

E_Russell_P151512_OOPS

by Edward Russell

FILE	2659_EDWARD_RUSSELL_E_RUSSELL_P151512_OOPS_108537_1350600062.DOCX		
TIME SUBMITTED	21-MAR-2018 01:43AM (UTC+0000)	WORD COUNT	1361
SUBMISSION ID	85185534	CHARACTER COUNT	9195

Object Orientated Programming

Development of a manufacturing system simulation.

Jaymie Phillips

Ted Russell
3-17-2018

Contents

Introduction.....	2
Specifications.....	2
Theory and application	3
Classes	3
Class Inheritance and overloading.....	3
Objects.....	4
Composition	4
Aggregation.....	5
File access	5
Patterns.....	6
Structural Diagram	7
Behavioural Diagram.....	7
Bibliography	11

Introduction

The assignment was to create a prototype computer system for a company that sells electrical components, which it manufactures itself. These components include resistors, capacitors, transistors, diodes and inductors. The prototype software should be designed according to the specifications given, these have been detailed in the specification section. The creation of this software should show sufficient understanding of object orientated programming techniques. These techniques and the unified modelling language (UML) diagrams will be outlined and analysed below.

Specifications

The program is required to have the following features:

- Users
 - Users should be able to create an account.
 - These accounts should be distinguishable as personal, business or admin accounts.
 - Users should be able to log into their account.
- Stock visibility and monitoring
 - Stock table viewable by all user types.
 - Price editing options available only to administration.
 - The ability to restock components which are low, this option is only available to administration.
- Sales
 - Personal and Business users should be able to make purchases.
 - The description, quantities and prices at time of purchase should all be printed to a receipt specific to that customer.
 - The receipt should also contain the final price including any and all applicable discounts.
 - These sales should decrement the stock table.

Theory and application

Many different object orientated programming methods were implemented during the design of the prototype program, these are outline below with examples of where and how they were implemented within the code.

Classes

A class is considered to be a blueprint for a type of data [1]. It defines what an object derived from that class will consist of and what can be done to that object. Within a class the data members are defined. These can be defined in three sub-sections; public, private and protected. These sub-sections determine if they can be accessed from outside the class but within the scope of the object. If a data member is defined in public, it accessible by any object or function that is able to access that class. If the data member is defined in protected it is only accessible to classes that inherit from the class in which it is defined. If the data member is defined in private, it is only accessible to that class. A class requires both a .cpp file and a .h file. One example of a class implemented within the program is the component class. Figure 1 shows the .cpp file whilst figure 2 shows the .h file of the class component. In the .h file the data members for the class are defined, in the .cpp file the constructor and destructor are set.

```
1 #include "Component.h"
2
3
4
5 Component::Component()
6 {
7 }
8
9 Component::Component(char* componentId)
10 {
11     componentid = componentId;
12 }
13
14
15 Component::~Component()
16 {
17 }
18
19 char* Component::GetComponentId()
20 {
21     return componentid;
22 }
23
24 void Component::SetComponentId(char* componentId)
25 {
26     componentid = componentId;
27 }
28
29
```

Figure 1 Shows the .cpp file of the Component class.

```
1 #ifndef COMPONENT_H
2 #define COMPONENT_H
3 class Component
4 {
5 private:
6     char* componentid;
7
8 public:
9     Component();
10    Component(char* componentId); // An example of Overloading.
11
12    char* GetComponentId();
13    void SetComponentId(char* componentId);
14
15    ~Component();
16 };
17 #endif
18
```

Figure 2 Shows the .h file of the Component class.

Class Inheritance and overloading

It is possible for one or more classes to inherit from another class, known as the base class. This makes it possible for one data member to be defined in one class and then used by all of the classes that derive from that base class. This allows for faster implementation of code and more efficient code as it is being reused by multiple classes. In the program designed by the author the component class (shows in figures 1 and 2) is the base class, one of the

classes that inherits from the component class is the resistor class shown in figures 3 and 4.

```
#include "Component.h"

class Resistor: public Component // An example of Inheritance.
{
private:
public:
    Resistor() : Component("") {};
    Resistor(char* componentId) : Component(componentId) {}; // An example of overloading on the resistor side.
    ~Resistor();
};
```

Figure 3 Shows the .h file of the resistor class.

As can be seen in figure 3, the resistor class is inheriting the data member componentId from the component class through overloading. At this point in the code the value of that data type is simple "", in the stockTable function however the overload is completed as seen in figure 5. This is where the overloaded data member componentId is given the value "RES01\t\t" for the resistor through the GetComponentId() function originally declared in the Component.cpp file.

```
1 #include "Resistor.h"
2 #include "Component.h"
3
4
5 Resistor::~Resistor()
6 {
7 }
8
9
10
```

Figure 4 Shows the .cpp file of the resistor class.

