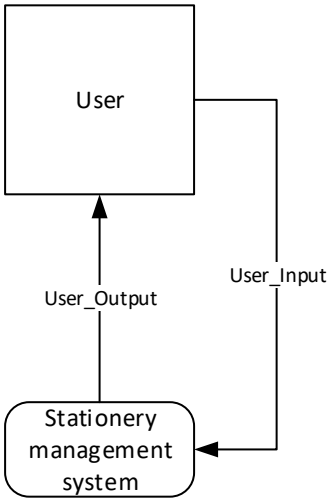
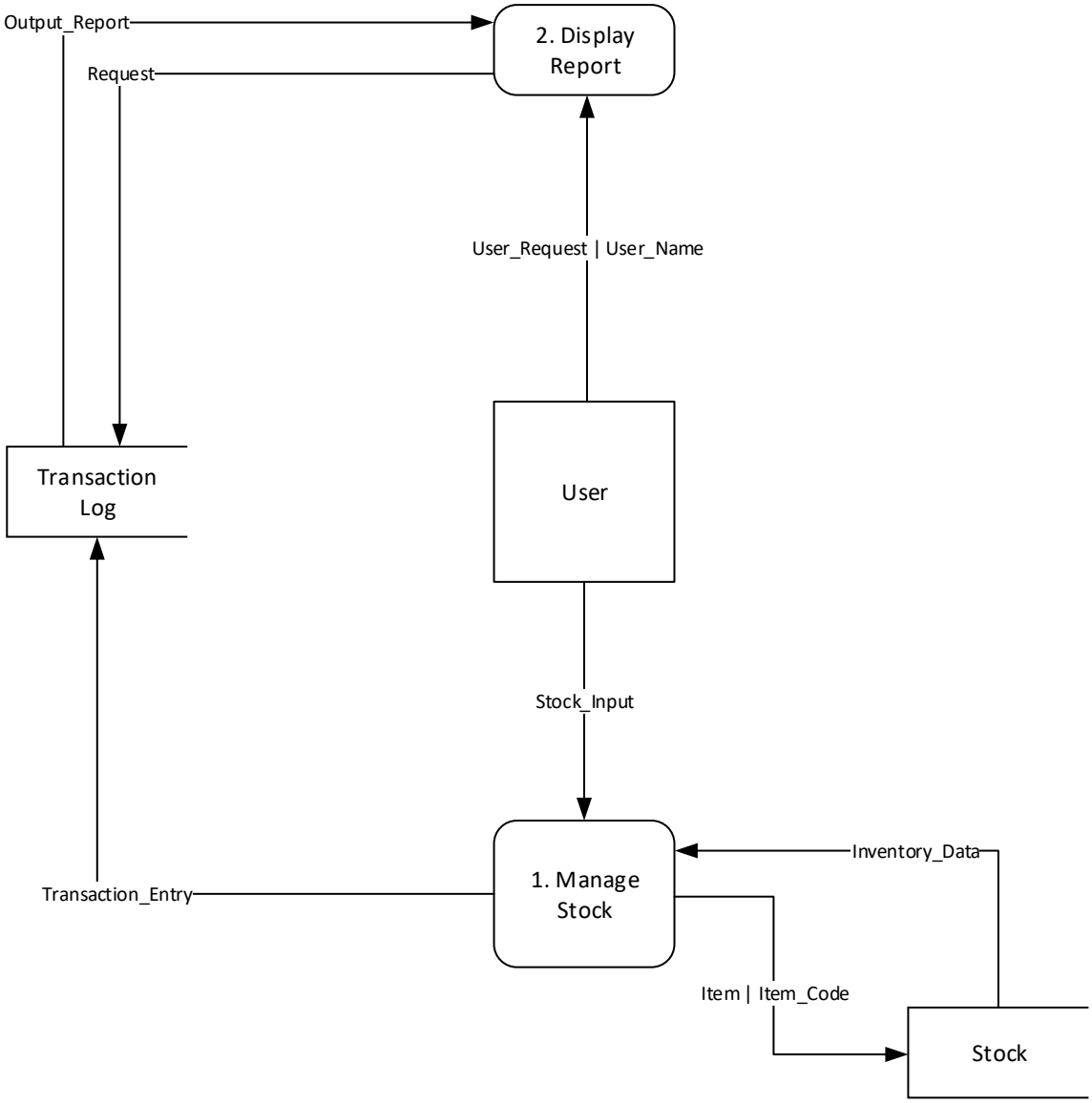
