FEUP Universidade do Porto
Faculdade de Engenharia

Industrial Informatics

[Informática Industrial]

2022/23 edition

intro Lazarus planning module

v2

José Faria, Andry Pinto

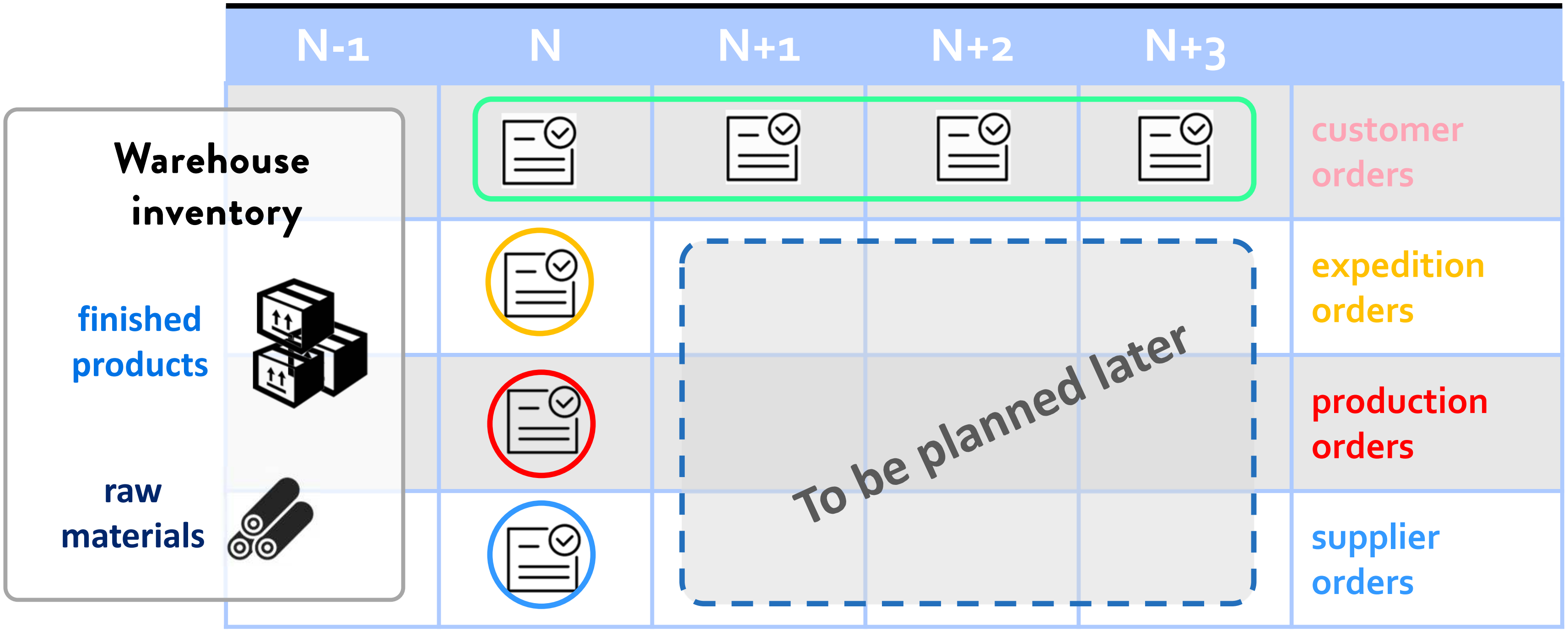
Contents

1. Introduction to the planning module
2. Important points to pay (lots of) attention
3. 1st example: Expedition orders – insert
4. 2nd example: Expedition orders – insert & update
5. 3rd example: Expedition orders – 1st with layers
6. 4th example: Expedition orders – ALL with layers

1. Introduction to the planning module

Introduction

- In a previous class, we introduced the planning module of the miniERP.



N-1 N N+1 N+2 N+3

Warehouse inventory

finished products

raw materials

customer orders

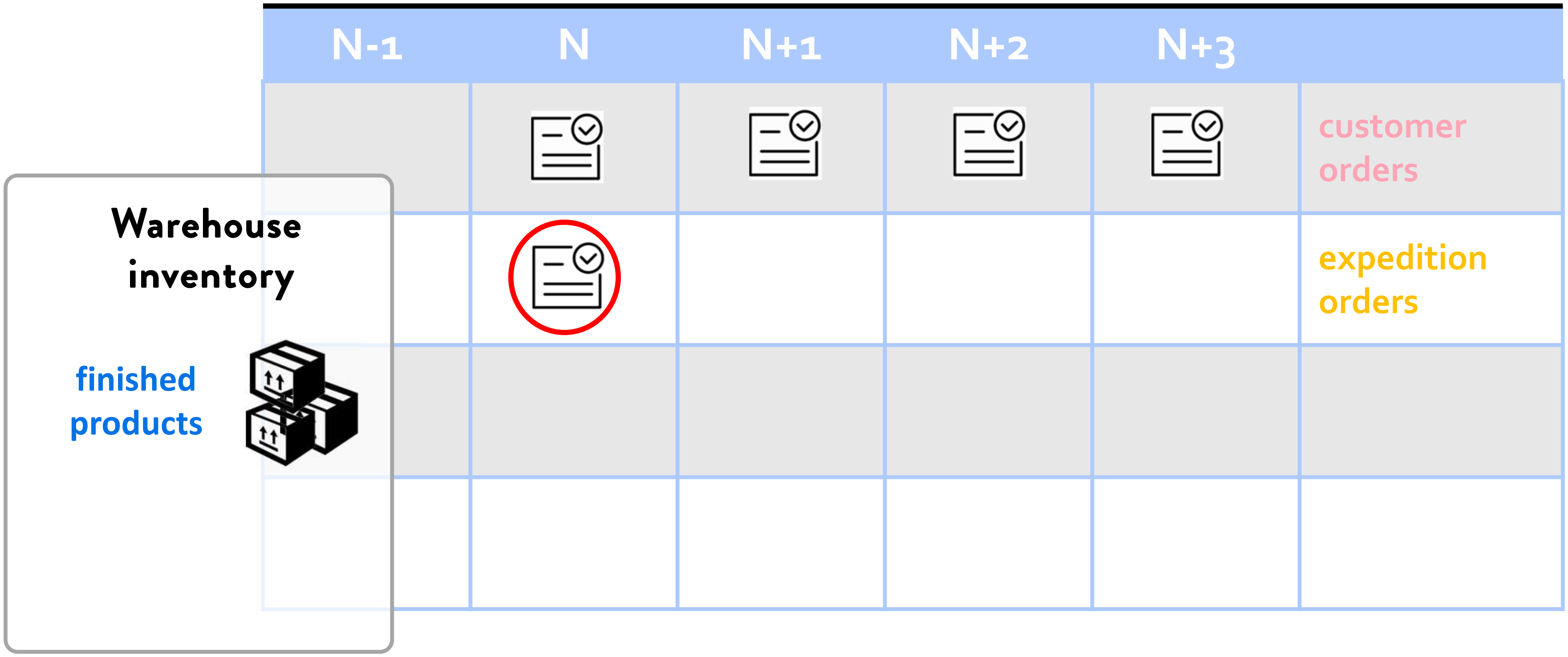
expedition orders

production orders

supplier orders

To be planned later

- Today, we will see **how to implement it in Lazarus**.
- In the class, we will **just** analyze the planning of the **expedition orders**.



N-1 N N+1 N+2 N+3

customer orders

expedition orders

Warehouse inventory

finished products

Expedition orders planning

Assuming
we “are” here

Quantities for product type P1

Quantities for product type P2

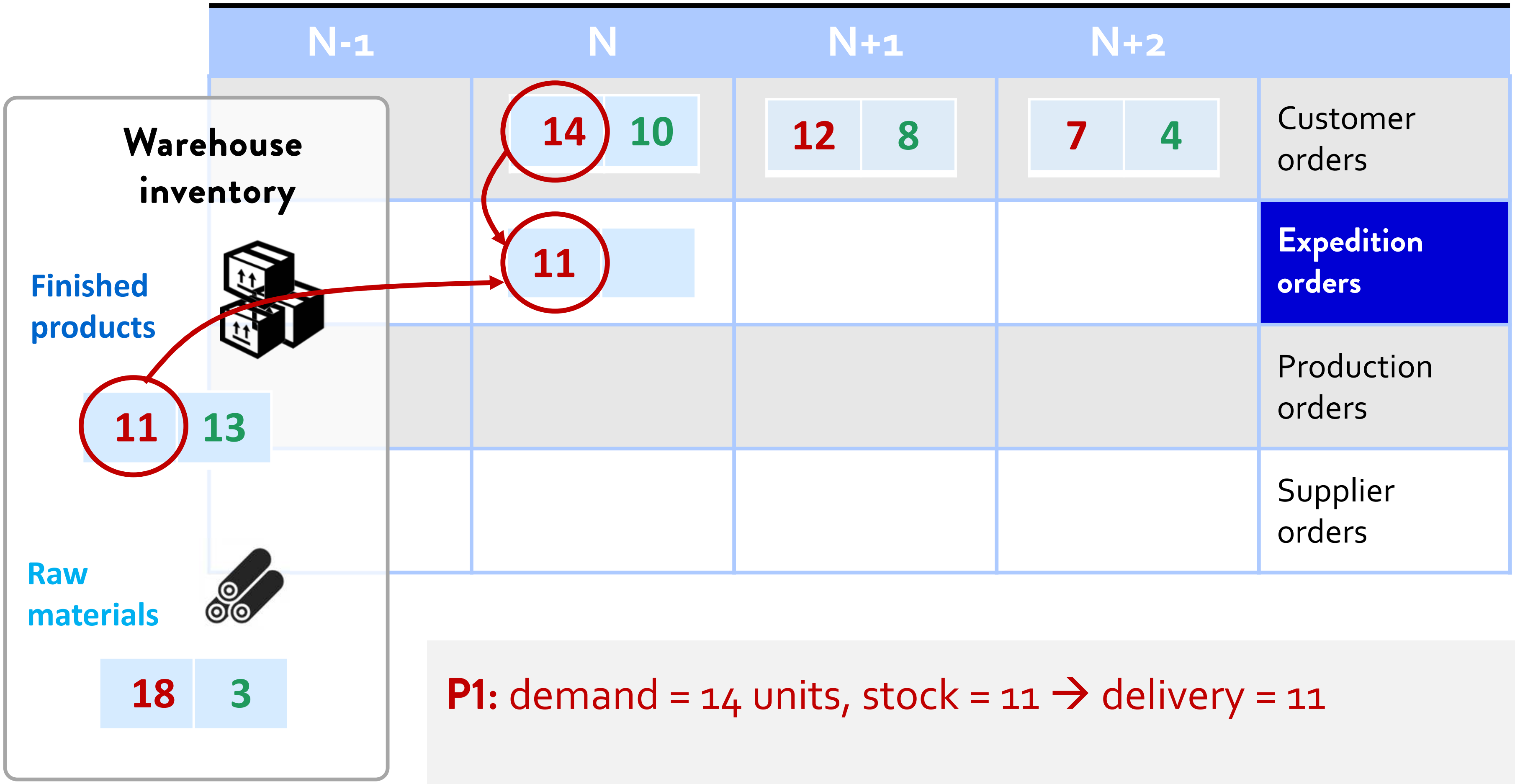
N-1	N	N+1	N+2	
	14 10	12 8	7 4	Customer orders
	?? ??			**Expedition orders**
				Production orders
				Supplier orders

Warehouse inventory

	P1	P2
Finished products	11	13
Raw materials	18	3

How many items of P1 and P2 should be planned for expedition in week N?

