EECS3311-W15 — Project Report

Submitted electronically by:

Team members	Name	Prism Login	Signature
Member 1:	Yuanxin He	yuanxinh	
Member 2:	Zhongran Deng	dzhongr	
*Submitted under Prism account:		dzhongr	

* Submit under **one** Prism account only

Also submit a printed version with signatures in the course Drop Box

Contents

Documentation must be done to professional standards. See OOSC2 Chapter 26: *A sense of style*. Code and contracts must be documented using the Eiffel and BON style guidelines and conventions. *CamelCase* is used in Java. In Eiffel the convention is *under_score*. Attention must be paid to using appropriate names for classes and features. Class names must be upper case, while features are lower case. Comments and header clauses are important. For class diagrams, use the BON conventions, and use clusters as appropriate. Use the EiffelStudio document generation facility (e.g. text, short, flat etc. RTF views), suitably edited and indented to prevent wrapping, to help you obtain appropriately documentation (e.g. contract views). Each diagram must be at the appropriate level of abstraction. Use Visio for the BON class diagrams. See model solution for Assignment 1, posted outside LAS2056.

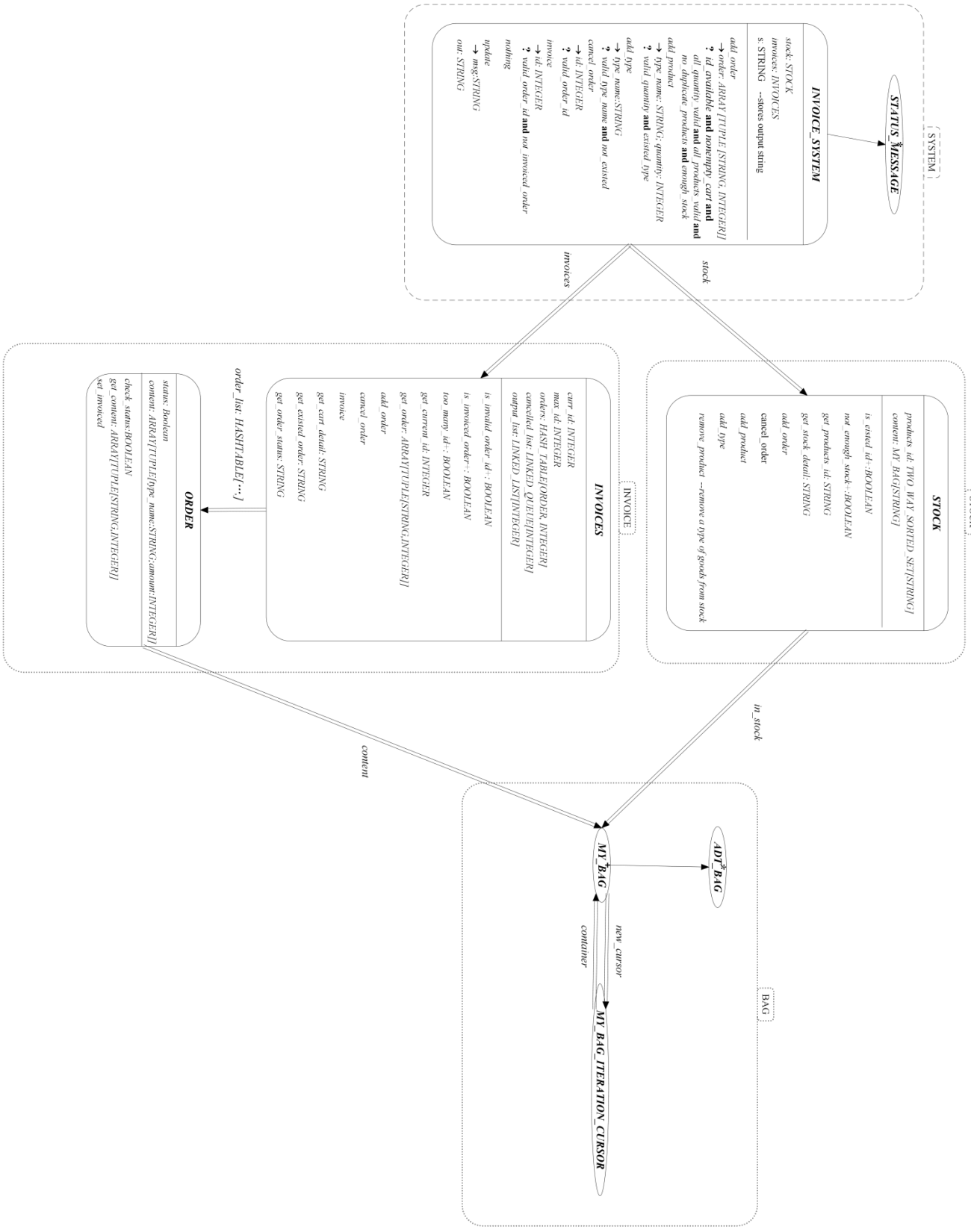
Your signature attests that this is your own work and that you have obeyed university academic honesty policies. Academic honesty is essentially giving credit where credit is due, and not misrepresenting what you have done and what work you have produced. When a piece of work is submitted by a student it is expected that all unquoted and uncited ideas and text are original to the student. Uncited and unquoted text, diagrams, etc., which are not original to the student, and which the student presents as their own work is considered academically dishonest.