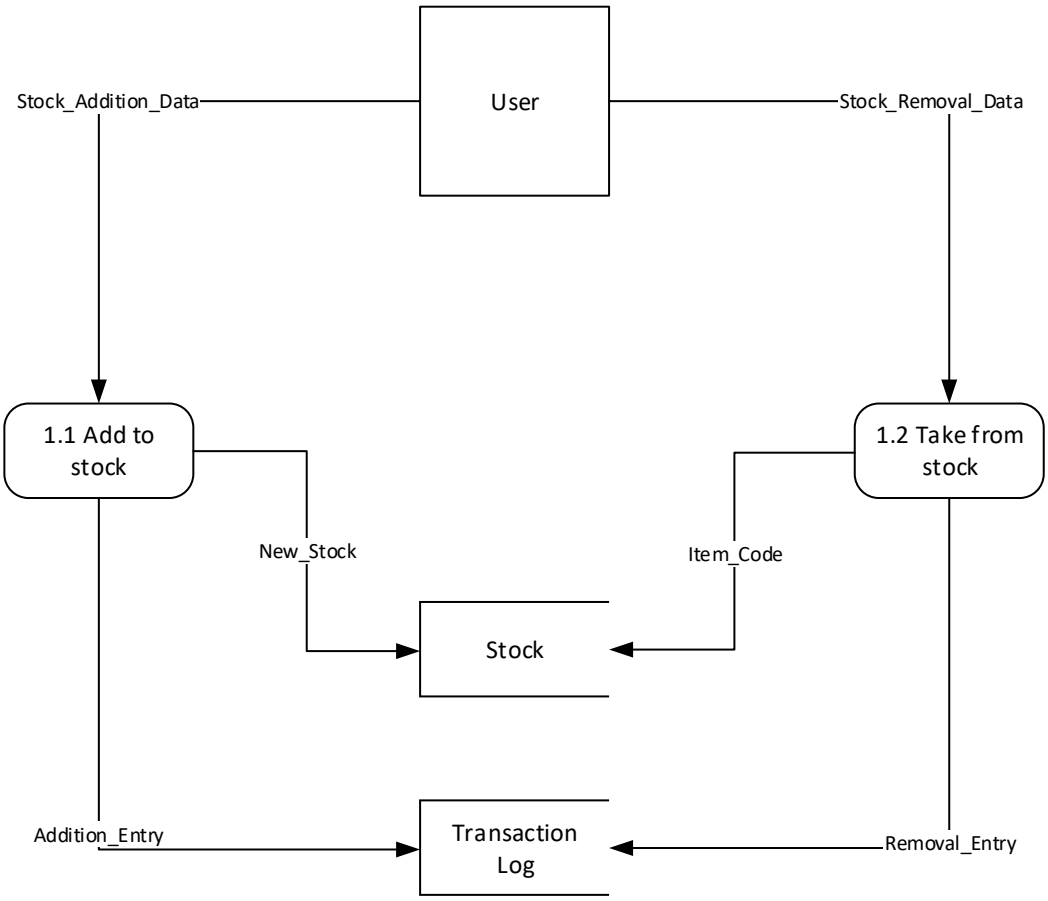
# Context Diagram



