Danny's Pizza Place

Daily Reporting Design

Version 0.7

1. Introduction

1.1. System Overview

This short design document outlines the design for the daily reporting system designed for Heimdahl's Pizza place. Given the limited scope of the system, the design is purposely kept rather sketchy and does not include all the details included in a design for a more complex system.

1.2. Design Objective

The objective of this design document is twofold:

- To provide enough information for a novice C++ programmer to implement the system.
- To provide enough detail to develop a comprehensive test suite before the actual code is available.

1.3. References

The Pizza Place Requirements available from the web page for CSci 5802.

2. Glossary

There are no new terms introduced in this document.

3. Use Cases

TBD

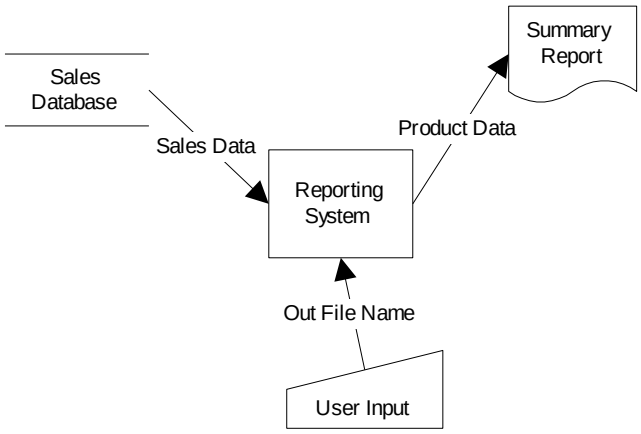
4. Design Overview

4.1. Introduction

The design of this simple system is a hybrid design with the main processing defined through functional decomposition and the abstract data types representing the sales data and sales reports designed in an OO fashion.

4.2. System Architecture

The architecture is very basic. The software will extract the sales data from the sales database and produce an output file.



4.3. System Interfaces

There is only one interface to another software system; the sales database.

4.3.1. Sales Database Interface

The interface to the Sales Database is not known at this time and this must be taken into account in the design an implementation of the report generator.

4.4. Constraints and Assumptions

We make the following assumptions:

- The performance of the implementation is not an issue given the limited processing and limited amount of data.
- If the user issues a command with an output file that already exists, that output file can be safely overwritten.
- The data provided from the database will have been checked for compliance with any constraints. That is, the reporting system does not have to perform any data analysis.

5. Design

5.1. Introduction

The design is split into two sections; (1) the functional decomposition of the reporting system and (2) the design of the abstract data types on which the reporting system operates.

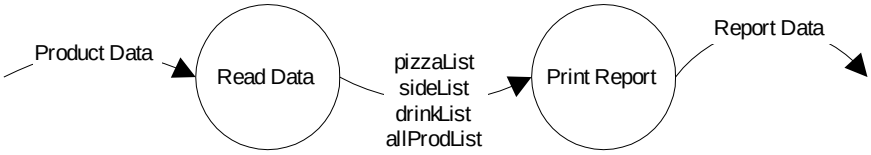
5.2. Data Flow Diagram

5.2.1. Context Diagram

At the context level, the reporting system will read the Product Data from the Product Database. Currently, the format of the Product Data is unknown.

The Reporting System will produce Report Data in the Sales Report. The Sales Report shall be a text file.

