Peter Stefan
40489803

Software architecture

Table of Contents

Introduction

For this paper we are tasked with designing a distributed store management system for a company, with proposed business goals that need to be met. For this problem two possible architectures will be presented to demonstrate the advantages and disadvantages of each and why the chosen architecture prevailed. After the comparison, a more detailed implementation and description of the latter will be shown. At the end of this paper the chosen architectural design will be further scrutinised to evaluate its viability in contrast with realistic scenarios. For comparing the designs, we will use SAAM, and to evaluate the chosen one ATAM will be utilized.

1.Two possible architectures:

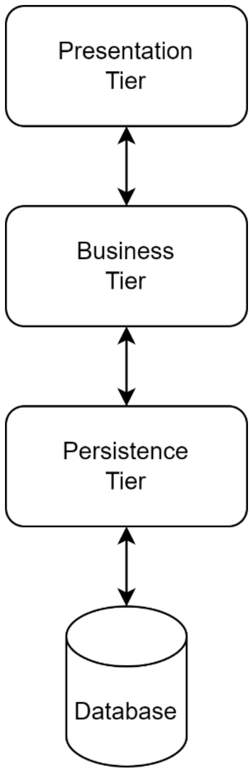
We will be comparing a client-server (2-tier), and a 3-tiered architecture design in this paper. Both are distributed systems and are very popular solutions for the kind of problem presented in this coursework. In a distributed system transparency is the most important principle, and these two styles will demonstrate this characteristic. For this program in particular 3-tiered architecture will be implemented as it is a clear winner between the two.

Client-server architectural style:

Client-server architectural style is a 2-tiered style where the components are the client, who requests services and the server who provides them. They are, however, independent in the way they work. They communicate with messages between each other called requests, this naming also shows the characteristic, that (compared to a main program subroutine architecture style for example) the components don't control each other. The message passing the components are linked by are the connectors. A server can provide service to multiple clients and in this case must keep a record of connections and current clients. However, this can cause a bottleneck, where if a problem arises with the server the recoverability of the system is poor, so the availability and continuity of this architecture could be a factor to take into consideration when designing the system. For example, if the DE-store server crashed all the information connected to the server via a database would be inaccessible and possibly lost if the problem was that severe. Another problem is that messages may be delayed or lost, so an implementation of message error handling can also be considered when designing such a system. There is also the drawback of handling messages which are unreliable and could be malicious.

Three-tiered heterogenous architecture style:

Heterogenous architecture involves the usage of multiple architecture styles in one system, this of course has the advantage of applying the best possible features of a style in the area where they fit best. In our implementation, we propose an architecture with three tiers and those tiers themselves are components of other styles, this is makes it a hierarchically heterogeneous design. This fits the distributed design required because the different tiers will serve different purposes, each only concerned with their own inner workings. The three tiers as proposed are:



The topmost tier, the presentation tier handles the UI, and any interaction with the user, this tier is designed using the model view controller schema, where the model is provided by the tier below. The second tier is the business logic (data processing and analysis mostly in our case), handling all the requests sent by the client and acting as a mediator sending them on to the third tier. The third is the information logic, where access to the database is handled, which is situated below it, and can only be accessed through this tier. It can also be described as a database management system.

This design could also be looked at as an advanced version of the client-server design, as the first two tiers have a client-server relationship. The business logic being the server/provider for the UI. But business logic can also be viewed as the client for the information logic tier of the system, which handles its requests in turn.

This division of layers, creates an abstraction around each task that needs to be done by the system, thus satisfying the separation of concerns concept. It is very beneficial to separate our design into these layers compared to the two-tiered architecture. By each layer having a clear function, they are easier to maintain and create the system in the first place, since the individual layers do not need to know about how the other layers below or above them function. They are provided with the information and data needed to do their business request, from the tiers above and below respectively. In a bigger system we can even employ different experts to develop different parts of the system making it a more manageable task. There is a trade off with applying this separation of concerns however, because it may be easier to implement the system, but changing it in the future might be slightly more complex. However horizontal scalability is great on a system design like this since the tiers may be divided further or more can be added if future upgrades require it, and the individual layers can be scaled without having to worry about any of the other ones. Thus, in the cases of individual layers, the changes and upgrades are easier in the system. Other benefits of division include better security for our database, because the presentation layer is not directly connected to the database itself, so we can employ additional security checks against malicious users in the business logic, such as handling SQL injections. (*What Is Three-Tier Architecture | IBM*, n.d.)

Compared to the two-tier architecture, we can conclude that this design is more beneficial in many ways. The most important part is perhaps, that a separate business tier provides many capabilities that a more complex such as this would require. The ease of development for a more complex system is also substantial compared to the other architecture, but there are trade offs too, for example the upgradability of the system might be slightly worse than other designs. The centralised business logic for many end users, makes the application maintenance centralised too, thus easier, compared to the client-server design where this could be more complicated as the number of clients a server has.

2.Comparison of architectures:

A decision must be made between the proposed architectures, in this case the decision will be made and justified using the Software Architecture Analysis Method also known as SAAM. Some comparison of the two architectures proposed in the previous section was already done, however a more formal way of comparing two possible architectures can be done using a method called SAAM which helps us show more elegantly and more clearly the differences between the systems and why one of them is superior. This method reveals relative scores of architectures rather than using a precise point system from 1-10 for example. Thus, it is useful for comparing the two proposed architectures.

