The University of Melbourne
SWEN90007 Software Design and Architecture

Reflective report

Student Name	Student Number	Student Username
Ziqi Jia	693241	ziqij
Catherine Jiawen Song	965600	cjsong

Report

Our system used identity map and we believe this improved our system's performance significantly. Our identity map saved data objects that were loaded from the database every time it was accessed which meant that it prevented duplicate retrieval of the same object from the database. This improved our system's performance as it reduced the number of calls we made to the database which typically slowed down the overall system's latency. We were unable to use identity map for all accesses to database as some functions that required access to multiple objects that were related to particular user id or product id were difficult to achieve through identity maps. When using these functions such as viewing products available for purchase for registered users, we saw a significant decrease in our system's performance in terms of latency which suggested that our identity map had significant impact on the overall system's performance.

Unit of work is also a pattern that affected the system performance. Unit of work takes notes of all the changes to objects within a database and commits all the changes with caller registration. We believe this pattern may have had a positive impact on the system performance if our system was not as simplistic as it's likely each commit only committed insignificant changes to the overall system that would have potentially slowed down the system as a result. For example, when we created an order, we registered the object as a new object and immediately called the commit function afterwards. This would mean that every small change we made whether its users adding products to their carts or admins modifying particular users would result in a call to the database and also modification of the identity map so that these changes would be reflected within the system. As a result of additional method invocations and identity map alterations this would have increased the latency of the system performance for each task.

Lazy load was a pattern which we did not use for our code base. Lazy load would only load objects when they're needed by the users which helps system performance and reduces wastage of system resources. We believe that lazy load alone may improve performance such as only loading objects when it is calling to find specific objects for particular users or products and thus increased the overall system performance in terms of response time. However together with identity map it unlikely would improve performance as both have similar functions and they may cause an overlap of storage of information within the system. This could introduce overhead that may negatively impact the system performance as data may be loaded twice, once through identity map and once through lazy load. Therefore, data mappers which called the database to find particular objects may have their performance slowed down and thus potentially increase the latency of the system overall.

DTO and remote façade were patterns which we wrote code for but did not implement them within our system. Remote façade provides course-grained interface over fine-grained objects while DTO holds all data required for a single call. Remote façade and DTO would result in less network calls which would mean that the response time for the overall system may increase significantly. However, as our systems were relatively simplistic there was no

need for the use of these patterns because they might have instead increased the overall complexity of the system and according to Bell's principle discussed later this could increase the burden on performance instead of improving performance.

Our system used a pessimistic offline lock instead of an optimistic offline lock. An optimistic lock assumes that the chances of conflict is low while a pessimistic lock assumes chances of conflict is high. We chose to use a pessimistic lock as our applications allows multiple administrators to edit details of users, themselves and products. An optimistic lock would likely have caused a burden of the system performance and increased the application latency of the system because multiple administrators who edits the same users would send requests to the server which means the server needs to process through all the request by checking the version numbers of each user relative to the current version number and make a decision to whether abort or commit their changes. Contrastingly using a pessimistic lock would mean our system ensures that only a single user is editing information and that their information will be saved as other users do not have the ability to co-edit the information and potentially cause their changes to be lost. This may cause inconvenience to users who would like to edit the same information but is still beneficial to the overall application latency as only a single request will be sent to the server regarding changes made by a single client instead of multiple clients sending multiple request and the server having to process each request.

While developing our prototype we did not consider principles to affect our prototype's performance. Our system operates in a typical synchronous request pattern whereby the response to the client's request is returned within single HTTP connections. As our enterprise application is a typical online shop for purchasing makeup, we believe that asynchronous requests or pipelining is highly unsuitable to improve the performance of our system. Asynchronous requests may be an unsuitable principle for our system to use as asynchronous requests require all the client request to be independent from each other. The majority of our requests are dependent on each other because for our system users user will select items to be put into the cart and if they view the cart they would expect the items to be within the cart and if they purchase items they would expect to see the items within the order history. The same also is also for the administrator users whereby if they edit the details of any user, products or orders they would expect to see the changes from the server immediately. The use of an asynchronous request is thus unsuitable as after the user request to add items to a cart, wish list or purchase it, their following request may likely be dependent on these previous request and with asynchronous requests a user may not expect to see these changes until at a later time. Additionally, the problem with asynchronous request is that responses from the server may arrive at varying orders which could be problematic in the sense that a user may see their response from the server of purchasing an item first before it was put into the cart. Therefore, asynchronous requests is unsuitable for our system because majority of objects in our system are interdependent thus meaning client request are also dependent on each other.

Bell's Principle states that simpler systems perform faster than complex systems. Our enterprise application is relatively complex as it is made up of 5 layers consisted of the presentation layer with the controller and view components, service layer with service components, domain logic layer with domain objects, data mapper layer which contains data mappers that accesses data stored in the data source layer. In additional we have concurrency

and security components, unit of work and identity maps, data transfer objects and remote facades. Despite the service layer providing ease of access to the domain object methods however this may be unnecessary due to the simplicity of the system of only editing user details or putting items into cart and this could instead result unnecessary method invocations from the service layer and thus slowed down the systems performance. Our unit of work also made insignificant impact to the overall system performance because each unit of work commitment only committed a single change instead of the multiple changes as desired by the unit of work to reduce calls to the database system. Thus, multiple calls to the data source layer through the unit of work pattern may have also reduced system performance rather than improved it. Additionally, our DTO and remote façade were completely obsolete as they were never used and only increased the complexity of our system without bringing any changes or benefits. If we were to follow Bell's Principle closely in the future then there may be certain features such as the service layer, unit of work, DTO, remote façade and potentially even concurrency components which we may exclude in the system which should help to increase the overall performance of the system.However, as the assignment required us to utilise multiple components such as unit of work and identity maps, presentation layers and so on to understand design patterns therefore Bell's principle would be unsuitable to be used in this scenario.

