**International Journal of Engineering Research and Development**
**e**-ISSN: 2278-067X, **p**-ISSN: 2278-800X, www.ijerd.com
Volume 6, Issue 11 (April 2013), PP. 36-42

# Building an Efficient and Flexible Architecture for Caching Data Objects in Distributed Environment Using RMI and Observer Design Pattern

Mukesh D. Parsana[1], Jayesh N. Rathod[2], Jaladhi D. Joshi[3]
[1]M.E. (C.E., Pursuing), Atmiya Institute of Technology & Science (Gujarat Technological University), INDIA.
[2]HOD (Department of Computer Engineering), Atmiya Institute of Technology & Science, Gujarat, INDIA.
[3]M.S. (CE), Florida Atlantic University, Boca Raton, FL, USA.
[3] Research Assistant, Motorola Inc., Sunrise, FL, USA.

**Abstract: -** Database connection management is one of the important issues for any web application. It involves retrieving initial context, retrieving appropriate data source, opening and closing connection with the database and so on. For the large database which involves many modules and functionalities and so many data sources and DAOs available, it becomes very complex to manage database connections and data sources. Caching is one of the primary techniques to decrease the response times of applications that frequently access data stores. As database operations are time consuming, it is better to store data access objects DAOs in cache and retrieve as and when required. But at the same time consistency of cache with database and is an important issue to consider. This paper discusses how RMI (Remote Method Invocation) and design and programming expertise of Observer design pattern can be used to build and efficient and flexible architecture for caching data objects in distributed environment. The Observer design pattern defines a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.

**Keywords:-** FIFO, LRU, LRU2, 2Q, LFU, ARC, MRU, TTL, RMI, Observer Pattern

## I. CACHING IN DISTRIBUTED ENVIRONMENT

A distributed system is a heterogeneous system. Diverse system components are connected to each other via a common network. Applications using TCP/IP-based Internet are examples of open distributed systems.

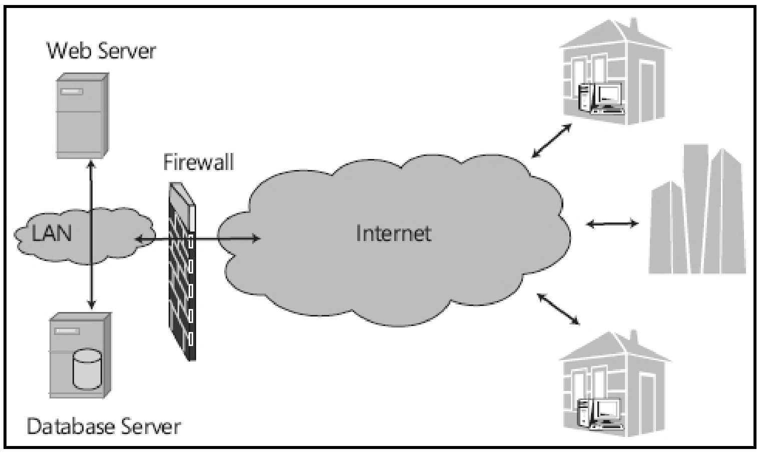

**Fig. 1:** Typical Distributed Architecture

In the distributed environment, different activities occur in concurrent fashion. Usually, common resources like the underlying network, Web/application servers, database servers, and cache servers are shared by many clients. Distributing the computing load is the hallmark of distributed systems. Resource sharing and allocation is a major challenge in designing distributed architecture. For example, consider a Web-based database-driven business application. The Web server and the database server are hammered with client requests.

Caching, load-balancing, clustering, pooling, and time-sharing strategies improve the system performance and availability.

Any frequently consumed resource can be cached to augment the application performance. For example, caching a database connection, an external configuration file, workflow data, user preferences, or frequently accessed Web pages improve the application performance and availability. Many distributed-computing platforms offer out-of-the-box caching infrastructure. Java Caching System (JCS) is a distributed composite caching system. In Microsoft .NET Framework, the System.Web.Caching API provides the necessary caching framework. The Microsoft project code-named "Velocity" is a distributed-caching platform. [6]

The performance of a caching system depends on the underlying caching data structure, cache eviction strategy, and cache utilization policy. Typically, a hash table with unique hash keys is used to store the cached data; JCS is a collection of hash tables.