SAAM will help us put our architectures into situations which may occur in the lifetime of the system and compare how the two would react differently and how their quality attributes would be put in context. Drawbacks and advantages can be demonstrated more clearly and justified with the method.

2.1. Develop scenarios:

For choosing scenarios one must make sure to present the different perspectives of stakeholders of the project, such as maintenance engineers who might be interested in performance of the system and future expansion of it being easier to implement, or the marketing management, who might be interested in future opportunities for implanting marketing strategies, and not so interested in the elegancy of the written code, more so in the fact that it works well.

2.2. Describe candidate architectures:

This step has been explored in the first section of this paper, but more detail can help the comparison and the justification of the choice made in this section.
The first architecture proposed is using a client-server architecture to implement the DE-store system. In short: the client side of the operation would work as an interactive UI for the client (in our case managers using the system) to communicate and make requests to

the server side. This server would work as a provider of services to the client, accessing the database and providing its information to the client. There are things that can be improved upon this design to better fit the given plans for the DE-store system.
The second architecture improves on this first one in several ways. It is a heterogeneous three-tiered architecture design implementing different architecture styles in the different tiers of the system. Each tier deals with one task and they don't need to understand how any of the other one's work. The topmost tier works as a client-server architecture with the tier below and the second tier works the same way with the third. The first tier is the frontend of the system, the presentation layer, which provides the user the ability to request information from the second tier. This tier is the business logic layer, where the data is transformed to fit the client's requests and sent to the tier below, most of the business logic is found. The next tier is the persistence layer where the database access is made. Separating these tiers based on their functionality creates a logical abstraction, making the system much easier to implement a huge benefit for the engineers. And this idea also supports the separation of concerns concept because each layer has its own limited scope and doesn't care about the logic implemented in the other layers. However, this design style comes at the cost of less overall agility in the future: some things will be harder to change in the future.

2.3. Classify scenarios:

Each of the chosen scenarios can be one of two types: they are either direct or indirect. Direct scenarios would be ones that are supported by the proposed architecture without making any changes to it. Indirect scenarios are ones where the system would need to go through some changes to handle the problem, these changes can be architectural – components or connectors might need changing to fit better in certain situations. Our choice of better architecture could be made by determining which option has less indirect scenarios, which option is more flexible, which can handle the most scenarios with minimal or no changes.

2.4. Scenario evaluations:

Scenarios and their potential impact on the architectures can be summarized in a table, this will show the types of scenarios evaluated and their predicted cost to implement the indirect ones.

Architecture	Scenario	Direct/ Indirect	Cost of implementation
Client - server	1.Expanding on the business logic	I	Maintenance with an increased number of clients can be expansive. Business logic will also need to be changed on each client if it needs expanded.
	2. Adding additional security to the system	I	Implementing this wouldn't be too difficult, however the overall security of client -server systems is not the most reliable, because the client directly communicates with the server.
	3.Adding a store front for customers to use	I	This might be hard to implement, since changes done will impact the processing of data and a different UI needs to be implemented.
	4. Adding log in system	I	Implementation is possible in the client side and checks are made in the server side. Since both need changing may make changes more difficult.
3-tiered heterogenous	1.Expanding on the business logic	I	The business layer can be expanded easily since, updating it won't affect the other tiers. Also, if needed, extra tiers can be implemented to expand the system.
	2. Adding additional security to the system	I	Implementing more security is really easy, since more security checks can be added, not just in the business layer, but in the persistence layer as well.
	3.Adding a store front for customers to use	I	Changing the presentation layer to implement UI would not impact the other two layers, so it can be done by a frontend specialist without them caring about any other logic in the system. The same way the underlying business logic can be done by a separate team.
	4. Adding log in system	I	It would be easy to implement in the presentation layer, but the business layer would need to be changed as well to check the log in information, thus issues may come from this.

2.5. Reveal Scenario Interaction:

In the case of the client-server architecture multiple scenarios require changes mainly in the client side, which proves that it is an overly general component, meaning that it serves too many functionalities in one place, thus making the maintainability of the system poorer. This is somewhat addressed in the 3-tiered heterogenous architecture by way of using a separate layer to handle the business logic. However, it must be noted that it may be beneficial to separate the system into further tiers based on new business drivers in the future.

