Project - Business register and database

Group 23: Jesper Saxer, Sebastian Lhådö , Grim Moström

February 27, 2017

1 Introduction

Conducting business on a professional level can be overwhelming without the proper tools and systems. It is essential that business actions and cash flows are tracked. Without a functional system it can be complicated to trace how eventual mistakes or disadvantageous business actions are affecting the business and even worse, to track inventories and cash flows during the declaration of taxes. A proper business tool such as a Business register can be helpful to administrate multiple users in the business in order to keep track of actions and cash flows. This project aims to create a complete system that keeps track of stock, purchases, identification of users and most importantly the in and outgoing cash flows. In addition it aims to provide a customer based cart system in order to assist the customer, improving the user experience, guiding the customer in the decision and payment process.

The project was conducted during a limited time span, which in some aspects has hasted the process, affecting the product. The project resulted in a complete business system with the desired functions. However, some flaws are included, such as limited test cases. The general quality of the product is correlating to the project goals, but a more vast timespan had improved the visual interaction and more test cases.

The project has contributed well to the learning of functional programming and use of functions. Even some aspects of visual programming has been learnt. Thus, giving increased experience of both front- and backend program development.

Contents

2 Illustration of the Business system

In order to deliver a complete understanding of the business system, we have chosen to map our system in graphical context. This way we can guide the reader through the build of the system and it's hidden functions.
The following graphical illustration is a brief summary of the main files of the business system.

3 Flow chart

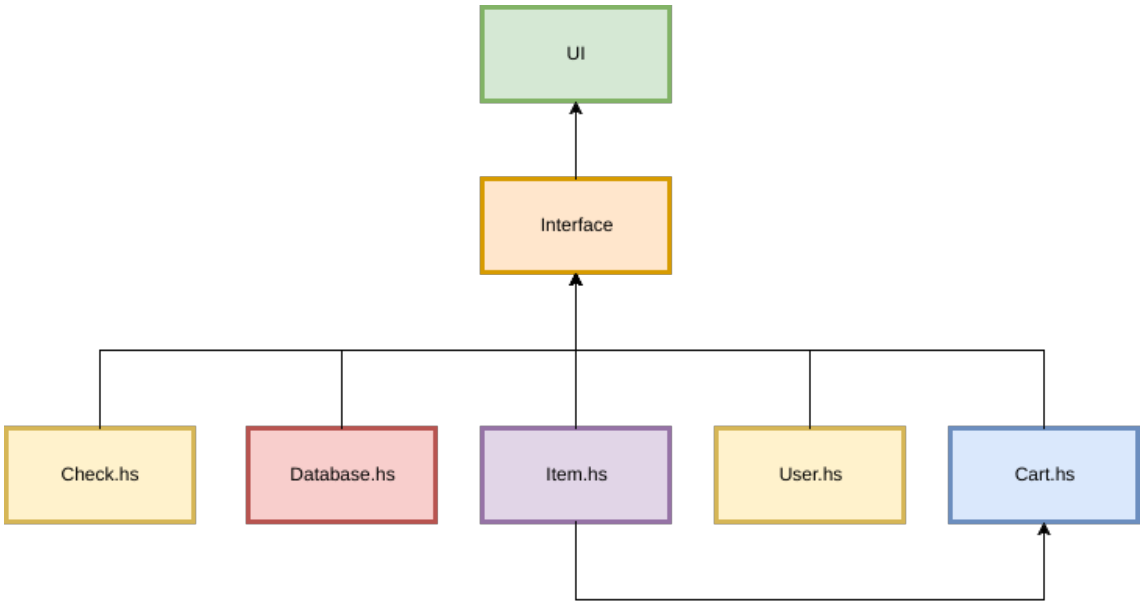

Figure 1: Flow chart over the shop

4 Branch declarations

4.1 Database

The database administrates and stores the main functions. Through the database we can store the inventory, create users, and receive purchases or changes in stock. The database is functioning as an information bank in which information can be edited such that it fits the needs of the administrator, thus correlating to the reality of the business process in practice. Without the database, it is impossible to store the flow of information that is added into the system. This implies that it is an essential piece of the complete business system.
Please read further in Branch & Data structure specification X:X for further information