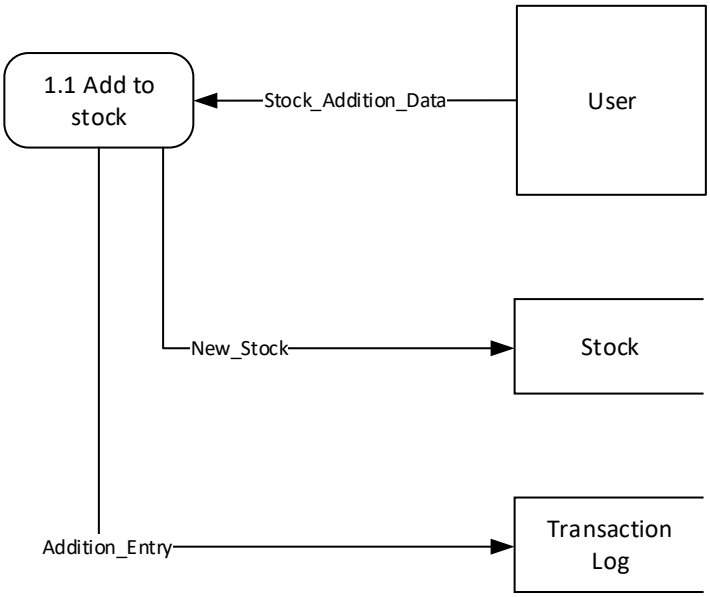
# 0. Stationery Management System



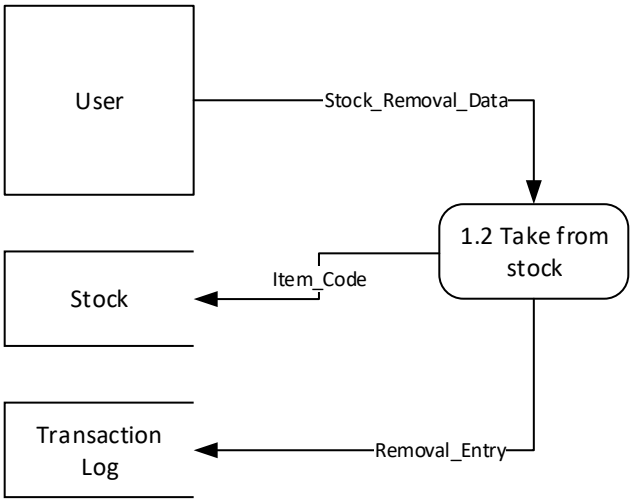
# 1. Manage Stock



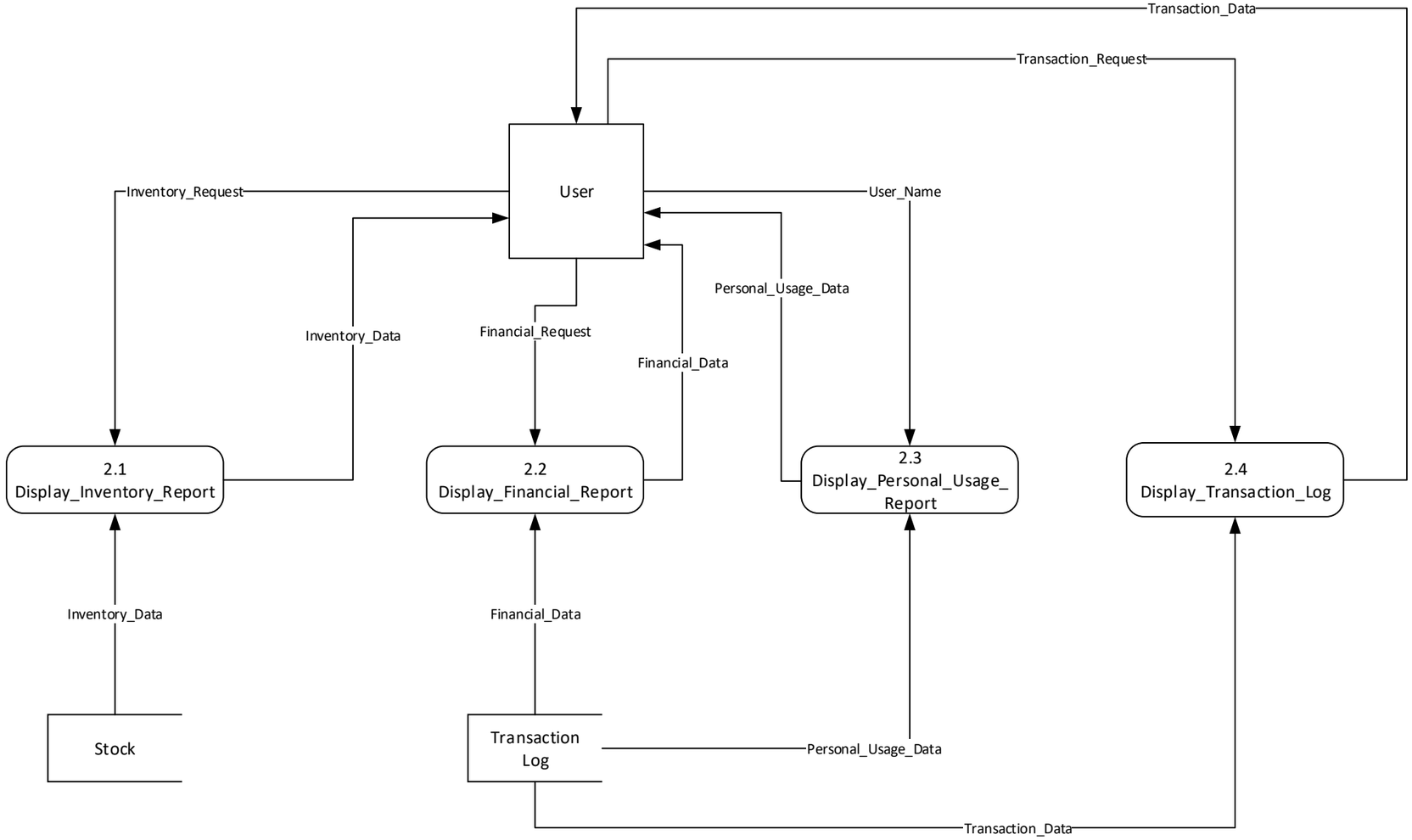
# 1.1 Add to Stock



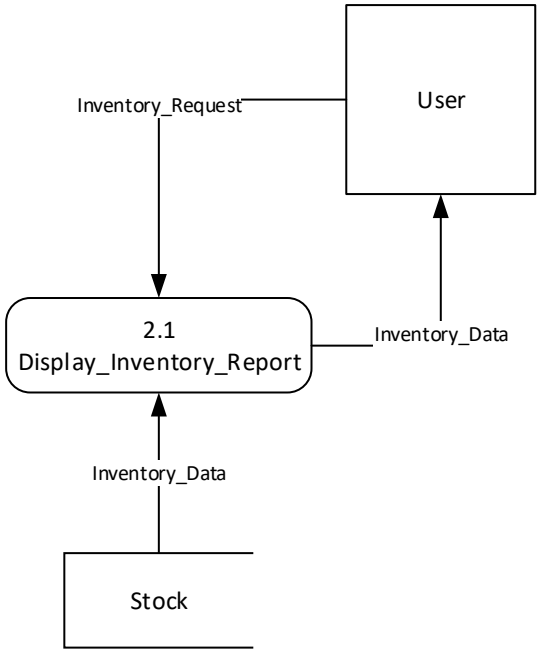