Objects

An object is created from a class, with the class providing the blueprint.

The declaration of an

object occurs in the main function of the program, one such declaration is shown in figure 5 and again more clearly in figure 6.

```
Resistor* resistor = new Resistor("RES01\t\t ");
cout << resistor->GetComponentId() << resD << "\t"
```

Figure 5 Shows the final stage of the overload shown in figures 3 and 4.

```
Resistor* resistor
```

Figure 6 Shows the declaration of the object Resistor of type Resistor.

Composition

Composition can be viewed as a "contains" or "has a" relationship between two objects, for example a resistor and tinned copper wire. A resistor contains tinned copper wire within it. This can be viewed in figure 7. Within

composition the object (in this case the resistor) is the reason for the existence of the parts (the tinned copper wire) ergo if the resistor is destroyed, the tinned copper wire is also destroyed. A part object (tinned copper wire) can only exist within this one object.[2]

```
1 #include "Component.h"
2
3 class Resistor: public Component // An example of Inheritance.
4 {
5 private:
6     Resistor* tinnedcopperwire;
7 public:
```

Figure 7 Shows composition between the resistor and tinned copper wire.

Aggregation

Aggregation is similar to composition in that it is a "has a" relationship, however it does not rely on the whole object (the resistor) to exist, and if the resistor is destroyed it can still exist. An example of this within the program is seen in figure 8. Within an aggregate relationship the parts object can exist in more than one object, so if Resistor was destroyed, the componentId object could still be used by other objects such as Capacitor.

```
Resistor(char* componentId)
```

Figure 8 Shows aggregation between Resistor and componentId.

File access

When a variable is initiated (e.g. num1 = 0) that value is stored in memory, meaning it is volatile and will be wiped when the program is closed. This is where file access is useful as it allows data to be stored in a non-volatile format, in this case, within a .txt file. File access has two main types, opening to read and opening to write. These are implemented through ifstream and ofstream respectively. An example of ifstream can be seen in figure 9 where the file is opened for reading, read and then closed again.

```
ifstream stockFile("stockList.txt"); // Opens the stockList file for reading, this puts all of the stock into variables so the stock quantities can be tracked.
stockFile >> resD >> resP >> resS >> capD >> capP >> capS >> indD >> indP >> indS >> dioD >> dioP >> dioS >> traD >> traP >> traS;
stockFile.close();
```

Figure 9 Shows the use of ifstream to open a file for reading.

When the file is being read, the data is read into variables defined within the ifstream in the order defined. Figure 10 shows an example of ofstream being used to write to a file. Here variables are written into a file in the order defined within the ofstream.

```
ofstream stockreceipt("stockreceipt.txt", ofstream::app); // Opens the stockre
stockreceipt
<< "These components need to be restocked in these quantities." << endl
<< to.description << "\t" << toU << endl
<< cr.description << "\t" << crU << endl
<< tow.description << "\t" << towU << endl
<< af.description << "\t" << afU << endl
<< ao.description << "\t" << aoU << endl
<< s.description << "\t" << sU << endl
<< t.description << "\t" << tU << endl;
stockreceipt.close();
```

Figure 10 Shows the use of ofstream to write to a file.

Data Structures

Data structures or structs, are used to represent a record as they are able to hold multiple data types. Structures have two sections, the definition, shown in figure 11 and accessing the structures members. The definition is where all the attributes of the member are defined. These attributes are later accessed and linked to a member. Figure 12 shows the creation of the members, figure 13 shows the attributes being linked to those members.

```
Res.description = resD; //
Res.price = resP; //Takes 1
Res.stockquantity = resS;

Cap.description = capD;
Cap.price = capP;
Cap.stockquantity = capS;

Ind.description = indD;
Ind.price = indP;
Ind.stockquantity = indS;

Dio.description = dioD;
Dio.price = dioP;
Dio.stockquantity = dioS;

Tra.description = traD;
Tra.price = traP;
Tra.stockquantity = traS;
```

```
struct stockStruc
{
    string description;
    float price;
    int stockquantity;
    int stockrestock;
}; // The struct is used
```

Figure 11 Defining the data structure.

Figure 13 Shows the linking of the structures members and the attributes defined within the structure.

```
struct stockStruc Res;
struct stockStruc Cap;
struct stockStruc Ind;
struct stockStruc Dio;
struct stockStruc Tra;
```

Figure 12 The members of the structure are created.

