Purchase Order System

Purchase Order System manages the way an organisation buys goods & services. The system is automated – where users have access to preferred suppliers and product/service category master list; create, modify, approve purchase orders, and sign off for goods received; eliminating the need for paper. Purchase Order System is a solution to generate, track & manage your purchase orders from one central location.

Such a system requires the hight security requirements based ob contexts (the history of the system). For example, it is important to restrict the person who receives goods is different from the person who created the order to avoid the fraud. The system is divided into two parts: functional and security logics, which are detailed as follows:

A. Functional data (work on MS SQL Server)

In SQL server, create table **purchase_order** which stores order information: identification, items (to simplify the list of items is a string), the state indicating if an order is already approved or not (isApproved), and the state indicating if goods are already received (isReceivedGoods) or not. The sql statement is formed as follows:

```
CREATE TABLE purchase_order(
oId int primary key identity(1,1) not null,
oItems varchar(25) not null,
isApproved bit not null,
isReceived bit not null
);
```

Table purchase_order

oId (auto_increment)	**Items**	**isApproved**	**isReceivedGoods**

B. Security data(work on MS SQL Server)

The security logic composes of the core RBAC (i.e. users, roles, permissions) and the history-based constraints.

Taking advatages of the existing security management of DBMSs, the core RBAC: the users are created as database users; the roles are database roles and are granted plivileges.

The history-based constraints are defined in Java class called **ADTranslation.** This class is called in the AspectJ class called **AspectJForSecurity.** The role of this class is to check the history-based constraints before executing a method and log its execution after the method is *successfully* performed. This checking is called when a method is invoked.

Note that the permission to execute a method of a user is checked at the database level.

1. User

In order to test Purchase Order System, three login users are created: allice_login, bob_login, and tom_login. The follow is the syntax to create these users in SQL Server.

- Allice :

  ```
  create login "allice_login" with password = 'pwdallice';
  create user "allice_login" for login allice_login;
  ```
- Tom

  ```
  create login "tom_login" with password = 'pwdtom';
  create user "tom_login" for login tom_login;
  ```
- Bob

  ```
  create login "bob_login" with password = 'pwdbob';
  create user "bob_login" for login bob_login;
  ```

2. Role

- In Purchase Order System, the users are employees in the organisation, which are divided into two roles: He/she can be a manager or a normal staff . So we create three roles: employee which has right to view the table purchase_order and view and log execution information of methods; manager can approve orders; staffs are allowed to create, modify orders, receive goods of an order. The SQL statements that create those roles are employee, manager, staff ::

  ```
  create role manager;
  create role staff;
  create role employee;
  ```

3. User_Role

The users of the software system are employees of the organization in Purchase Order System. Then all users are members of the role ‘employee’. The aim is to grant general permissions on the database, for example, view the table ‘purchase_order’. Each user is then asigned to a specific role (manager or staff), which has certain permissions.

- All users are employees of the enterprise

  ```
  ALTER ROLE employee ADD MEMBER bob_login;
  ALTER ROLE employee ADD MEMBER tom_login;
  ALTER ROLE employee ADD MEMBER allice_login;
  ```

- Allice is a staff
- Tom is a manager
- Bob is a staff

  ```
  ALTER ROLE manager ADD MEMBER tom_login;
  ALTER ROLE staff ADD MEMBER bob_login;
  ALTER ROLE staff ADD MEMBER allice_login;
  ```

4. Grant plivileges

a. Support permissions

These permissions support the test cases. i.e. view the results after executing SQL statements.

- All employees can view table purchase_order
- All employees can view, insert into table method_autdit

```
/*all employees can see table purchase_order */
grant SELECT on dbo.purchase_order to employee;

/*all employees can see and create log table method_audit */
grant SELECT, INSERT on dbo.method_audit to employee;
```

b. Permission polices:

Assuming the security policy is defined in the B specification:

```
Permissions ={
      StaffRole|-> Purchase_order_create_Label
      , StaffRole|-> Purchase_order_modify_Label
      , StaffRole|-> Purchase_order_receiveGoods_Label
      , ManagerRole|-> Purchase_order_approve_Label
}
```

This means we must grant these permissions as follows:

- **Purchase_order_create** to **staff**
- **Purchase_order_modify** to **staff**
- **Purchase_order_receiveGoods** to **staff**
- **Purchase_order_approve** to **manager**

The idea is that each time one of these methods is invoked by a (connecting) user at the application level (call a method in the Java class) we must ensure this user has the right to perform it. Such a method acts on the database, but granting permissions at the method-level (application-level) is not supported by DBMS. They support only granting plivileges (insert, delete, update, select) on tables and columns, or granting EXECUTE on stored procedure which are stored in the database .