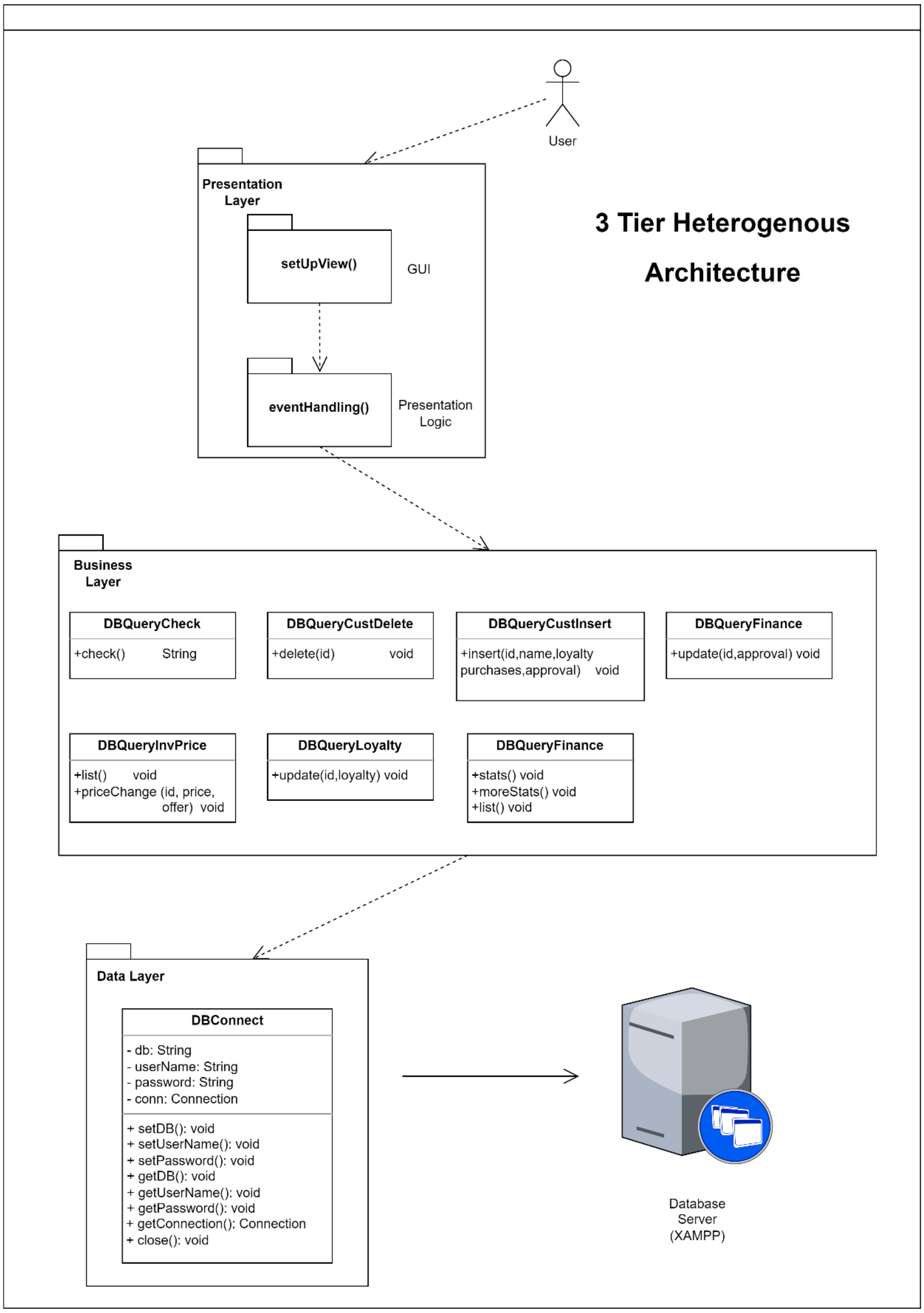
2.6. Overall Evaluation:

We can compare the two architectures based on how well they are suited to implement changes for the proposed scenarios, our scores would be 0, +, - respectively. 0 meaning neither architecture ranks better than the other one in that situation. + meaning one is better and – meaning the opposite.

Architecture	Scenario 1.	Scenario 2.	Scenario 3.	Scenario 4.	Overall
Client - server	-	0	-	0	-
3-tiered heterogenous	+	+	+	0	+

From this table we can conclude that the 3-tiered heterogenous architecture is a better option for a more complex system such as our example. It also copes better with changes, and expansions in the future.

3. Description of chosen design:

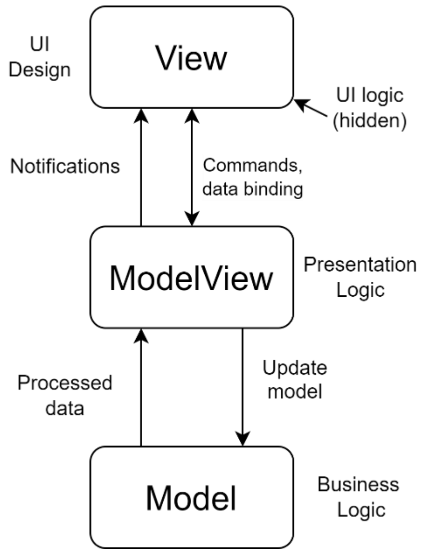


Full design UML diagram

Now that the design has been determined between the two provided architectural styles. Below is a look at the prototype design in more detail. The system fulfils the distributed requirement, there are 3 tiers (and the addition of the database being on a separate XAMPP server), which can be implemented to work on different machines even. The 3-tiered architecture promotes the separation of concerns, meaning tasks are allocated to individual parts of the system, which can function on their own, and need not know about the others. This is also called creating an abstraction around the different business tasks making it easier to understand the implementation as a whole. This also means, from a development point of view that the system can be developed easier. Even if the overall idea for it is complex, the separate parts can be created by specialists, who like the code itself, do not need to know how any other segment of the system works. The architecture incorporates different styles applying them in the 3 different tiers wherever their advantages can be used the best. This makes the architecture hierarchically heterogeneous. This layered design means the components are grouped by technical role in the system, making it a technically partitioned architecture. The three tiers can be further analysed to show their advantages and disadvantages.