4.2 User

When registering business-transactions it is important for the system to determine whether the user is an administrator, employee or a customer in order to offer the right services and properties. The system is designed such that it offers an administrator the full access to change all variables in the system, including the users available to administer the system. A customer on the other hand only has access to information needed to purchase goods, eg. stock, price and products.

Please read further in Branch & Data structure specification X:X for further information

4.3 Item

One of the most essential parts of the business system is the Item specification. The items may be changed or modified as the stock or price changes. Further you upload one of these changes to the database in order to update price and inventory. An item holds all the information needed to specify the products values, such as the stock, EAN code, price, and name.

Please read further in Branch & Data structure specification X:X for further information

4.4 Cart

The cart is a temporary list of chosen products by the User. This cart holds products in order to sum up the list of chosen products and giving the user an overview of its choices. From here it is easy to navigate in order to add more products or remove them. This way the usability of the system increases since the customer can iterate the shopping list while being able to keep on shopping.

Please read further in Branch & Data structure specification X:X for further information

4.5 Interface

4.5.1 Functional Interface

The interface is divided in two levels of functionality. The function interface, and the graphical menu. Interface functions is our gathering channel from our systems flow chart. This is the channel in which the User interacts with the complete business system. By hiding all help functions in the back end systems you can easily communicate in a front-end manor to the user, simplifying the systems usability, improving the user experience. This way we enable the customer to handle the system without any further practical guidance than what is offered by the business system itself and its users guide.

Please read further in Branch & Data structure specification X:X for further information

4.5.2 Graphical Interface

The graphical interface is a visualization of the interface functions to get the complete front-end product. With a graphical interface the User is further being guided in the useage of the system. This is the final level of the product, surfacing to the interaction with the User.

Please read further in Branch & Data structure specification specification X:X for further information

5 Branch & Data structure specification

5.1 Database

Database.hs is the way we choose to represent a Database in our Cashier-System. Our Database.hs has its own datatypes which is defined the following preset:

```
type Id = Int -- Ean for item, userId for users
type Database a = [(a,Id)]
```

The identity of a user is defined by a User code, based on integers. These numbers are unique for each User and makes it easy to identify two different users with similar names. The identity of Items are identified by a matching EAN code, scanned on the products back. This serial code defines the sort of product and is unique for the specific type of product specified.
The data type Database a = [(a,Id)] is polymorphic. This implicates that it is non type specific. This way, Database can hold both user identities and items. This is fitting to the needs of information storage to be held in the business system. In summarization the Database takes one polymorphic argument and returns a tuple containing a product, or user together with an ID.

The following functions are reachable if Database.hs is imported

```
empty :: Database a
deleteWithID :: Id -> Database a -> Database a
delete :: Eq a => a -> Database a -> Database a
insert :: a -> Id -> Database a -> Database a
grab :: Eq a => a -> Database a -> a
grabWithID :: Id -> Database a -> a
```

Notation: In the structure of the function specifications for the functions above. You can see a collection of types with an arrow pointing right. The value presented after the last arrow indicates what the complete function returns. The other arrows is simply separating arguments that the function takes while being called.

The same fundamental constructional frame of functions is maintained through the whole project. For specified code, please read attached file.