5.2.2. Level 1



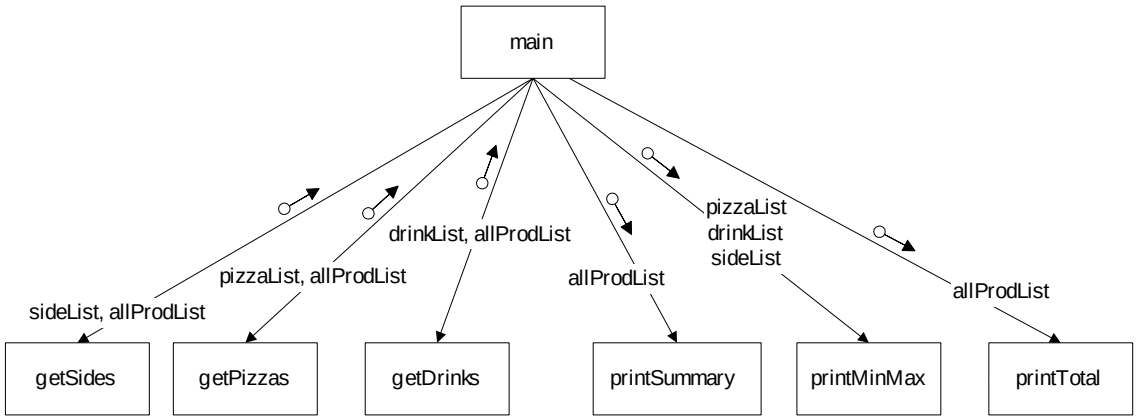
The Read Data process is responsible for extracting the data from the database and producing four lists of products. Read Data will produce one list for each product line and one list containing all the products. The lists are all of the type TContainer defined later in this design document.

The Print Report process will traverse each list and produce the appropriate output on the text file.

5.3. Functional Decomposition

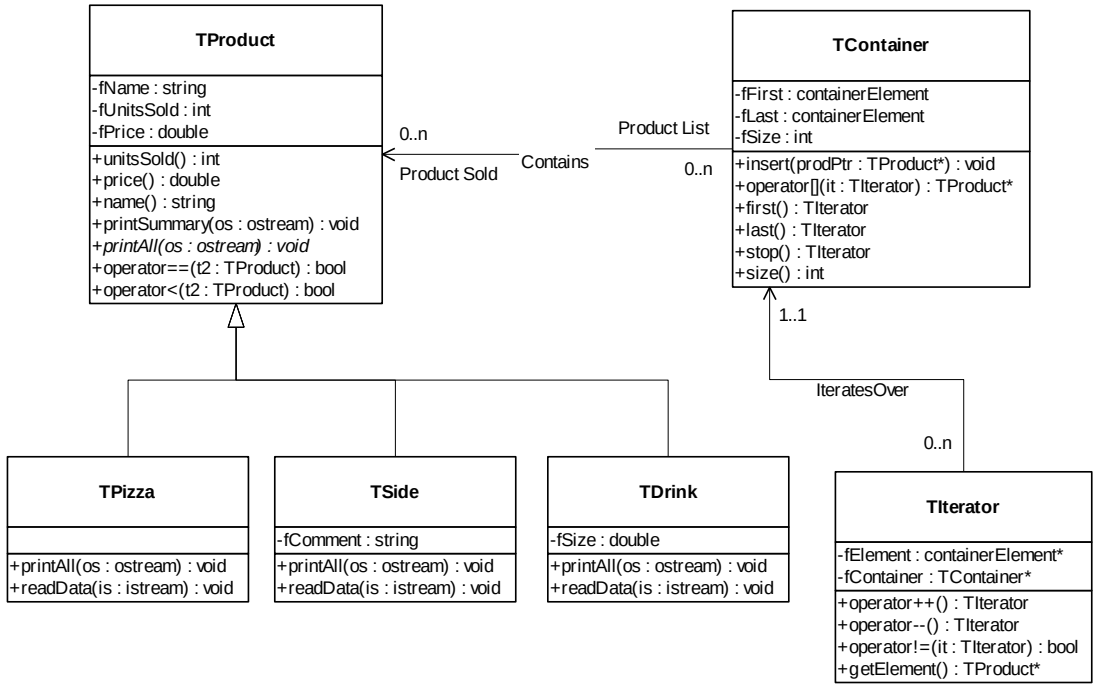
The program will be very simple and all processing will be governed from the main function. The main function will first invoke the get routines to extract the data from the database and then invoke the print routines to produce the report.

