SET10101 Software Architecture Coursework Report

Architectures Considered

For this development contract there are many approaches that can be taken to develop the business management system that utilise different software architectures. The system described in the contract is a distributed system, meaning the different processes that make up the system reside on different machines or networks. The two architectures considered are Data Centred Architecture for creating a centred database that can be changed and queried by services within the distributed system and Three-Tiered Architecture which is effective for complex systems integrating a graphical user interface.

Data Centred Architecture

In a complex distributed system, each process will need data to perform their written functions. Data must be easily accessible to the processes running without relying on the existence of other processes. Data centred architecture uses the existence of a central database to store all of this data. The data kept in the central database can then be easily accessed and manipulated by other processes.

What is Data Centred Architecture

Data Centred Architecture is split into two categories: Blackboard and Repository. In the case of blackboard, the central data store acts as a knowledge base for a problem, with multiple knowledge sources updating its' content to provide information that other sources can use to contribute to a solution.

A repository data-centred application uses a passive data store (the repository) which contains the necessary information for processes to carry out tasks or solve problems. It is this style of data-centred architecture that will be considered for the project.

The components for a data-centred system are agents and a database. Agents connect to a database via product protocol (The JDBC API for Java as an example) and can send updates or queries as required.

Benefits of using Data Centred Architecture

Having all of the systems data situated in one place has many benefits:

- Easy to make secure – security measures to protect any data would not have to be implemented on each process, just the repository.
- Easy to backup – backups only need to be administered to the repository. All data used by processes comes from this repository so only one component needs backing up
- Code reuse – if two different processes need to manipulate the database in the same way the code used is the same if not very similar. This makes it easy to implement methods for CRUD operations that take different arguments to perform the same task.

Drawbacks of using Data Centred Architecture

There are also drawbacks to having a centralised repository:

- Difficult to Evolve Data – if the structure of the repository must change, every process in the system that has access to the database must have its' code updated appropriately. This can be difficult and very expensive for large distributed systems.
- High dependency on a single component – although other agents should function fine if another fails, in the event that the repository fails or becomes corrupted in some way the entire distributed system will fail.
- Bottlenecks – high traffic to the repository, which would be common in a large distributed system, can lead to decreased in the systems efficiency. This drawback can be negated by relocating the database to a machine capable of managing the increased traffic.

Suitability for System

Implementing a repository data-centred architecture for the DE-Store would be useful for storing any information about products, customers and sales. Given the DE-Store is a distributed system each process running would act as agent. Having all agents connected to and manipulating the same repository would prevent processes from using data that is out dated.

Three-Tiered Architecture

For a system that needs to be expanded or adapted, it is important that code can easily be updated or replaced. If a piece of software's concerns overlap, it can be incredibly difficult and time consuming to make changes to code. Three-tiered architecture is used to create modular software that splits up code into one of three tiers based on its function.

What is Three-Tiered Architecture

Three-tiered architecture is an architecture with which a developer can divide a programs functions into three components that can be maintained independently of one another.

The presentation tier includes code relevant to the user interface of the system. The role of the presentation tier is to act as a bridge for the user to exchange information with the system and vice versa.

The application tier is responsible for the logic behind the application. It is where decisions are made and calculations are performed. Data is moved between the data tier and the presentation tier via the application tier.

The data tier is responsible for any data manipulation in a program. Whether this be file editing or database manipulation, the data tier often provides persistence for a system.

Each tier in a program developed with three-tiered architecture does not have to be located on one machine/process; modules may run on a server on another network.

There are many options for protocols for data transfer between the layers. These may include UDP, RMI, peer to peer.

Benefits of using Three-Tiered Architecture

Having a program split up into three tiers has the following benefits:

- Ease of maintenance – tiers can be easily modified without requiring changes in other tiers. This reduces maintenance down time and cost.
- Specialised development – if multiple developers or teams of developers are working on a three-tiered system they can each work on areas they are most experienced in.
- Separates concerns – less instances of code becoming tangled or scattered.