Basically, a stored procedure is prepared SQL code that is stored in the database and can be reused over and over again. It can be called from the application level (via methods). In addition, we can define permissions of executing a stored procedure at the database level.

Therefore, the propose solution is to create a stored procedure for each above method which executes SQL statements. Then, we grant executing permission of these stored procedures to specific roles (users). This stored procedure is called in the corresponding Java method. Consequently, when a user logins and calls a method, the system will check his permission of executing the corresponding stored procedure at the database level.

Create stored procedure:

- **Purchase_order_create :** do statement insert
- **Purchase_order_modify :** do statement update column items
- **Purchase_order_approve :** do statement update column isApproved
- **Purchase_order_receiveGoods :** do statement update column isReceivedGoods

SQL statements to create these stored procedures are as follows:

```sql
--create a procedure for method "purchase_order_create"
--drop procedure purchase_order_create;
create procedure purchase_order_create(
@items varchar(25)
)
AS
BEGIN
       INSERT INTO dbo.purchase_order VALUES
       (@items, 0, 0)
       PRINT 'The order is successfully created!'
END
GO

/*create procedure "purchase_order_approve"*/
--drop procedure purchase_order_approve;
create procedure purchase_order_approve(
@oId int
)
AS
DECLARE @count int
SELECT @count = 0

--check if the order (@oId) is existing?
SELECT @count = COUNT(*) FROM purchase_order WHERE oId = @oId

--update isApproved = true if the order is existing
IF(@count = 0)
       PRINT 'The order is not existing!'
ELSE
       BEGIN
              UPDATE purchase_order
              SET isApproved = 1
              WHERE oId = @oId
              PRINT 'The order is successfully approved!'
       END
GO

/*create procedure "purchase_order_receiveGoods"*/
--drop procedure purchase_order_receiveGoods;
create procedure purchase_order_receiveGoods(
@oId int
)
AS
DECLARE @count int
SELECT @count = 0

--check if the order (@oId) is existing?
SELECT @count = COUNT(*) FROM purchase_order WHERE oId = @oId
```

```
--update isReceived = true if the order is existing
IF(@count = 0)
	PRINT 'The order is not existing!'
ELSE
	BEGIN
		UPDATE purchase_order
		SET isReceived = 1
		WHERE oId = @oId
		PRINT 'The order is successfully completed!'
	END
GO

/*create procedure "purchase_order_modify"*/
--drop procedure purchase_order_modify;
create procedure purchase_order_modify(
@oId int,
@items varchar(25)
)
AS
DECLARE @count int
SELECT @count = 0

--check if the order (@oId) is existing?
SELECT @count = COUNT(*) FROM purchase_order WHERE oId = @oId

--update isReceived = true if the order is existing
IF(@count = 0)
	PRINT 'The order is not existing'
ELSE
	BEGIN
		UPDATE purchase_order
		SET oItems = @items
		WHERE oId = @oId
		PRINT 'The items of this order if successfully modified!'
	END
GO
```

An example of Call stored procedures in java methods (work on Eclipse)

Here is an example of a method calls a stored procedure, defined in class **FSStoreProcedure**

```
	/**
	 * Call the stored procedure 'purchase_order_approve' in the database,
	 *  which  execute  the  statement  of  modifying  the  value  of  the  column
'isApproved' to 'TRUE' on the table 'purchase_order'
	 * of a given order.
	 * @param _oId the identification of the order
	 * @return true if the stored procedure is executed (the statement returns the
row count),
	 * otherwise, return false (the statement returns nothing)
	 */
	public boolean purchase_order_approve(int _oId){
		boolean ret = false;
		CallableStatement statem;
		try {
```

```
                    statem  =  EnvironmentSystem.getDBConnection().prepareCall("{call
purchase_order_approve(?)}");
                    statem.setInt("oId", _oId);
                    int result = statem.executeUpdate();
                    if(result == 1){
                          System.out.println("[INFO]  Order  "  +  _oId  +  "  is
approved!");
                          ret = true;
                    }
                    statem.close();
                    return ret;
             } catch (SQLException e) {
//                  e.printStackTrace();
                    System.err.println("[SQL]  ERROR:  "  +  e.getErrorCode()  +":  "  +
e.getMessage());
                    return false;
             }
       }
```