The detailed definition of each function is given in Section 6.1.



5.4. Abstract Data Types

The data types needed to keep track of the sales data are designed using OO techniques and UML.



The TProduct hierarchy is used to keep track of the products in the sales database. We are using inheritance so that we can treat all products in a uniform way. With this structure, we can insert any product into the various lists and treat them all as products. The only difference between the different products is the way we print all the information since they contain slightly different data. This is captured through the abstract function printAll that must

be implemented for each class derived from TProduct. The detailed definition of the classes can be found in Section 6.2.

The TContainer class will be used to implement the various lists we need to keep track of the products. The class will maintain a sorted list (based on product name) of the products inserted in the list. The list will contain pointers to the products so that we can insert the same product in several lists without duplication. This may not be the most elegant solution, but it will do for this simple system.

TIterator is a helper class used to iterate over TContainer. The iterator class is used the same way an integer would be used to iterate over an array. This is a common way of implementing access into a container class and is extensively used in the STL. Again, the detailed definition of the classes can be found in Section 6.2.

6. Detailed Function and Class Definitions

6.1. Function Definitions

6.1.1. Function main

Signature: int main(int argc, char* argv[])

Parameters: int argc: the number of words on the command line (in)

char* argv[]: string array containing command line arguments (in)

Return Value: int: Indicates the success of the computation, 0 means OK

Processing: main will validate the output file name from the command line. main will open the database for reading and call the read functions to extract the data. main will then call the print functions to print the report. Finally, main will close all databases and files.

Functions Called: getPizzas, getSides, getDrinks, printSummary, printMinMax, printTotal

6.1.2. Function getPizza

Note: This function shall in the future read the data from the DBMS. The database is not ready at this time and we do not know what it will look like. Thus, we will for now read from a file.

Signature: void getPizzas(ifstream &is, TContainer &pizzaList, TContainer &allProdList);

Parameters: ifstream &is: The input stream with the pizza information(in)

TContainer &pizzaList: The list containing all pizzas (out)

TContainer &allProdList: The list containing all products sold (out)

Return Value: NA

Processing: The function will go through is and extract the pizza information until the end of file is reached. We assume the file is formatted correctly and no read errors will occur. Note that this function is used only for testing purposes. It will be replaces with a new one reading from the database.

Functions Called: TPizza::insert, TContainer::insert, TPizza::readData

6.1.3. Function readSides

This function works like readPizza except it will read the sides.

6.1.4. Function readDrinks

This function works like readPizza except it will read the drinks.

6.1.5. Function printMinMax

Signature: void printMinMax(ofstream &of, TContainer &prodSet, const char *header);

Parameters: ofstream &of: The output file for the report

TContainer &prodSet: The list of products we want to print

char *header: The name of the product family we are printing

Return Value: NA

Processing: The function will traverse the prodSet and find the best and worst selling products in the list. These two products will then be outputted on the output file with eth appropriate heading.

Functions Called: TProduct::printAll (note that this is a purely virtual method)

6.1.6. Function printSummary

Signature: void printSummary(ofstream &of, const TContainer &prodList);

Parameters: ofstream &of: The output file for the report

TContainer &prodSet: The list of products we want to print

Return Value: NA

Processing: The function will traverse the prodList and print out the information about each product. The function will rely on that the prodList is sorted.

Functions Called: TProduct::printSummary

6.1.7. Function printTotal

Signature: void printTotal(ofstream &of, const TContainer &prodList);

Parameters: ofstream &of: The output file for the report

TContainer &prodSet: The list of products we want to print

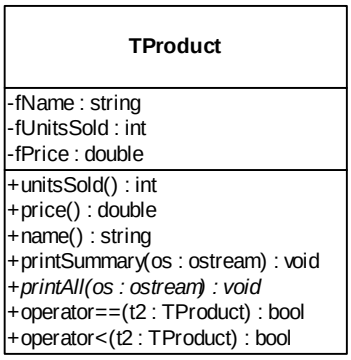
Return Value: NA

Processing: The function will traverse the list and compute the sum of the sales of the products in the list.