Drawbacks to using Three-Tiered Architecture

Some of the drawbacks to using three tiered software architecture are:

- Initial development is time consuming – to implement a new small part of an application new code is usually required for each tier.
- Complex structure – when programming with a three-tiered architecture there are many methods spread across three layers, this can be difficult to keep up with when designing the program.

Suitability for System

The development contract states that the DE-Store must be “expandable and adaptive”. Three-tiered architecture would enable this as code can be modified easily to accommodate for changes in the system design. One tier can be modified without breaking other areas of the application, making maintenance cheaper and easier. This makes adapting the system and adding new functions on top of the ones described in the contract a simpler process.

Final Architecture Recommendation

The architectural style chosen to implement the software development contract is heterogeneous, meaning it will be built using multiple architectural styles. The system is Repository Data-Centred with the main DE-Store process utilising three-tiered architecture.

The system uses remote method invocation (RMI) as a means of inter-process communication. RMI is a Java API which allows processes to invoke methods contained within another process, it is used within the prototype to demonstrate how the processes within the system can connect with each other across different machines.

Design

The prototype for this contract uses multiple separate processes to emulate a distributed system. The main application (DE-Store) is the prototype for the contract, other processes act as the agents that might be a part of the functions described. These processes connect to a central database using JDBC and use SQL statements to get, add or remove information needed to carry out tasks. See the diagram: Class Diagram.

DE-Store

The DE-Store process is the main program described in the development contract. It is a three-tiered system made up of a user interface layer, an application layer and a data layer. These are not located on separate servers; the three layers are all contained within the one program:

- Presentation Layer: This layer defines the UI components that the user interacts with. The program has tabs for different aspects of the program (Products, Inventory Control, Customers and Sales).
- Application Layer: This layer is used to implement the logic behind the user interface, for example, clicking a button on the UI may call a method from the application layer to process data inputted to the UI by a user. This layer also creates the strings for logging user activities.
- Data Layer: This layer is responsible for manipulating the central database.

The purpose of using these three tiers was to make programming the application simpler. The different code for the three layer categories are kept separate making it easier to manipulate methods that may be called multiple times by other layers.

The DE-Store process also acts as an RMI server/client. So that it can communicate with the other processes to send and receive information.

Distributed System Components

Central Inventory System

The Central Inventory System is a simple RMI server with a basic GUI which was implemented to fulfil the inventory control function specified in the contract. The system can receive requests for stock replenishment from the DE-Store and respond by sending selected requests.

Sales System

The Sales System is another simple RMI server with a basic GUI which emulates a sale being made. When a sale is made the sales system invokes a method within the DE-Store process, passing on information about a new sale. This process is used to demonstrate the DE-Store's ability to handle incoming sales.

Database

The database for the system was hosted using WAMPserver64. Processes in the prototype system access the database via the DE-Store's data tier to query and update the hosted database using JDBC. Product, customer, sales and order information is all stored on the SQL database, which serves as the repository. See database diagrams.

Tools Used

Eclipse – IDE for Java application development

WindowBuilder – Plugin Java GUI designer for Eclipse

ObjectAid – Plugin Java Class Diagram Creator for Eclipse

WAMPserver64 – Database hosting software

Evaluation

From the list of aims in the development contract for what functions the DE-Store should have the following have been implemented in the prototype:

- Price control - Using the prototype a store manager can add, remove and update product information. Products are displayed on a table in the “Products” tab and a manager can search for a product to access it easily to see its details, edit it or remove it from the system entirely.
- Inventory control - The prototype has an “Inventory Control” tab with tables for products that are low on stock and products that are out of stock. When a products quantity is changed it will be automatically placed into a table if it meets the conditions. By default a product is classed as low stock if its quantity falls below 20 but this can be redefined easily.
  A store manager can request stock from the Central Inventory System, choosing the product and the quantity to request. This order will show up in the Central Inventory System and it can then be sent to the store, product information and inventory control tables will all update to display the new relevant information.
- Loyalty card – When a new customer is added to the system, they start with 0 loyalty points. For purpose of the prototype whenever the customer spends £10 they receive 1 loyalty point. This is calculated by the prototype and the databases are updated to reflect the gain in points.