In production, many applications buckle down because they treat the database as their cache. Web server-based application-level cache can be effectively used to mitigate this problem. An effective caching mechanism is the foundation of any distributed-computing architecture.

ORM technologies are becoming part of the mainstream application design, adding a level of abstraction. Implementing ORM-level cache will improve the performance of a distributed system. Different ORM caching levels such as transactional cache, shared cache, and the details of inter-cache interaction are discussed below. The impact of ORM caching on application design is also explored.

## II. GENERAL ARCHITECTURE OF CACHING MECHANISM

Caching is one of the primary techniques to decrease the response times of applications that frequently access data stores. These data stores could be directory servers, relational databases or file systems.

Entries in the cache are stored with a key for each entry. Entries are read or added or removed based on this key. When an application using a caching mechanism needs to access the data store, it recreates the key for the entry and checks if it is present in the cache. If the entry is present, the entry from the cache is returned. If the entry is not present, it is retrieved from the data store and is added to the cache.

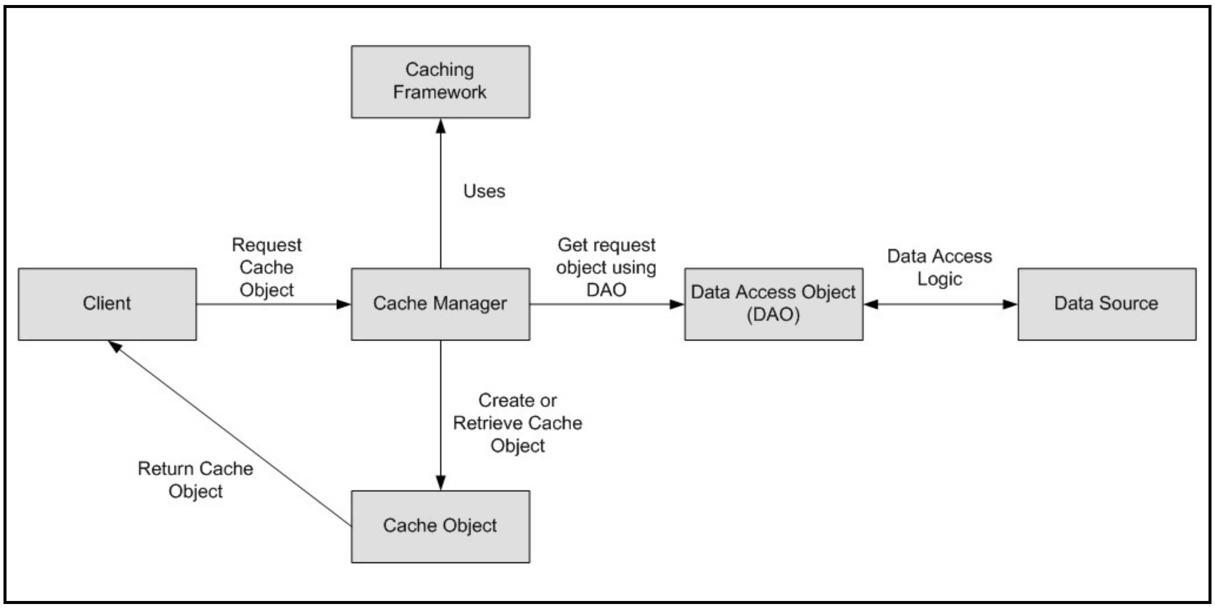

**Fig. 2:** General Architecture of Caching Mechanism

HashMap or LinkedList structures can be used for implementing caching algorithms. The HashMap stores the cache entries and the LinkedList will be used for keeping a record of the most recent cache access. The LinkedList will store the keys for cache entries. Most recent accesses will be moved to the top of the list and entries will be removed from the end of the list. The Hashmap stores entries that are retrieved based on their key values.