Patterns

There are a variety of different design patterns, each having a different scenario in which they are particularly useful. One such pattern is the singleton pattern, which provides a global access to an application or class that only requires a single instantiation. Like all classes a singleton pattern requires both a .h file and a .cpp file and to be called in the main function. Figures 14 and 15 show the singleton pattern being created whilst figure 16 shows it being called. This singleton will allow the file "stockList.txt" to be opened, the data read into the variables and then closed whenever it is called. This means that this does not have to be done everytime the file needs to be opened.

```
1 #include <fstream>
2 #include <string>
3 using namespace std;
4 class Singleton
5 {
6 private:
7     Singleton();
8
9 public:
10     int fileOpen();
11     string resD, capD, indD, dioD, traD;
12     float resP, capP, indP, dioP, traP;
13     int resS, capS, indS, dioS, traS;
14     ~Singleton();
15 };
16
17
18
19
```

Figure 14 Shows the .h file for the singleton pattern.

```
1 #include "Singleton.h"
2
3
4 Singleton::~Singleton()
5 {
6 }
7
8 int Singleton::fileOpen()
9 {
10     ifstream stockFile("stockList.txt"); // Opens the stockList file for reading, this puts all of the stock into variables so the stock quantiti
11     stockFile >> resD >> resP >> resS >> capD >> capP >> capS >> indD >> indP >> indS >> dioD >> dioP >> dioS >> traD >> traP >> traS;
12     stockFile.close();
13     return 0;
14 }
```

Figure 15 Shows the .cpp file of the singleton pattern.

```
Singleton::fileOpen();
```

Figure 16 Shows the singleton being called.

Structural Diagram

To show the structural layout of the class system and inheritance used within the design of this prototype program, a class diagram has been created. This is shown in figure 17. As can be seen in the diagram, all of the classes (Resistor, Capacitor, Inductor etc) derive from and inherit componentId from the class Component.

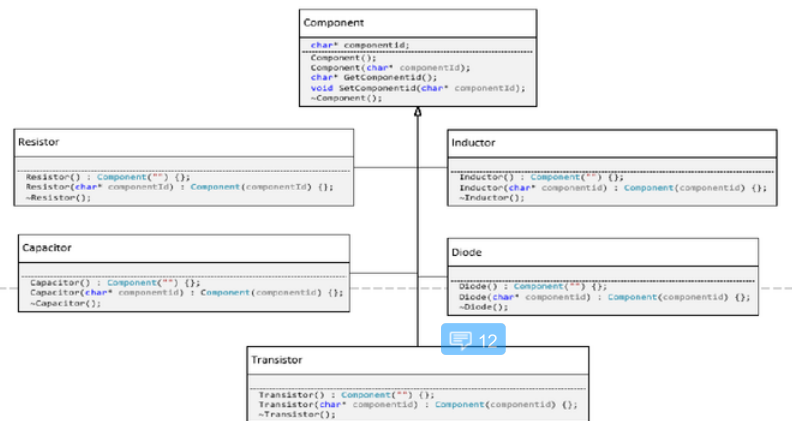


Figure 17 The class Diagram.

Behavioural Diagram

To show the general flow of the program and how it interacts with each of its individual functions, a use case diagram was created. This is seen in figure 18.

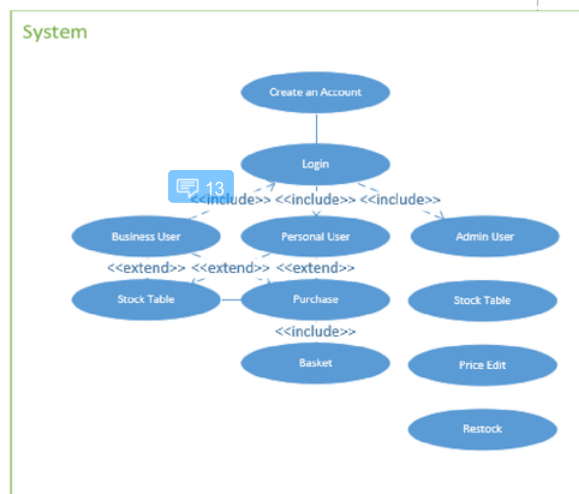


Figure 18 Shows the use case diagram for the entire prototype system.

To extend this, for each function with user interaction, a use case diagram was also created.