As the DE-Store is a repository based system, it is important that the data remains correct and uncorrupted. The following checks are in place within the prototype to maintain the integrity of the repository:

- A store manager cannot leave fields blank when inputting new data or the system will throw a warning dialog notifying them of the mistake. It will not allow any changes to the database that would ruin the integrity of the data.
- The system checks that users definitely want to delete items from the database; this is to avoid accidental deletion of any vital information. This is in the form of a warning dialog.
- Selecting any options without completing any required prerequisites (selecting an item from a table) will alert the user of any actions they need to take first.

This prototype defines the architecture of the system and provides an excellent starting point for developing the full system in the future. The generated class diagram shows progress so far.

Future Development

Using the architecture style demonstrated using the prototype it would be simple to add the following features:

- Finance Approval – by connecting the system to the “Enabling” finance system, a store manager could utilise buy now and pay later schemes by sending the relevant information to the system via a communication protocol such as RMI. The Sales System and Central Inventory Systems used in the prototype demonstrate how RMI is a viable choice for this exchange. The customers table in the repository and any associated methods may have to be modified but thanks to the modular nature of three-tiered architecture this should be an easy change.
- Reports and Analysis – the prototype already displays sales received by a mock sales system and automatically adjusts stock levels and customer loyalty points as necessary. It would be possible to implement a function to generate multiple reports based on types of products, time frames, customers etc. The sale class may need to be updated with more attributes depending on what information the company wants to see in these reports.
- Loyalty Card Special offers – it would be possible to add a loyalty to card reward for the store that would work by checking a customer’s total loyalty points when they make a sale and if they meet the threshold they would receive a special offer such as free delivery.

The prototype is currently connected to mock versions of a sales system and central inventory system. If the contract is won and DE-Store was to be developed further it could be linked to the company’s actual systems easily due to its three tiered architecture, only requiring changes to the application layer.