Expedition orders planning



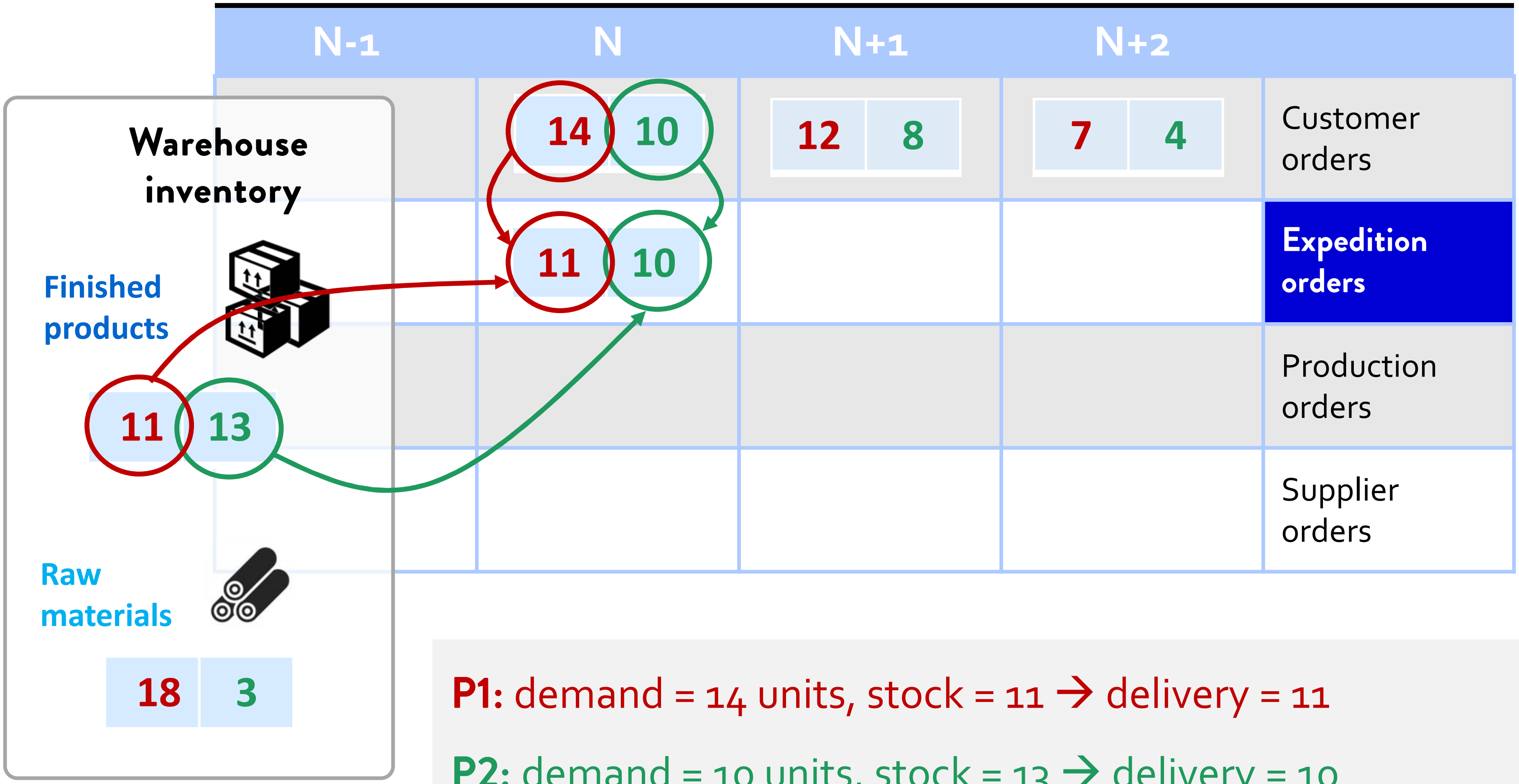
	N-1	N	N+1	N+2	
		14 10	12 8	7 4	Customer orders
		11			**Expedition orders**
					Production orders
					Supplier orders

Warehouse inventory

Finished products	11	13
Raw materials	18	3

P1: demand = 14 units, stock = 11 → delivery = 11

Expedition orders planning



	N-1	N	N+1	N+2	
		14 | 10	12 | 8	7 | 4	Customer orders
		11 | 10			**Expedition orders**
					Production orders
					Supplier orders

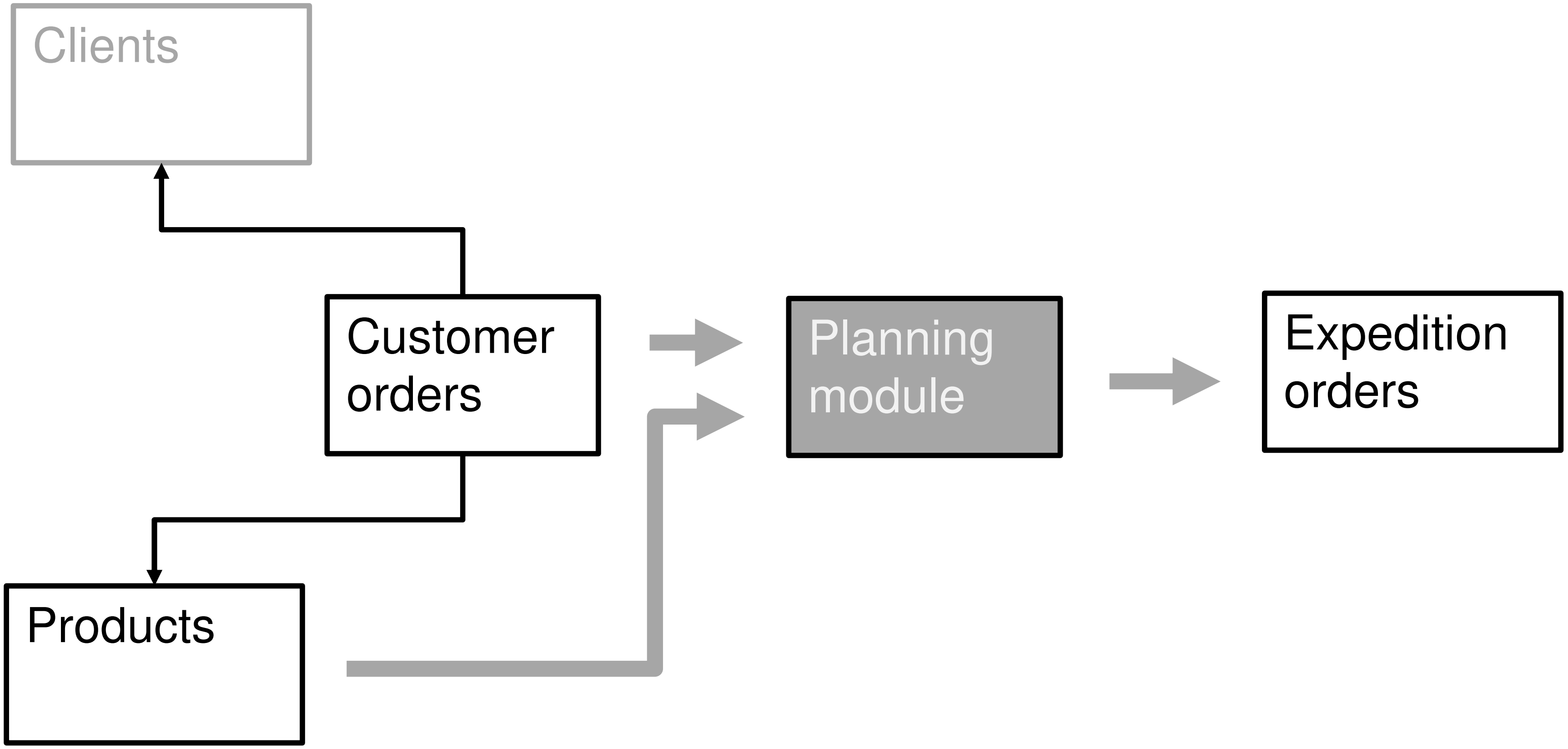
Warehouse inventory

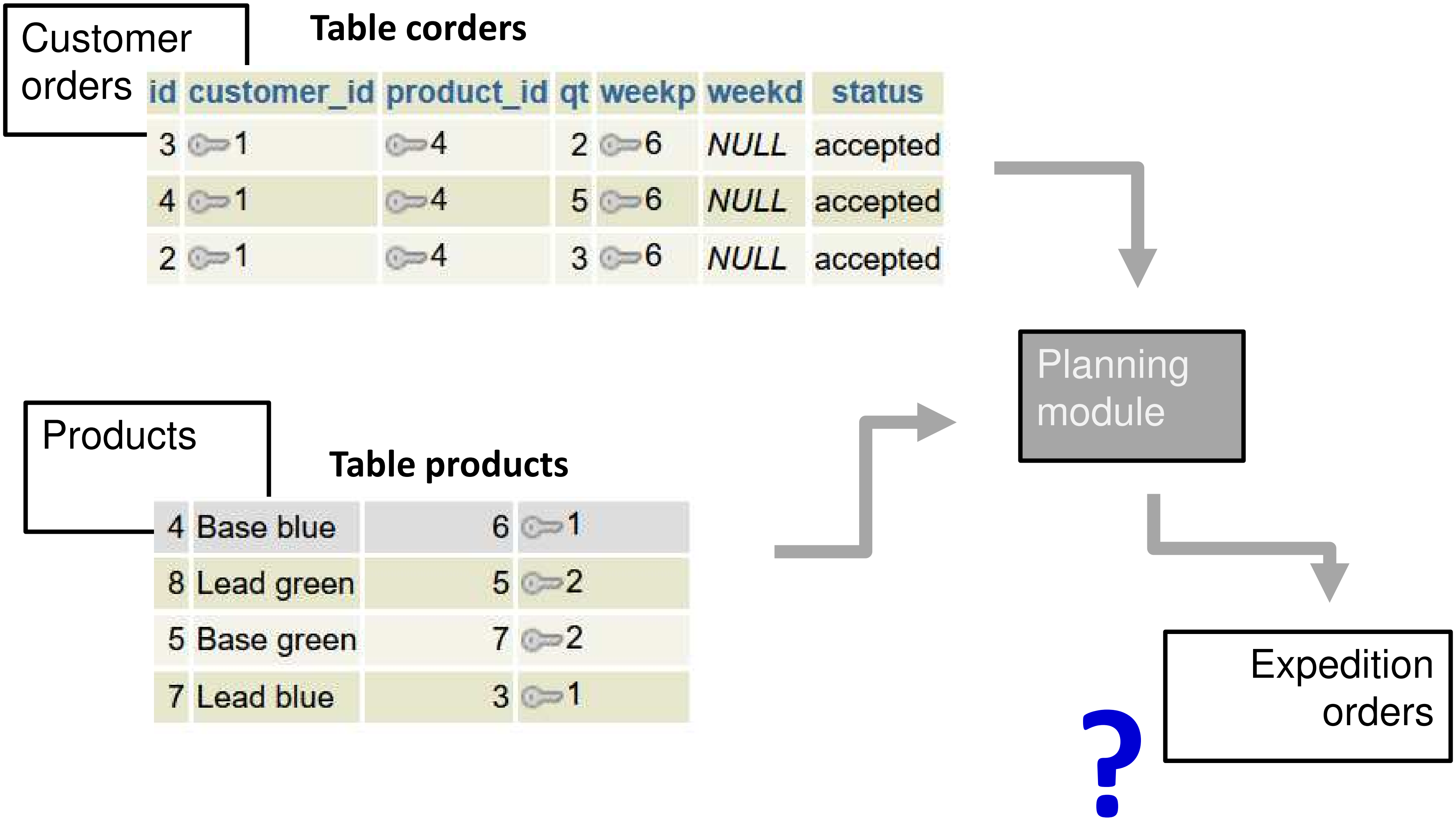
Finished products	11	13
Raw materials	18	3