The presentation tier is the only tier the user interacts with, it acts as a client to the 2nd tier (the application/business tier) this design ensures the user cannot directly communicate with the database, they only see a representation of the system, and through a UI they can send requests to the business tier server. The presentation tier does not do any of the data processing it only deals with setting up the UI, getting inputs from the user, and once the data comes from below, it shows it to the user. Since this tier is "thin" it also does not need expansive hardware to function, so the deployment costs can be reduced.

In the prototype the presentation layer is implemented in Object Oriented Style, the specific pattern used is Model-View ViewModel *(figure 1.)*. There is an abstraction between the View (in the prototype this is called UI) setting up the main window the ModelView, this contains the presentation logic. This logic handles events happening when interacting with the user, and calls methods from the Model, essentially sending requests to it. The Model is where the business logic happens, where the data processing is done.



The ViewModel (named eventHandling in the prototype) calls methods of the Model to handle requests done by the user. It implements the mediator pattern as it sits between the front-end and back-end sorting out access based on the business cases of the application (*figure 2.)*

Figure 1

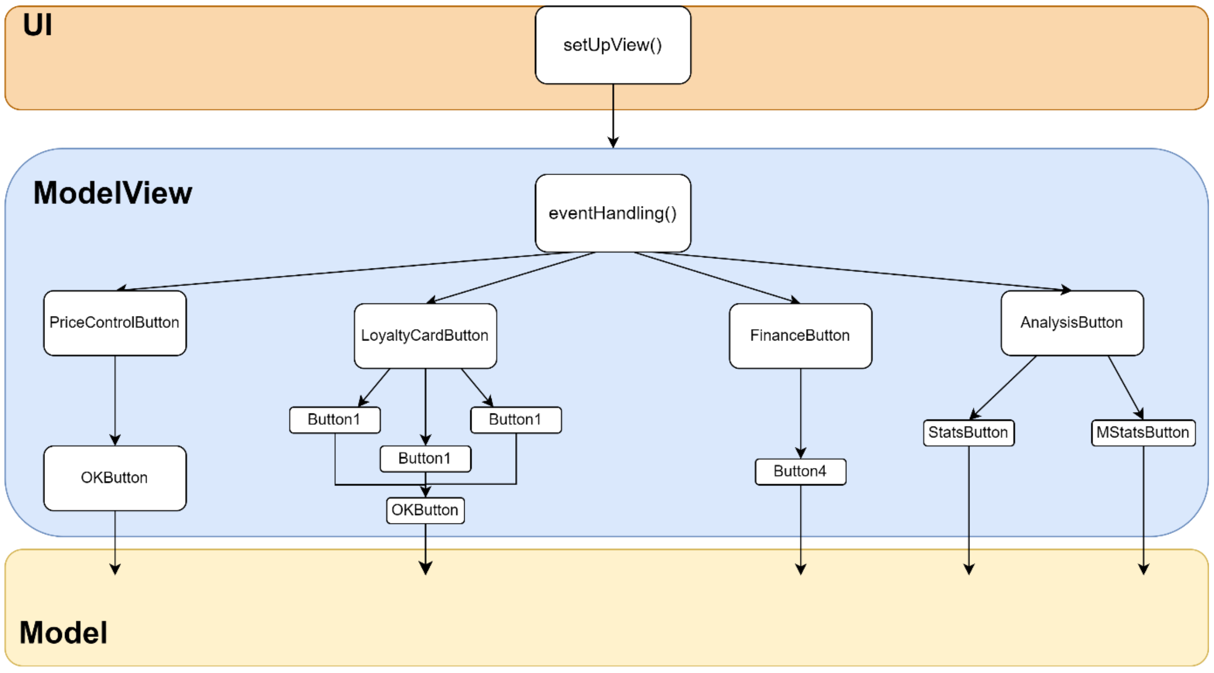


Figure 2

The business layer receives the inputs from the client, it processes them. This tier is designed in an object-oriented way, this fits well with our method of separating concerns via the tiers. Object-oriented design ensures that parts of the system are separated based on the tasks they do, the business goals they achieve. Through creating objects, the real world - or a more complex design such as our example - can be implemented and understood easier. Since the system is modular, it promotes reusability, and this in turn makes development easier. Deployment of the whole system becomes more cost effective too, for the same reason. Potential expansions on the designed system can be implemented easier due to OO as well, the system is easier to maintain too. If errors arise in one business area, then that individual tier (or object in the case of the business layer) responsible for that task can be investigated. In the same vein if updates are needed, or the system needs to scale the whole system does not necessarily have to be redesigned or changed.

3.1. Business requirements handled by the prototype:

Price control:

Once price control is accessed the prototype lists off all the items available with their prices and offers, so the user can examine the current data before changing it. The interface then asks for three inputs for this: the id of the item, the price to be set, and the offer type for the item (this is implemented as an enum datatype in the database, 'N/A', '3 for 2', 'buy one get one free', 'free delivery'). After these are given pressing the OK button these inputs are sent to the business layer, which than sends the database manager an SQL query updating the effected rows. Once this is done an update message is sent to the console, and the details of the change are presented.