# 1.2 Take From Stock



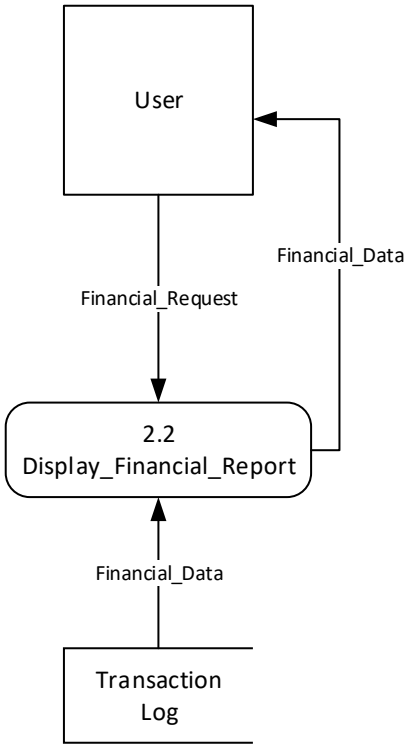
# 2. Display Report



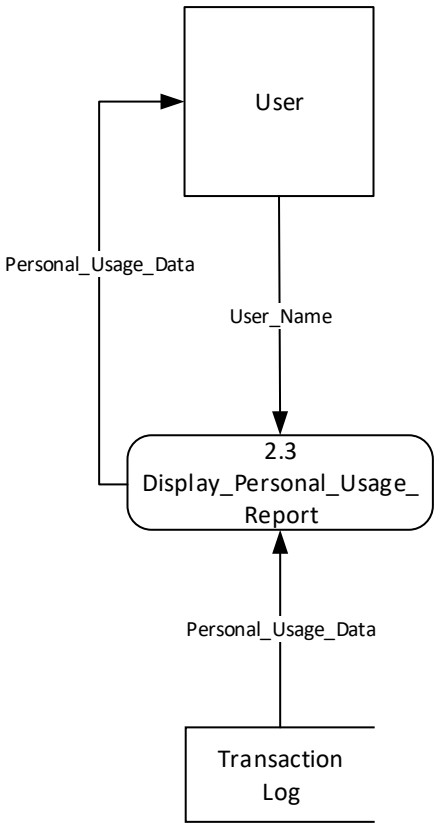
# 2.1 Display Inventory Report



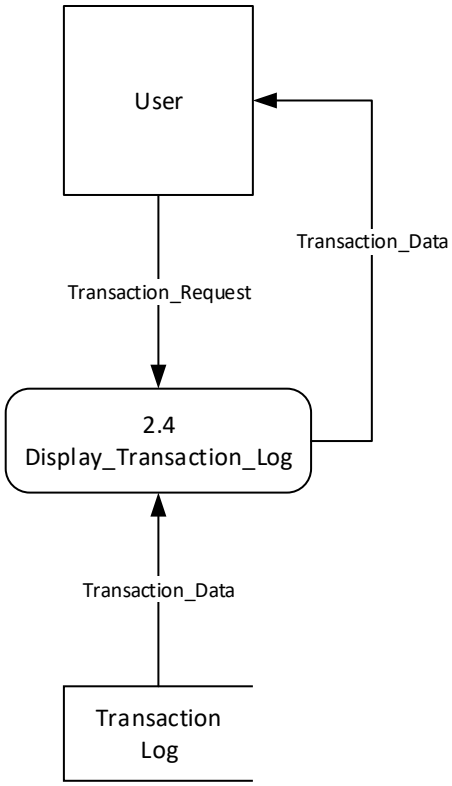