1. Requirements for Invoicing System

Our customer provided us with the following statement of their needs: The subject is to invoice orders. To invoice is to change the state of an order (to change it from the state “pending” to “invoiced”). On an order, we have one and one only reference to an ordered product of a certain quantity. The quantity can be different to other orders. The same reference can be ordered on several different orders. The state of the order will be changed into “invoiced” if the ordered quantity is either less or equal to the quantity which is in stock according to the reference of the ordered product. You have to take into account new orders, cancellations of orders, and entries of quantities in the stock. A console based application for user input suffices.

Analysis of the requirements and further description may be found at the following URL:
https://wiki.eecs.yorku.ca/course_archive/2014-15/W/3311/protected:assign:project:phase1

2. BON class diagram overview (architecture of the design)



In this project, we need to divide stock and invoices so that they can store data independently without worrying about messing up the data with each other. In stock section, apart from storing the content of stock using class MY_BAG, we also need to establish a namespace of product types for fast query. In invoices section, we need three lists to record existing orders collection, cancelled orders collection and output sequence respectively. Last but not lease, we need a common inherited class that groups up functions that is required for exception detection.

STOCK, INVOICES are designed to handle stuffs related to stock and invoices respectively. First, they are well separated so that when the system is asked to execute some commands, system could unambiguously direct the commands to a correct specific classes instead of calling a superclass all the time. This manifests the principle of *separation of concerns*. Second, it also helps to improve *reliability* because reduced complexity of a class may significantly decrease the likelihood of bugs. Third, class **STOCK** doesn't need to know the operations and attributes in **INVOICES** and vice versa. Operation related to state queries must must go through their internal query routine. That is to say, all External classes cannot directly get accessed to their classes states(they are not allow to get access to each other's state either). It strengthens *information hiding* as well as safety.

INVOICE_SYSTEM takes the role of MODEL in an ETF pattern. It is used to receive commands and pass them towards corresponding module. First, this design facilitates *extendibilities*. Whenever we want to add new functions, we simply add a new class and write finite code modification in scope of **INVOICE_SYSTEM** to bring this new class into effect. We change least possible codes and thus minimizes the workload for adaption. Also note that defensive programming is applied before the function is called. Therefore, preconditions specified in the class are just for completeness of contract and normally makes no sense. Also note that postconditions specified in this class would not be so specific as those in **STOCK** and **INVOICES** because we weaken the conditions here for future modification.

STATUS_MESSAGE is an interface that groups up the majority of ancillary boolean queries and error messages together. It is created for the need of shared data and all three classes described above would inherit from it. Therefore, we are *programming to interface instead of programming to implementation.* This is particularly helpful because it greatly prevents the risk of negligent data alteration. Thus it provides a more reliable and efficient way to handle exception message and makes the system more maintainable. Also note that, queries involved internal state query would be directed to specific class and thus would not be fully implemented here.

MY_BAG is an abstract data type designed to hold a collection of goods specifying certain amount. It adopts an iterator pattern so class **MY_BAG_ITERATION_CURSOR** is also well implemented. It's a container for both stock and orders. However, in some function's implementation, we would pass the data using an array instead of using **MY_BAG** because it would greatly reduce the job of converting.

ORDER is a class holding content of order as well as its status and basic operations.

3. Table of modules — responsibilities and information hiding

a) STOCK module

1	TWO_WAY_SORTED_SET[STRING]	Responsibility: namespace of product type for fast query.	Alternate: ARRAYED_SET[G]
	Concrete	Secret: use of set ensures no types duplicate.	

