Design Optimizations of Enterprise Applications – A Literature Survey

Willie (Phong) Tu
Dept. of Computer Science, University of Calgary
w.tu@acm.org

Abstract

Software design patterns have been defined as possible solutions to reoccurring problems. One aspect of the problems is performance. This can be arguably true in high volume enterprise applications. This paper will explore the current discussions and research in utilization of software design patterns to optimize enterprise applications.

1. Introduction

First, let's define what is design optimization. In this context, design optimization is the improvement of an application's design and performance using software design patterns. Design improvements can be subjective, thus the main qualifier is the utilization of software design patterns to improve the performance of an application. The concentration would be software patterns improving the performance of enterprise applications.

Software design patterns are solutions to recurring design problems in software development. Utilization of these patterns provides a more elegant and reusable object-oriented design [11]. However, here we will investigate its utilization in terms of optimizing the performance of enterprise applications. This is not limited to patterns that can improve performance but also patterns that can cause performance degradation.

Performance improvements to an application can be a measure in different dimensions. In one dimension there is the subjective perceived running time of operations versus the objective actual running time of an operation. Perceived running time of the system is where the user perceives the operation time to be. This could be due to providing more feedback on what the system is doing, occupying the user with other tasks, or displaying partial results from the operation, among many other strategies. We will not be concerned with the perceived running time improvements but rather the actual execution time of an operation.

Another dimension is the performance improvement for the single user versus multiple users, in other words, scalability. The design optimization patterns noted here will focus on actual running time of operations for single user performance and multi-user scalability improvements.

The organization of the findings is categorized through the usage of an architectural pattern, which is defined as layer [5]. The layers used are the primary three-layer architecture for information systems [10,21], which includes the data access layer, domain layer, and presentation layer. Design optimization patterns that spans layers will be elaborated first, then a bottom up approach starting with the data access layer, followed by the domain layer, and finally the presentation layer. We will then continue on with findings from different enterprise applications. After which, we will summarize the current state in design optimizations of enterprise applications based on the findings.

2. Crossing Layers

In this section design patterns will be presented that are considered applicable to more than one layer of an enterprise application. Most often these patterns are used in a set of components belonging to a layer called System Frameworks; however, this may not always be the case, thus we will categorize these design patterns as crossing layers.

2.1. State

In terms of state, there is the application state and user state. The application state contains required information for all users of the system in order to function correctly and efficiently. The user state records the conversational information between the user and the application. This conversational state may be required to maintain conversational history and user information. User state is commonly referred to as session state. State can be costly as it may incur communication overhead, resource utilization, and added maintenance.

Yoder and Barcalow [26] discuss a Session pattern to contain user information that can be shared throughout the system. This pattern provides an object to encapsulate all conversational information required for a client. There is little discussion on performance implications for the Session pattern; however, [26] and [13] mentions the consequence of reducing the number of object by utilization of the Session pattern.

Fowler [10] suggests several patterns to address user conversational state, which includes the Client Session State, Server Session State, and Database Session State pattern.

Using the Client Session State pattern would mean storage of the conversational state on the client's machine and the state is transferred back to the server on each request. By storing and maintaining the state on the client side frees server resource to improve scalability. However, this can degenerate when the state becomes large and the cost of transferring the state on each request may become a performance hindrance [10].

On the other hand, the Server Session State pattern enables the state to be maintained by the server. This requires resource from the server to hold this information. If memory resource is required for containing this state, it may become problematic when memory resource is insufficient [10].

The user conversational state can also be stored in the database by applying the Database Session State pattern. The performance aspect of this approach is highly dependant on the database itself [10].

Fowler [10] suggests possible performance issues and trade offs regarding different patterns in providing user conversational state; however, does not demonstrate concrete examples with execution time tradeoffs. Session states can be misused, Dudney et al. [8] provides some guidance in identifying possible problematic scenarios. In any case reducing the usage of conversational state in its resource consumption and maintenance, can improve the scalability of an enterprise application [1].

As for application state patterns this can be addressed in terms of improving caching, distribution and in other layers of the application, namely the data access and domain layers. These will be discussed in later sections.

Gamma et al. [11] provides a general state pattern, the Memento pattern, which is a pattern that can degrade the performance of an application. Its purpose is to provide serialization and de-serialization of an object's state externally without having to violate encapsulation. This is accomplished by having an object, the memento object, to hold the state of another object, the originator. The originator would contain a method to set the memento appropriately, effectively copying its state into the memento object. The process of copying the originators state may incur high costs if the data is large with highly frequent copy operations.

Another pattern discussed by Gamma et al. [11], is the State pattern. This pattern provides different behaviors depending on the state of an object. The State pattern itself does not seem to suggest any performance issues or improvements; however, the authors of [11] does highlight possible performance issue with implementation choices of the pattern. The implementation choice between a polymorphic call to a table lookup can have a performance impact.

One aspect of providing state functionality is the maintenance of its consistency in concurrent scenarios. This will be explored later on in the Concurrency section.