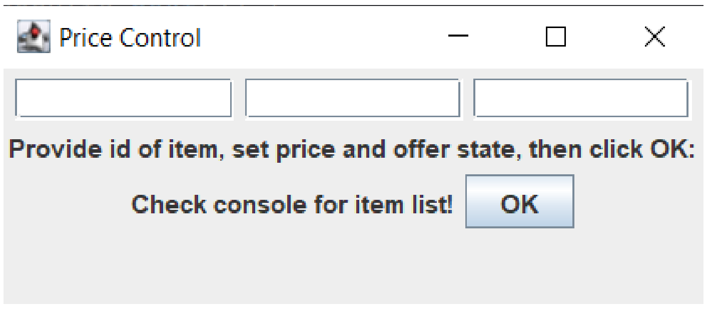
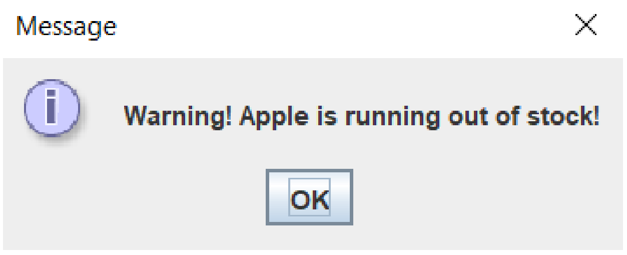

Price Control
Provide id of item, set price and offer state, then click OK:
Check console for item list!
OK

Figure 3.: price control UI

Inventory control:

The stock is monitored with an SQL query checking if an item's quantity falls below 5. If it does, then a messageDialog pops up to warn the manager.



Warning! Apple is running out of stock!

Figure 4: Inventory warning message

Loyalty cards:

All customers in the database have a loyalty card attribute, which through this system can be changed. The UI allows the manager to input the id of a customer, if this is an existing one, they can update their loyalty card status and press OK sending this input to query which handles it and updates the database the way as it was described in the price control business goal. In this interface new customers can also be added, or existing ones deleted.

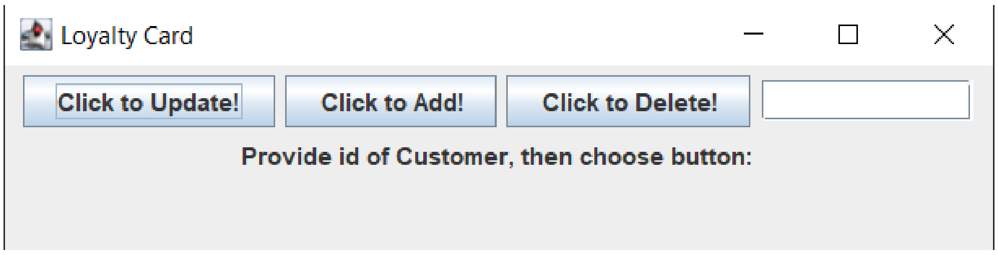


Figure 5: Loyalty card and customer handling

Finance approval:

Customers can get finance approval on their orders, this interface lets managers set this attribute, then a query is sent to the business layer to handle the change in data, which is then updated in the database, same way as the previous queries.

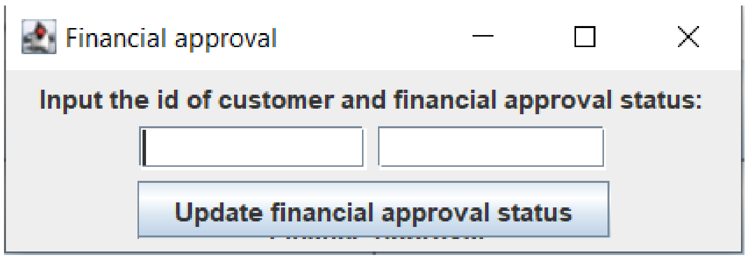
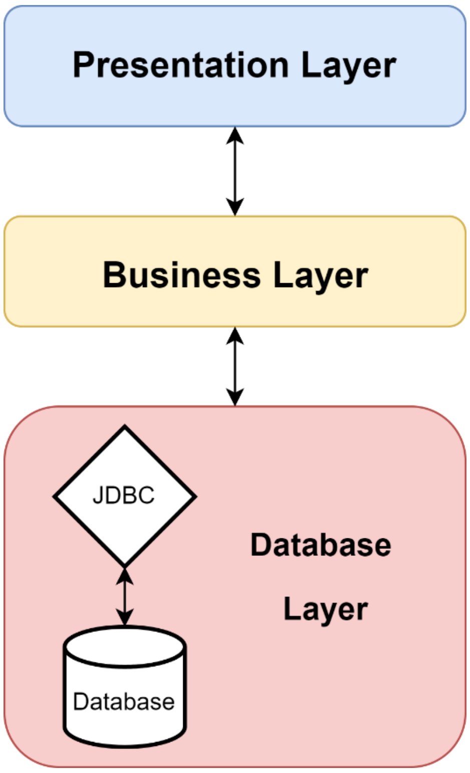