P1: demand = 14 units, stock = 11 → delivery = 11

P2: demand = 10 units, stock = 13 → delivery = 10

Planning module





Customer orders

Table corders

id	customer_id	product_id	qt	weekp	weekd	status
3	1	4	2	6	*NULL*	accepted
4	1	4	5	6	*NULL*	accepted
2	1	4	3	6	*NULL*	accepted

Products

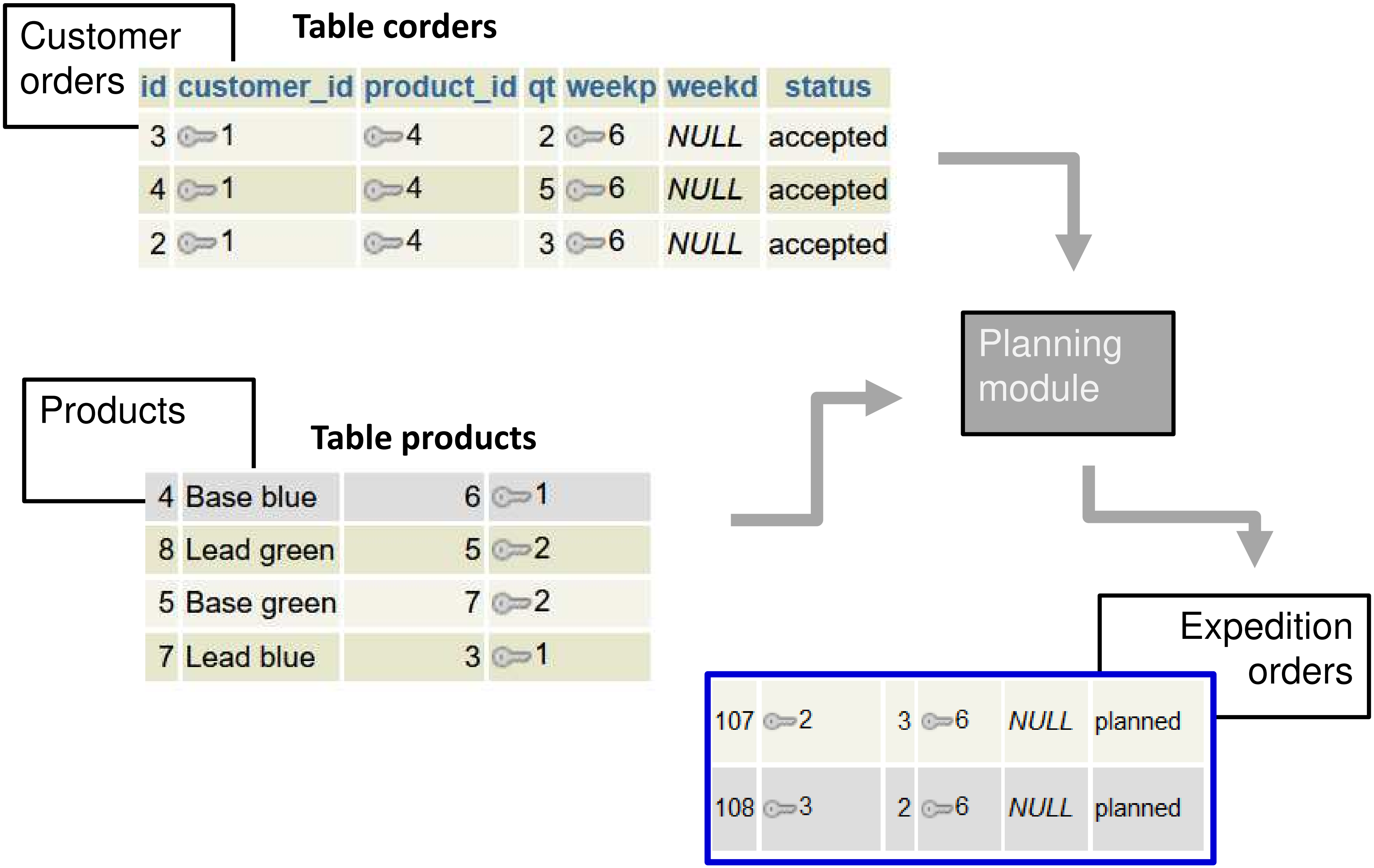
Table products

4	Base blue	6	1
8	Lead green	5	2
5	Base green	7	2
7	Lead blue	3	1

Planning module

Expedition orders

?



Customer orders

Table corders

id	customer_id	product_id	qt	weekp	weekd	status
3	1	4	2	6	*NULL*	accepted
4	1	4	5	6	*NULL*	accepted
2	1	4	3	6	*NULL*	accepted

Products

Table products

4	Base blue	6	1
8	Lead green	5	2
5	Base green	7	2
7	Lead blue	3	1

Expedition orders

107	2	3	6	*NULL*	planned
108	3	2	6	*NULL*	planned

2. Important points to pay (lots of) attention

- Before going to the examples,

- let’s emphasize some **very important points** when analyzing them and developing your own applications

1. Quality of the software

We will pay **a lot of attention** to the quality of the software, namely:

1. Architecture (global organization)
2. Coherence (naming of variables, procedures, etc.)
3. Indentation
4. Comments (explain the code and identify its main sections)

2. 3 Layers architecture

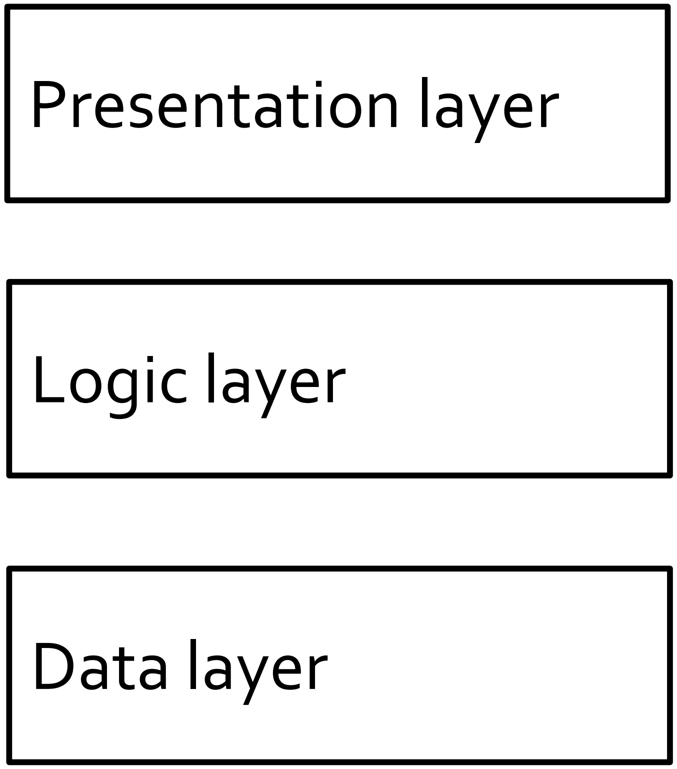
The code will be organized in 3 layers:

Presentation layer → user interface, controls

Logic layer → business rules, events

Data layer → database access, SQL functions

2. 3 Layers architecture



Presentation layer

Logic layer

Data layer

You should **understand** the rationale of the 3 layers architecture and **apply** it in the miniERP application!

3. Step by step approach

As in the two previous class, we will proceed a step-by-step:

- there is an accompanying **version** of the application **for each step**
- each version adds a **new functionality on top** of the previous version, as so …

… you should only move to the next version after having understood well the previous one

4. Coding patterns

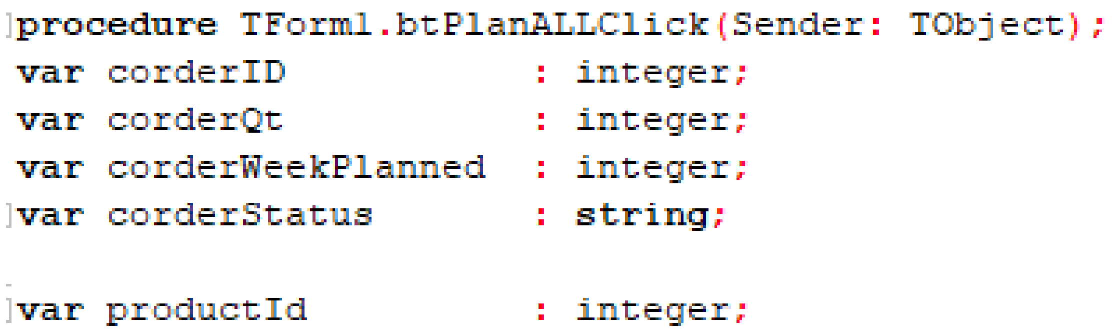
- The programming **patterns followed in the class examples were those in** the next slides will be followed.
- When developing your own miniERP application, you may adopt other patterns but …

… as far as possible, you should be consistent along the code and stick to the patterns you chose.