# 2.2 Display Financial Report

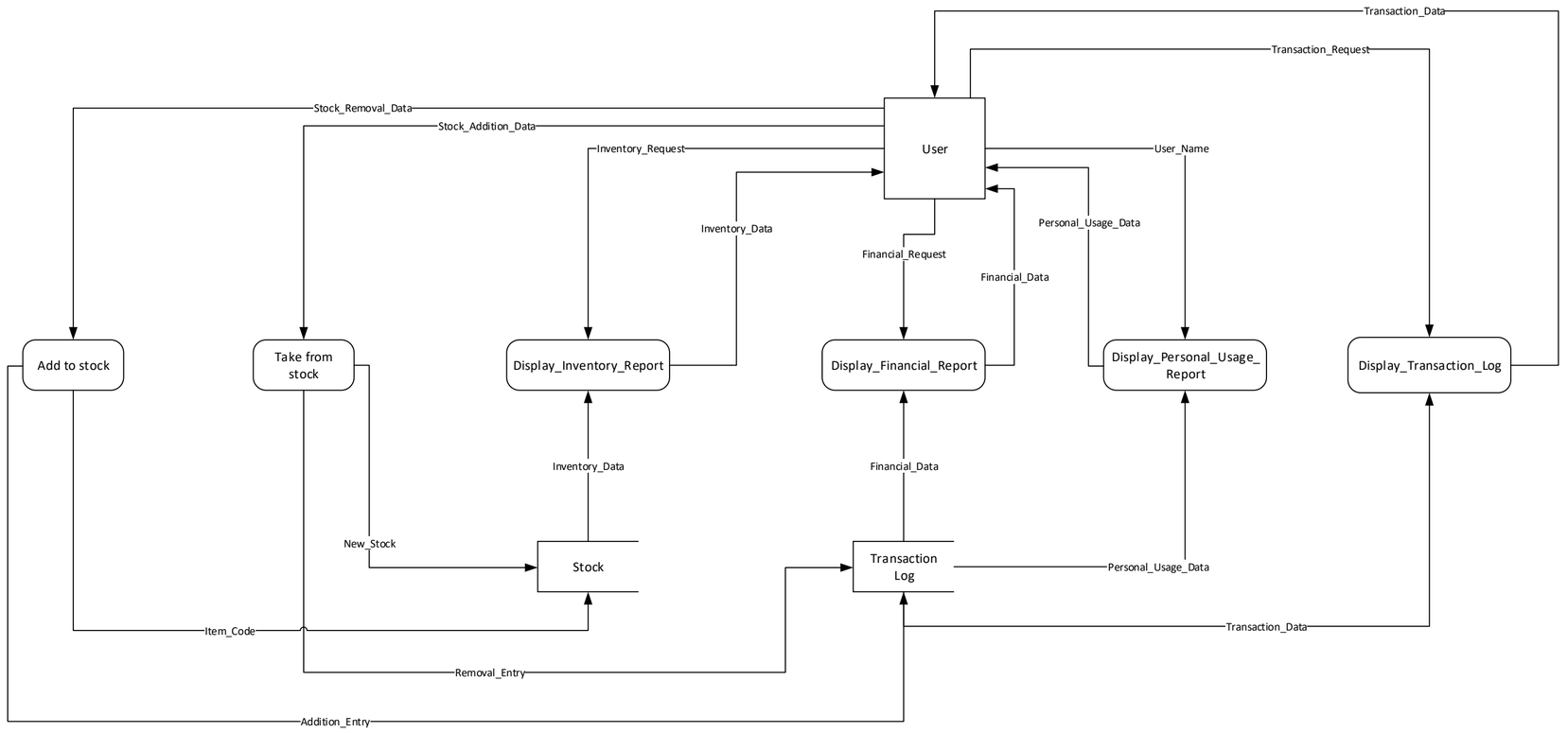


# 2.3 Display Personal Usage Report

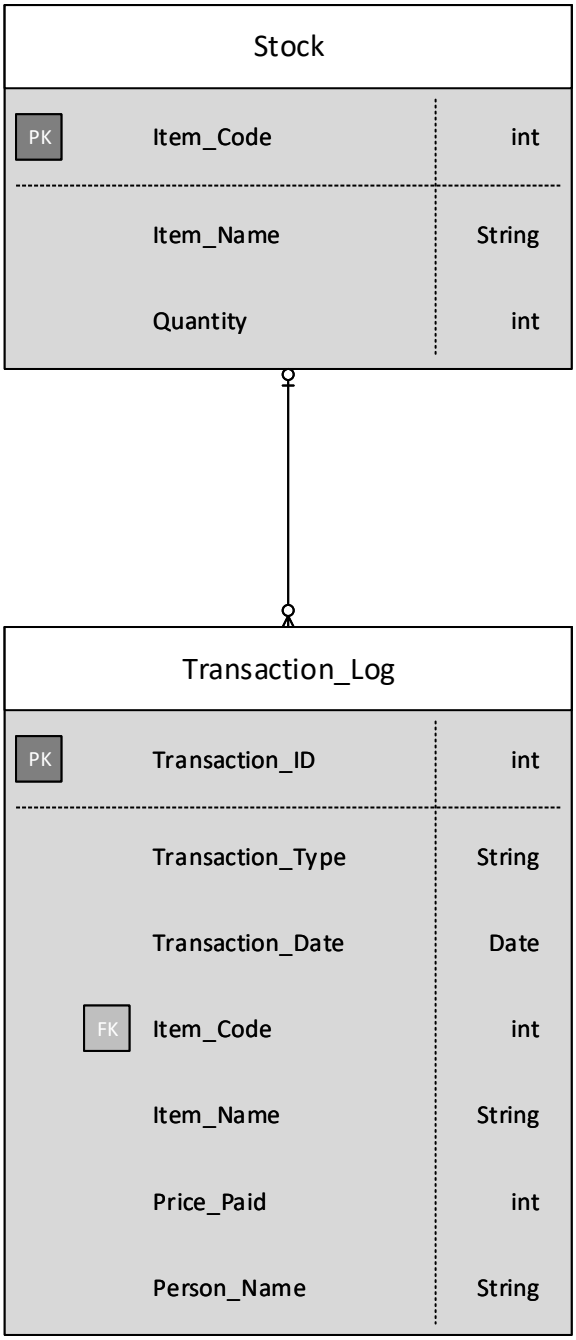


# 2.4 Display Transaction Log




Transaction_Data
Transaction_Request
Stock_Removal_Data
Stock_Addition_Data
Inventory_Request
User
User_Name
Inventory_Data
Personal_Usage_Data
Financial_Request
Financial_Data
Add to stock
Take from stock
Display_Inventory_Report
Display_Financial_Report
Display_Personal_Usage_ Report
Display_Transaction_Log
Inventory_Data
Financial_Data
New_Stock
Stock
Transaction Log
Personal_Usage_Data
Item_Code
Transaction_Data
Removal_Entry
Addition_Entry