5.2 User

User.hs is the representation of User identities in the Cashier-System. Our User.hs has its own data structure which is defined in the following way.

```
type Name = String
type Id = Int
type Wallet = Int
type Spent = Int
type IsAdmin = Bool

data User = User Name Id Wallet Spent IsAdmin deriving (Show, Eq)
```

Every User takes the following arguments: Name is represented by a string of characters bound between two " " symbols. Example: "Name" Id is represented by a series of numbers, identifying the user by unique numerical identification by integers. Example: Id "Sebastian" = 1234 Wallet: The Wallet is a data type representing the amount of money a user have stored in the system. This virtual money represented by integers is later at disposal on the products put into the system. Spent represents the amount of money a user have spent in our system. The purpose of this datatype is to track cash flows in the system and follow up on the systems revenues. IsAdmin keeps tracks the user itself have admin properties or not. This is very important in order to keep the business systems integrity level. By having administrator properties, the system disables customers access to sensitive data and essential product maintenance functions.

The following functions are reachable if User.hs is imported

```
newUser       :: Name  -> Id  -> Wallet -> Spent -> IsAdmin -> User
setName       :: Name  -> User -> User
getName       :: User  -> Name
setId         :: Id    -> User -> User
getId         :: User  -> Id
fillWallet    :: Wallet -> User -> User
removeWallet  :: Wallet -> User -> User
removeSpent   :: User  -> User
makeAdmin     :: User  -> User
removeAdmin   :: User  -> User
getAdminStatus :: User -> Bool
addSpent      :: Spent -> User -> User
reduceSpent :: Spent -> User -> User
removeSpent :: User -> User
getWallet     :: User  -> Wallet
clearWallet   :: User  -> User
```

These data types are all related to preset functions explained in previous section. The data types are all enabling changes in the main datatype of its kind.

5.3 item

Item.hs is the way we chose to represent a product of in the Cashier-System. Our Item.hs has its own datastructure which is defined in the following way.

```
type Name = String
type Ean  = Int
type Price = Int
type Stock = Int
data Item = Item Name Ean Price Stock deriving (Show,Eq)
```

This means that every item has the following Arguments.
Name, simply the name in the form of a string.
Ean, the barcode that is on the item itself.
Price, the price we want to take for the item.
Stock, the amount of items we have in storage.
The following functions is reachable if Item.hs is imported.

```
createItem :: Name -> Ean -> Price -> Stock -> Item
setName :: Name -> Item -> Item
getName :: Item -> Name
setEan :: Ean -> Item -> Item
getEan :: Item -> Ean
setPrice :: Price -> Item -> Item
getPrice :: Item -> Price
addToStock :: Stock -> Item -> Item
removeFromStock :: Stock -> Item -> Item
replaceStock :: Stock -> Item -> Item
getStock :: Item -> Stock
```

An example would be setName "Coca-Cola" (Item "cola" 1234 10 0) -> (Item "Cola-Cola" 1234 10 0)
Here you can see that setName is being called with two arguments, first a Name and secondly an Item and with these two arguments it returns a new Item. For specified code, please read attached file.

5.4 Cart

In order to enable customers to interact and store their product choices in the system, we have created the branch Cart. Cart.hs imports both Item.hs and Database.hs which is stated through the flowchart.
Cart has only one datatype which is declared in the following way:

```
type Cart = [Item]
```

The datatype Cart is simply put the linked list of items. This is to create a simple structure where we can easily get an overview of chosen products.
Cart.hs is imported and makes the following functions reachable.

```
empty :: Cart
addToCart :: Item -> Cart -> Cart
removeFromCart :: Item -> Cart -> Cart
removeFromCartAUX :: Item -> Cart -> Cart -> Cart
calculatePrice :: Cart -> Int
getFirst :: Cart -> (Item,Cart)
```

All functions specified aims to make Cart edible for Users. The names of functions are relatively self explanatory. As mentioned in previous branch and data specifications, the types put in are contained among the first arrows, whilst the last output is specified last in each function. For specified code, please read attached file.