Few authors have ventured into prescribing patterns specifically addressing state; however, that is not to say other suggested patterns cannot be used in providing state functionality. Even if patterns were mentioned in supporting state, little discussion surrounds their effect on performance let alone attributing sample implementations, and their execution times.

2.2. Caching

There can be performance impact on repeated operations of object creations, computations, and communications in retrieving data, which are previously executed. To address such problems, an optimization technique, Caching, is suggested [4,6,13,23,24,25]. This technique involves preserving an object and its data for later use without having to re-fetch its required data. It also reduces the need to re-setup an object's required state and re-compute otherwise costly computations.

Gamma et al. [11] discusses several patterns, Singleton, Flyweight, and Proxy, to solve the problems surrounding object creation.

The Singleton pattern provides an access point to a single instance of a class [11]. This ensures the required resources for the singleton is retrieved and initialized only once. Gamma et al. [11] did not provide performance implications surrounding the application of this pattern. Grand [12] discusses the implementation of the Singleton pattern using lazy instantiation, whereby the singleton instance is instantiated only when needed. This defers the utilization of resources only when needed. Grand [12] also mentions it functions analogous to a cache with a single object.

The Flyweight pattern provides sharing of objects in different contexts [11]. A flyweight object is an object that can be shared simultaneously in different contexts. The flyweight object itself can contain data that is context independent. It may also require operating on data that is context specific. By utilizing the Flyweight pattern it reduces the number of objects created, which

frees server resources for other functions; however, it can be costly in providing context specific data it requires [11].

The Proxy pattern is a surrogate in providing access to another object [11]. Two versions of the proxy may provide performance improvements, the virtual proxy and smart reference. A virtual proxy creates an object on demand, thus only acquiring resources when needed. The smart reference proxy provides maintenance on keeping track the references of object, creating and releasing its resources when necessary. Another version of the Proxy pattern is the remote proxy. Remote proxy provides a local representational access to an object in a different process, by hiding the details to access the remote object. Fowler [10] discusses the performance implication of not realizing the usage of a remote proxy, because of its seamless nature in having the exact signature of the remote object itself. Gamma et al. [11] also mentions another optimization it provides, copy-on-write. The copy-on-write variation of the proxy, aid in reducing the expense of copying objects with lots of data continuously. The copying expense is only incurred on operations, which modifies the subject of the proxy.

Gamma et al. [11] also provides other patterns relating to object creation, which includes Abstract Factory, Factory Method, and Prototype, but did not mention performance implications surrounding these patterns.

Grand [12] discusses the pattern, Object Pool, to manage the creation and access to a limited number of objects that can be reused. By applying this pattern it reduces the creation of possibly expensive objects to a limited set. Objects are retrieved from a pool when required and released back into the pool when done. Multiple retrievals by different clients can be done up to the limit of the pool, at which time the client would have to wait for an object to be released back into the pool in order to use the pooled object. Grand [12] provides an example in pooling connections to the database, which is also suggested by [8] and [1] to increase performance and scalability.

Grand [12] also discusses the Cache Management pattern in providing fast access to objects without having to re-fetch from possibly expensive sources. This pattern provides storage for already fetched objects and a mechanism to retrieve it. If an object does not exist in the cache then the required fetch and setup of the object is executed, after which the newly created object would be added to the cache. This limits the expense of constructing the object only on first retrieval of the objects not existing in the cache. Object non-existence in the cache can be due to not being added into the cache or it has been expired from the cache. The method to determine the expiration of a cache item is when the maximum limit of objects in the cache is reached, and an algorithm to determine if it is to be pulled out of the cache. The author discusses a least recently used algorithm as a means to determine which object to discard from the cache. Developers can then tune the limit of the cache to achieve the optimal cache hit rate for the application.

Grand [13] discusses the following patterns in regards to object creation, which includes the Shared Object, and Ephemeral Cache Item pattern.

The Shared Object pattern provides a central access to limited resources that can be shared between clients [13]. The author presents the example of telephone lines as the shared resource and the clients are point of sale cash registers requiring credit card transactions. There would be a manager managing the set of telephone lines to the credit card clearinghouse. Each cash register would then request access to a telephone line through the manager. This centralized access to shared resource can produce a bottleneck. On the other hand, the set of shared objects can be increased to meet the client's demands, but only within constraints such as the system resources. The Shared Object pattern provides a more manageable access to the limited set of resources.

The Ephemeral Cache Item pattern is a refinement of the Cache Management pattern in [12]. It provides the same features of Cache Management except for the limit of the cache and expiration of. The Ephemeral Cache Item pattern expires the cache item based on the lifetime it exists in the cache not by the number of objects in the cache. A scheduler would check the items in the cache if its lifetime in the cache has expired and would discard expired objects from the cache. This provides a cache with relatively recent object data [13].

These patterns discussed by Grand [12] and [13] address the problem of shared objects and to reduce the amount of object creation. The author provides example implementation and usage of these patterns; however, does not provide different example applications utilizing the patterns, and execution time measurements.