Financial approval
Input the id of customer and financial approval status:
Update financial approval status

Figure 6: Financial approval interface

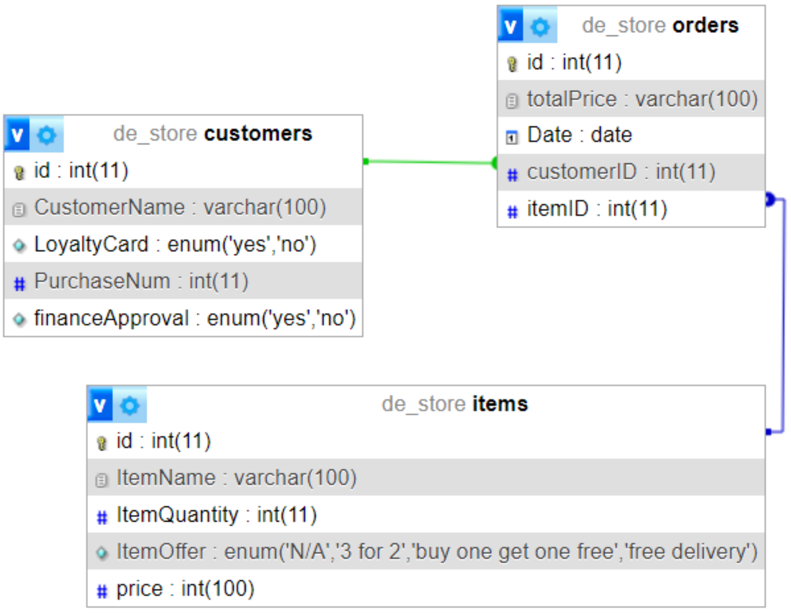
Reports and analysis:
In the database there is an order table connected to the customers and the items tables. The prototype prints out analysis of the purchase history of customers. These are just a couple of examples of what the system could be used for. First it gives back the average amount spent by loyal customers, then the total income of the company in the last year is calculated, as is the number of orders done that year, and the number of orders which have been financially approved. All these calculations are handled in the business logic, which sends SQL queries to the database layer, which accesses the information through the database server. This area of the design could be improved in the future to analyse the database further and improve business for the company. The layered architecture allows these improvements to be made with ease, since if the database is given, only the business layer needs to be expanded to incorporate more statistical queries.

The third tier is the data tier, this level deals with accessing and changing the database, it has all the benefits of separating concerns as the layers above. Extra databases can be added, the existing one can be expanded and only this layer would need to be updated, and the two above would not know about it. This tier being separated from the user, and them only accessing it in this indirect way through the middleware also introduces added security for the database. The prototype implements the database on a separate server using XAMPP server. The DBConnect class creates a connection to this sever using JDBC.



JDBC integration

First a connection object is used to load a driver in. A statement object is then initialized to accept the queries, which are created below, these are then executed and the ResultSet object stores the query results, which can be iterated to then be sent to the presentation layer. From a data centred perspective this architecture could make the middle tier a database client for the layer below.



de_store **orders**

- id : int(11)
- totalPrice : varchar(100)
- Date : date
- customerID : int(11)
- itemID : int(11)

de_store **customers**

- id : int(11)
- CustomerName : varchar(100)
- LoyaltyCard : enum('yes','no')
- PurchaseNum : int(11)
- financeApproval : enum('yes','no')

de_store **items**

- id : int(11)
- ItemName : varchar(100)
- ItemQuantity : int(11)
- ItemOffer : enum('N/A','3 for 2','buy one get one free','free delivery')
- price : int(100)

Database architecture

Design drawbacks:

However, there are drawbacks to this system as there would be with any design. The system trades off scalability, maintainability, and ease of deployment for performance. Because the layers or tiers are separated there is a degree of performance loss from having the data travel through all the layers. When the system needs to incorporate more users, or more database, it will be relatively easy to scale each tier on their own, however the performance of the system is going to be reduced if more and more layers are added, and in an increasingly complex system it may be required to implement further layers, thus this is important to note.

4. Evaluation of the design:

The proposed heterogenous architecture can be evaluated using the ATAM (Architecture Trade-off Analysis Method). This method helps us analyse critical scenarios which the system might encounter. These can describe drawbacks of the chosen architecture and might aid a developer with designing safeguards for the scenarios and problems described throughout the analysis. The method also might give a better understanding of the system.

4.1. Present ATAM to participants:

ATAM is a structured method utilizing potential scenarios, that test the critical parts of the system. Through the scenarios we can showcase how well the system would do in operation, this analysation system is used for investigating problems the system will have to deal with in the future, thus finding good scenarios is a key in setting up the investigation. A good choice would describe a certain problem where a trade-off must be made by the system, competing qualities of the system come into contact and a choice is needed, also investigating cases, where handling the problem wrong might cause critical failure of the system.