5.5 Interface

5.5.1 Functional Interface

The functional interface is the systems most advanced branch as it collects and handles all other branches as a root of the system.Interface.hs handles all branches and is imported

and called through different functions. To make it more readable we have added headings in order to understand the different specifications. For specified code, please read attached file.

```
imports
import Item
import User
import Cart
import Database
import Test.HUnit
```

As mentioned, you can see that all branches are imported into the functional interface. One addition to this is our automated tests conducted through test.Hunit. To read more about debugging and testing, please read chapter about testing and debugging. For specified code, please read attached file.

```
Data types

type Name    = String
type Ean     = Int
type Price   = Int
type Stock   = Int
type Id      = Int
type Wallet  = Int
type Spent   = Int
type IsAdmin = Bool
data Interface = Interface User (Database User) (Database Item) Cart deriving (Show,Eq)
```

Many of these data types are mentioned previously and are relatively self explanatory in its context. They are empirically needed to be restated to increase readability of code. Each branch specifies data types used in the program. This way the usability and quality of the system increases significantly for future edits. For specified code, please read attached file.

```
Interface
newInterface :: User -> Database User -> Database Item -> Cart -> Interface
```

new interface specifies what types the interface takes in the system. The system is as you can see built of the other branches, calling user, databases, cart and creates the interface. For specified code, please read attached file.

```
User administration
getUser :: Interface -> User
createUser :: Name -> Id -> Wallet -> Spent -> IsAdmin -> Interface -> Interface
removeUser :: User -> Interface -> Interface
findUser :: Id -> Interface -> User
```

The user administration calls all the functions needed to handle users from the User branch. This way the interface simplifies the user administration as it is gathered and called with simple functions. For specified code, please read attached file.

```
setUserName :: Name -> User -> Interface -> Interface
setUserId :: Id -> User -> Interface -> Interface
makeUserAdmin:: User -> Interface -> Interface
removeUserAdmin:: User -> Interface -> Interface
```

```
Wallet
getWallet :: User -> Interface -> Wallet
fillWallet :: Int -> User -> Interface -> Interface
reduceWallet :: Int -> User -> Interface -> Interface
clearWallet :: Int -> User -> Interface -> Interface
```

```
createItem :: Name -> Ean -> Price -> Stock -> Interface -> Interface
removeItem :: Item -> Interface -> Interface
findItem :: Ean -> Interface -> Item
```

The interface calls all wallet functions that is needed to adjust or create wallet values. For specified code, please read attached file.

```
stock handling
addToStock :: Int -> Item -> Interface -> Interface
removeFromStock :: Int -> Item -> Interface -> Interface
replaceStock :: Int -> Item -> Interface -> Interface
```

The interface calls all stock functions that is needed to adjust or create stock values. For specified code, please read attached file.

```
Cart Handling
addToCart :: Item -> Interface -> Interface
removeFromCart :: Item -> Interface -> Interface
buy :: Interface -> Interface
```

The interface calls all cart functions that is needed to adjust or create temporary cart values.

5.5.2 Graphical Interface

The graphical interface file Menu.hs withholds all information regarding the terminal actions that the user is interacting with to purchase products. This implicates that our following Users guide will mainly be concerning this file. To make a functional menu system we have had to make some additional imports to the file:

```
import Interface
import System.Process
```

The functional interface file is imported in order to make its content implemented in the users interface menu. Its code is rather large but its usability is quite simple. System.Process is imported simply as a standard help function in order to be able to clear the terminal and increasing its usability.
To understand the graphical interface, please read Users guide and attached file.

6 User Guide

This system requires that you have a terminal and that you can open it. When you have located your terminal you want to find the directory where the folder Cashier-System is stored in our case it's stored in:
/Users/Jesper/Documents/uppsalaDV/pkd/project/Cashier-System therefore to locate it in the terminal we have to write:
cd Documents/uppsalaDV/pkd/project/Cashier-System/ .
You should now have your terminal looking something like this.