Fowler [10] provides several patterns surrounding object creation; these include Identity Map, Lazy Load, and Registry.

The Identity Map pattern ensures that an object is loaded from the database only once by maintaining a map containing loaded objects that can be retrieved by its identity [10]. It acts as a cache for database reads, which eliminates the need to communicate with the database on every access to get the object. It also ensures there is only one instance of the object's data in the system, if the path to retrieve the object always checks the identity map first.

The Lazy Load pattern provides an object that may not contain all its data, but can load its necessary data when needed [10]. Fowler [10] suggests it can be implemented as lazy initialization [2], virtual proxy [11],

value holder, or a ghost. The lazy initialization approach simply checks if a field is loaded when accessed, if not it retrieves the data. The virtual proxy approach provides an indirection to check if an object is loaded, if not it will load the object. The client to the virtual proxy is transparent to the loading process and treats the virtual proxy as the real object. The value holder approach manages the loading of the underlying data by wrapping it. The value holder will only retrieve the underlying data from the database on first access. The ghost approach is having an object within its partial state. It will load its required data on first access either as groups or its entirety.

The Registry pattern provides a global object to access common objects and services [10]. A Registry can centralize control of common services and objects. This can be quite useful in loading immutable objects and can be loaded on start up of the application, such as all the states in the United States. Fowler [10] provides little discussion around the performance implications of the Registry pattern.

Fowler [10] suggests several patterns in reducing object creation and deferring loading data when needed. The discussions provide performance implications, and sample applications utilizing the patterns, but lack their variations and measured effect.

Alur, Crupi, and Malks [1] discuss these patterns in relation to object creation, which includes the Service Locator, and Value List Handler pattern.

The Service Locator pattern provides a uniform access to locate services and objects [1]. This hides the details from the client on the actual lookup and creation of required objects. The Service Locator can improve network performance as it can aggregate the required network calls in order to create the object. It also improves the client performance by utilization of a cache to eliminate the expense in creating the required services and objects.

The Value List Handler pattern provides a structure to search, cache, and iterate through the results [1]. This pattern was previously known as the Fast Lane Reader. It provides a mechanism to execute queries to retrieve a set of results, bypassing business object lookups and retrieval operations, maintains this collection as a cache and returns the required subset of the result to the client. This pattern provides a facility to perform search operations otherwise would be costly using the business object finders. This improves network performance as only the subset of the result is returned to the client on demand. By not using the transaction managed path to retrieve the business objects, it can defer and possibly eliminate unnecessary costs in transaction management. The performance issue with this pattern is in creating large sets of Transfer Objects [1]. To alleviate the performance impact the authors suggest limiting the number of results returned from the database or other strategies for the Data Access Object pattern, which includes the Cached RowSet and RowSet Wrapper List [1].

Alur, Crupi, and Malks [1] discusses different strategies for implementing the suggested patterns and their performance impact; however, did not elaborate on the effects and trade offs of the different approaches.

As well as the Service Locator and the Value List Handler pattern there were other patterns and their implementation strategies that can have performance impact [1], which will be elaborated in later sections.

Limiting the creation of object expense by reusing objects can improve performance as shown by Halter and Munroe [14] and Bulka [4]. Halter and Munroe [14] and Bulka [4] discussed and provided measurements in the cost of object creation, and its performance impact. The authors of the mentioned patterns in this section omitted the quantitative analysis of the patterns and its performance impact. Even though it is shown by others [4,14] it can improve performance, on the other hand, in a concurrent environment, sharing of objects and its resource can cause bottlenecks as well [4,14,17,23].

2.3. Concurrency

Concurrency is an important aspect of any enterprise application. It allows multiple users and operations to be serviced at the same time. Without such notion it would become a queue servicing one request at a time. Amdahl's law states the sequential portion of a computation will bind the performance gain of the parallel portion [16]. In other words, the sequential processing portion of an application can limit the scalability of an application [4]. To demonstrate this, [4] describes the scenario of matrix multiplication in three phases, which includes Initialization, Multiplication, and Presentation. The only phase that can perform in parallel is the Multiplication phase. In theory, given unlimited number of processors processing the Multiplication phase in parallel, the time spend can be reduced to negligible. However, the Initialization and Presentation phase remains constant, as it can only be processed sequentially. Thus no matter how much parallelism is utilized by the application, the operations will converge to the sequential portions of the application. Essentially, patterns in this section provide options to aid in enabling the parallel portions more performant.

Grand [12] discusses a number of patterns in relation to concurrency and performance; this includes the Single Threaded Execution, Read/Write Lock, Double-Buffering, and Asynchronous Processing pattern.

The Single Threaded Execution pattern provides the mean to protect concurrently accessed resources to ensure its consistency [12]; also known as Critical

Section. The possible performance impact to this approach is guarding operations that do not require such protection. This is costly in that it incurs the protection overhead of critical sectioning the operation, and enforcing a serial execution of the operation that can otherwise be executed simultaneously [12].