Grant EXECUTE stored procedure:

Providing the right to execute the above stored procedures by executing the following SQL statements:

GRANT EXECUTE ON OBJECT::dbo.**purchase_order_create** TO staff;
GRANT EXECUTE ON OBJECT::dbo.**purchase_order_modify** TO staff;
GRANT EXECUTE ON OBJECT::dbo.**purchase_order_receiveGoods** TO staff;
GRANT EXECUTE ON OBJECT::dbo.**purchase_order_approve** TO manager;

5. Table method_audit

This table stores execution information of defined methods, including: the id of the object that the method acts on, the method name, the list of parameters of the method, the user who executed it, and the moment the method is completed. The SQL statement that creates this table is:

```
create table method_audit(
audit_id int identity(1,1) not null,
obj_id int not null,
method_name varchar(50) not null,
method_params varchar(50),
executed_user varchar(25),
executed_moment datetime DEFAULT CURRENT_TIMESTAMP,
primary key(audit_id)
);
```

List of considered methods in Purchase Order System :

- **Purchase_order_create :** insert new record into table purchase_order
- **Purchase_order_modify :** update value of column items= new items list of a given order on table purchase_order
- **Purchase_order_approve :** update the value of column isApproved = TRUE of a given order on table purchase_order
- **Purchase_order_receiveGoods :** update the value of column isReceivedgoods = TRUE of a given order on table purchase_order

Table method_audit

audit_id	Obj_id	Method_name	Method_params	Executed_user	Executed_moment

C. Test cases(work on Eclipse)

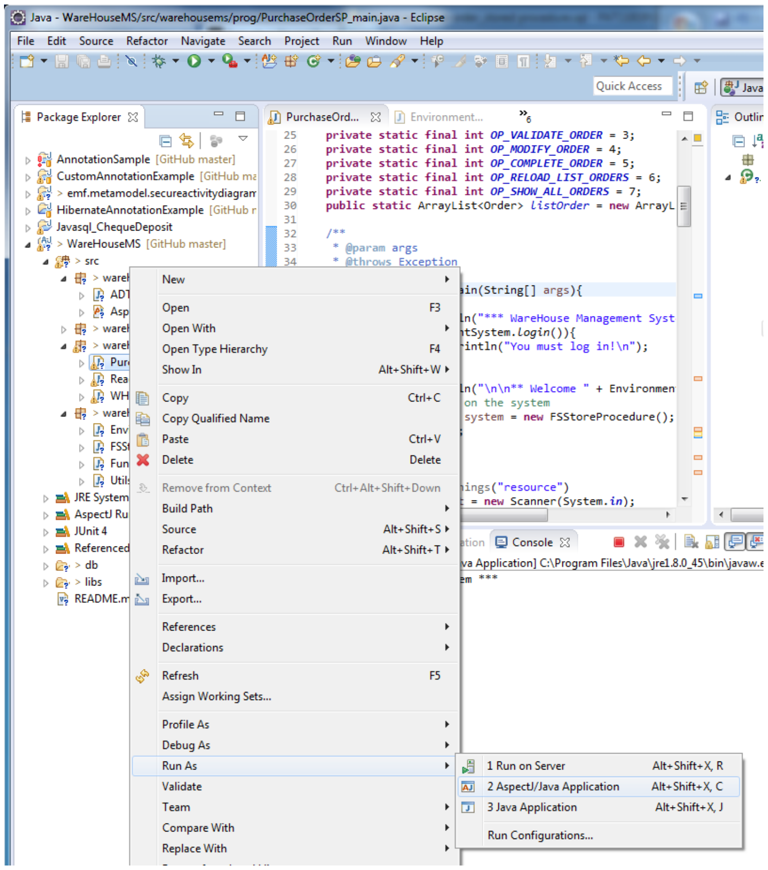
1) Run test cases

- Open Eclipse after dowload and extract it (see file *Software requirements*)
- Install AspectJ into Eclipse (see file *Software requirements*)
- Import the zip file **warehouseMS2.0.zip** into Eclipse
- Change ***DB_HOST*** to the IP address of computer where the DBMS is running:

private final static String ***DB_HOST*** = "???";