Functions Called: TProduct::unitsSold, TProduct::price

6.2. Class Definitions

6.2.1. Class TProduct



Overview: This class stores all the common information for the products on sale in the pizza place. The various products will inherit from this class.

Parent Class: None.

6.2.1.1. Attribute Description

Attribute: fName
Type: string
Description: The name of the product
Constraints: cannot be empty

Attribute: fUnitsSold
Type: int
Description: The number of units sold for this product
Constraints: Must be >=0

Attribute: fPrice
Type: double
Description: The unit price for the product
Constraints: Must be >=0

6.2.1.2. Method Descriptions

6.2.1.2.1. unitsSold

Signature: int unitsSold() const;

Parameters: NA

Return value: The number of units sold

Pre-condition: none

Post-condition: return value is equal to the number of units sold

Attributes read/used: fUnitsSold

Methods called: none

Processing logic: Return fUnitsSold.

6.2.1.2.2. price

Signature: double price() const;

Parameters: NA

Return value: The price of the product

Pre-condition: none

Post-condition: return value is equal to the price of the product

Attributes read/used: fPrice

Methods called: none

Processing logic: Return fPrice.

6.2.1.2.3. name

Signature: string name() const;

Parameters: none

Return value: The name of the product

Pre-condition: none

Post-condition: The return value is the name of the product

Attributes read/used: fName

Methods called: none

Processing logic: Return fName.

6.2.1.2.4. printSummary

Signature: virtual void printSummary(ostream &os) const;

Parameters: os: The output stream we want to print to

Return value: none

Pre-condition: The output stream is a valid ostream

Post-condition: The summary information is printed on the ostream

Attributes read/used: fName,fUnitsSold, fPrice

Methods called: none

Processing logic: The function outputs the name of the product, the number of units sold, the price per unit, and the total value of sales of this product.

6.2.1.2.5. printAll

Signature: virtual void printAll(ostream &os) const = 0;

Parameters: The output stream we want to print to

Return value: none

Pre-condition: The ostream is valid

Post-condition: All available data about the product is available on the ostream

Attributes read/used: NA

Methods called: NA

Processing logic: NA, this method is purely virtual, must be defined in derived classes

6.2.1.2.6. operator==

Signature: bool operator==(const TProduct &t2);

Parameters: The product with which we want to compare

Return value: true if the products are equal

Pre-condition: none

Post-condition: return value is true iff the products are equal

Attributes read/used: fName

Methods called: none

Processing logic: Return true if the names of the products are the same. The name is the only key for a product.

6.2.1.2.7. operator<

Signature: bool operator<(const TProduct &t2);

Parameters: The product with which we want to compare

Return value: true if the parameter product is greater that this product

Pre-condition: none

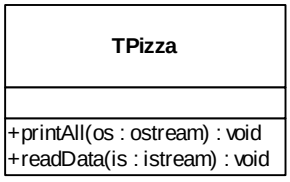
Post-condition: return value is true iff the parameter is greater than this

Attributes read/used: fName

Methods called: none

Processing logic: Return true if the parameter's name is lexicographically greater than the name of this product.

6.2.2. Class TPizza



Overview: This class stores the information for a pizza.

Parent Class: TProduct.

6.2.2.1. Attribute Description

No new attributes defined

6.2.2.2. Method Descriptions

6.2.2.2.1. printAll

Signature: virtual void printAll(ostream &os) const;

Parameters: The ostream we want to print on

Return value: none

Pre-condition: The ostream is valid

Post-condition: All available information regariding the product is printed on the ostream

Attributes read/used: fName, fUnitsSold, fPrice

Methods called: none

Processing logic: Output the information about the product on the ostream.