7 User cases

8 Algorithms

8.1 Recursive algorithms

The business system is built on various functions and mathematical algorithms. One of the most common function types applied is recursive algorithms, or recursion. We specify recursive algorithms as a function that takes its own values to generate an infinite loop until the desired answer is reached. To apply the recursion it is important that the programmer specifies grid values in which the algorithm should operate, and end cases onto the recursion returns values. (http:
www.dictionary.com/browse/recursion)Retrieved: 2017-02-20

```
1 grabWithId.
2 grabWithId :: Id -> Database a -> a
3 grabWithId i [] = error "not in our database"
4 grabWithId i (x:xs)
5   | i == (snd x) = fst x
6   | otherwise = grabWithId i xs
```

grabWithId initially takes an ID from and the database it's supposed to search. Further it makes fault values if the database is empty. What the algorithm then does is using recursion to walk through the list of users searching for a match to the identity code. for each comparison made it either goes to the next value, or gets a match and returns the User holding the right identification. If the list is compared completely without result it interprets the list as empty and returns an error. namely "not in our database".

This way we can see a typical recursion example where the algorithm uses itself to walk through the information in order to return a desired value or result. To understand Recursions further, please read attatched pre- and post conditions on applied functions in attached file.

8.2 Pattern matching

in our business system we are applying pattern matching functions. The specification of the function type defines by giving the computer a given sequence, or pattern in which it matches the functions definitions. By giving the computer a parameter, or matching datatype to what we want to achieve, it simply replaces the original value or adds a new one.(https:
www.haskell.org/tutorial/patterns.html) To exemplify this we have attached an example out of our own code of a pattern matching function.

```
setName :: Name -> Item -> Item
setName x (Item name ean price stock) = Item x ean price  stock
```

setName is hands on in its command as it takes an input that will replace the x in the fitting pattern of an item. for example in setName it takes a String and an Item, onto it wants the user to clarify what String of characters it wants to declare as the items name. In application setName "coca-Pepsi" returns an item with the name "coca-Pepsi" with attatched specifications such as the data type Item is supposed to hold. eg. (Name, Ean, Price, Stock). To understand Pattern matching further, please read pre- and post conditions on applied functions in attached file.

9 Tests & Debugging

We have applied automated tests by the module HUnit.
Hunit is a testing framework for the programming language Haskell. The methodology of the framework is to be able to easily create, and edit tests that are automatically conducted by the program in order find errors or problems with the code faster and intelligible (https: hackage.haskell.org/package/HUnit)
Please read further in attached file for all HUnit tests.
The ambition to this project is to have at least one automated test for each function and getting 0 errors, or faults in the program. However, we believe that this might be insufficient to account for complete functionality of the program.

10 Shortcomings of the program

10.1 Test cases

Due to the time frame of the project we have not satisfied the complete need for test cases throughout the whole system. However, we have conducted some tests. We are aware that they are not covering all bases. This implies that hidden faults are possible to interrupt the functionality of the system. However, so far we have not encountered any non-expected errors in the program. If more time was offered we would have conducted more tests on the system in order to cover the complete functionality of all functions.

10.2 Run time cost

The business system is currently handling the search and action based functions in an non-effective way. This is because the run-time cost of functions generally are T:(n) = n, which can be defined as linear run time cost . This means that most actions and recursive algorithms are processed one unit or value at a time. Thus, making the system slower than necessary. However, since haskell is generally processing values fast and the database is currently handling small amounts of information, it is not recognisable at this development state and is therefore not a priority in the project.

10.3 Advancement time frame

As many projects, This business system project was initially striving to obtain a complete high quality business register program with a highly scaleable framework. As we completed the base-platform for the system we discovered that only a few additional functions was able to be done in time and we decided to focus more on the quality of the surrounding project issues other than expanding the systems functionality. Thus, the goal was obtained to complete a scaleable project. However we believe that more time would benefit the projects usability and quality.

10.4 Usability graphical framework

We are all agreeing that the system would benefit from a more graphical interface. If the system could access a more usable interface directed to the common non-technical user the quality of the program would increase significantly.