4. Coding patterns

Naming of Lazarus

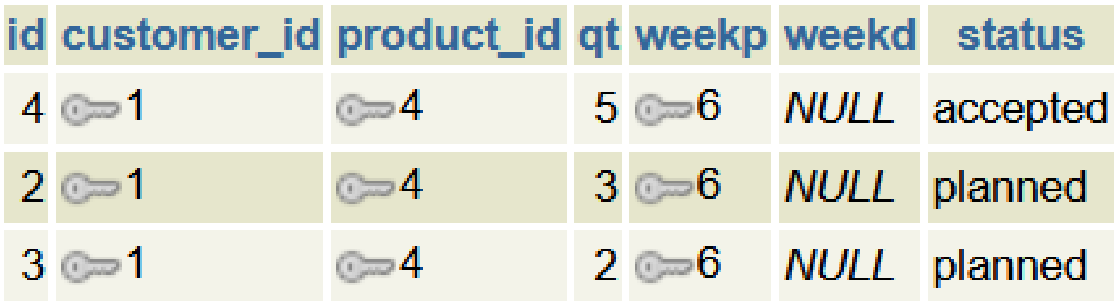
variables and procedures



```
procedure TForm1.btPlanALLClick(Sender: TObject);
var corderID            : integer;
var corderQt            : integer;
var corderWeekPlanned   : integer;
var corderStatus        : string;

var productId           : integer;
```

Naming of PostgreSQL

tables and columns

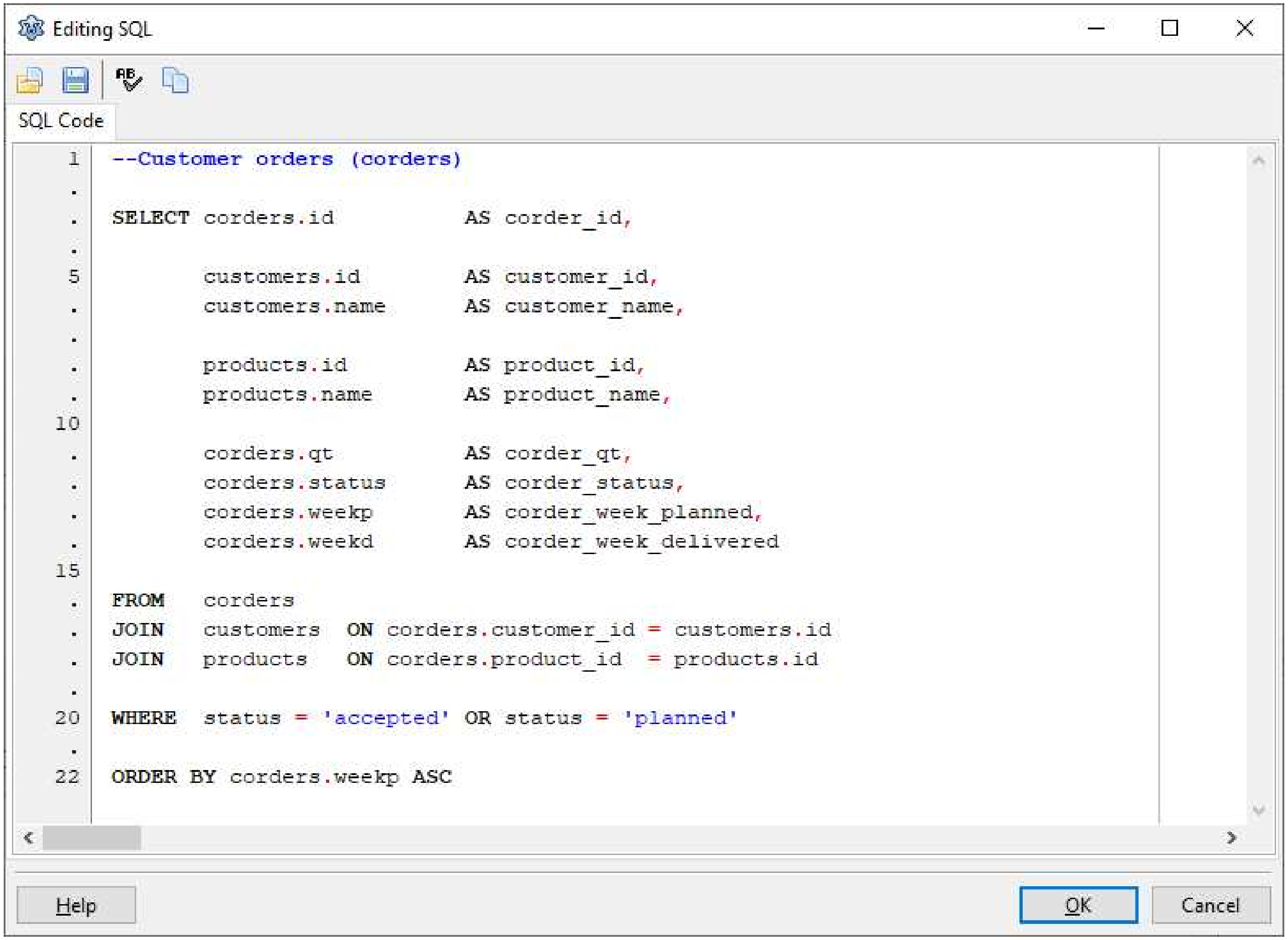


id	customer_id	product_id	qt	weekp	weekd	status
4	🔑1	🔑4	5	🔑6	*NULL*	accepted
2	🔑1	🔑4	3	🔑6	*NULL*	planned
3	🔑1	🔑4	2	🔑6	*NULL*	planned

4. Coding patterns

SQL

indentation

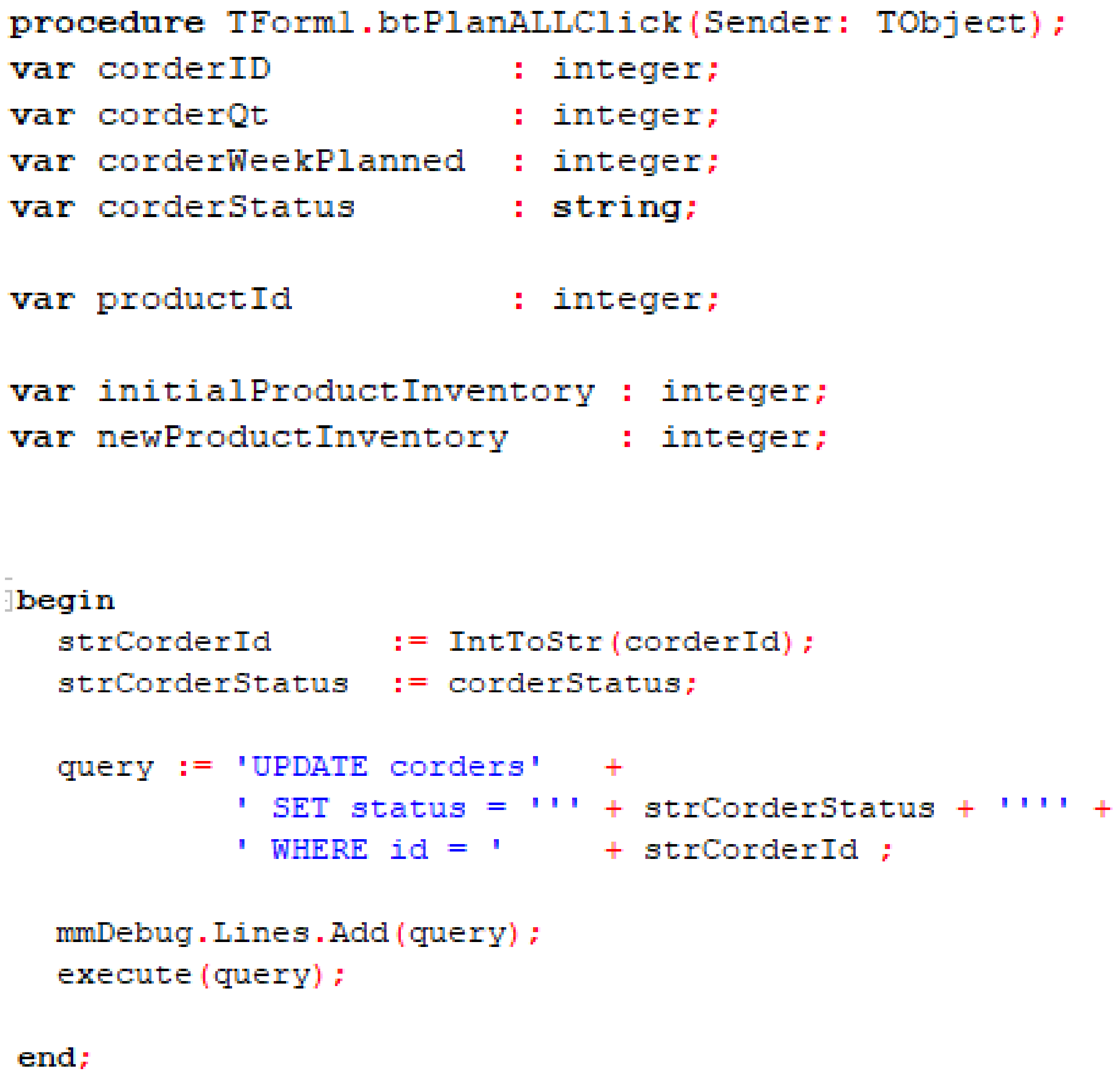


```
--Customer orders (corders)

SELECT corders.id          AS corder_id,

       customers.id        AS customer_id,
       customers.name      AS customer_name,

       products.id         AS product_id,
       products.name       AS product_name,

       corders.qt          AS corder_qt,
       corders.status      AS corder_status,
       corders.weekp       AS corder_week_planned,
       corders.weekd       AS corder_week_delivered

FROM   corders
JOIN   customers  ON corders.customer_id = customers.id
JOIN   products   ON corders.product_id  = products.id

WHERE  status = 'accepted' OR status = 'planned'

ORDER BY corders.weekp ASC
```

4. Coding patterns

Lazarus

indentation



```
procedure TForm1.btPlanALLClick(Sender: TObject);
var corderID              : integer;
var corderQt              : integer;
var corderWeekPlanned     : integer;
var corderStatus          : string;

var productId             : integer;

var initialProductInventory : integer;
var newProductInventory     : integer;



begin
  strCorderId       := IntToStr(corderId);
  strCorderStatus   := corderStatus;

  query := 'UPDATE corders'   +
           ' SET status = '''  + strCorderStatus + '''' +
           ' WHERE id = '      + strCorderId ;

  mmDebug.Lines.Add(query);
  execute(query);

end;
```