6.2.2.2.2. readData

Signature: void readData(istream &is);

Parameters: The istream we want to read data from

Return value: none

Pre-condition: The istream is valid and all data in the istream is valid and formatted correctly

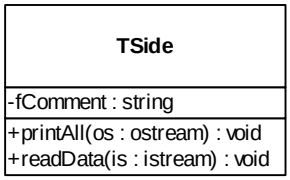
Post-condition: The information from the istream is stored in the proper files in the product

Attributes read/used: fName, fUnitsSold, fPrice

Methods called: none

Processing logic: Store the data in the right field.

6.2.3. Class TSide



Overview: This class stores the information regarding a side order.

Parent Class: TProduct.

6.2.3.1. Attribute Description

Attribute: fComment
Type: string
Description: A comment regarding the side order
Constraints: Must be shorter than 256 characters

6.2.3.2. Method Descriptions

6.2.3.2.1. printAll

Signature: virtual void printAll(ostream &os) const;

Parameters: The ostream we want to print on

Return value: none

Pre-condition: The ostream is valid

Post-condition: All available information regarding the product is printed on the ostream

Attributes read/used: fName, fUnitsSold, fPrice, fComment

Methods called: none

Processing logic: Output the information about the product on the ostream.

6.2.3.2.2. readData

Signature: void readData(istream &is);

Parameters: The istream we want to read data from

Return value: none

Pre-condition: The istream is valid and all data in the istream is valid and formatted correctly

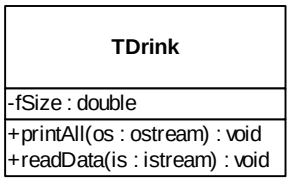
Post-condition: The information from the istream is stored in the proper files in the product

Attributes read/used: fName, fUnitsSold, fPrice, fComment

Methods called: none

Processing logic: Store the data in the right field.

6.2.4. Class TDrink



Overview: This class stores the information regarding a side order.

Parent Class: TProduct.

6.2.4.1. Attribute Description

Attribute: fSize
Type: double
Description: The size of the drink in fluid ounces
Constraints: Must be >=0

6.2.4.2. Method Descriptions

6.2.4.2.1. printAll

Signature: virtual void printAll(ostream &os) const;

Parameters: The ostream we want to print on

Return value: none

Pre-condition: The ostream is valid

Post-condition: All available information regarding the product is printed on the ostream

Attributes read/used: fName, fUnitsSold, fPrice, fSize

Methods called: none

Processing logic: Output the information about the product on the ostream.

6.2.4.2.2. readData

Signature: void readData(istream &is);

Parameters: The istream we want to read data from

Return value: none

Pre-condition: The istream is valid and all data in the istream is valid and formatted correctly

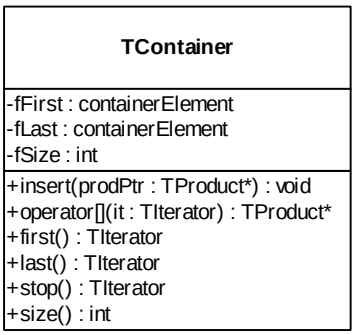
Post-condition: The information from the istream is stored in the proper files in the product

Attributes read/used: fName, fUnitsSold, fPrice, fSize

Methods called: none

Processing logic: Store the data in the right field.

6.2.5. Class TContainer



Overview: This is a container class that will maintain a sorted list of products. The TIterator class is a friend class of TContainer. TIterator is the iteraator for TContainer.

TContainer is made up of a linked list. The elements in the list contain a forward and backward pointer, as well as a pointer to a product (TProduct).

Parent Class: None.

6.2.5.1. Attribute Description

Attribute: fFirst
Type: containerElement*
Description: A pointer to the first element of the container.
Constraints: none

Attribute: fLast
Type: containerElement*
Description: A pointer to the last element of the container.
Constraints: none

Attribute: fSize
Type: int
Description: The number of elements in the container.
Constraints: fSize>=0

6.2.5.2. Method Descriptions

6.2.5.2.1. size

Signature: int size() const;

Parameters: none

Return value: The number of elements in the container

Pre-condition: none

Post-condition: The return value contains the number of elements in the container.