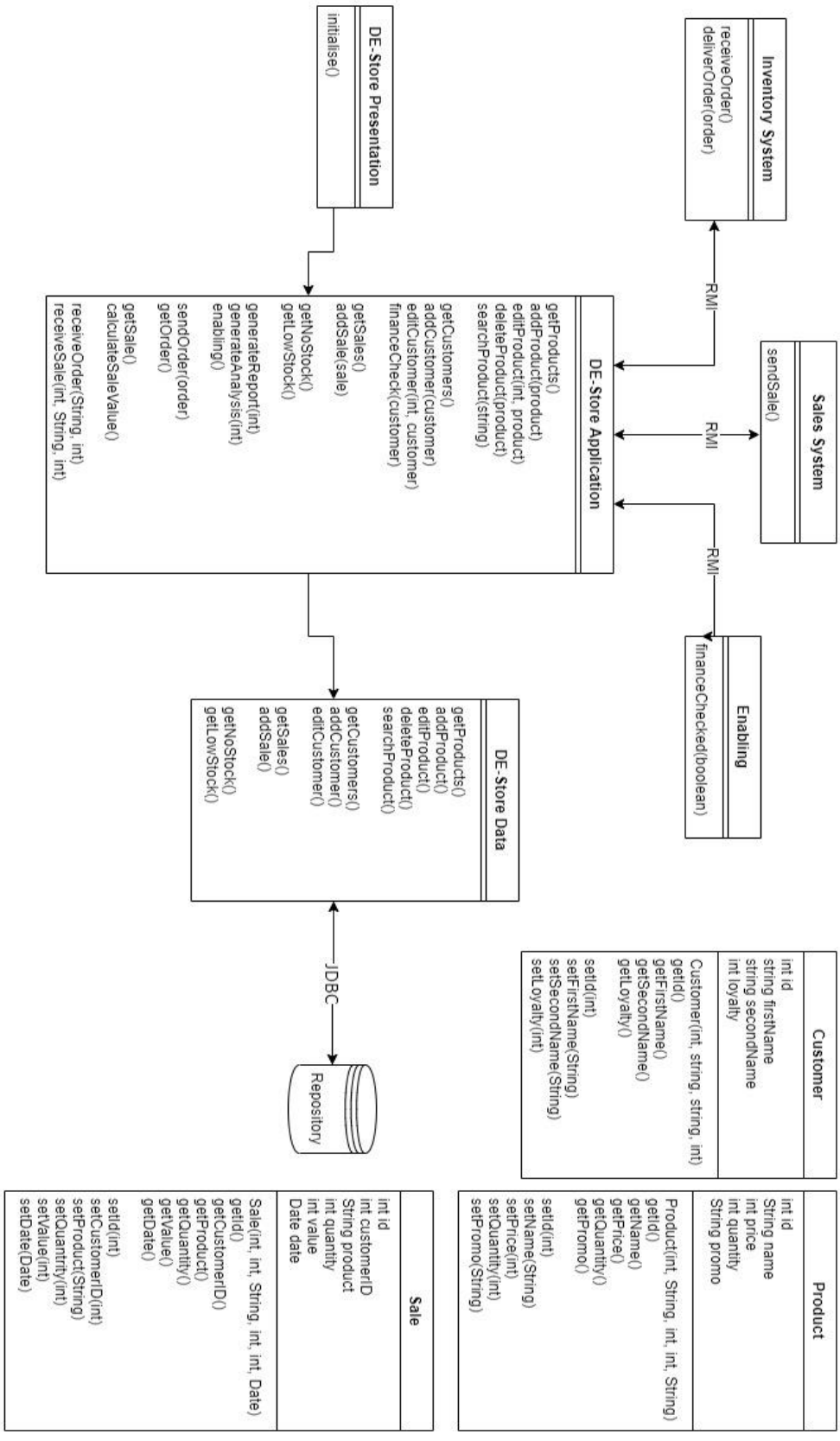
At the moment, the prototype contains hover-over tooltips to help guide a store manager using the program however should the contract be won a full user guide would be written.

If developed into a fully functional system, user accounts (with encrypted password management) with different permissions could be integrated to the system. Currently product names cannot be changed by the store manager but this is the same for any user using the program. An admin account or “Area Manager” account could be implemented to grant the user access to more functions within the program.