The Read/Write Lock pattern provides an external class to manage the concurrent read and exclusive writes operations on the data of a class [12]. This pattern is also mentioned in [17] as a lock utility. In utilizing the Read/Write Lock pattern instead of Single Threaded Execution pattern can result in better throughput; however, if there are few calls to read the object data then will result in a lower throughput than the Single Threaded Execution pattern.

The Double-Buffering pattern is a specialization of the Producer-Consumer pattern in that it produces objects that is put into multiple buffers in which consumers can consume from; also known as Exchange Buffering [12]. The producers, produces asynchronously, the required objects to be put into multiple buffers for consumers to consume. The multiple buffers allow the producer to continuously pre-fetch data to satisfy the consumer's needs. This pattern can reduce the situation where the application has to wait for data to be fetched to continue operation; however, it does incur more memory resource being utilized.

The Asynchronous Processing pattern provides a queue to perform operations asynchronously [12]. This enables a client of a request to continue processing without having to wait for the processing of the request. This also reduces the resources necessary to operate requests asynchronously by not having to allocate a thread for each request. This pattern is analogous to usage of J2EE's Message Beans for concurrent processing [19], which can also have performance implications as discussed by [8].

Grand [12] provides many concurrency patterns to explore and their possible performance implications. The lacking aspect is in the different examples of applying them, and the measurements of their performance effects.

Grand [13] discusses these patterns in relation to concurrency and performance, which includes the Optimistic Concurrency, and Thread Pool pattern.

The Optimistic Concurrency pattern allows modification of transaction managed data by modifying local copies and acquiring the lock to update the shared data [13]. By modifying the local copy of the data and only abort the transaction when failure to commit the data allows better throughput than a pessimistic approach. This is not always the case, in situations where concurrent transactions modify the same piece of data, in an optimistic scenario the whole transaction will need to be restarted if aborted whereas the pessimistic approach would wait until it can perform the operation. Fowler [10] also discusses this pattern.

The Thread Pool pattern provides a managed set of threads to service operations [13]. This pattern reduces the time to create new threads by reusing created threads. It also controls the number of possible executing threads to reduce to much system resources being utilized. On the other hand, if the demand for the threads is exceeding the number of threads in the pool, the thread pool may lower the throughput of the application.

Grand [13] provides many examples and varying implementation of the concurrency patterns, the one missing aspect is in measuring the different variations and its performance impact.

Fowler [10] discusses the pattern, Coarse-Grained Lock, to address concurrency and performance problem.

The Coarse-Grained Lock pattern provides a lock on a set of objects [10]. This reduces the cost in having a separate lock for operations that operate on the same set of shared objects frequently. However, if the lock is too coarse-grained then the possibility of having other requests wait can be higher.

Fowler [10] also provides other concurrency patterns, which includes Implicit Lock, Optimistic Offline Lock, and Pessimistic Offline Lock, but little surrounds the discussion of their performance impacts.

Overall, the choice in providing concurrent access can have many performance implications. If not done correctly deadlocks may occur, which can ultimately halt the application [10,14,17,23]; however, if managed correctly it can provide consistent data and improved throughput of the application [12,13,17].

2.4. Distribution

Distribution is the placement of objects in different process space. This placement can be within the same machine or through network communication to another machine. A purpose for this is to utilize another processing node's resource [9].

Brown, Eskelin, and Pryce [3] presented these patterns for distributed component design; this includes the Replicated Object, Distributed Façade, Object Factory, and Distributed Command pattern.

The Replicated Object pattern provides an object containing a replicate of the original business entity instead of having to make a remote call [3]. This eliminates the need for a Proxy [11] to forward the call to a remote location, instead make the business entity serializable and transfer it to the client to retrieve its required data. The authors note there is the issue of synchronization of modified data in the replicated object and where the data centrally resides. This may require the sending back of the modified replicated object back

to the server, if this occurs often enough it may cause performance issues.

The Distributed Façade pattern provides a remote interface to access highly affiliated objects without having to expose all the objects to the client [3]. Fowler calls this pattern, Remote Façade [10]. This reduces the network communication and dependence between the clients with the business objects. It provides a coarse-grain approach to access the business objects instead of the finer-grain approach.

A complimenting pattern to the Replicated Object is the Object Factory. It creates the Replicated Object by making the necessary operations to the business entities thus reducing the coupling between the Distributed Façade and the business entities [3]. The authors did not discuss any performance implications with this pattern.

The Distributed Command pattern compliments the Replicated Object as well. Instead of having to transfer the modified Replicated Object back to the server, commands are created for what has been updated [3]. In the case the command object only containing "deltas" to objects, it can reduce the amount of network traffic and server processing to determine what has been modified [3]. If all the data of an object has been modified then the command object may not provide much performance increase.

Brown, Eskelin and Pryce [3] discussed some of the early patterns on distributed components. Also discussed their performance implications. Examples were given; however not in variant forms of their possible implementations and performance impact.