Cache algorithms are optimizing instructions that a computer program can follow to manage a cache of information stored on the computer. The efficiency of a cache is largely dependent on the caching algorithm that is used. Cache size is usually limited, and if the cache is full, the computer (that is, the programmer) must decide which items to keep and which to discard to make room for new items.

The most efficient caching algorithm would be to always discard the information that will not be needed for the longest time in the future. Since it is impossible to predict how far in the future information will be needed, this is not implementable in hardware, however (with pre-defined simulations) it can be used as a gauge as to the effectiveness of other algorithms.

Important factor in determining an effective caching strategy is the lifetime of the cached resource. Usually, resources stored in the temporal cache are good for the life of an application. Resources that are stored in the spatial cache are either time-dependent or place-dependent. Time-dependent resources should be purged as per the cache expiration policy. Place-specific resources can be discarded based on the state of the application. In order to store a new resource in the cache, an existing cached resource will be moved out of the cache to a secondary storage, such as the hard disk. This process is known as paging. Replacement algorithms such as least frequently used resource (LFU), least recently used resource (LRU), and most recently used resource (MRU) can be applied in implementing an effective cache-eviction strategy, which influences the cache predictability. The goal in implementing any replacement algorithm is to minimize paging and maximize the cache hit rate. The cache hit rate is the possibility of finding the requested resource in the cache. In most cases, LRU implementation is a good enough solution. JCS and ASP.NET caching is based on the LRU algorithm. In more complex scenarios, a combination of LRU and LFU algorithms such as the adaptive replacement cache (ARC) can be implemented. The idea in ARC is to replace the least frequently and least recently used cached data. This is achieved by maintaining two additional scoring lists. These lists will store the information regarding the frequency and timestamp of the cached resource. ARC outperforms LRU by dynamically responding to the changing access pattern and continually balancing workload and frequency features. Some applications implement a cost-based eviction policy. For example, in Microsoft SQL Server 2005, zero-cost plans are removed from the cache and the cost of all other cached plans is reduced by half. The cost in SQL Server is calculated based on the memory pressure. [4][5]

A study of replacement algorithms suggests that a good algorithm should strike a balance between the simplicity of randomness and the complexity inherent in cumulative information. Replacement algorithms play an important role in defining the cache-eviction policy, which directly affects the cache hit-rate and the application performance.

Examples of some good cache replacement algorithms and their features are listed down below.

**Table 1:** Caching Algorithms and their Features

| Algorithm Name | Characteristics and Features |
|---|---|
| FIFO (First In First Out) | The advantage of this algorithm is that it's simple and fast; it can be implemented using a simple array and an index. The disadvantage is that it's not very smart; it doesn't make any effort to keep more commonly used items in cache. |
| LRU (Least Recently Used) | This is the most common and straightforward caching algorithm. There is a fixed amount of objects that can be cached, and this is configurable. When the caching of a new object is requested but the cache is full, the object that has least recently been requested from the cache (or updated in the cache) is ejected to make space.<br>**Automatic:** This algorithm works with the JVM's garbage collector to allow cached objects to be automatically garbage-collected. The cache maintains soft references to cached objects. These allow the cached objects to be garbage collected by the system as necessary. This allows for a cache size that is bounded only by the amount of memory available to the JVM. However, unlike the LRU algorithm, it does not necessarily guarantee that frequently accessed objects will always be available in the cache.<br>**Timeout:** This algorithm times out objects that have been untouched for a number of milliseconds. The timeout amount is specified via the property cache timeout.<br>**Hybrid:** This combines the benefits of both the LRU and Automatic algorithms. Two levels of caching are provided. At the first level, an LRU cache ensures that a fixed number of the most recently used objects are available in the cache. At the second level, an Automatic cache holds cached objects that are available to be reclaimed by the garbage collector. |
| LRU2<br>(Least Recently Used Twice) | The advantage of this algorithm is that it's adaptive, like LRU, and in addition won't fill up from scanning accesses, since items aren't retained in the main cache unless they've been accessed more than once. |

