. The proper decomposition and encapsulation of system functionality at various levels of abstraction enables software developers to reach new heights in terms of the complex systems they model with their designs and software code. These aspects of software development - cohesion, normalization, polymorphism, decomposition, reuse, and software problem domains - should be taken to heart by any software designer that aspires to model real world processes of ever increasing complexity.

Breaking Down the Risk Barrier

Generally speaking, specialization yields scalability. As globalization gains steam, it will be more important for industry to accommodate the increasing demand that comes with access to world markets. One way to do this is through specialization through development of supporting industries.

The purpose of DCI is to provide software services to other industries by maintaining a relatively narrow focus in the way it develops software.

By developing techniques for producing reliable, scalable, flexible software, DCI has positioned itself to be competitive within its own industry, and to provide competitive advantage to its clients by being highly specialized and passing on savings from economies of scale to its customers.

As a unique outsourcing option because of the way in which it conducts business, DCI provides value by reducing risk associated with traditional outsourcing methods, or with internal software development approaches. For many companies, the failure rate of IT projects is around 70% [12]. The lack of confidence in the abilities of internal IT departments to deliver may have resulted in many lost opportunities for operational enhancement. Through its innovative business model, and through its optimized internal software development practices, DCI is breaking down the barriers to process optimization. As a result, firms, industries, and nations will become more competitive as they look forward to the introduction of affordable, low risk, highly scalable DCI information technology solutions into their organizations.

Bibliography

- [1] Wikipedia encyclopedia (Source lines of code)
http://en.wikipedia.org/wiki/Source_lines_of_code
- [2] Gupta, A., Mitra, A., Agile Systems with Reusable Patterns of Business Knowledge 2005
- [3] Larman, C., Applying UML and Patterns 1998
- [4] Postel, Jon. Transmission Control Protocol, IETF, September 1981
- [5] Spendolini, S., Using Virtual Private Database in an Oracle HTML DB Application,
http://www.oracle.com/technology/pub/notes/technote_htmlldb_vpd.html
- [6] Porter, M., The Competitive Advantage of Nations 1990
- [7] Apache Tomcat <http://tomcat.apache.org/>
- [8] Wikipedia encyclopedia (Open Source)
http://en.wikipedia.org/wiki/Open_source
- [9] Java 2 Enterprise Edition <http://java.sun.com/j2ee/>
- [10] Class Loader
<http://java.sun.com/j2se/1.4.2/docs/api/java/lang/ClassLoader.html>
- [11] Datasource connection pool
http://www.developer.com/db/article.php/10920_2172891_3
- [12] Pearlson, K., Saunders, C. Managing and Using Information Systems (a strategic approach) 2nd edition Wiley, 2004