Grand [13] discusses the following patterns in relation to distribution and performance; they include the Object Replication, Mobile Agent, and Heavyweight/Lightweight pattern.

The Object Replication pattern provides replicas of an object while maintaining the illusion that there is only one single object to different computing nodes [13]. This pattern can serve to improve the throughput and availability of an application. By providing a local copy of the object to a computing node, the access to the object does not incur network communication overhead to communicate with the object. The performance impact comes in synchronizing the objects between the computing nodes to provide consistency in the event the object state is modified.

The Mobile Agent pattern enables an object to be located where its required data is [13]. By collocating an object with its required resources it conserves network bandwidth thereby reducing the communication overhead between the object and its required resources. A downside to this is the cost in maintaining the communication with the mobile agent if it requires centralized management.

The Heavyweight/Lightweight pattern provides the mechanism to retrieve required attributes of an object on demand [13]. This pattern is similar to Lazy Initialization [2], and variations of Lazy Load [10]. The Heavyweight/Lightweight pattern encapsulates the required data in a data object and is only retrieved when requested. This reduces the time to retrieve data and conserves memory resources. Also, since the data is copied to the client on a batch set of data it is less costly than having the client retrieving the data individually and incurring the network overhead. On the other hand, [13] suggests this can have a negative effect if the data is retrieved frequently, which can cause more time spent in downloading the required data.

The Registry pattern, different from the one mentioned by Fowler [10], provides a name lookup mechanism to access shared services [13]. The registry would provide a proxy, which encapsulates the access details to the service. One hindrance to this lookup service is the indirect nature as it provides an external service in order to refer to another external service [13].

Grand [13] explored in depth the variations of these patterns, which can aid in distributed scenarios and their different variations in implementations.

Alur, Crupi, and Malks, [1] provides a pattern, the Transfer Object pattern, to reduce communication overhead. Fowler [10] calls this pattern, the Data Transfer Object pattern. The Transfer Object pattern provides performance improvements primarily by reducing the communications between processes [1,10,18,20]. Instead of having to incur additional inter-process communication overhead for information required, a transfer object is created for a more coarse grain approach. The transfer object contains the information otherwise requiring additional communication to retrieve. This type of coarse grain communication has been recognized to improve the performance as shown by Halter and Munroe [14] in an example to demonstrate granularity and performance impact. Different implementation of this pattern can also have negative performance impact. This can occur when the design of the transfer object contains insufficient data and requires additional fetches for information. Such scenario may indicate the transfer object is too fine grain. On the other hand if the transfer object is too coarse grained, that is to say the transfer object was designed to transfer unnecessary data, then the overhead incurred may diminish any performance improvements in reducing the number of communication trips.

A complimenting pattern to the Transfer Object is the Transfer Object Assembler [1]. Its purpose is to package a Transfer object by bringing together required information from different components within a process space to be transferred and used by another process, thereby reducing the inter-process communication

overhead. Without the Transfer Object Assembler, there may be finer grain value objects, which requesting processes will have to fetch from different components in another process space.

The Session Façade pattern provides an aggregated point of access on a number of business components to expose these business components and available services to remote clients [1]. This pattern is similar to the Distributed Façade discussed in [3]. It enables a more coarse-grained access to services provided by the business components. This reduces the network access from the remote client to the business components. Another effect of this pattern is by having a Session Façade aggregating the results of the business components is faster than having the client making the requests. This is often due to having the Session Façade collocated with the business components, which results in local communication instead of inter-process communication with the business components.

The Service Activator pattern provides a way to invoke services asynchronously [1]. The pattern utilizes a listener to receive asynchronous messages and delegates the appropriate service to handle the message. Upon completion of handling the message request the Service Activator would then send a response back to the client indicating its completion and results of the operation. On top of which [1] discusses a Service Activator Aggregator strategy. This strategy enables a business service to be broken down into subtasks and invoked asynchronously. This enables the task to be performed by subtasks in parallel. Upon completion of the subtasks, their results are aggregated together to produce the main result.

Alur, Crupi, and Malks [1] discuss in detail the different approaches in providing distributed computing. The patterns target the J2EE platform; however, as a design pattern its basic structure can be implemented by different platforms supporting object-oriented designs.

Trowbridge et al. [24] discussed common enterprise application patterns and its implementation using the Microsoft .Net technology. This provides more sample implementation and variation in approaches to apply the patterns discussed by others previously in this paper. In particular it provides some infrastructural patterns related to distribution and performance. Specifically, they are the Server Clustering, Load-Balanced Cluster, and Failover-Cluster.

3. Data Access Layer

The data access layer provides the necessary transformation from the persisted data types to an object-oriented model. This transformation involves serialization and deserialization of objects.

Fowler [10] discusses many patterns related to data access and transforming to an object-oriented model. But little is focused on the performance impact on utilization of these patterns.

Grand [13] discusses these patterns in relation to the data access layer and performance implications, which includes the Persistence Layer, CRUD, isDirty, and Lazy Retrieval pattern.