| | |
|---|---|
| 2Q (Two Queues) | Items are added to an LRU cache as they are accessed. If accessed again, they are moved to a second, larger, LRU cache. Items are typically ejected so as to keep the first cache at about 1/3 the size of the second. This algorithm attempts to provide the advantages of LRU2 while keeping cache access overhead constant, rather than having it increase with cache size. |
| LFU (Least Frequently Used) | Frequency of use data is kept on all items. The most frequently used items are kept in the cache. Because of the bookkeeping requirements, cache access overhead increases logarithmically with cache size. The advantage is that long term usage patterns are captured well, incidentally making the algorithm scan resistant; the disadvantage, besides the larger access overhead, is that the algorithm doesn't adapt quickly to changing usage patterns, and in particular doesn't help with temporally clustered accesses. |
| ARC (Adaptive Replacement Cache) | ARC improves the basic LRU strategy by splitting the cache directory into two lists, T1 and T2, for recently and frequently referenced entries. In turn, each of these is extended with a ghost list (B1 or B2), which is attached to the bottom of the two lists. These ghost lists act as scorecards by keeping track of the history of recently evicted cache entries, and the algorithm uses ghost hits to adapt to recent change in resource usage. Note that the ghost lists only contain metadata (keys for the entries) and not the resource data itself, i.e. as an entry is evicted into a ghost list its data is discarded. |
| MRU (Most Recently Used) | The 'Most Recently Used' technique is a simple algorithm that limits the size of the cache to a pre-defined number of entries or a pre-defined size (in memory). In this algorithm, the most recently used entry is at the top of the list and entries are removed from the end of the list. As is obvious, we will need to maintain a record of most recent cache accesses. |

## III. CACHING ISSUES

Constraining the size of the cache is the biggest issue. If we had caches without any limitations on the size of the cache, they will take up a lot of memory space. In addition, searching for an entry in the cache will take longer and would defeat the whole purpose of caching. We need to do is come up with algorithms that constrain the size of the cache to either a specific number of entries or a specific memory size.

Another issue is that of synchronization with the data in the store (RDBMS, file system etc.). What if the data in the store itself changes? Our cache will become dirty, i.e. the cache will no longer contain the latest data. There are two methods to resolve this - maintaining a TTL for cache entries or using event notification.

A Time-To-Live (TTL) for a cache specifies the time after which the cache becomes invalid. The cache is emptied when the TTL for the cache expires. The cache is then rebuilt again or all the entries in the cache at the time it was flushed are read back into the cache. We could also specify a TTL for each entry and rewrite the entry into the cache from the store when the TTL for the entry expires.

Event notification is a better way to synchronize caches with data stores. In this mechanism, an event listener is registered with the data store. This listener tracks the changes occurring on the data store. If any entry present in the cache is changed (in the store) the entry is re-written into the cache.

## IV. OBSERVER DESIGN PATTERN

The Observer Pattern defines a one-to-many dependency between objects so that when one object changes state, all of its dependents are notified and updated automatically. The Observer Pattern provides an object design where subjects and observers are loosely coupled. Loosely coupled designs allow us to build flexible OO systems that can handle change because they minimize the interdependency between objects. When two objects are loosely coupled, they can interact, but have very little knowledge of each other. [7][8]

Some important points about Observer Design Pattern shown in the above figure are as below: [1][2][3]

(1) **The only thing the subject knows about an observer is that it implements a certain interface (the Observer interface).** It doesn’t need to know the concrete class of the observer, what it does, or anything else about it.