4.2. Present business drivers:

The presented system as described in the previous section is designed to work for managers of the DE-store, it makes use of a database storing item information that the store has to offer. There is also a table for storing customer information and purchase information that can be used for analysing statistics of the how the company is performing. Through a User Interface the managers can access different parts of the system through SQL queries get information from the database and modify it, the UI is designed for good usability for anyone who might work with the system. The system is designed with flexibility and extensibility in mind because each layer can be later modified or updated and no other tier would be affected, and if needed new tiers can be incorporated to support additional features or security. This design is also much easier to implement in the first place than other more complicated architectures, thus the installability of the system is great. But this creates a trade-off of course between easier implementation and possibly better future modifiability. Because of the separated concerns of each tier is easier to understand and this can make maintainability for the engineers easier in the future. Project's scope currently is using one database storing all data required for the system, and only one client system (UI) where all the proposed business features can be accessed. Expanding the UI with additional views or implementing multiple databases, however, is possible with the current design, so upgradability of a the 3-tiered architecture works well in this case.

4.3. Present architecture:

The proposed design uses a three-tiered heterogeneous architecture style, it has a presentation layer, a business layer, and a third layer below the persistence layer, which access the database. The top two tiers work with each other in a client-server style architecture, as do the second and third. The persistence layer functions the same in relation to the database.

4.4. Identify architecture approaches:

This is a heterogeneous style since it uses multiple architectures. The different tiers work as separate architecture styles as described above.

4.5. Generate quality attribute tree:

Utility	Maintainability	Database maintenance	(H, L) Detect and fix issue with database access
			(H, H) Detect and fix issue with queries returning wrong data (involves many tiers)
		General maintenance	(H, L) User input issue detection and fix
	Flexibility	Enhance User Interface	(M, L) Implement web-based UI frontend
			(M, L) Changing GUI implementation from java Swing to WPF
		Changing design	(M, L) Changing the system to an n-tier architecture design
	Performance	Increase system efficiency	(M, L) Implementing load balancing in the business layer
	Scalability	Implementing additional functionality	(M, L) Using separate databases for separate functionalities
			(M, H) Implementing new databases for different stores under the same company
			(M, M) Implementing connection to multiple client servers
	Security	Data integrity	(H, M) Implementing additional checks to improve data integrity.
		Data confidentiality	(H, L) System sabotage done by malicious user.
			(H, L) Implementing a user login page before accessing the system.

4.6. Analyse architectural approaches:

Another table below shows individual analysis of 2 architectural approaches in relation to scenarios in more detail.

Analysis of architectural approach	
Scenario #: Security 1	**Scenario:** System sabotage done by malicious user.
Attributes:	Security, data confidentiality
Environment:	Normal operation
Stimulus:	An attack against the system is issued through the interface
Response:	Additional security checks set up against potential attacks

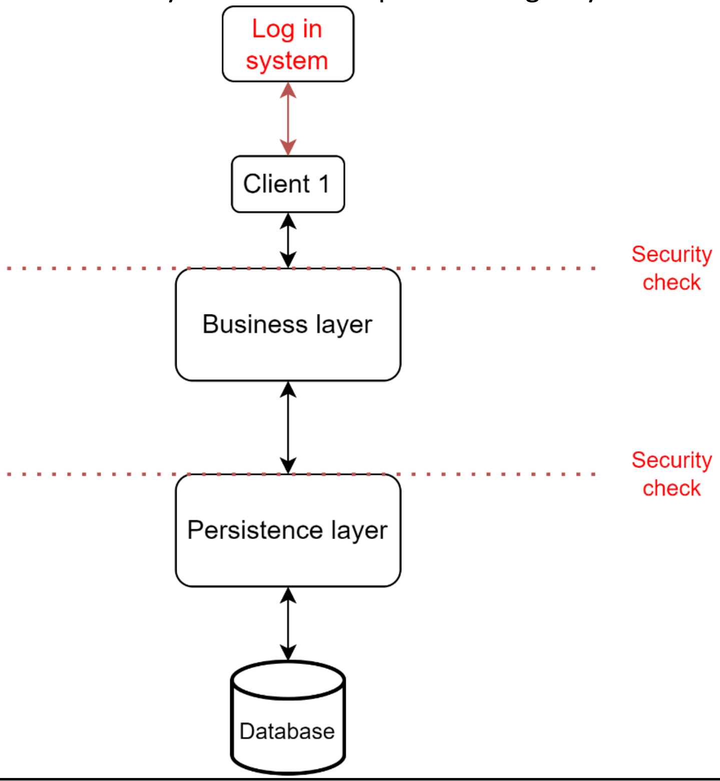
Architectural decisions	**Sensitivity**	**Trade off**	**Risk**	**Non-risk**
Separate client tier	S1			N1
No backup server	S2	T1	R1	
No login system	S3		R2	

Reasoning:

- Because of the 3-tierd system, the client is separated from the database, and additional checks are easily implemented in the business layer and protect the database from malicious users (Non-risk 1)
- If an attack succeeds however, the lack of a data backup would damage availability and data integrity (Risk 1)
- Login system could also add another layer of security. This implementation shouldn't cause problems because of the separated concerns of tiers.