2	ADT_BAG[G -> {COMPARABLE, HASHABLE}]	Responsibility: unordered collection of hashable items with possible multiplicity and a sorted domain	Alternate: a more generic bag without the constraint of a sorted domain, and/or without the constraint of hashable items.
	Abstract	Secret: none	

2.1	MY_BAG	Responsibility: store products as inventory.	Alternate: implement with two arrays, the first for the data item and the second to store the multiplicity. This would not take advantage of the look-up efficiency of hashable items.
	Concrete	Secret: implemented with hashing and counting to take multiplicity into account.	

b) INVOICES module

1	HASH_TABLE[ORDER, INTEGER]	Responsibility: record of orders in increasing order by id	Alternate: ARRAYED_LIST[TUPLE[INTEGER, ORDER]]. Drawback of using this structure is that when you remove an element from a position, position would not be reserved.
	Concrete	Secret: position is reserved when some elements are removed which facilitates future insertion.	

1.1	ORDER	Responsibility: record of orders and their status	Alternate: ARRAY[TUPLE[STRING,INTEGER]]
	Concrete	Secret: status of the order is a Boolean value, return true if invoiced; return false if pending	

2	LINKED_QUEUE[INTEGER]	Responsibility: record of cancelled ids.	Alternate: ARRAYED_LIST[INTEGER]
	Concrete	Secret: first come first out principle applies the sample ETF output.	

3	ARRAYED_LIST[INTEGER]	Responsibility: record of output sequence	Alternate: LINKED_LIST[INTEGER]
	Concrete	Secret: none	

c) SYSTEM module

1	STATUS_MESSAGE	Responsibility: an encapsulation of ancillary Boolean queries and error messages for common use.	Alternate: none.
	Abstract	Secret: queries involved internal state query would direct to specific classes instead of implementing in this class.	

1.1	INVOICE_SYSTEM	Responsibility: console of the whole program	Alternate: none
	Concrete	Secret: defensive programming is applied here ensuring no precondition violation would occur.	

4. Expanded description of design decisions

In below would provide more details of description for module **INVOICES**.

4.1. INFORMATION HIDING

The feature of initialization and attributes will be set to private so that they can only be invoked in scope of the class. However, it also provides routine for external state query like get_order_amount, get_current_id and get_order. Thus, it ensures that the client can access states through public routine while the private states still keeps invisible to the clients. This mechanism also applies to class **STOCK**.

4.2. INHERITANCE

Class **INVOICES** inherits from **STATUS_MESSAGE**. It may take advantage of all exception judgement function defined in **STATUS_MESSAGE.** However, functions involved state query would not be allowed to access to **INVOICES** attributes directly. Therefore, functions involved state query in **STATUS_MESSAGE** would be directed to function in **INVOICES** with same signature where concrete function is implemented here.

4.3. STATES

Class **INVOICES** has five attributes to describe class's state. They are:

order_list	a HASH_TABLE, stores orders in order and its id, reserving vacant position of cancelled orders
canceled_id_list	a LINKED_QUEUE, records index of removed orders in order_list
output_sequence	an ARRAYED_LIST, records the output sequence
max_id	an INTEGER, records the largest id that order_list has ever created
current_id	an INTEGER, records the id of last created order

4.4. OPERATION MECHANISM

The operation mechanism will be described as below.

When the system adds a new order while canceled_id_list being empty, add_order command will be executing in **INVOICE_SYSTEM**. Soon it will be split into two independent commands directing to **INVOICES** and **STOCK** respectively. In **INVOICE** class, since this is the 1st order being created, a new order will be added in order_list meanwhile its order id

will be automatically generated as well (in this case it's 1). output_sequence will also add a same order of STRING to record it. max_id will be added by 1. current_id will record the index of the newly added order in order_list.

When the system wants to cancel an order, the cancel_order will also direct the command to **INVOICE** like that is in add_order. First, order will be removed from output_sequence and its next order will cover its place. Second, the id will be pushed into canceled_id_list to record the vacant position. Finally, order will be removed from order_list. Note that in order_list, the next order will not cover its predecessor's position.

If the system wants to add orders again given the canceled_id_list is not empty, systems will operate differently. First, canceled_id_list pops up the id in its first position(queue observes first come first out principle). order_list will assign the pop-up id to the newly added order and add them together into the table. output_sequence will add the order at the end of the array. max_id will not be changed since no larger id is generated. current_id will be assigned with the pop-up id.