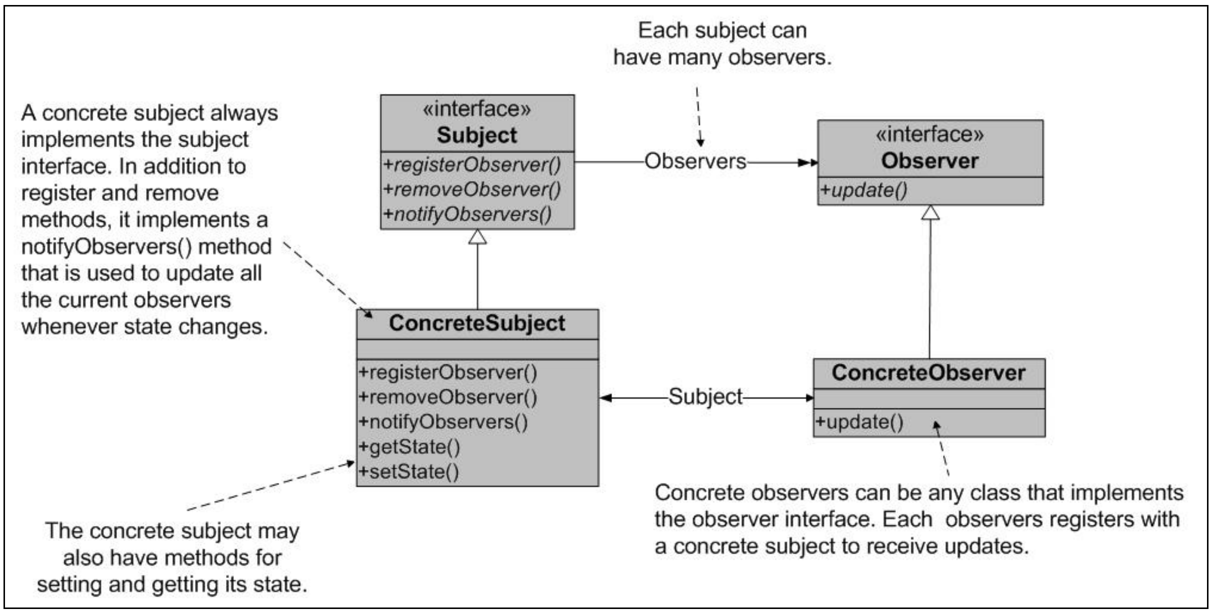
Each subject can have many observers.
A concrete subject always implements the subject interface. In addition to register and remove methods, it implements a notifyObservers() method that is used to update all the current observers whenever state changes.
«interface» Subject
+registerObserver()
+removeObserver()
+notifyObservers()
Observers
«interface» Observer
+update()
ConcreteSubject
+registerObserver()
+removeObserver()
+notifyObservers()
+getState()
+setState()
Subject
ConcreteObserver
+update()
Concrete observers can be any class that implements the observer interface. Each observers registers with a concrete subject to receive updates.
The concrete subject may also have methods for setting and getting its state.

**Fig. 3:** Observer Design Pattern Architecture

(2) **We can add new observers at any time.** Because the only thing the subject depends on is a list of objects that implement the Observer interface, we can add new observers whenever we want. In fact, we can replace any observer at runtime with another observer and the subject will keep purring along. Likewise, we can remove observers at any time.

(3) **We never need to modify the subject to add new types of observers**. Let's say we have a new concrete class come along that needs to be an observer. We don't need to make any changes to the subject to accommodate the new class type, all we have to do is implement the Observer interface in the new class and register as an observer. The subject doesn't care; it will deliver notifications to any object that implements the Observer interface.

(4) **We can reuse subjects or observers independently of each other.** If we have another use for a subject or an observer, we can easily reuse them because the two aren't tightly coupled.

(5) **Changes to either the subject or an observer will not affect the other.** Because the two are loosely coupled, we are free to make changes to either, as long as the objects still meet their obligations to implement the subject or observer interfaces.

## V. USING RMI AND OBSERVER DESIGN PATTERN TO BUILD AN EFFICIENT AND FLEXIBLE ARCHITECTURE FOR DISTRIBUTED CACHING MECHANISM

JCS (Java Caching Mechanism) provides a way cache objects using LRU algorithm. But research has proved that ARC (Adaptive Replacement Cache) outperforms LRU in many contexts. So it is always better to have a flexible mechanism in which cache replacement algorithm can be configured as per requirements and need. As the same time managing cache instances and keeping them informed about the changes are important issues to consider. Following figure shows an efficient and flexible manner to build a caching mechanism that solves all these problems. This architecture will work as per the steps given below.