1st example: Expedition orders - insert

1. Download from Moodle:

- The database dump minierp
- The zip file of the first example

2. Change the owner of the database in the dump file, and import it to your database in PostgreSQL

…

…

3. Unzip the file with the example

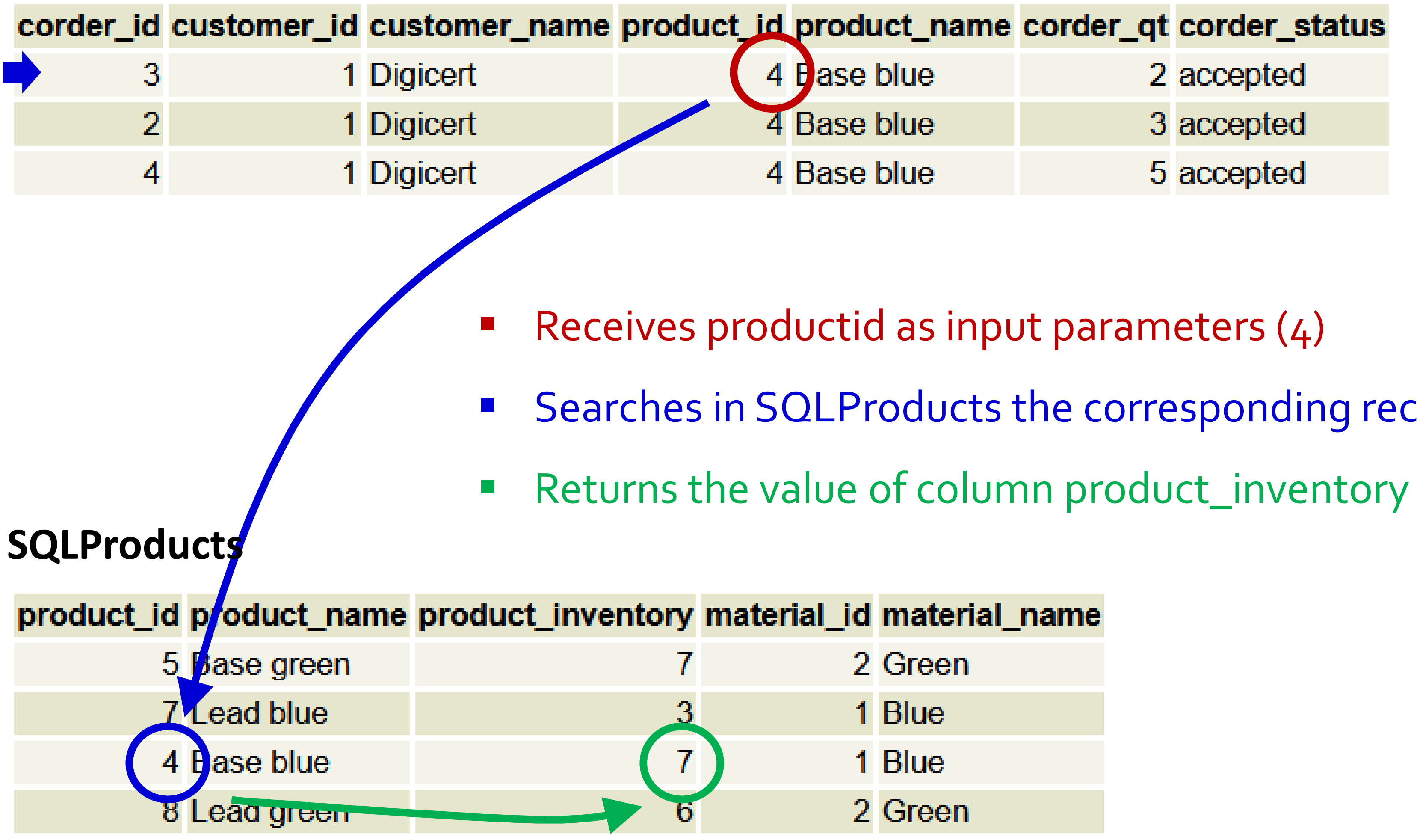
4. Open the Final version

5. Change the database name and your credentials in PQConnectionFEUP

6. Run de application

7. Read the **Help** and analyze the **code** of the application, in particular function **getInventoryByProductId()**

function getInventoryByProductId()



SQLCorders

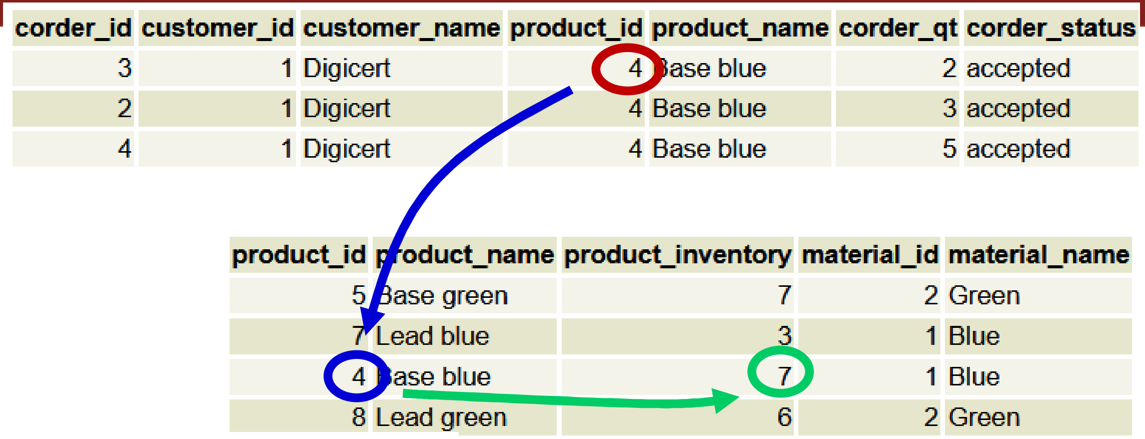
RecNo 1

corder_id	customer_id	customer_name	product_id	product_name	corder_qt	corder_status
3	1	Digicert	4	Base blue	2	accepted
2	1	Digicert	4	Base blue	3	accepted
4	1	Digicert	4	Base blue	5	accepted

- Receives productid as input parameters (4)
- Searches in SQLProducts the corresponding record (3)
- Returns the value of column product_inventory (38)

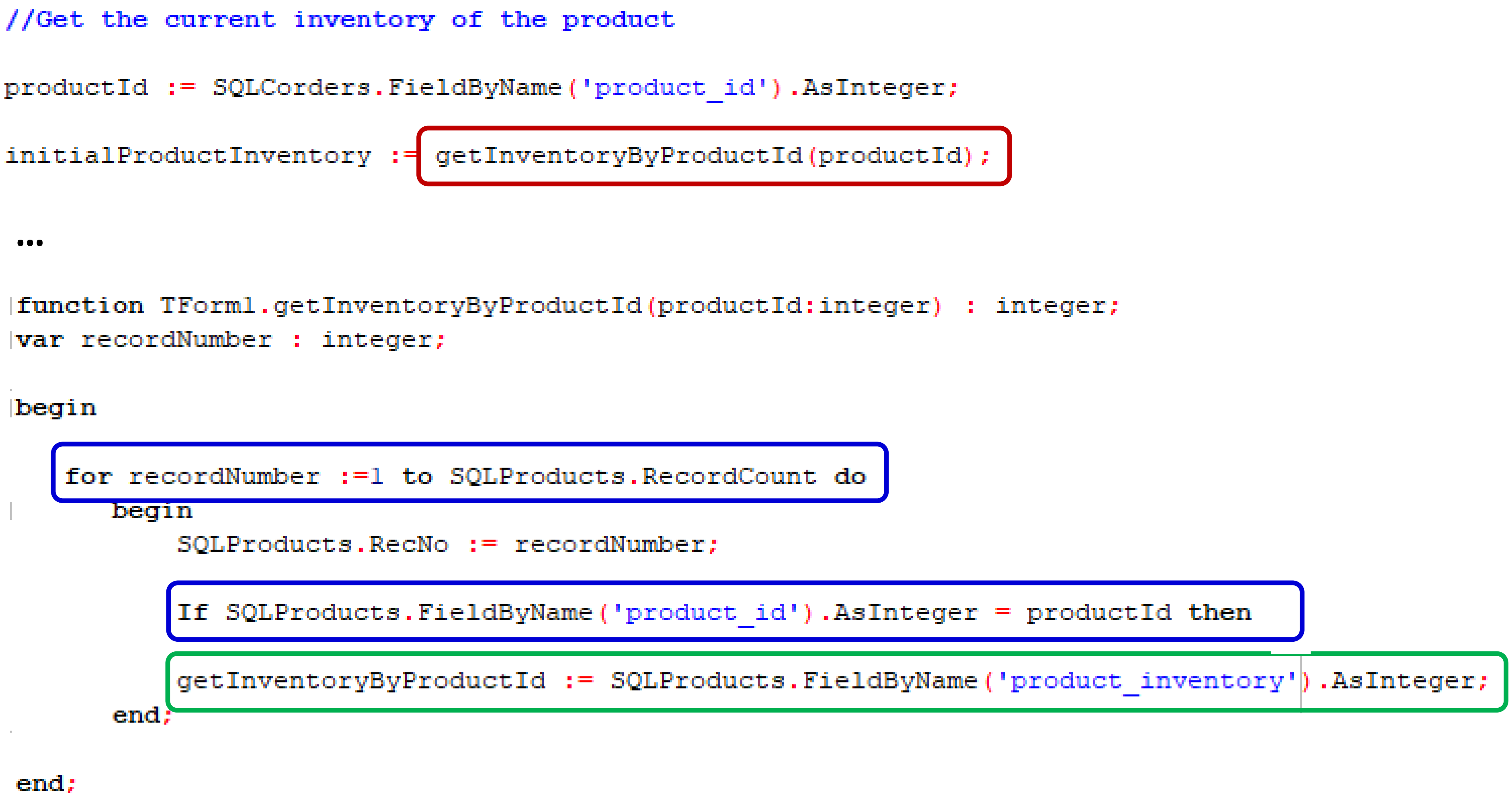
SQLProducts

product_id	product_name	product_inventory	material_id	material_name
5	Base green	7	2	Green
7	Lead blue	3	1	Blue
4	Base blue	7	1	Blue
8	Lead green	6	2	Green



corder_id	customer_id	customer_name	product_id	product_name	corder_qt	corder_status
3	1	Digicert	4	Base blue	2	accepted
2	1	Digicert	4	Base blue	3	accepted
4	1	Digicert	4	Base blue	5	accepted

product_id	product_name	product_inventory	material_id	material_name
5	Base green	7	2	Green
7	Lead blue	3	1	Blue
4	Base blue	7	1	Blue
8	Lead green	6	2	Green



```
//Get the current inventory of the product

productId := SQLCorders.FieldByName('product_id').AsInteger;

initialProductInventory := getInventoryByProductId(productId);


 ...

function TForm1.getInventoryByProductId(productId:integer) : integer;
var recordNumber : integer;

begin

  for recordNumber :=1 to SQLProducts.RecordCount do
      begin
          SQLProducts.RecNo := recordNumber;

          If SQLProducts.FieldByName('product_id').AsInteger = productId then

          getInventoryByProductId := SQLProducts.FieldByName('product_inventory').AsInteger;
      end;

end;
```

2nd example: Expedition orders – insert & update

1. Download from Moodle the zip file of the second example

2. Open the Final version, run de application and read the Help memo

3. Open the Working version and complete the missing sections of the code

Help on the creation of the dynamic query

If you are having difficulties, proceed as follows:

1. Start by editing a **static query**, test it in the SQL window of PostgreSQL
2. Copy/paste it to the application and **test it** by running the application
3. If it is ok, **replace the 1st static parameter** by a dynamic parameter and run the application
4. If it is ok, then **replace the next parameter**

...