- Run class **PurchaseOrderSP_main** to login:

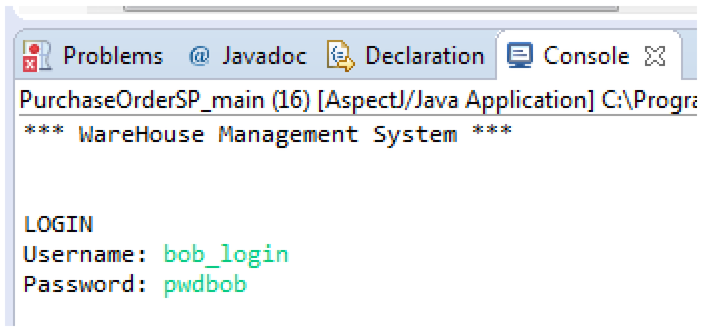
- Right click on **PurchaseOrderSP_main,** choose **Run as ➔ AspectJ/Java Application**



```
private static final int OP_VALIDATE_ORDER = 3;
private static final int OP_MODIFY_ORDER = 4;
private static final int OP_COMPLETE_ORDER = 5;
private static final int OP_RELOAD_LIST_ORDERS = 6;
private static final int OP_SHOW_ALL_ORDERS = 7;
public static ArrayList<Order> listOrder = new ArrayL

/**
 * @param args
 * @throws Exception
ain(String[] args){
ln("*** WareHouse Management Syst
ntSystem.login()){
rintln("You must log in!\n");
ln("\n\n** Welcome " + Environmen
on the system
system = new FSStoreProcedure();
nings("resource")
t = new Scanner(System.in);
```

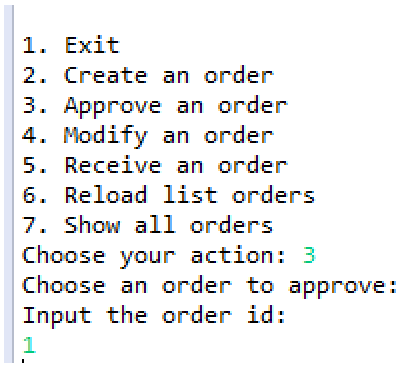
```
va Application] C:\Program Files\Java\jre1.8.0_45\bin\javaw.e
em ***
```

- Provide login information: Username and Password, which are defined in section **Security ➔User**



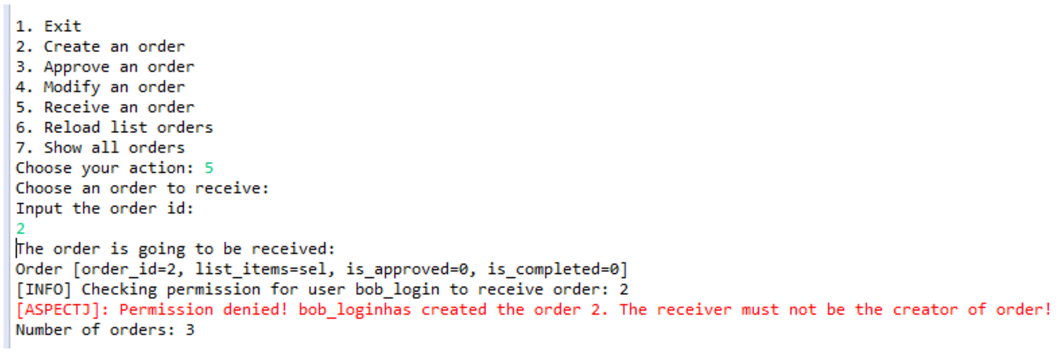
```
PurchaseOrderSP_main (16) [AspectJ/Java Application] C:\Progra
*** WareHouse Management System ***


LOGIN
Username: bob_login
Password: pwdbob
```

- Choose a method to execute from the list of actions
  - Give a number to call a method. For example, 3 to invoke the method **purchase_order_approve.**
  - Give a number representing the order id to choose the specific object that the method acts on.