If the system keeps on adding orders while the canceled_id_list is empty, the system will act in a way supposing canceled_id_list is empty described above

The invoice command will visit order_list and change the corresponding order from pending to invoiced.

5. Significant Contracts (Correctness)

(only for the module with the most significant contracts)

INVOICE_SYSTEM

```
                        -- Automatic generation produced by ISE Eiffel --
note
        description: "Summary description for {MODEL}."
        author: ""
        date: "$Date$"
        revision: "$Revision$"

class interface
        INVOICE_SYSTEM

create {INVOICE_SYSTEM_ACCESS, STUDENT_TEST1}
        make

feature --attributes

        stock: STOCK
                        -- track the quantity of product left in our stock and store the product id

        invoices: INVOICES
                        -- handle all orders and change their status.

        s: STRING_8
                        --store output string

feature --commands using defensive programming

        add_order (order: ARRAY [TUPLE [type_name: STRING_8; amount: INTEGER_32]])
                        --first remove corresponding goods in stock, second add new order into invoices;
postconditions here are WEAK, deatailed postconditions are hidden in respective classes in order to
achieve information hiding. Same as all commands below
                require
                        id_valid: not too_many_id (invoices)
                        nonempty_cart: not is_empty_cart (order)
                        all_quantity_valid: all_quantity_valid (order)
```

```
                        all_products_valid: all_products_valid (order, stock)
                        no_duplicate_products: not has_duplicate_products (order)
                        enough_stock: not not_enough_stock (order, stock)
                ensure
                        stock_decrease: stock.get_total_amount < old stock.get_total_amount
                        invoices_increase: invoices.get_order_amount = old invoices.get_order_amount +
1


        add_product (type_name: STRING_8; quantity: INTEGER_32)
                        --invoke corresponding command in STOCK if preconditions are satisfied
                require
                        valid_quantity: not is_invalid_quantity (quantity)
                        existed_type: is_existed_id (type_name, stock)
                ensure
                        stock_increase: Current.stock.get_total_amount = old
Current.stock.get_total_amount + quantity


        add_type (type_name: STRING_8)
                        --invoke corresponding command in STOCK if preconditions are satisfied
                require
                        valid_type_name: not is_invalid_type_name (type_name)
                        not_existed: not is_existed_id (type_name, stock)
                ensure
                        type_increase: Current.stock.get_type_amount = old
Current.stock.get_type_amount + 1
                        type_exist: Current.stock.is_existed_id (type_name, stock)


        cancel_order (id: INTEGER_32)
                        --first remove order from invoices, second restore corresponding goods in stock
                require
                        valid_order_id: not is_invalid_order_id (id, invoices)
                ensure
                        invoices_decrease: Current.invoices.get_order_amount = old
Current.invoices.get_order_amount - 1
                        stock_increase: Current.stock.get_total_amount > old
Current.stock.get_total_amount


        invoice (id: INTEGER_32)
                        --invoke corresponding command in INVOICES if preconditions are satisfied
```

```
                require
                        valid_order_id: not is_invalid_order_id (id, invoices)
                        not_invoiced_order: not is_invoiced_order (id, invoices)
                ensure
                        order_invoiced: Current.invoices.get_odr (id).check_status

        nothing
                        --no state changes after operation

feature --output handling

        update (msg: STRING_8)
                        -- Perform update to the model state.

        out: STRING_8
                        -- New string containing terse printable representation
                        -- of current object

end -- class INVOICE_SYSTEM
                        -- Generated by ISE Eiffel --
                        -- For more details: http://www.eiffel.com --
```

6. Summary of Testing Procedures

a) Acceptance test

Test	Description	Passed
at1.txt	Normal scenario where product types are created, orders are placed and invoiced.	Passed
at2.txt	Adding invalid orders.	Passed
at3.txt	Cancel non-existed order.	Passed
at4.txt	Test how does the output sequence change by adding and cancelling orders.	Passed
mytest.txt	Test all possible violation cases	Passed

b) Unit test

Test Run:04/04/2015 11:53:18.169 PM

ROOT

Note: * indicates a violation test case

PASSED (18 out of 18)