(1) Clients request the requested object. This request will go to the Cache Manager.

(2) Cache manager checks whether requested object is presented in the cache or not.

(3) If the requested object is there in the cache, it will be returned to the client. Cache will be updated and entries will be rearranged accordingly to the caching rules.

(4) If the requested object not there in the cache, it will be retrieved from the database and appropriate entry will be deleted in the cache for future use. Cache will be updated and entries will be rearranged according to caching algorithm rules.

(5) Application may have more than one instances of cache (Distributed caching mechanism). Observer pattern is used to update all these instances automatically if there are any changes in the database by application or by some external operations. In this case, each cache instances will be treated as the observer. They will register themselves to main interface (subject) to receive any notifications about the data changes in the database.

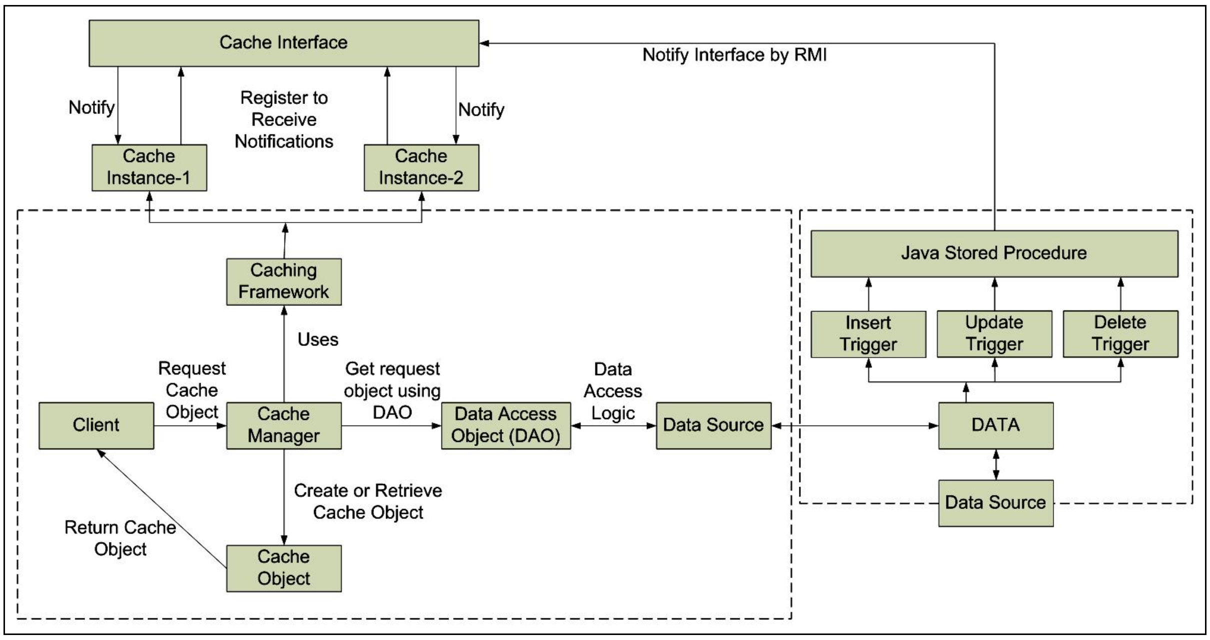

**Fig. 4:** Efficient and Flexible Architecture for Distributed Caching Mechanism

(6) If data is changed in the database by any of the database operations i.e. insert, update or delete, appropriate trigger will be fired. These triggers will call appropriate java stored procedures loaded in the database. These Java Stored Procedures will notify the main interface (subject) by using RMI (Remote Method Invocation) mechanism about the changes occurred in the database.
(7) Main interface will notify and update all the cache instances registered with it at that particular time.
(8) As expertise of the observer design pattern is going to be used in this architecture, new cache instances can be created and registered with the main interface without major code changes. The newly added cache instance will start getting notifications from the main interface after registration.