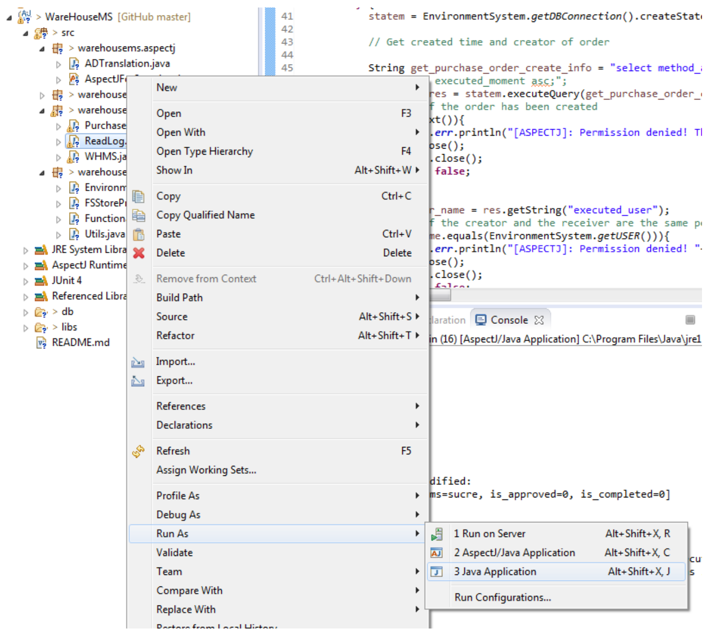
```
1. Exit
2. Create an order
3. Approve an order
4. Modify an order
5. Receive an order
6. Reload list orders
7. Show all orders
Choose your action: 3
Choose an order to approve:
Input the order id:
1
```

- Returned result
  - Invocation of a method checks:
    - the history-based constraints (by AspectJ) at the application level.
    - the user permission at the database level.
  - Result:
    - Denied right at the application level, then it does not reach the database level. Nothing is changed.
    - Granted (passed) the application level (fulfilled the history-based constraints), but denied at the database level. Nothing is changed.
    - Granted at both application and database levels, then this execution is logged.



```
1. Exit
2. Create an order
3. Approve an order
4. Modify an order
5. Receive an order
6. Reload list orders
7. Show all orders
Choose your action: 5
Choose an order to receive:
Input the order id:
2
The order is going to be received:
Order [order_id=2, list_items=sel, is_approved=0, is_completed=0]
[INFO] Checking permission for user bob_login to receive order: 2
[ASPECTJ]: Permission denied! bob_loginhas created the order 2. The receiver must not be the creator of order!
Number of orders: 3
```

- View the execution log
  - Choose class **ReadLog:** right click the **ReadLog ➔ Run As ➔ Java Application**
  - The result shows the list executed methods as well as the person who executed it and the timestamp when it is completed.



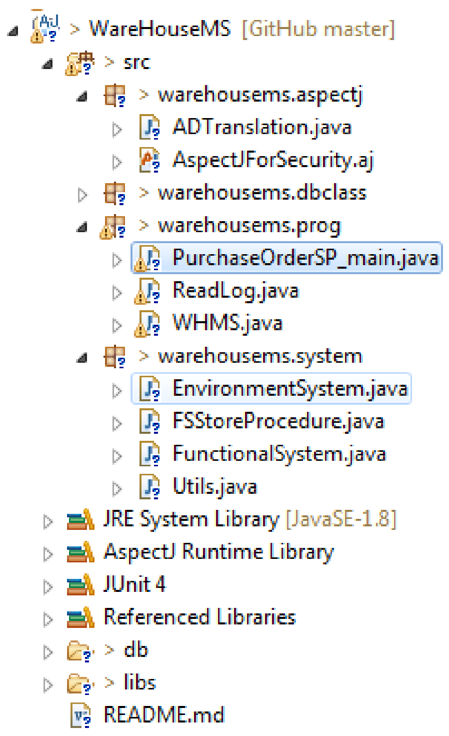
```
statem = EnvironmentSystem.getDBConnection().createStat

// Get created time and creator of order

String get_purchase_order_create_info = "select method_
 executed_moment asc;";
res = statem.executeQuery(get_purchase_order_
f the order has been created
xt()){
.err.println("[ASPECTJ]: Permission denied! T
ose();
.close();
 false;

r_name = res.getString("executed_user");
f the creator and the receiver are the same p
me.equals(EnvironmentSystem.getUSER())){
.err.println("[ASPECTJ]: Permission denied! "
ose();
.close();
```

```
n (16) [AspectJ/Java Application] C:\Program Files\Java\jre1
dified:
ms=sucre, is_approved=0, is_completed=0]
```