Case Type	Passed	Total
Violation	13	13
Boolean	5	5
All Cases	18	18

State	Contract Violation	Test Name
Test1	STUDENT_TEST1	
PASSED	NONE	t1: check whether products type list is sorted alphabetically Input order: nuts,bowl,apple Output order: apple,bowl,nuts
PASSED	NONE	t2: check whether stock detail list is sorted alphabetically by product name; ensure zero product would not be shown Input order: nuts->300,apple->100; add type bowl but do not add product Output order: apple->100,nuts->300
PASSED	NONE	t3: check whether output of "carts" is sorted alphabetiaclly regardless of input order Input order: 1: bowl->4,nuts->5 2: nuts->6,apple->10 Output order: 1: bowl->4,nuts->5 2: apple->10,nuts->6 1: bowl->4,nuts->5 2: apple->10,nuts->6
PASSED	NONE	t4: check whether output sequence keeps position vacant after cancelling order If not reserved: 1,2,3,4,5,6,7 If reserved(output): 1,2,3,5,4,6,7
PASSED	NONE	t5: check whether the output of order state is shown correctly given the same condition of t4; also check whether invoice function working correctly Input: output order is 2, 4, 3, 1, 5; invoice 3 and 5: Output: 2->pending,4->pending,3->invoiced,1->pending,5->invoiced
PASSED	NONE	*t_1: add type with empty string

PASSED	NONE	*t_2: add the same type twice
PASSED	NONE	*t_3: add negative amount of products into stock
PASSED	NONE	*t_4: add untyped products into stock
PASSED	NONE	*t_5: add 10001 orders
PASSED	NONE	*t_6: add an empty order
PASSED	NONE	*t_7: add order with negative amount of items
PASSED	NONE	*t_8: add order with untyped items
PASSED	NONE	*t_9: add an order with duplicate items
PASSED	NONE	*t_10: order exceeds the amount in stock
PASSED	NONE	*t_11: invoice an invalid order
PASSED	NONE	*t_12: repeative invoicing
PASSED	NONE	*t_13: cancel not existed order

Note: t_5 is technically removed from the actual code because computer is not competent to compute 10001-loop in short time (add 10001 orders).

7. Appendix (Contract view of all classes)

(Only classes that you created; do not include user input command classes, only model classes)

INVOICES:

```
                        -- Automatic generation produced by ISE Eiffel --
note
        description: "Summary description for {INVOICES}."
        author: ""
        date: "$Date$"
        revision: "$Revision$"

class interface
        INVOICES

create {INVOICE_SYSTEM}
        make

feature --ancillary boolean queries

        is_invalid_order_id (id: INTEGER_32; invoices: INVOICES): BOOLEAN
                        --if id not exist, return true; otherwise return false

        is_invoiced_order (id: INTEGER_32; invoices: INVOICES): BOOLEAN
                        --check whether the order is invoiced or not

        too_many_id (invoices: INVOICES): BOOLEAN
                        --check whether the amount of ids exceeds 10000

feature --state queries

        get_order_amount: INTEGER_32
                        --return the amount of order in the order_list.
                ensure
                                Result = Current.order_list.count

        get_current_id: INTEGER_32
                        --return current id
                ensure
                                Result = current_id

        get_max_id: INTEGER_32
                        --return max id
                ensure
                                Result = max_id

        get_odr (id: INTEGER_32): ORDER
                        --return order by given id

        get_order (id: INTEGER_32): ARRAY [TUPLE [STRING_8, INTEGER_32]]
                        --return content of order by given id
                ensure
                                not Result.is_empty

feature --commands

        add_order (order: ARRAY [TUPLE [STRING_8, INTEGER_32]])
                        --generate new order in order_list, modify output_sequence
                require
                        id_valid: not too_many_id (Current)
```

```
                        nonempty_cart: not is_empty_cart (order)
                        all_quantity_valid: all_quantity_valid (order)
                        no_duplicate_products: not has_duplicate_products (order)
                ensure
                        order_increase: Current.order_list.count = old Current.order_list.count +
1
                        id_unique: not Current.order_list.found
                        order_inserted: attached Current.order_list [current_id] as ol implies
ol.equal_order (create {ORDER}.make (order))
                        output_correct: across
                                        Current.output_sequence as o2
                                all
                                        Current.order_list.has (o2.item)
                                end
                        max_id_correct: old canceled_id_list.is_empty implies current_id = max_id

        cancel_order (id: INTEGER_32)
                        --remove order from order_list, create a new record in canceled_id_list,
modify output_sequence
                require
                        valid_order_id: not is_invalid_order_id (id, Current)
                ensure
                        order_removed: not Current.order_list.has_key (id)
                        cancel_added: Current.canceled_id_list.count = old
Current.canceled_id_list.count + 1
                        output_removed: across
                                        Current.output_sequence as o3
                                all
                                        o3.item /= id
                                end

        invoice (id: INTEGER_32)
                        --alter state of order from pending to invoiced
                require
                        valid_order_id: not is_invalid_order_id (id, Current)
                        not_invoiced_order: not is_invoiced_order (id, Current)
                ensure
                        order_invoiced: attached Current.order_list [id] as o4 implies
o4.check_status

feature --output handling

        get_cart_detail: STRING_8
                        --output a string describing detail of the cart

        get_exsited_order: STRING_8
                        --output a string illustrating product types in the invoice list

        get_order_status: STRING_8
                        --output a string illustrating status of orders

invariant
        all_positive_order: across
                        Current.order_list as o
                all
                        o.item.is_all_positive
                end
        max_id: Current.get_max_id <= 10000

end -- class INVOICES
                        -- Generated by ISE Eiffel --
                        -- For more details: http://www.eiffel.com --
```