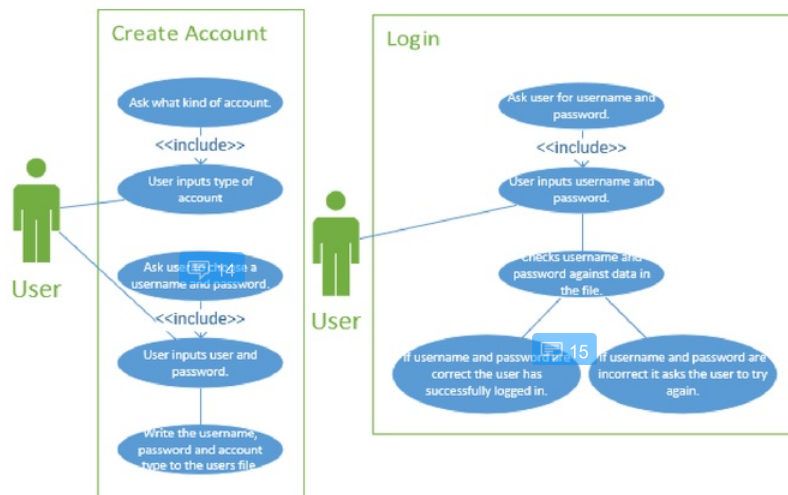


Figure 19 Shows the use case diagrams for the Create Account and Login functions.

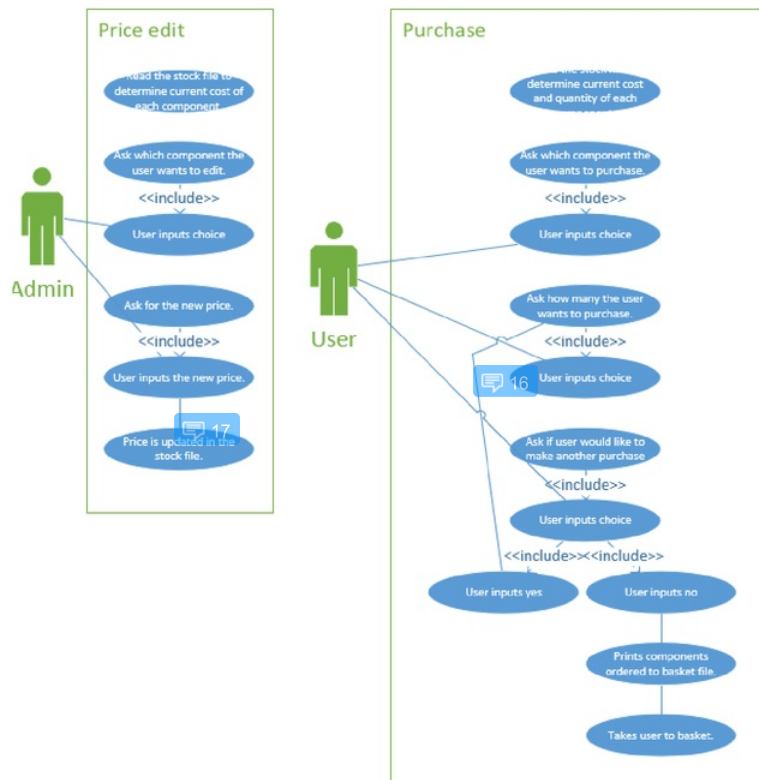


Figure 20 Shows the use case diagrams for the Price Edit and Purchase functions.

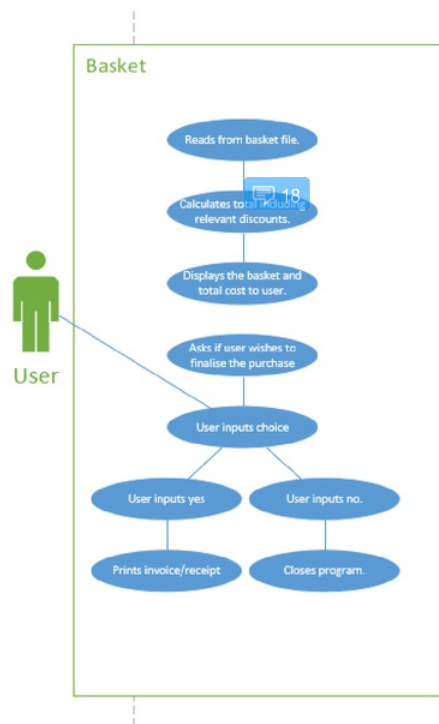


Figure 21 Shows the use case diagram for the Basket function.