The Persistence Layer pattern provides an abstraction to the details of persisting objects [13]. If the persistence layer is not tuned for an application the consequence may be difficult in optimizing access to the database. One implementation of the persistence layer is to cache retrieved data and the transformation of an object. This reduces the trips to the database to retrieve the data and reconstruct the object.

The CRUD pattern is the acronym for Create, Retrieve, Update and Delete. It organizes the persistence functions into Create, Retrieve, Update, and Delete operations in the persistence layer [13]. By composing complex transactions from the basic CRUD operations it may impose performance penalties. This can result in retrieving unnecessary amounts of data and performing multiple round trips to the database in order to complete the transaction.

The isDirty pattern provides the facility to avoid unnecessary expense of updating objects to the database that does not require it [13]. For objects that require persistence, a status is added to indicate if any of its fields has to be modified. Upon submitting the object for update to the database, the persistence layer would check if the object modification status has been set. This prevents unnecessary update to the database when no modifications have been made. As possible implementation option, Grand discusses providing more than one status for large or complex objects. This provides a more specialized update instead of blindly sending the entire object and its data to the database for update [13].

The Lazy Retrieval pattern retrieves only a subset of the essential data and load the rest of the data when required [13]. This pattern is similar to the Lazy Load pattern discussed by Fowler [10]. It avoids the overhead of the time and memory resource in retrieving data that may not be used.

Grand [13] explores these data access patterns, which was discussed by [27] and their relationship with each other in depth. The author provides plenty of examples and implementation approaches, especially surrounding the Persistence Layer pattern.

Keller [15] discusses these patterns in relation to data access and performance, which includes the Cluster Read, Bundled Write, Store for Forward, and Flat File Write pattern. The author also mentions optimization

using table structural patterns; these will not be highlighted in this paper.

The Cluster Read pattern groups a series of database queries into a stored procedure to avoid multiple database calls [15]. It provides a single access point for operations requiring many database queries in order to retrieve its required data. Keller states it can expedite complex scenarios by up to 90% by grouping the queries as a stored procedure in the database and return the requested data in one trip [15].

The Bundled Write pattern packages required updates to the database in one trip as a bundled query [15]. This reduces the number of trips to the database in cases where many objects are modified and requires update to the database. It forwards the generated queries to update the individual objects into a Bundle Manager, which would then package them together to be sent to the database as one request.

The Store for Forward pattern provides a mechanism to store updates in a buffer for asynchronous processing to the database, while allowing the client to continue with their tasks [15]. Updates are pushed into a buffer when the client requests an update, another thread would act on this buffer and utilize a Bundle Manager to bundle multiple entries in the buffer as one request to the database. This provides an asynchronous processing of update requests by the client.

The Flat File Write pattern provides a facility for batch operations to write the required inserts into a file, after which, a database would load the data from the file [15]. The main assumption of this pattern is that a database would operate on a file faster than sending individual requests to the database.

Keller [15] discusses an overview of some of the patterns involved in the data access layer and their performance implications. Keller did not go into much detail of the different variation and implementation strategies for the patterns, and provides no code sample for the patterns.

The data access layer is a critical path in any enterprise application. It handles an expensive channel to other resources, namely a database. Most often this layer cannot avoid the overhead in inter-process communication as well as network communication. Design choices in this layer can have varying performance consequences as discussed by the different authors and should not be taken lightly.

4. Domain Layer

The domain layer provides the domain specific logic in serving the business requirements. This logic includes the entities, rules, and constraints pertaining to the domain of the application. It communicates with Data Access layer in order to persist and retrieve relevant data. It is the point of access for the Presentation layer to request for business services.

Alur, Crupi, and Malks [1] discuss some patterns in relation to the Domain layer and their performance implications, namely, the Business Delegate, and Composite Entity.

The Business Delegate pattern provides the abstraction to lookup and access to business services for clients [1]. This abstraction performs the required lookup for the service using a Service Locator [1], and delegates any requests to the actual business service. An implementation variation discussed is to cache the reference to the business service on first lookup, subsequent access would not need to perform the overhead of finding the service.

The Composite Entity pattern comprises multiple business objects as a coarse-grained entity object [1]. By aggregating multiple persisted business objects it can provide a better representation of entities and compositional relationships for the domain of the application. Some performance related implementation strategies would be utilization of the Lazy Load [10] pattern, and the isDirty pattern [13]. These aggregations of business objects can improve performance in that it all communicates between the aggregated objects within the Composite entity without having to incur the cost of communication to different entities. Also, this produces fewer entities for the client to access and lookup, which decreases the network overhead. If used in conjunction with the Transfer Object pattern [1], it can produce coarse-grain requests, which also reduces the number of network communications.

Fowler [10] describes the Transaction Script pattern as a mechanism to organize business logic into procedures catered to different requests from the Presentation layer [10]. It organizes a request from the presentation into a single procedure, which can make calls directly to the database or through a wrapper. This can reduce the number of business entity interaction and communicate directly with the database.

Fowler [10] also discusses other patterns relevant to the domain layer; however, there were no discussions as to their performance implications.