STOCK:

```
                        -- Automatic generation produced by ISE Eiffel --
note
        description: "Summary description for {STOCK}."
        author: ""
        date: "$Date$"
        revision: "$Revision$"

class interface
        STOCK

create {INVOICE_SYSTEM}
        make

feature --ancillary boolean queries

        is_existed_id (type_name: STRING_8; stock: STOCK): BOOLEAN
                        --check whether the type has already existed

        not_enough_stock (order: ARRAY [TUPLE [type_name: STRING_8; amount: INTEGER_32]]; stock:
STOCK): BOOLEAN
                        --check whether the order exceed the quantity of goods in the stock

feature --state queries

        get_total_amount: INTEGER_32
                        --return total amount of all items in the stock

        get_type_amount: INTEGER_32
                        --return amount of types in the stock

        occurrence alias "[]" (type_name: STRING_8): INTEGER_32
                        --return quantity for given type name

        get_products_id: STRING_8
                        --output namespace (goods that can be imported) in the stock

        get_stock_detail: STRING_8
                        --output a string illustrating all states of the stock

feature --commands

        add_order (order: ARRAY [TUPLE [type_name: STRING_8; quantity: INTEGER_32]])
                        --export goods according to the order
                require
                        nonempty_cart: not is_empty_cart (order)
                        all_quantity_valid: all_quantity_valid (order)
                        all_products_valid: all_products_valid (order, Current)
                        no_duplicate_products: not has_duplicate_products (order)
                        enough_stock: not not_enough_stock (order, Current)
                ensure
                        stock_decrease: Current.get_total_amount < old Current.get_total_amount

        remove_product (type_name: STRING_8; quantity: INTEGER_32)
                        --export one type of good with certain quantity
                ensure
                        product_removed: not Current.in_stock.has (type_name) or Current.in_stock
[type_name] = old Current.in_stock [type_name] - quantity

        cancel_order (order: ARRAY [TUPLE [type_name: STRING_8; quantity: INTEGER_32]])
                        --import goods according to the order
```

```
                ensure
                        stock_increase: Current.get_total_amount > old Current.get_total_amount

        add_product (type_name: STRING_8; quantity: INTEGER_32)
                        --import one type of good with certain quantity
                require
                        valid_quantity: not is_invalid_quantity (quantity)
                        existed_type: is_existed_id (type_name, Current)
                ensure
                        stock_increase: Current.get_total_amount = old Current.get_total_amount +
quantity

        add_type (type_name: STRING_8)
                        --define certain type of goods that can be imported
                require
                        valid_type_name: not is_invalid_type_name (type_name)
                        not_existed: not is_existed_id (type_name, Current)
                ensure
                        type_increase: Current.get_type_amount = old Current.get_type_amount + 1
                        has_type: Current.products_id.has (type_name)

invariant
        positive_product_quantity: across
                        Current.in_stock.domain as p
                all
                        Current.in_stock [p.item] > 0
                end

end -- class STOCK
                        -- Generated by ISE Eiffel --
                        -- For more details: http://www.eiffel.com --
```