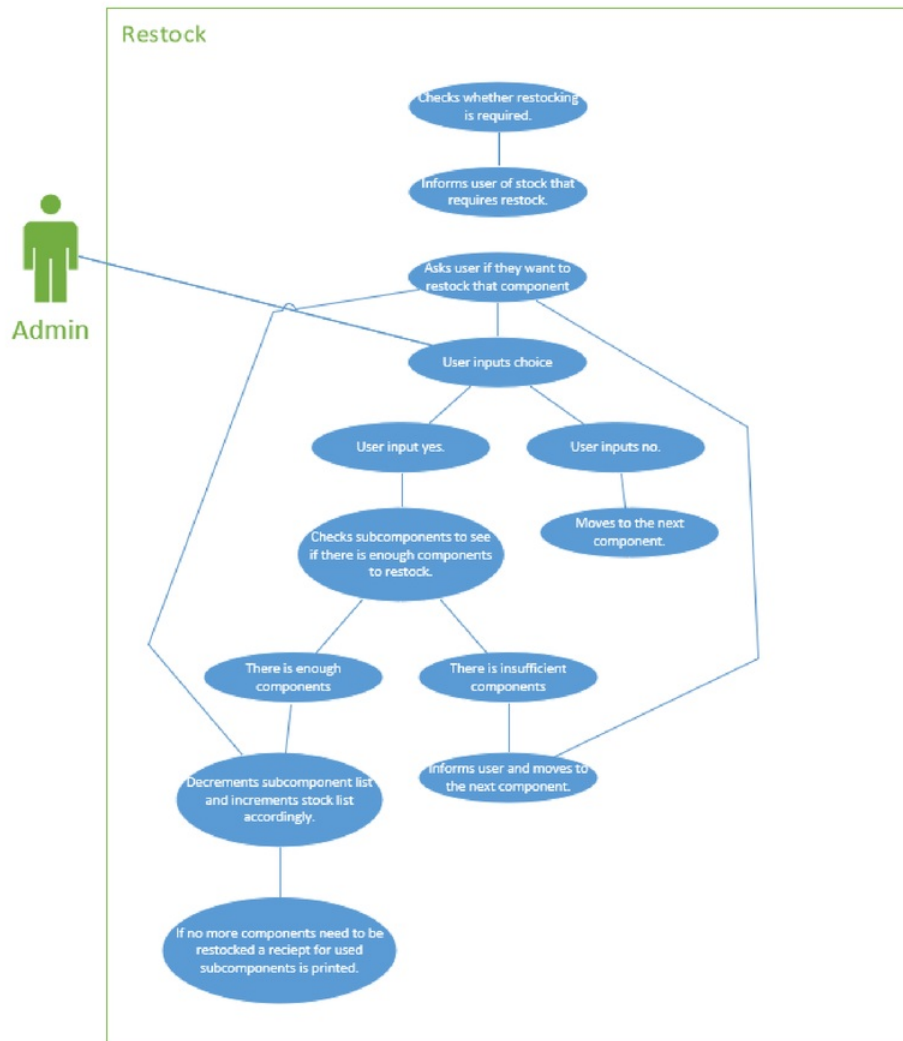


Figure 22 Shows the use case diagram for the Restock function.

Bibliography

- [1] "Classes and Objects," 2018. [Online]. Available: https://www.tutorialspoint.com/cplusplus/cpp_classes_objects.htm. [Accessed: 21-Mar-2018].
- [2] Alex, "10.2 — Composition | Learn C++," *Learncpp.com*, 2017. [Online]. Available: <http://www.learncpp.com/cpp-tutorial/102-composition/>. [Accessed: 21-Mar-2018].

FINAL GRADE

GENERAL COMMENTS

31/50

Instructor

PAGE 1

PAGE 2

PAGE 3

PAGE 4



Comment 1

Good understanding shown here with very good use of references to support assertions made



Comment 2

Not always both can be defined in either file, however, this is a far easier way of reading and sharing code



Comment 3

Very good understanding shown here

PAGE 5



Comment 4

Good understanding shown but use of the word instantiation may have made it clearer



Comment 5

Careful of the language here, although latin in nature, it does not follow academic convention



Comment 6

very good use of references



Comment 7

this shows again good understanding but could have benefitted from references



Comment 8

initialised maybe ? remember semantics here



Comment 9

why is file access handled in this way ?

(i.e: why open and close)

and refs



Comment 10

Good understanding but refs needed



Comment 11

Good understanding of patterns shown. May have wanted to elucidate the difference between creational and behavioural and include references for such



Comment 12

very good class diagram



Comment 13

Unfortunately this is not representative of a use-case diagram there does not appear to be any actors for the system.

Is this a general system overview ?



Comment 14

These are far better than the first diagram



Comment 15

Not sure what these lines represent



Comment 16

unsure what this line represents



Comment 17

either extends or includes, what is the solid line for

PAGE 10



Comment 18

erroneous lines

PAGE 11



Comment 19

Activity Diagrams ?

Sequence Diagrams ?

PAGE 12