2) Purchase Order (Ware House) System components:

- **PurchaseOrderSP_main:** the main class
- **FSStoreProcedure:** the functional class which defines methods calling stored procedures in the database.
- **ADTranslation** defines history constraints of the methods (operations) to be secure.
- **AspectJForSecurity** defines a pointcut to each method to be secure. i.e. An interception occurs when a method to be secure is called to check its dynamic constraints before execute the method ifself. It also logs the execution of this method if it is successfully executed.
- **ReadLog** the class reads the execution log.
- **EnvironmentSystem** connects to JDBC. Remember to change ***DB_HOST*** to the IP address of computer where the DBMS is running:

private final static String ***DB_HOST*** = "157.159.110.133";



```
WareHouseMS [GitHub master]
  src
    warehousems.aspectj
      ADTranslation.java
      AspectJForSecurity.aj
    warehousems.dbclass
    warehousems.prog
      PurchaseOrderSP_main.java
      ReadLog.java
      WHMS.java
    warehousems.system
      EnvironmentSystem.java
      FSStoreProcedure.java
      FunctionalSystem.java
      Utils.java
  JRE System Library [JavaSE-1.8]
  AspectJ Runtime Library
  JUnit 4
  Referenced Libraries
  db
  libs
  README.md
```

3) Security rules:

1) *Only users whose role is staff can create, modify, receive an order*
2) *Only manager can approve an order*
3) *The order is signed off only after it is approved and the person receives goods should not be the person who created the order*
4) *The person modifies an order must be the person who created it. Modification of an order can be performed several times, but not after the order is approved*

a. Authorized actions

✓ User logins as "allice_login" calls method **purchase_order_create(string _items)** **(***assuming the order id is _oId***)**

- Expected result: **succeeded**

Because Allice has permission to execute method **purchase_order_create (**rule 1).

✓ User logins as "allice_login" calls method **purchase_order_modify (int** _oId**)**

- Expected result: **succeeded**

Because Allice has right to perform method **purchase_order_modify** (rule 1), and Allice created the order _oId (rule 4).

✓ User logins as "tom_login" calls method **purchase_order_approve (int** _oId**)**

- Expected result: **succeeded**

Because Tom has right to perform method **purchase_order_approve (**rule 2)

- ✓ User logins as “bob_login” calls method **Purchase_order_modify (int _oId)** when the order is not yet approved.
  - ○ Expected result: **succeeded**

  Because Bob is a staff who has right to perform **Purchase_order_modify ,** and the order is not yet approved (rule 4).

- ✓ User logins as “bob_login” calls method **Purchase_order_receiveGoods (int _oId)** after the order is approved by Tom.
  - ○ Expected result: **succeeded**

  Because Bob didn’t create the order _oId and the order is already approved (rule 3).

b. Malicious actions

- ✓ User logins as “tom_login” calls method **purchase_order_create:**
  - ○ Expected result: **failed**

  Because Tom is a manager which does not have permission to execute method **purchase_order_create,** according to rule 1.

- ✓ User logins as “bob_login” calls method **purchase_order_modify (int _oId)**
  - ○ Expected result: **failed**

  Because althought Bob has permission to perform method **purchase_order_modify, the order** _oId is created by Allice. Execution failed according to rule 3.

- ✓ User logins as “bob_login” calls method **purchase_order_approve (int _oId)**
  - ○ Expected result: **failed**

  Because Bob is a staff which does not have right to perform method **purchase_order_approve** (rule 2)

- ✓ User logins as “allice_login” calls method **Purchase_order_receiveGoods (int _oId)**
  - ○ Expected result: **failed**

  Because Allice created the order (rule 3)

- ✓ User logins as “bob_login” calls method **Purchase_order_receiveGoods (int _oId)** when the order is not yet approved.
  - ○ Expected result: **failed**

  Because the order is not yet approved (rule 3).

- ✓ User logins as “bob_login” calls method **Purchase_order_modify (int _oId)** when the order is approved by Tom.
  - ○ Expected result: **failed**

  Because the order cannot be modifed when it is already approved (rule 4).