ORDER:

```
                        -- Automatic generation produced by ISE Eiffel --
note
        description: "Summary description for {ORDER}."
        author: ""
        date: "$Date$"
        revision: "$Revision$"

class interface
        ORDER

create
        make

feature --queries

        equal_order (other: like Current): BOOLEAN
                        --check whether two orders are equal

        is_all_positive: BOOLEAN
                        --check whether if all products' quantity positive.

        check_status: BOOLEAN
                        --check order status, false if pending, true if invoiced

        get_content: ARRAY [TUPLE [type_name: STRING_8; amount: INTEGER_32]]
                        --return the content of the order

feature --commands

        set_invoiced
                        --alter the order status from pending to invoiced

invariant
        all_positive: across
                        Current.content as p
                all
                        p.item.amount > 0
                end

end -- class ORDER
                        -- Generated by ISE Eiffel --
                        -- For more details: http://www.eiffel.com --
```

STATUS_MESSAGE:

```
                        -- Automatic generation produced by ISE Eiffel --
note
        description: "Error Messages for invoice"
        author: "JSO"
        date: "$Date$"
        revision: "$Revision$"

class interface
        STATUS_MESSAGE

create
        make_ok,
        make_empty_string,
        make_already_exsit,
        make_negative_quantity,
        make_no_product,
        make_no_more_id,
        make_empty_cart,
        make_not_valid_product,
        make_duplicate_product,
        make_not_enough,
        make_invalid_order_id,
        make_invoiced_order

feature --ancillary boolean queries (sorted alphabetically)

      all_products_valid (order: ARRAY [TUPLE [type_name: STRING_8; amount: INTEGER_32]]; stock:
STOCK): BOOLEAN

        all_quantity_valid (order: ARRAY [TUPLE [type_name: STRING_8; amount: INTEGER_32]]):
BOOLEAN

        has_duplicate_products (order: ARRAY [TUPLE [type_name: STRING_8; amount: INTEGER_32]]):
BOOLEAN

        is_empty_cart (order: ARRAY [TUPLE [STRING_8, INTEGER_32]]): BOOLEAN

        is_existed_id (type_name: STRING_8; stock: STOCK): BOOLEAN

        is_invalid_order_id (id: INTEGER_32; invoices: INVOICES): BOOLEAN

        is_invalid_quantity (quantity: INTEGER_32): BOOLEAN

        is_invalid_type_name (type_name: STRING_8): BOOLEAN
                ensure
                                Result implies type_name.count < 1

        is_invoiced_order (id: INTEGER_32; invoices: INVOICES): BOOLEAN

        not_enough_stock (order: ARRAY [TUPLE [type_name: STRING_8; amount: INTEGER_32]]; stock:
STOCK): BOOLEAN

        too_many_id (invoices: INVOICES): BOOLEAN

feature -- output handling

        out: STRING_8
                        -- string representation of current status message

end -- class STATUS_MESSAGE
```

MY_BAG:

```
                        -- Automatic generation produced by ISE Eiffel --
note
        description: "bag application root class"
        date: "$Date$"
        revision: "$Revision$"

class interface
        MY_BAG [G -> {HASHABLE, COMPARABLE}]

create
        make_empty,
        make_from_tupled_array

convert
        make_from_tupled_array ({attached ARRAY [attached TUPLE [G, INTEGER_32]]})

feature -- creation queries

        new_cursor: MY_BAG_ITERATION_CURSOR [G]
                        -- Fresh cursor associated with current structure

        is_nonnegative (a_array: ARRAY [TUPLE [x: G; y: INTEGER_32]]): BOOLEAN
                        -- Are all the `y' fields of tuples in `a_array' non-negative

feature -- bag equality

        bag_equal alias "|=|" (other: like Current): BOOLEAN
                        -- equal to current object?

feature -- queries

        domain: ARRAY [G]
                        -- sorted domain of bag

        count: INTEGER_32
                        -- cardinality of the domain

        occurrences alias "[]" (key: G): INTEGER_32
                        -- Anything out of the domain can simply be considered out of the bag,
                        -- i.e. has a number of occurrences of 0.

        is_subset_of alias "|<:" (other: like Current): BOOLEAN
                        -- current bag is subset of `other'
                        -- <=

feature -- commands

        extend (a_key: G; a_quantity: INTEGER_32)
                        -- add [a_key, a_quantity] to the bag
                        -- add additional quantities if item already is in the bag

        add_all (other: like Current)
                        -- add all elements in the bag `other'

        remove (a_key: G; a_quantity: INTEGER_32)
                        -- remove [a_key, a_quantity] from the bag

        remove_all (other: like Current)
                        -- bag difference
                        -- i.e. no. of items in Current
```

```
                        -- minus no. of times in other,
                        -- or zero

        debug_output: STRING_8

end -- class MY_BAG
                        -- Generated by ISE Eiffel --
                        -- For more details: http://www.eiffel.com --
```