## VI. CONCLUSIONS

Response time can be significantly reduced by building efficient cache architecture for distributed application. Data Access Objects and Data stores can be stored in the cache for retrieving it quickly as and when retrieved. But at the same time synchronizing cache with actual database is an important issue to consider. Using java stored procedure, cache instances and interfaces can be notified about the changes occurred in the database. Remote method invocation APIs of Java are used to notify remote cache interfaces and instances. Observer Design Pattern can be used to notify all distributed cache instances automatically. New cache instance can be added easily due to Observer design pattern without having code change in the current architecture. Thus efficient caching mechanism for distributed environment can be implemented in J2EE architecture with the help of following technologies and design patterns: SQL Trigger Procedures, Java Stored Procedures, Remote Method Invocation (RMI), and Observer Design Pattern to notify all cache instances automatically in distributed environment.

## ACKNOWLEDGMENT

The author would like to express his earnest gratitude to his guide Jayesh N. Rathod, Head of Computer Engineering department at Atmiya Institute of Technology & Science, Gujarat, INDIA for his constant guidance, encouragement and moral support which helped the author to accomplish this research work. The author would also like to extend his sincere gratitude to Mr. Jaladhi D. Joshi (M.S. (CE), Florida Atlantic University, Boca Raton, FL, USA) Who is working as Research Assistant at Motorola Inc., Sunrise, FL, USA for helping him in all possible ways.

## REFERENCES

[1]. Design Patterns in Software Development by MU Huaxin & JIANG Shuai, Beijing, IEEE 2011 (ISBN: 978-1-4244-9699-0)
[2]. An efficacious software design method based on pattern and its application by Chuanjun Li & Qing Wang & Wenwen Cai & Jun He, Chengdu, China, August 2010 (Print ISBN: 978-1-4244-7324-3)

[3]. Some domain patterns in Web application framework by Liu Chao & He Keqing & Liu Jie & Ying Shi, Proceedings of the 27th Annual International Computer Software and Applications Conference, 2003 IEEE (ISSN : 0730-3157)

[4]. Design Space Exploration of Object Caches with Cross-Profiling by Martin Schoeberl & Walter Binder & Alex Villaz on, 2011 14th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing (ISSN : 1555-0885)

[5]. A Time-Predictable Object Cache by Martin Schoeberl, 2011 14th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing (Print ISBN: 978-1-61284-433-6)

[6]. Web Reference: http://code.msdn.microsoft.com/velocity

[7]. Head First Design Patterns by Eric Freeman, Elisabeth Robson, Bert Bates, Kathy Sierra, O'Reilly Media inc.

[8]. Head First Object-Oriented Analysis and Design by Brett D. McLaughlin, Gary Pollice, Dave West, O'Reilly Media inc.

## ABOUT THE AUTHOR

Mr. Mukesh Parsana completed his graduation in Information Technology discipline from one of the most reputed engineering colleges in India, Nirma Institute of Technology Ahmedabad. Currently he is pursuing his master degree in Computer Engineering form Atmiya Institute of Technology and Science Gujarat. He has worked with India’s top multinational software firm, Infosys Technologies, for four years as senior software developer. Sun Microsystems/Oracle awarded him with 3 international certificates (SCJP, SCWCD and SCBCD) for his expertise in different J2EE technologies. Due to his expertise on j2EE technologies, he has received offer of employment from multinational companies like Wipro Technologies, TCS, IBM and Persistent Technologies.

**SCJP: Sun Certified Java Programmer for Java 2 Platform 1.4 (CX-310-035)**
**SCWCD: Sun Certified Web Component Developer for J2EE V1.4 (CX-310-081)**
**SCBCD: Sun Certified Business Component Developer for J2EE V1.3 (CX-310-090)**