There seems to be few discussions surrounding design patterns to improve the performance of domain layers. This could be due to it utilizing patterns in the Crossing layers section to optimize the execution of the domain logic. Also patterns pertaining to the domain layer may be more catered to specific domains. In later sections of this paper where patterns for optimization specific application will be discussed, some of which could be applicable in this layer as well.

5. Presentation Layer

The presentation layer provides the interface to the end user. This allows the user to interact and perform functions provided by the applications.

Alur, Crupi, and Malks discuss the Intercepting Filter as a pattern to provide a facility to manipulate a request before and after the processing of a request [1]. This pattern provides the pre and post process filters required for a request. As each filter is meant to be pluggable and independent the cost surrounding the sharing of information between them can be costly.

A pattern discussed by Alur, Crupi, and Malks [1] that pertains to the presentation layer is the Value List Handler pattern. It provides a caching mechanism to handle paging of large results set, which otherwise would incur repeating calls to the database.

There were many patterns discussed by Alur, Crupi, and Malks [1] but little discussions related them to performance impact or implications.

Trowbridge et al. [24] suggests using the Page Cache pattern to optimize performance on web pages. The Page Cache pattern caches dynamic web pages that can change infrequently. It includes the following performance benefits [24]:

- using this facility conserves CPU processing resource;
- eliminates unnecessary processing, and conserves client connections by serving cached pages and freeing resources to process other dynamic pages;
- enables concurrent access to the read-only versions of cached pages.

There are many patterns which are applicable to the presentation layer, but few discussions revolves are performance centric ones. Having a more responsive interacting layer to the user would seem like a prime candidate for performance optimization; however, this could be due to having intense processing logic elsewhere in the application, and any optimization on the presentation will have minimal impact in the overall application performance.

6. Applications

In this section two applications and their performed optimization patterns will be discussed. These applications are the IBM SanFrancisco, and the RUBiS auction site.

6.1. IBM SanFrancisco

The IBM SanFrancisco is a project to provide flexible application frameworks to service different implementations for specific domains [6].

One of the patterns used is the Cached Aggregate [6]. This pattern provides fast access to retrieve the result of a calculation that is dependant on numerous objects. When the individual elements of a calculation are modified it notifies the cached aggregate, this approach allows the total calculation to be spread out instead of recalculating the individual elements each time the end result is required. In the case where there is little data, caching the result may end up more costly then traversing through the objects to derive the result, as it may take more time to maintain the cached aggregate. A pattern discussed in [22], Cache Balance, suggests an optimal range where the Cache Balance would be performant.

The utilization of the Command [11] pattern can improve performance [22]. Command objects encapsulate the information, and actions to execute a command. The ability to forward an entire command object to the required location reduces the number of remote calls, and all required information to execute the operation can be accessed locally with information provided by the command object.

The Extensible Item pattern provides a mechanism to dynamically change an object's behavior, and data [6]. By providing such flexibility this suffers the cost of performance. The invocation of a method in an Extensible Item is on average 20 times slower then calling a method directly [22].

There are other patterns discussed along with their performance impact; however, they seem heavily tied to the SanFrancisco framework and usage.

6.2. RUBiS

RUBiS (Rice University Bidding System) is a sample auction site with different versions developed by researchers at Rice University, France [7]. It is used to illustrate the different implementation choices and their effects on a web application.

The concentration of this particular research [7] is not pattern; however, the Session Façade pattern was applied and measured against other implementations. This provides the possible effect on the selection of using the Session Façade design pattern. The result of the research showed that the implementation using the Session Façade design pattern is slow compared to other implementations. The experiment indicated even though Session Façade was implemented to provide a coarse grain access to the business components, the Session Façades are still going through all the communication layers to access the business components as if they are remote objects. Thus even though the Session Façade pattern indicated by others [1,3] as providing a coarse grained access to remote objects, which can reduce network communication, can be less performant depending on how it is implemented.

7. Summary

There are many discussions in the area of software design pattern. They provide example implementations and different strategies in implementing the pattern. They also provided discussions on issues pertaining to their usage and applicability. Some provided discussions on the performance implications of implementation variants of the patterns; however, many do not. The discussion on design patterns affecting performance is provided in all the three layers of an information system [10,21]. In particular, [1], [10] and [13] discussed patterns spanning all the layers. The layer with the least amount of discussion on performance impact of design patterns is the Presentation Layer. This could provide opportunity to mine new patterns or explore the possible performance implications of existing presentation specific design patterns. In any event, this finding demonstrates there is little evidence quantifying the performance impact, if any, of applying the patterns and their variations. Much of the measurements provided are by micro-level performance tuning and optimization [4,14,23].

The design choices developers make, involves many aspects in the decision, understanding their possible performance implications is one of them.

8. References

[1] D. Alur, J. Crupi, and D. Malks, Core J2EE Patterns: Best Practices and Design Strategies, 2nd Edition, Prentice Hall PTR, Upper Saddle River, NJ., 2003.