Architectural design:

Extra security measures and potential login system:



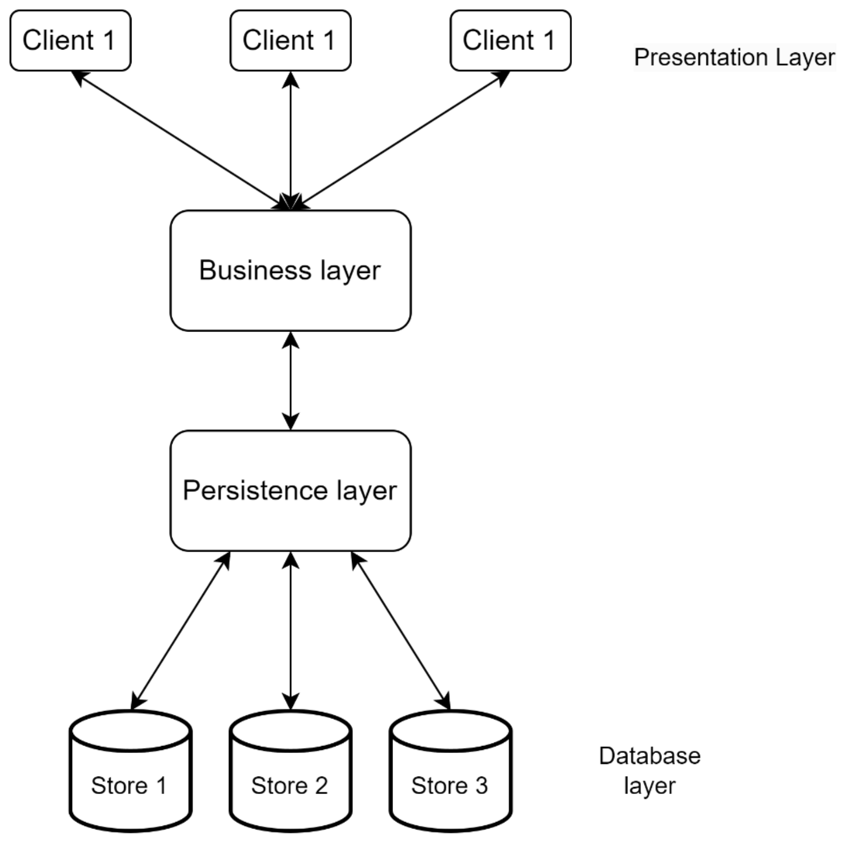
Analysis of architectural approach	
Scenario #: Scalability 2	**Scenario:** Implementing new databases for different stores under the same company
Attributes:	Scalability (and upgradability)
Environment:	Company expansion
Stimulus:	Company wants to expand, new stores are introduced, the system needs to incorporate them as well.
Response:	New store databases are added to the system, modification of individual layers is easier, changing the whole system might cause issues.

Architectural decisions	**Sensitivity**	**Trade off**	**Risk**	**Non-risk**
Separated persistence layer	S1	T1	R1	
Data processing separate	S2		R2	
Object oriented design	S3	T2		N2

Reasoning:

- The 3-tiered architecture provides scalability if only one functionality is changed, but in this case, because more than one layer is involved the change is more complicated. (T1) Both the database access, and the data processing need to be upgraded to incorporate new stores. (R1, R2)
- Because the business processing layer was created in OO style, parts of the system can be reused and only slightly changed to fit for new stores. Same is true for how the database is accessed, even if new databases are added the implementation can be reused and upgrades can be made easier. Extra clients may be added for the new stores too, if required and the same benefits apply. (T2)

Architectural design:
New design with incorporated extra databases, and different clients:



4.7. Summary:

Through this evaluation we can conclude that the architecture works in a way it was predicted, the growth scenarios show that making updates to the system is reasonably easy, and in the this could help with fixing its negative sides as well. For example, security is an issue in the current prototype, but with the architecture, its implementation is not an issue. Another discovery is that if many layers are involved in an issue, or the system needs to be scaled in all aspects, that can prove very difficult with the layered architecture. While changing or expanding one layer is easy, with many it gets confusing and can lead to system failure very easily.

References

1. Layered Architecture—Software Architecture Patterns [Book]. (n.d.). Retrieved 24 November 2023, from https://www.oreilly.com/library/view/software-architecture-patterns/9781491971437/ch01.html

Kazman, R., Klein, M., Barbacci, M., Longstaff, T., Lipson, H., & Carriere, J. (1998). *The Architecture Tradeoff Analysis Method*. 68–78. https://doi.org/10.1109/ICECCS.1998.706657

Kazman, R., Klein, M., & Clements, P. (2000). *ATAM: Method for Architecture Evaluation:* Defense Technical Information Center. https://doi.org/10.21236/ADA382629

Lo, P. (2010). *Software Architecture Analysis Method (SAAM)*.

Metsker, S., & Wake, W. C. (2006). *Design Patterns in Java*. Addison-Wesley Professional.

Singh, E. (2021, October 2). Architectures in Distributed Systems. *CodeX*. https://medium.com/codex/introduction-to-distributed-systems-66502ac8289

Software Architecture & Design Introduction. (n.d.). Retrieved 24 November 2023, from https://www.tutorialspoint.com/software_architecture_design/introduction.htm

Tyson, M. (2022, May 13). *What is JDBC? Introduction to Java Database Connectivity*. InfoWorld. https://www.infoworld.com/article/3388036/what-is-jdbc-introduction-to-java-database-connectivity.html

What is Three-Tier Architecture | IBM. (n.d.). Retrieved 14 November 2023, from https://www.ibm.com/topics/three-tier-architecture