Help on the creation of the dynamic query

- At first glance, this approach **may seem more time consuming** than directly editing the dynamic query
- However, if you take into account the time you will need to **debug it**,
- **I assure you** that it is **very worthwhile !**

3rd example: Expedition orders – 1st with layers

The code will be organized in 3 layers:

Presentation layer → user interface, controls

Logic layer → business rules, events

Data layer → database access, SQL functions

1. Download from Moodle the zip files of the third version

2. Open the Final version

3. Run the application and read the Help memo

4. Carefully analyze the code, paying special attention to the layered organization of the code, particularly to the **invocation of the data layer functions**…

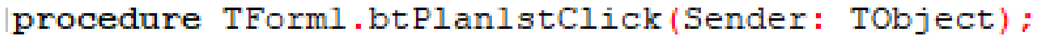
no layers architecture
(standard):

--

the SQL queries are mixed with the logic code (plan procedure)

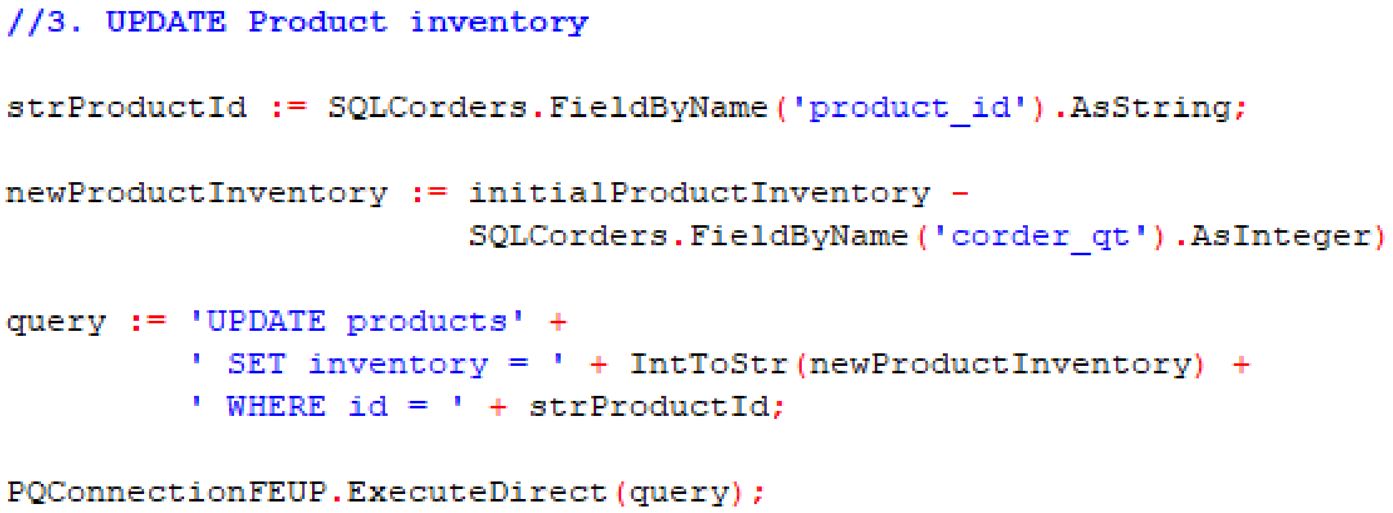
--

versions 1 and 2 of the example



```
procedure TForm1.btPlan1stClick(Sender: TObject);
```

...



```
//3. UPDATE Product inventory

strProductId := SQLCorders.FieldByName('product_id').AsString;

newProductInventory := initialProductInventory -
                       SQLCorders.FieldByName('corder_qt').AsInteger)

query := 'UPDATE products' +
         ' SET inventory = ' + IntToStr(newProductInventory) +
         ' WHERE id = ' + strProductId;

PQConnectionFEUP.ExecuteDirect(query);
```

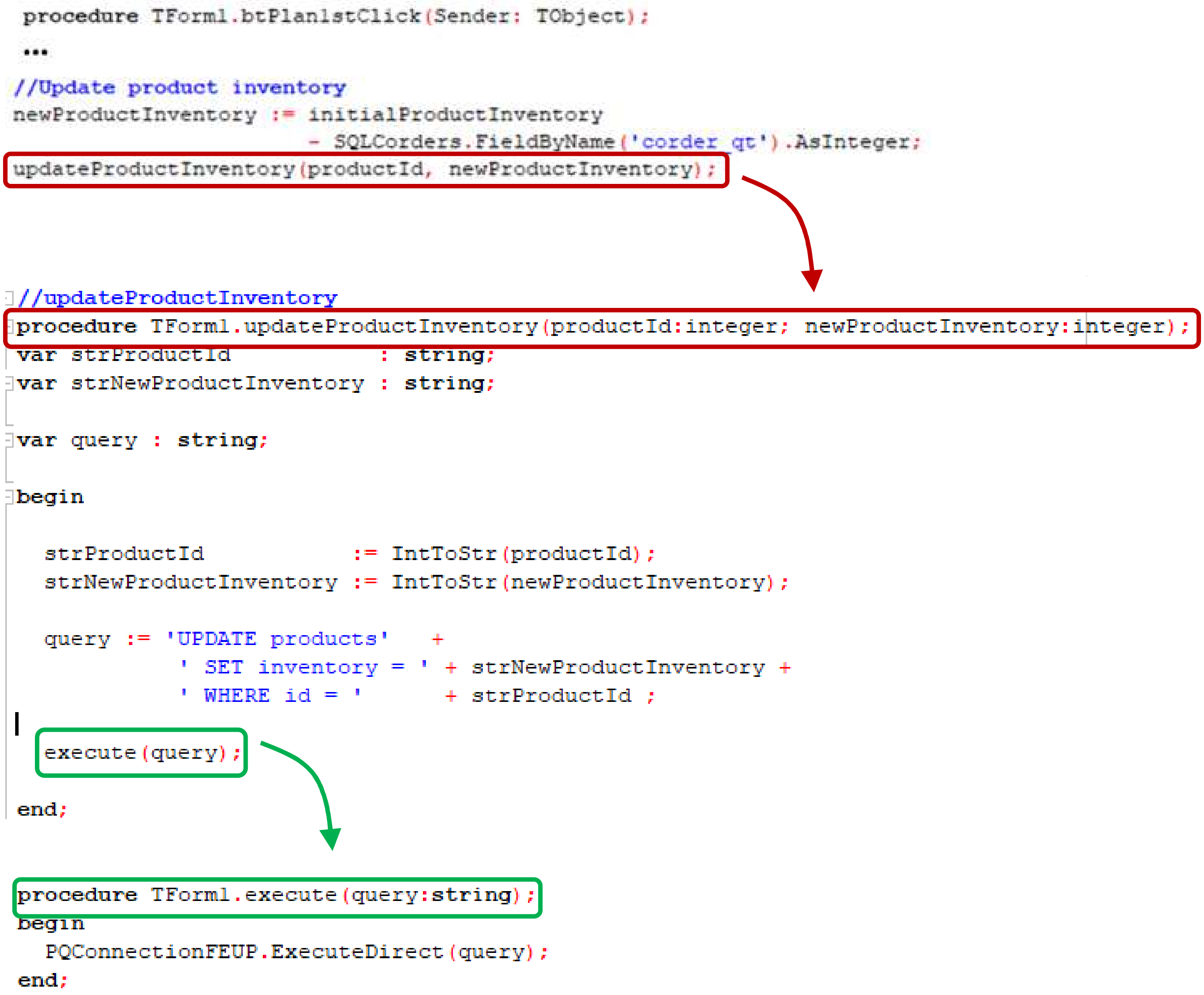
layered architecture

--

the SQL queries in the data layer are apart from the logic layer

--

versions 3 and 4 of the example



```
procedure TForm1.btPlan1stClick(Sender: TObject);
...
//Update product inventory
newProductInventory := initialProductInventory
                       - SQLCorders.FieldByName('corder_qt').AsInteger;
updateProductInventory(productId, newProductInventory);
```

```
//updateProductInventory
procedure TForm1.updateProductInventory(productId:integer; newProductInventory:integer);
var strProductId           : string;
var strNewProductInventory : string;

var query : string;

begin

  strProductId           := IntToStr(productId);
  strNewProductInventory := IntToStr(newProductInventory);

  query := 'UPDATE products'    +
           ' SET inventory = ' + strNewProductInventory +
           ' WHERE id = '      + strProductId ;

  execute(query);

end;
```

```
procedure TForm1.execute(query:string);
begin
  PQConnectionFEUP.ExecuteDirect(query);
end;
```

…

5. Now, open the Working version

6. Edit the sections highlighted in the code to:

a) replace the dynamic query code in
section 3. UPDATE Product inventory of btPlan1stClick()
by the invocation of the procedure updateProductInventory()

b) in the data layer, edit the code of updateProductInventory()

4th example: Expedition orders – ALL with layers

1. Download from Moodle the zip files of the fourth version

2. Open the Working version and complete the missing sections of the code that all the customer orders aer planned, not only the 1st one

5th example: Expedition orders – with Units

Introduction

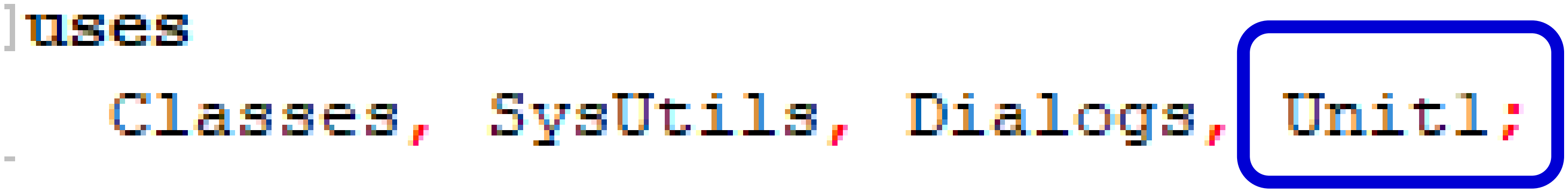
- In this final version, you are going to move the data and presentation procedures to 2 new units – DataLayer and PresentationLayer – so the code becomes much easier to maintain.
- As usual:
  - Start by downloading the 5th example from Moodle
  - Open the application and analyze the new organization of the code

Introduction

- Once you are familiar to the organization of the code, open the WORKING version of the example.
- This example is identical to the FINAL version of example 4.
- Follow the instructions in the following slides to create the 2 new units and move the data and presentation procedures in Unit1 to these new units.