Attributes read/used: fSize

Methods called: none

Processing logic: Return fSize.

6.2.5.2.2. insert

Signature: virtual void insert(const TProduct *prodPtr);

Parameters: A pointer to the product we want to insert into the list

Return value: none

Pre-condition: The parameter is a valid pointer to a TProduct of derived class

Post-condition: The product is inserted in the right place in the container. The container must remain sorted based on the lexicographic order of the product name.

Attributes read/used: fFirst, fLast, fSize

Methods called: none

Processing logic: Traverse the list until we find the right place (all elements before this spot are smaller than the element to be inserted and all elements after this spot are larger). Create a new containerElement and insert in the right spot.

6.2.5.2.3. operator[]

Signature: TProduct* operator[] (const TIterator &it) const;

Parameters: An iterator for this container.

Return value: A pointer to the product at the spot in the container the iterator is referring to.

Pre-condition: The iterator is referring to an element in the container. That is, th eoperator was created for this container and the iterator is within bounds.

Post-condition: The pointer to the element at the point the iterator is referring to is returned.

Attributes read/used: none

Methods called: TIterator::getElement

Processing logic: The iterator can extract the element to which it is referring; use that method.

6.2.5.2.4. first

Signature: TIterator first() const;

Parameters: none.

Return value: An iterator referring to the first element of the container.

Pre-condition: none.

Post-condition: The iterator refers to the first element of the list. If the container is empty, first shall return a special value that is not indexing into the container. This value will be the same as the one returned from TContainer::stop().

Attributes read/used: fFirst

Methods called: TIterator::TIterator

Processing logic: Create an iterator that refers to the first element of the container.

6.2.5.2.5. last

Signature: TIterator last() const;

Parameters: none.

Return value: An iterator referring to the last element of the container.

Pre-condition: none.

Post-condition: The iterator refers to the last element of the list. If the container is empty, last shall return a special value that is not indexing into the container. This value will be the same as the one returned from TContainer::stop().

Attributes read/used: fLast

Methods called: TIterator::TIterator

Processing logic: Create an iterator that refers to the last element of the container.

6.2.5.2.6. last

Signature: TIterator stop() const;

Parameters: none.

Return value: An iterator not referring to any element in the container.

Pre-condition: none.

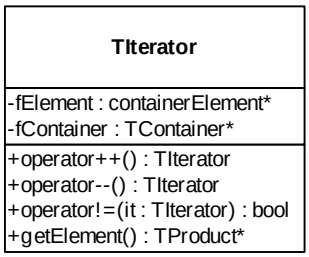
Post-condition: stop shall return a special value that is not indexing into the container.

Attributes read/used: none

Methods called: TIterator::TIterator

Processing logic: Create an iterator that refers to the NULL element.

6.2.6. Class TIterator



Overview: This is the iterator that goes with TContainer. An iterator works the same way an integer does when it comes to arrays. We can increment and decrement an iterator, we can find the iterator for the first element in the container, the last element, and the out of bounds element. WE can also check of two iterators are not the same and we can use the iterator to extract a value from the container.

TIterator is a friend class of TContainer since it needs access to the internals of the TContainer.

Parent Class: None.

6.2.6.1. Attribute Description

Attribute: fElement
Type: containerElement*
Description: A pointer to the element in the container the iterator is currently referring to.
Constraints: NULL indicates the stop iterator

Attribute: fContainer
Type: TContainer*
Description: A pointer to the container this iterator works on.
Constraints: Cannot be NULL

6.2.6.2. Method Descriptions

6.2.6.2.1. operator++

Signature: TIterator &operator++();

Parameters: none

Return value: The iterator updated to refer to the next element in the container.

Pre-condition: this is a valid iterator and not the stop iterator.

Post-condition: The iterator now refers to the next element in the container. If the increment takes off the end of the container, we will return the stop iterator.