Diagrams

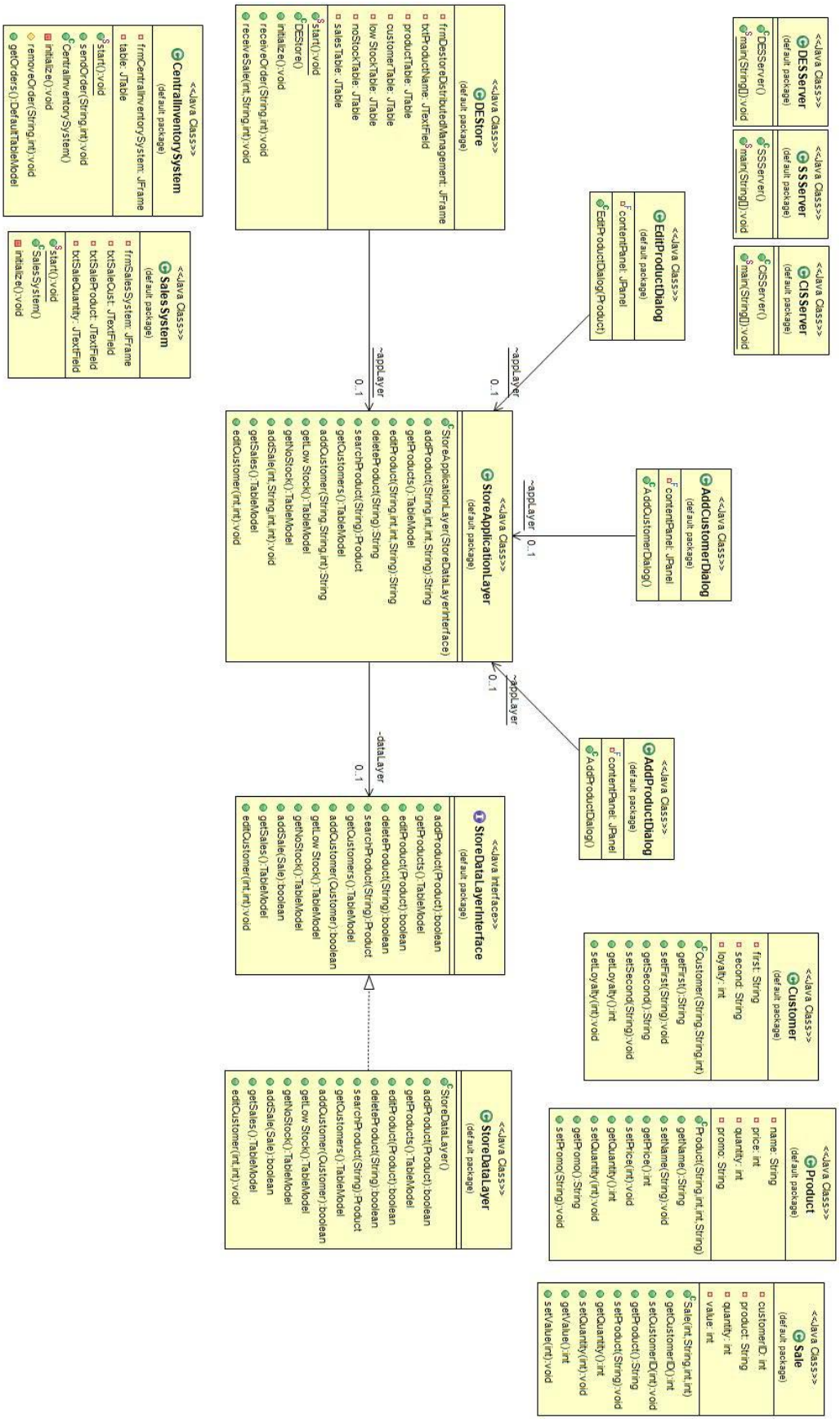
Class Diagram

The proposed class diagram for the chosen architectural style.



Generated Prototype Class Diagram

Class diagram generated by ObjectAid from the prototype's code.



Database Tables

Structure of tables within the repository.

Products:

	#	Name	Type	Collation	Attributes	Null	Default	Comments	Extra	Action
☐	1	Name 🔑	varchar(100)	latin1_swedish_ci		No	None			Change Drop More
☐	2	Price	int(11)			No	None			Change Drop More
☐	3	Quantity	int(11)			No	None			Change Drop More
☐	4	Promotion	varchar(100)	latin1_swedish_ci		No	None			Change Drop More

Customers:

	#	Name	Type	Collation	Attributes	Null	Default	Comments	Extra	Action
☐	1	id 🔑	int(11)			No	None		AUTO_INCREMENT	Change Drop More
☐	2	first_name	varchar(100)	latin1_swedish_ci		No	None			Change Drop More
☐	3	second_name	varchar(100)	latin1_swedish_ci		No	None			Change Drop More
☐	4	loyalty_points	int(11)			No	None			Change Drop More

Sales:

	#	Name	Type	Collation	Attributes	Null	Default	Comments	Extra	Action
☐	1	saleID 🔑	int(11)			No	None		AUTO_INCREMENT	Change Drop More
☐	2	customerID	int(11)			No	None			Change Drop More
☐	3	product	varchar(100)	latin1_swedish_ci		No	None			Change Drop More
☐	4	quantity	int(11)			No	None			Change Drop More
☐	5	value	int(11)			No	None			Change Drop More

Central Inventory System Orders:

	#	Name	Type	Collation	Attributes	Null	Default	Comments	Extra	Action
☐	1	Product	varchar(100)	latin1_swedish_ci		No	None			Change Drop More
☐	2	Quantity	int(100)			No	None			Change Drop More