Setup the 2 new units

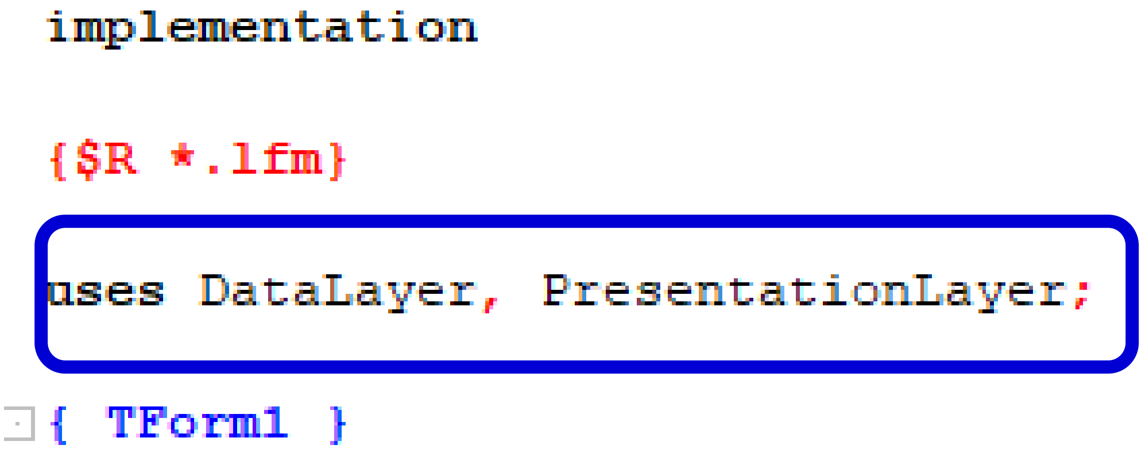
- Create two new units (File\New unit) and change their names to DataLayer and PresentationLayer
- Open the .lpr file and the change the names of the new units from unit2 and unit3 to DataLayer and PresentationLayer.
- In the use clause, add Unit1 so that this new units get access to objects in Unit1 and Form1



```
uses
  Classes, SysUtils, Dialogs, Unit1;
```

Setup the 2 new units

- Now, go to Unit1 and add the use clause as follow, so that Unit1 ge access to the procedures in the two new units

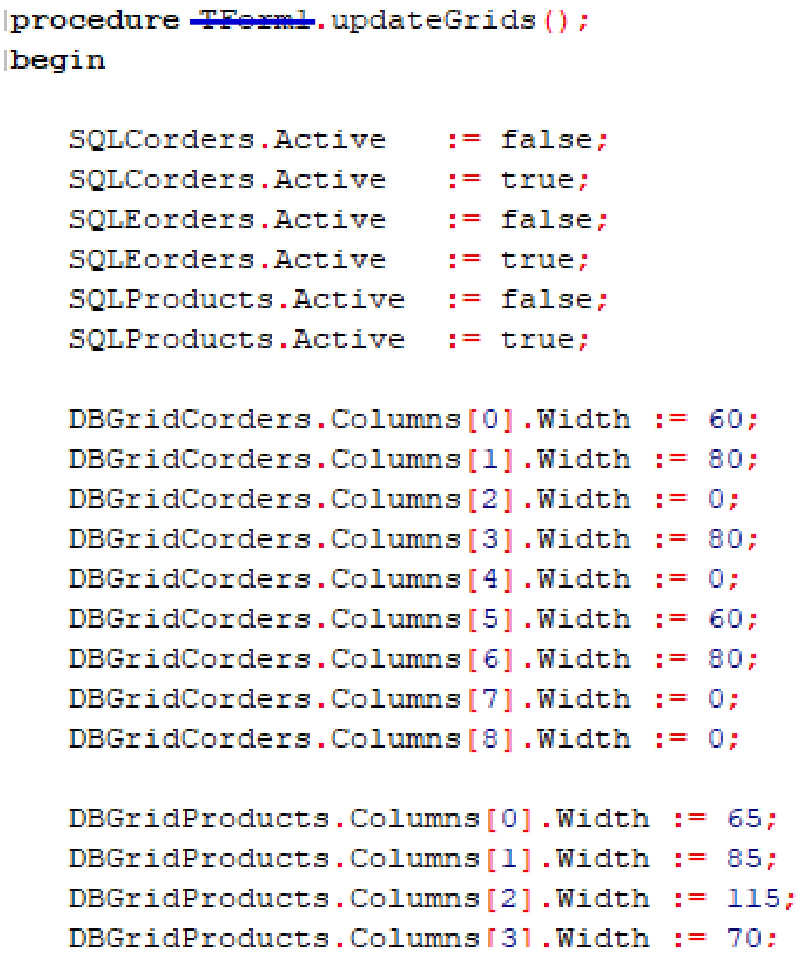


```
implementation

{$R *.lfm}

uses DataLayer, PresentationLayer;

{ TForm1 }
```

Moving the presentation layer

- Copy/paste the procedure updateGrids() from Unit1 to the new unit PresentationLayer:

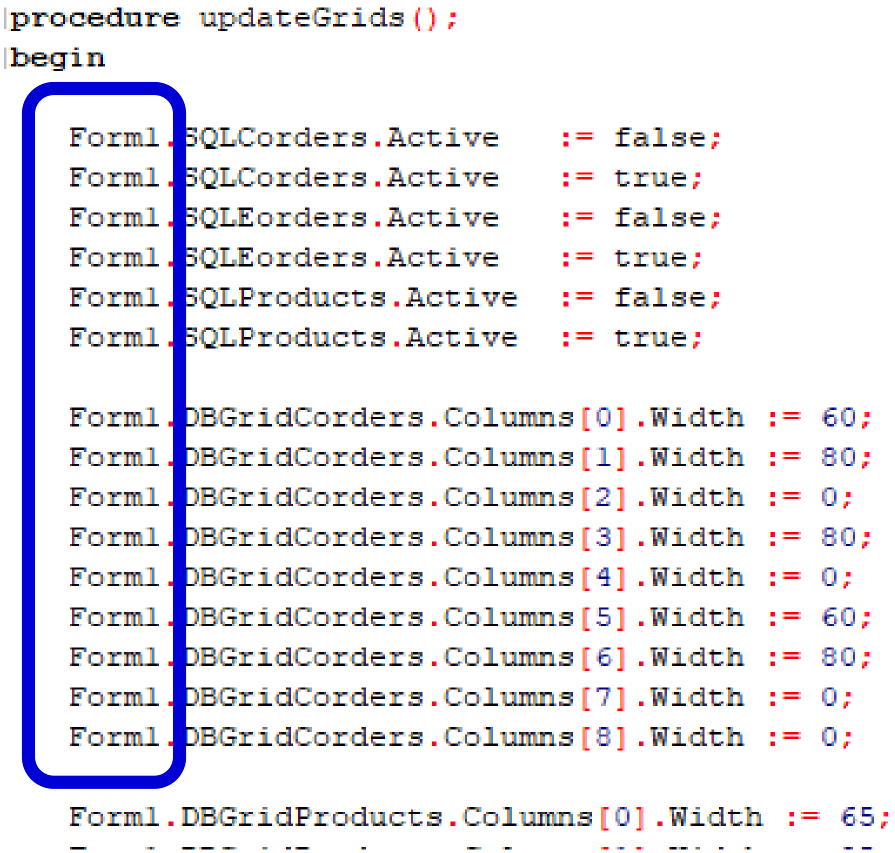
- Remove TForm1. from the name of the procedure as it no longer belongs to Form1.



```
procedure ~~TForm1~~.updateGrids();
begin

    SQLCorders.Active    := false;
    SQLCorders.Active    := true;
    SQLEorders.Active    := false;
    SQLEorders.Active    := true;
    SQLProducts.Active   := false;
    SQLProducts.Active   := true;

    DBGridCorders.Columns[0].Width  := 60;
    DBGridCorders.Columns[1].Width  := 80;
    DBGridCorders.Columns[2].Width  := 0;
    DBGridCorders.Columns[3].Width  := 80;
    DBGridCorders.Columns[4].Width  := 0;
    DBGridCorders.Columns[5].Width  := 60;
    DBGridCorders.Columns[6].Width  := 80;
    DBGridCorders.Columns[7].Width  := 0;
    DBGridCorders.Columns[8].Width  := 0;

    DBGridProducts.Columns[0].Width  := 65;
    DBGridProducts.Columns[1].Width  := 85;
    DBGridProducts.Columns[2].Width  := 115;
    DBGridProducts.Columns[3].Width  := 70;
```

Moving the presentation layer

- Add Form1. to the names of all object of Form1 accessed by the procedure in the PresentationLayer unit:

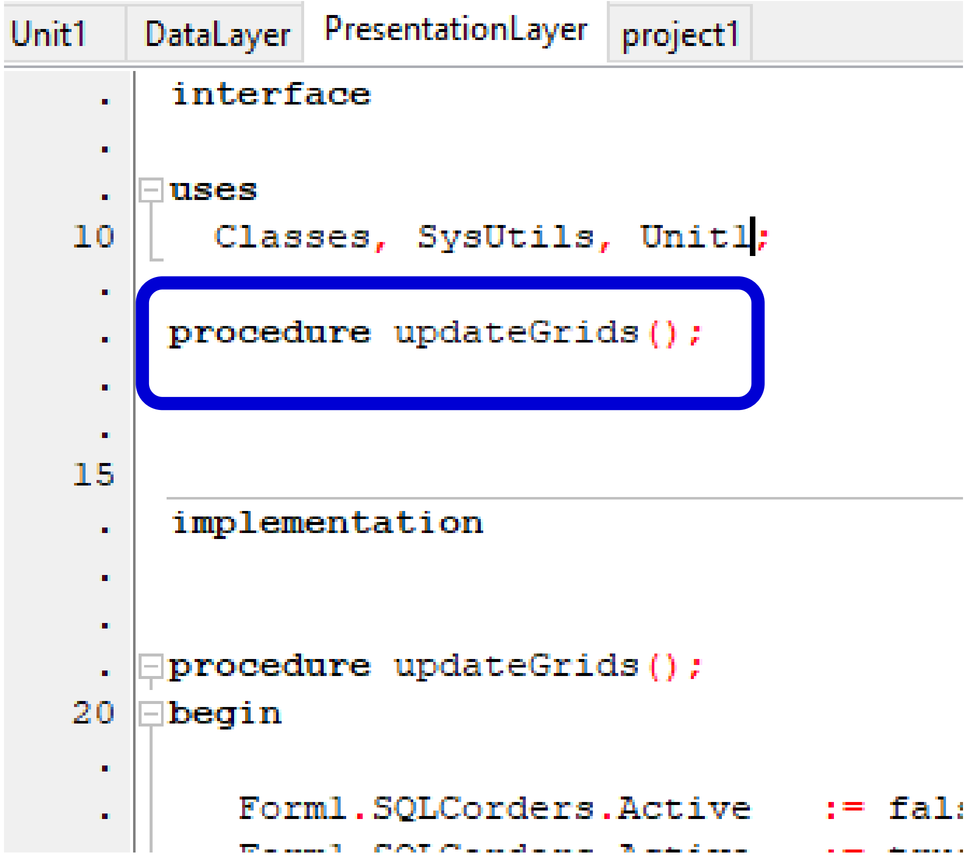


```
procedure updateGrids();
begin

   Form1.SQLCorders.Active   := false;
   Form1.SQLCorders.Active   := true;
   Form1.SQLEorders.Active   := false;
   Form1.SQLEorders.Active   := true;
   Form1.SQLProducts.Active  := false;
   Form1.SQLProducts.Active  := true;

   Form1.DBGridCorders.Columns[0].Width := 60;
   Form1.DBGridCorders.Columns[1].Width := 80;
   Form1.DBGridCorders.Columns[2].Width := 0;
   Form1.DBGridCorders.Columns[3].Width := 80;
   Form1.DBGridCorders.Columns[4].Width := 0;
   Form1.DBGridCorders.Columns[5].Width := 60;
   Form1.DBGridCorders.Columns[6].Width := 80;
   Form1.DBGridCorders.Columns[7].Width := 0;
   Form1.DBGridCorders.Columns[8].Width := 0;

   Form1.DBGridProducts.Columns[0].Width := 65;
```

Moving the presentation layer

- Add the declaration of the procedure in the interface section.