Attributes read/used: fContainer, fElement

Methods called: none

Processing logic: Update the iterator to have the fElement point to the next element in the container.

6.2.6.2.2. operator--

Signature: TIterator &operator--();

Parameters: none

Return value: The iterator updated to refer to the previous element in the container.

Pre-condition: this is a valid iterator and not the stop iterator.

Post-condition: The iterator now refers to the previous element in the container. If the decrement takes off the beginning of the container, we will return the stop iterator.

Attributes read/used: fContainer, fElement

Methods called: none

Processing logic: Update the iterator to have the fElement point to the previous element in the container.

6.2.6.2.3. operator!=

Signature: bool operator!=(const TIterator &it);

Parameters: The iterator with which we are comparing

Return value: true if the iterators are not the same

Pre-condition: both iterators are valid iterators.

Post-condition: Return true iff the iterators are the same

Attributes read/used: fContainer, fElement

Methods called: none

Processing logic: Compare fElement and fContainer between this and the parameter. If either one is not the same, return false.

6.2.6.2.4. getElement

Signature: TProduct* getElement() const;

Parameters: no

Return value: A pointer to which the iterator is referring to.

Pre-condition: This is a valid iterator and it is not the stop iterator.

Post-condition: Return the pointer to the product contained in the containerElement fElement is referring to

Attributes read/used: fElement

Methods called: none

Processing logic: Return fElement->prodPtr.

7. Test Input Files

The following three sections contain the input format for the test files. We will experiment with files until the DBMS folks get their job done. The files will be named pizza, sides, and drinks.

7.1. Input Format (pizza)

```
Sausage-large
10.95
8
Sausage-small
4.95
1
Sausage-medium
7.95
10
Calzone-large
12.95
4
Calzone-medium
10.95
2
Calzone-small
7.50
7
Pepperoni-large
9.95
3
Spinach-large
12.95
1
Anchovy-large
11.50
0
Cheese-medium
5.50
12
Cheese-large
8.50
7
```

7.2. Input Format (sides)

```
Fries-large
Remember to salt a lot
2.35
23
OnionRings-small
Medium salt
1.45
12
Fries-small
Extra oily
0.99
10
OnionRings-large
Medium salt
1.95
8
HashBrowns

1.45
4
```

7.3. Input Format (drinks)

```
Sprite-medium
20
1.85
0
Sprite-small
12
1.05
11
Coke-large
32
2.05
20
Coke-medium
20
1.85
10
Coke-small
12
1.05
23
Water-large
32
1.05
9
Coffee
16
0.95
33
```

8. Output Format (Suggested)

The formatting of the report may leave a little to be desired, but we are not going to be too critical.

```
Summary report from Heimdahl's pizza place
-------------------------------------------

Product:           #Sold:  Total:
Anchovy-large         0   0
Calzone-large         4   51.8
Calzone-medium        2   21.9
Calzone-small         7   52.5
Cheese-large          7   59.5
Cheese-medium         12  66
Coffee                33  31.35
Coke-large            20  41
Coke-medium           10  18.5
Coke-small            23  24.15
Fries-large*          23  54.05
Fries-small*          10  9.9
OnionRings-large*     8   15.6
OnionRings-small*     12  17.4
Pepperoni-large       3   29.85
Sausage-large         8   87.6
Sausage-medium        10  79.5
Sausage-small         1   4.95
Spinach-large         1   12.95
Sprite-medium         0   0
Sprite-small          11  11.55
Water-large           9   9.45


PIZZA:
Worst Selling:
Anchovy-large
11.5
0

Best Selling:
Cheese-medium
5.5
12


SIDE:
Worst Selling:
OnionRings-large*
Medium salt*
1.95
8
```

```
Best Selling:
Fries-large*
Remember to salt a lot*
2.35
23


DRINK:
Worst Selling:
Sprite-medium
20
1.85
0

Best Selling:
Coffee
16
0.95
33



Total Sales:  699.5
```