# Entity Relationship Diagram



# Data Dictionary

| | |
|---|---|
| Addition_Entry = | Transaction_ID + Transaction_Type + Transaction_Date + Item_Code + Item_Name + Price_Paid |
| Financial_Data = | {Item_Code + Item_Name + Price_Paid} + Total_Expenditure |
| Financial_Request = | Request |
| Inventory_Data = | {Item_Code + Item_Name + Item_Quantity} |
| Inventory_Request = | Request |
| Item = | Item_Code + Item_Name + Item_Quantity |
| Item_Code = | *integer, > 0* |
| Item_Name = | *String, length 20* |
| Item_Quantity = | *integer, > 0* |
| New_Stock = | Item_Code + Item_Name + Item_Quantity |
| Output_Report = | [Inventory_Data \| Financial_Data \| Personal_Usage_Data \| Transaction_Data] |
| Personal_Usage_Data = | {User_Name + Item_Code + Item_Name + Transaction_Date} |
| Price_Paid = | *number, 2 decimal places, > 0* |
| Removal_Entry = | Transaction_ID + Transaction_Type + Transaction_Date + Item_Code + Item_Name + User_Name |
| Request = | *No data, the beginning of a process* |
| Stock = | Item_Code + Item_Name + Item_Quantity |
| Stock_Addition_Data = | Item_Code + Item_Name + Item_Quantity + Price_Paid + Transaction_Date |
| Stock_Input = | [Stock_Removal_Data \| Stock_Addition_Data] |
| Stock_Removal_Data = | Item_Code + User_Name + Transaction_Date |
| Total_Expenditure = | *number, total addition of all Price_Paid of stock, 2 decimal places, > 0* |
| Transaction Log = | {Transaction_ID + Transaction_Type + Transaction_Date + Item_Code + Item_Name + Price_Paid + User_Name} |
| Transaction_Data = | {Transaction_ID + Transaction_Type + Transaction_Date + Item_Code + Item_Name + [Price_Paid \| User_Name]} |
| Transaction_Date = | *Date, current date at time of transaction* |
| Transaction_Entry = | [Removal_Entry \| Addition_Entry] |
| Transaction_ID = | *integer, > 0* |
| Transaction_Request = | Request |
| Transaction_Type = | *["Add" \| "Remove"]* |
| User_Input = | [Stock_Removal_Data \| Stock_Addition_Data \| User_Name] |
| User_Name = | *String, length 20* |
| User_Output = | [Inventory_Data \| Financial_Data \| Personal_Usage_Data \| Transaction_Data] |

```
/*
Add to stock
{
	read in item code, item name, item quantity, price paid, and date
	store the item code, item name and quantity in the list of stock
	store the item code, item name, item quantity, price paid and date in the list of
transactions
}

Remove from stock
{
	read in item code, users name, and date
	if item is in stock
	quantity = quantity - 1
	store the item code, item name, item quantity, users name and date in the list of
transactions, and give the transaction a transaction type of "Add" and an ID equal to the size
of the list.
}

Display inventory report
{
	for the entire list of stock
	{
		output the item code, name, and quantity
	}
}

Display financial report
{
	for the entire list of transactions
	{
		if the transaction type is 'add' then
		output each item code, item name and price paid
		store the price paid in the totalExpenditure variable
	}
	output the total of all price paid values as the total expenditure
}

Display transaction log
{
	for the entire list of transactions
	{
		output the date of the transaction
		if the transaction type is 'add' then
		output the item code, item name and price paid
		if the transaction type is 'remove' then
		output the item code, item name and the users name
	}
}

Display personal usage report
{
```

```
        read in a users name
        for the entire list of transactions
        {
                if the transaction type is 'remove' and the users name is the same as the users
name of the transaction
                output the item code, item name and transaction date
        }
}

*/
```

**Application Modelling**
**Assignment 1 Report**
**Rhys Jones**
COSE40574
Staffordshire University, College Road, Stoke-on-Trent ST4 2DE
Rhyswj94@gmail.com

**Linked Lists:**
Linked lists are useful for inserting into the middle of the list, or getting random access to any elements, also a great implementation if you are unaware of how many items will be in the list, unlike arrays or similar none dynamic collections, which may need to be re-created and have memory copied into them if they grow too big. This however isn't necessary for my program, I do not need to insert into the middle of the list, or get random access to any elements within the list. Whilst I am not aware of how large my lists will be, there are far easier and less complicated dynamic collections available for me to use.

**Trees:**
Trees are excellent for searching for data, however they are fairly complicated to balance and may result in what is essentially a linked list if done incorrectly, and although my program will search for specific data, it doesn't need something as complex as a tree to do it.

**Heaps:**
Heaps are essentially trees, and order elements by priority, nothing in my program will have priority, therefore it is no necessary. Also the previous comments about complexity of trees applies here too.

**Queues:**
Queues are a first in, first out data structure, meaning that the first element into the queue will be the first to be removed. I don't need this functionality in my program, since users will be able to take any item of stock they desire.

**Stacks:**
Stacks are similar to queues, except they work in the opposite order, they're a last in, first out data structure, utilising the pop and push operations, push adding to the top of the stack and pop removing the most recently added element. Once again, users will be able to take any item of stock they desire, and therefore this isn't useful functionality.

**Which data structures I will use:**
I have decided to simply use Vectors, they are easy to use, they are essentially a dynamic array, the memory is handled for you, including destruction of the vector once it goes out of scope (which it will not in my program, but it is worth mentioning). Vectors also store the elements contiguously, meaning the memory locations of elements are all stored alongside each other.

While vectors do have some overhead in terms of their memory allocation (The vector may allow itself to potentially contain 10 more elements than it actually has)

I will also use structs, to collect variables together in a similar fashion to OO programming. The vectors will hold these structs.