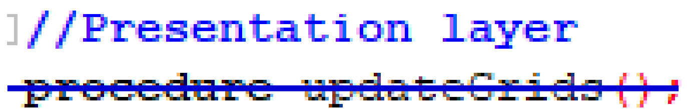
(if you copy/paste it from Form1, don't forget to delete Form1. from the name of the procedure)



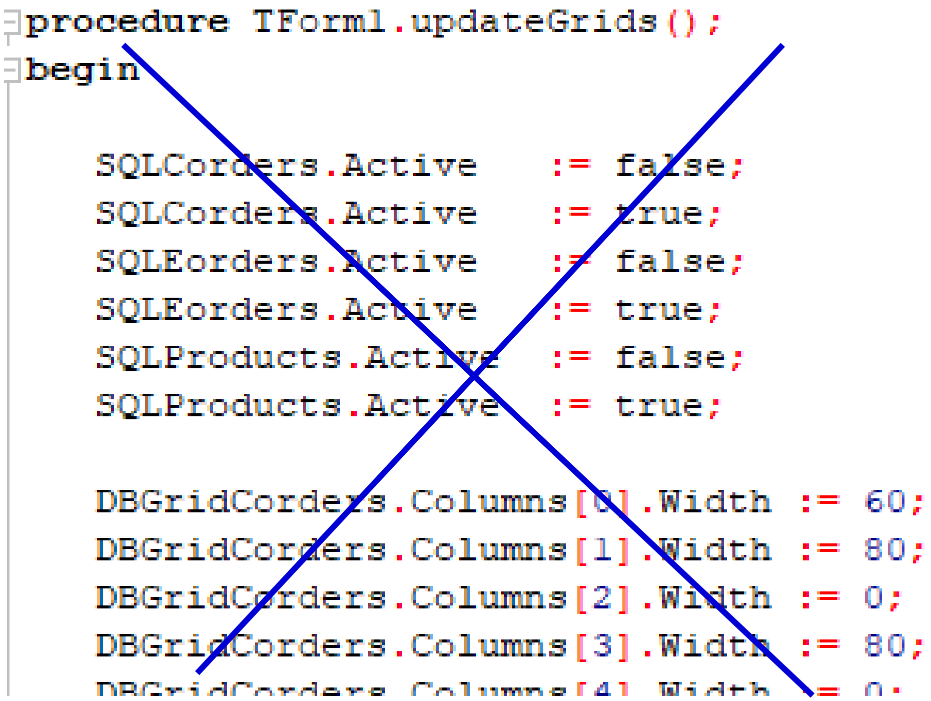
```
interface

uses
  Classes, SysUtils, Unit1;

procedure updateGrids();



implementation



procedure updateGrids();
begin

    Form1.SQLCorders.Active    := fal
```

Moving the presentation layer

- In Unit1, delete the definition and the declaration of the procedure updateGrids() , as this procedure no longer exists in Form1



```
//Presentation layer
~~procedure updateGrids();~~
```



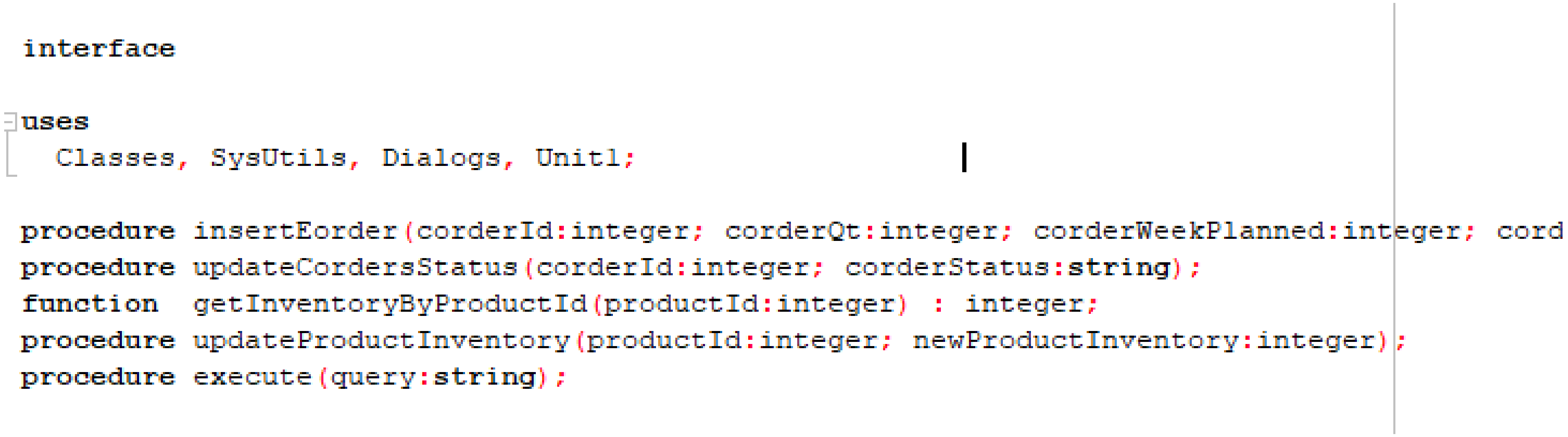
```
procedure TForm1.updateGrids();
begin

   SQLCorders.Active    := false;
   SQLCorders.Active    := true;
   SQLEorders.Active    := false;
   SQLEorders.Active    := true;
   SQLProducts.Active   := false;
   SQLProducts.Active   := true;

   DBGridCorders.Columns[0].Width  := 60;
   DBGridCorders.Columns[1].Width  := 80;
   DBGridCorders.Columns[2].Width  := 0;
   DBGridCorders.Columns[3].Width  := 80;
   DBGridCorders.Columns[4].Width  := 0;
```

Checking the presentation layer

- Run the application and confirm that procedure updateGrids() was successfully moved to the new unit.

Moving the data layer

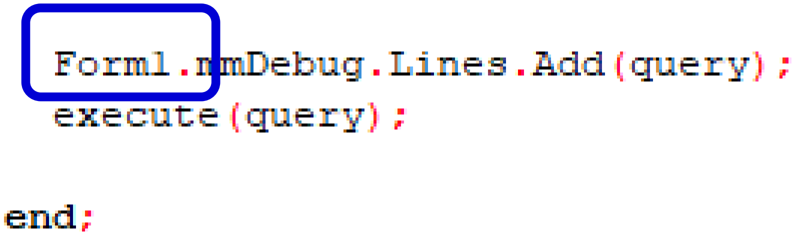
- Now, let’s do a similar operation for the data layer procedures and functions.
- Copy paste the code of the procedures and functions in the data layer section of Unit1 to the new DataLayer unit.
- Add their declarations to the interface section:



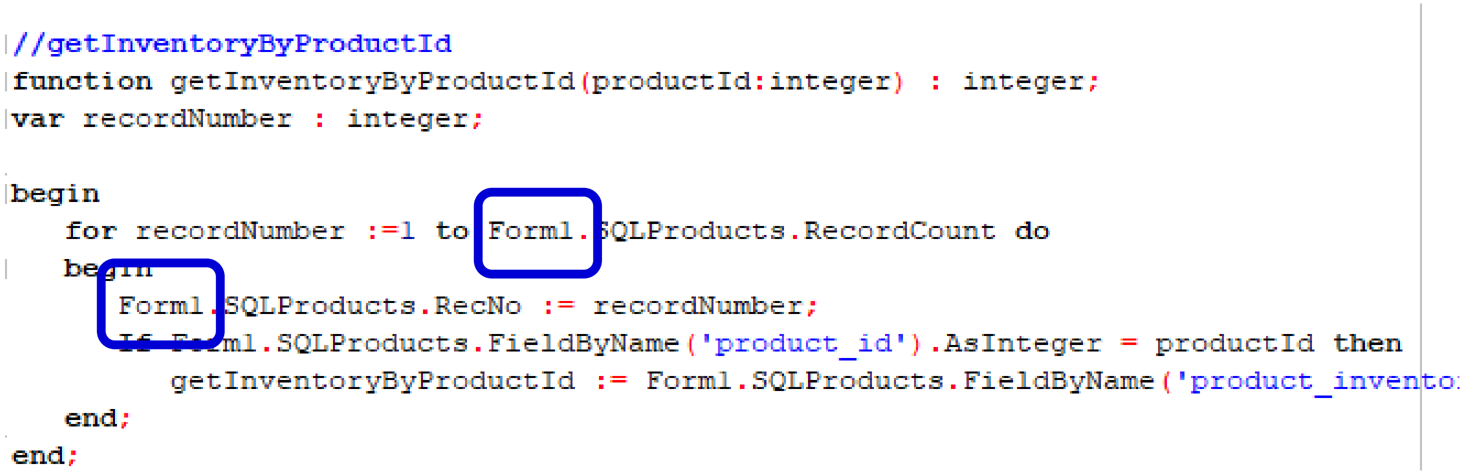
```
interface

uses
  Classes, SysUtils, Dialogs, Unit1;

procedure insertEorder(corderId:integer; corderQt:integer; corderWeekPlanned:integer; cord
procedure updateCordersStatus(corderId:integer; corderStatus:string);
function  getInventoryByProductId(productId:integer) : integer;
procedure updateProductInventory(productId:integer; newProductInventory:integer);
procedure execute(query:string);
```

Moving the data layer

- In the definitions of the functions and procedures, add Form1. to the names of the objects in Form1



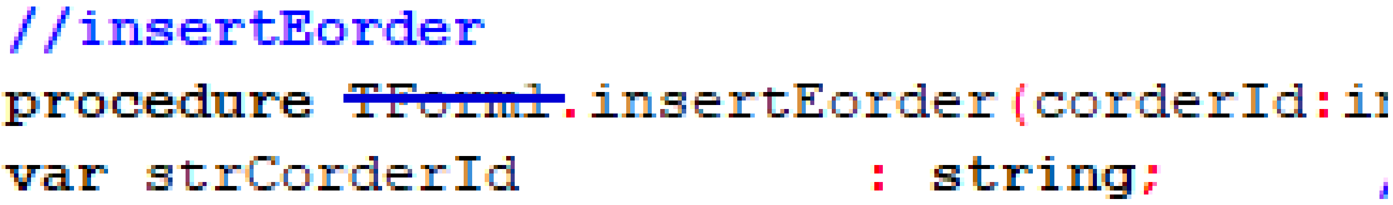
```
    Form1.mmDebug.Lines.Add(query);
    execute(query);

end;
```



```
//getInventoryByProductId
function getInventoryByProductId(productId:integer) : integer;
var recordNumber : integer;

begin
   for recordNumber :=1 to Form1.SQLProducts.RecordCount do
   begin
      Form1.SQLProducts.RecNo := recordNumber;
      If Form1.SQLProducts.FieldByName('product_id').AsInteger = productId then
         getInventoryByProductId := Form1.SQLProducts.FieldByName('product_invento
   end;
end;
```

Moving the data layer

- Also, remove TForm1 from the names of the procedures and functions:



```
//insertEorder
procedure ~~TForm1~~.insertEorder(corderId:i
var strCorderId              : string;      ,
```

- Run the application and confirm that the data layer was moved successfully to the new unit!

- To close …

- a **layered architecture** sets the **foundations** of the software that **make the difference** between

this …

… and

☺ !

thank you !