[2] K. Beck, *Smalltalk Best Practice Patterns*, Prentice Hall, Upper Saddle River, NJ., 1997.

[3] K. Brown, P. Eskelin, and N. Pryce, "A small pattern language for Distributed Component Design", Pattern Languages of Programs (PLoP'99) Conference, Monticello, Il., 1999.

[4] D. Bulka, *Java Performance and Scalability, Volume 1: Server-Side Progamming Techniques*, Addison-Wesley, Upper Saddle River, NJ., 2000.

[5] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal, *Pattern-Oriented Software Architecture: A System of Patterns*, John Wiley & Sons Ltd., West Sussex, England, 1996.

[6] J. Carey, B. Carlson, and T. Graser, *SanFrancisco Design Patterns: Blueprints for Business Software*, Addison-Wesley, Reading, MA., 2000.

[7] E. Cecchet, J. Marguerite, and Zwaenepoel, "Performance and Scalability of EJB Applications", Proceedings of the 17th ACM conference on Object-oriented programming, systems, languages, and applications (OOPSLA'02), Seattle, WA., November 2002, pp. 246-261.

[8] B. Dudney, S. Asbury, J.K. Krozak, and K. Wittkopf, *J2EE AntiPatterns*, Wiley Publishing, Inc., Indianapolis, IN., 2003.

[9] J. Farley, *Java Distributed Computing*, O'Reilly & Associates, Inc., Sebastopol, CA., 1998.

[10] M. Fowler, *Patterns of Enterprise Application Architecture*. Pearson Education Ltd., Addison-Wesley, Boston, MA., 2003.

[11] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, Reading, MA., 1994.

[12] M. Grand, *Patterns in Java, Volume 1, 2nd Edition*, Wiley Publishing, Inc., Indianapolis, IN., 2002.

[13] M. Grand, *Patterns in Java, Volume 3: Java Enterprise Design Patterns*, John Wiley & Sons, Inc., New York, NY., 2002.

[14] S.L. Halter, and S.J. Munroe, *Enterprise Java Performance*, Prentice Hall PTR, Upper Saddle River, NJ., 2001.

[15] W. Keller, "Object/Relational Access Layers: A Roadmap, Missing Links and More Patterns", In Proceedings of the 3rd European Conference on Pattern Languages of Programming and Computing (EuroPLoP'98), Bad Irsee, Germany, July 1998.

[16] S. Krishnaprasad, "Uses and Abuses of Amdahl's Law", *The Journal of Computing in Small Colleges*, Volume 17, Issue 2, December 2001, pp. 288-293.

[17] D. Lea, *Concurrent Programming in Java, 2nd Edition: Design Principles and Patterns*, Addison-Wesley, Reading, MA., 2000.

[18] N.N. "Best practices to improve performance using Patterns in J2EE", PreciseJava, 2003. Available at: http://www.precisejava.com/javaperf/j2ee/Patterns.htm.

[19] A. Poddar, "Add concurrent processing with message-driven beans", JavaWorld, July 2003. Available at: http://www.javaworld.com/javaworld/jw-07-2003/jw-0718-mdb.html.

[20] V. Ramachandran, "Design Patterns for Optimizing the Performance of J2EE Applications", *Java Developer's Journal*, Vol. 6, Issue 9, September, 2001. Available at: http://www.sys-con.com/java/article.cfm?id=1135.

[21] K. Renzel, and W. Keller, "Three Layer Architecture" in M. Broy, E. Denert, K. Renzel, M. Schmidt (Eds.), *Software Architectures and Design Patterns in Business*

Applications, Technical Report TUM-I9746, Technische Universität München, 1997.

[22] G. Schimunek, T. Fanto, M.C. Filorizzo, A. Rauch, R. Rolandsson, M. Sant, and J. Van der Sypt, "IBM SanFrancisco Performance Tips and Techniques", International Business Machines Corporation Redbooks, February 1999.

[23] J. Shirazi, *Java Performance Tuning*, O'Reilly & Associates, Inc., Sebastopol, CA, 2000.

[24] D. Trowbridge, D. Mancini, D. Quick, G. Hohpe, J. Newkirk, and D. Lavigne, "Enterprise Solution Patterns Using Microsoft .Net, Version 2.0", Microsoft Corporation, 2003. Available at: http://msdn.microsoft.com/practices/type/Patterns/Enterprise/default.asp.

[25] S. Wilson, and J. Kesselman, *Java Platform Performance: Strategies and Tactics*, Addison-Wesley, Upper Saddle River, NJ., 2000.

[26] J.W. Yoder, and J. Barcalow, "Architectural Patterns for Enabling Application Security", Fourth Conference on Patterns Languages of Programs (PloP'97), Monticello, Il., September 1997.

[27] J.W. Yoder, R.E. Johnson, and Q.D. Wilson, "Connecting Business Objects to Relational Databases", In Proceedings of the 5th Conference on the Pattern Languages of Programs (PloP'98), Monticello, IL., August 1998.