MY_BAG_ITERATION_CURSOR:

```
note
	description: "Summary description for {MY_BAG_ITERATION_CURSOR}."
	author: ""
	date: "$Date$"
	revision: "$Revision$"

class interface
	MY_BAG_ITERATION_CURSOR [G -> {HASHABLE, COMPARABLE}]

create
	make

feature

	make (bag: MY_BAG [G])

feature

	item: G
			-- Item at current cursor position.

	after: BOOLEAN
			-- Are there no more items to iterate over?

	forth
			-- Move to next position.

feature

	index: INTEGER_32

	target: MY_BAG [G]

end -- class MY_BAG_ITERATION_CURSOR
```

ADT_BAG:

```
                        -- Automatic generation produced by ISE Eiffel --
note
        description: "Abstract Data Type for BAG[G] where G is hashable and comparable"
        author: "JSO"
        date: "$Date$"
        revision: "$Revision$"

deferred class interface
        ADT_BAG [G -> {HASHABLE, COMPARABLE}]

feature -- creation queries

        is_nonnegative (a_array: ARRAY [TUPLE [x: G; y: INTEGER_32]]): BOOLEAN
                        -- Are all the `y' fields of tuples in `a_array' non-negative
                ensure
                        correct_result: Result = (across
                                        a_array as it
                                all
                                        it.item.y >= 0
                                end)

feature -- bag equality

        bag_equal alias "|=|" (other: like Current): BOOLEAN
                        -- equal to current object?
                ensure
                        symmetry: Result = (other |=| Current)

feature -- queries

        total: INTEGER_32
                        -- total number of items in the bag

        count: INTEGER_32
                        -- cardinality of the domain

        domain: ARRAY [G]
                        -- sorted domain of bag
                ensure
                        value_semantics: Result.object_comparison
                        correct_items: across
                                        1 |..| Result.count as j
                                all
                                        has (Result [j.item])
                                end
                        sorted: across
                                        1 |..| (Result.count - 1) as j
                                all
                                        Result [j.item] <= Result [j.item + 1]
                                end

        occurrences alias "[]" (key: G): INTEGER_32
                        -- Anything out of the domain can simply be considered out of the bag,
                        -- i.e. has a number of occurrences of 0.
                ensure
                                Result >= 0
                                has (key) implies Result > 0
```

```
        has (a_item: G): BOOLEAN
                        -- bag has element `a_item'
                ensure
                        has_item: Result = (occurrences (a_item) > 0)

        is_subset_of alias "|<:" (other: like Current): BOOLEAN
                        -- current bag is subset of `other'
                        -- <=
                ensure
                        correct_subset: Result implies across
                                        domain as g
                                all
                                        has (g.item) implies other.has (g.item) and then
occurrences (g.item) <= other.occurrences (g.item)
                                end

feature -- commands

        extend (a_key: G; a_quantity: INTEGER_32)
                        -- add [a_key, a_quantity] to the bag
                        -- add additional quantities if item already is in the bag
                require
                        non_negative: a_quantity >= 0
                ensure
                        extended: has (a_key) and then occurrences (a_key) = old (occurrences
(a_key)) + a_quantity

        add_all (other: like Current)
                        -- add all elements in the bag `other'

        remove (a_key: G; a_quantity: INTEGER_32)
                        -- remove [a_key, a_quantity] from the bag
                require
                        non_negative: a_quantity >= 0
                ensure
                        subtract: old occurrences (a_key) > a_quantity implies has (a_key) and
then occurrences (a_key) = old occurrences (a_key) - a_quantity
                        zero: old occurrences (a_key) <= a_quantity implies not has (a_key) and
then occurrences (a_key) = 0

        remove_all (other: like Current)
                        -- bag difference
                        -- i.e. no. of items in Current
                        -- minus no. of times in other,
                        -- or zero

feature -- countimng quantifiers

        number_of (f: PREDICATE [ANY, TUPLE [G, INTEGER_32]]): INTEGER_32

invariant
        consistent_count: count = domain.count
        nonnegative_items: across
                        domain as it
                all
                        occurrences (it.item) > 0
                end
        reflexivity: Current |=| Current

end -- class ADT_BAG
                        -- Generated by ISE Eiffel --
                        -- For more details: http://www.eiffel.com --
```