Caching is another principle which we did not consider for our system. We believe that caching within our system may not bring significant improvement to our performance due to the fact that our system is unlikely to be used by a significant number of users or need servers worldwide to allow users to access our system. Because majority of our system usage requires user login therefore most of the objects that has already been accessed are typically cached within identity maps specific to each user. Hence the addition of caching may be unnecessary and obsolete as our objects do not exhibit temporal locality or spatial locality and using caching in this scenario could even lead to a reduction in performance in terms of latency which should be avoided.

Reflection

Throughout the project the structure of our high-level architecture changed as we added more layers and patterns to make our system more maintainable and have better performance.

In the presentation layer, we changed the servlets to controller-view structure with the template view pattern and the front controller pattern. The addition of these two patterns makes the structure of the presentation layer clearer, makes the system more expandable, and increases the flexibility of system performance. This increased the maintainability of our system as future features which we would want to implement can be easily implemented by adding a view of the feature and corresponding command classes. Previously with servlets they had the difficulty of adding new features and also repetitive code structure within servlets such as error handling.

In order to ensure secure access to the data within our system, we add security patterns into the system, including authentication enforcer pattern, authorisation enforcer pattern, and secure pipe pattern. These security patterns are used to handle the authentication and authorization of users, and confidentiality of the users' requests. They improve the maintainability of the system as the code has been encapsulated into a single place to support reusability, and also decrease the likelihood of a successful attack from the opponents. With future development this means that the security component can be easily modified with other libraries without affecting the overall performance of the system while still ensuring that the overall system is secure.

The maintenance of session state in our system is supported by implementing the client session state into the system. The reason we choose the client session state is because it is easy to implement and supports stateless server objects well by giving them maximum clustering and failover resilience. However, there may be issues when dealing with large amount of data as the cost of where the data is stored and the time it takes to transfer all of it on each request may become very high. It also has a security issue as the data sent to client is vulnerable to being monitored and changed. Although the security issue can be covered by security patterns, there are still some risks that sensitive data can be attacked. Hence, the server session state can be used to avoid these issues in the future. Server session state stores all state in a replicated server side datastore, which can fully control the sessions without considering the issue of data size and hidden the implementation and user details. Due to the weaknesses of client session states within our system this could impact future development as the session state may needed to be modified if the traffic of our system increases if we were to commercialise the enterprise system as the system may be unable to handle significant data.

For service layer, domain logic layer and data mapper layer, there was not much changes over the duration of our project as we only added single classes to implement the feature B which were AdminService class, AdminUser class, and AdminMapper class. The use of model-view-controller pattern showed that our system was easily maintainable and that additional features could be added without impacting the overall system or having to change other classes to accommodate these new features as every layer is decoupled from each other. If we were to implement additional features such as guest users or other features within existing users our model-view-controller will allow us to extend our current architecture easily without significant risks of the overall system breaking or requiring significant modification.

In order to deal with concurrency, we implemented pessimistic offline lock pattern and implicit lock pattern into the system. Implicit lock is quite important as it aims to prevent forgetting the lock by putting the code for acquiring and releasing the lock in the mapper layer. Pessimistic locking locks a record for exclusive use until completion resulting in better integrity but increases the risks of deadlocks for the application. It also needs a direct connection to the database to complete the lock. If someone who reads a record and intending to update it placing an exclusive lock on the record, the user may be locked out for a long time if no one release the lock, slowing down the response of the entire system and causing frustration. For our enterprise application system, optimistic lock might be a better

choice as it is faster and it is not necessary to maintain a connection to the database for the session, which is suitable for the system have many layers of architectures. However, we also have to take consideration that due to the nature of our system an optimistic lock could also result in increased application latency in the scenario of multiple administrators editing the system. Therefore, future choices in concurrency patterns needs to be considered carefully by weighing the pros and cons of each locking pattern. From the perspective of architecture design, the implementation of lock also has some disadvantages. It increases the instability of the system as the best balance between lock overhead and lock contention is unique to the application and is sensitive to design, implementation, and even underlying system architecture changes. These balances can change over the life of an application and can lead to dramatic changes in updates. hence, implementation of lock patterns needs to be considered carefully in the future when design the architecture as it may impact a lot on maintainability of our system.

Overall, our system has achieved some improvements in overall performance, but the simplistic nature of the system meant that some of the patterns used increased the performance latency while principles of system performance that we did not consider could be beneficial for our system performance. Our system exhibits some maintainability and extensibility for future development through mode-view-controller patterns but features such as choices in concurrency locks and maintenance of client session state may need to be reviewed and potentially remodified for better maintenance and extensibility.