A DYNAMICSCON PRESENTATION

POWERED BY DUG

# DYNAMICSCON VIRTUAL

MARCH 2023

BUSINESS CENTRAL

DYNAMICSCON.COM

## Design patterns with object-oriented approaches

## Patrick Schiefer

- 12 years of experience as Software developer
- Blogging about BC and Azure DevOps https://patrickschiefer.com

@schiefer_p

## Agenda

- **General**
- **SOLID Principle**
- **Design patterns**
  - Template Method Pattern
  - Command Pattern
  - Command Queue Pattern
  - Dependency Inversion Principle

Object-oriented programming (OOP) is a computer programming model that organizes software design around data, or objects, rather than functions and logic. An object can be defined as a data field that has unique attributes and behavior.

Source: https://www.techtarget.com/searchapparchitecture/definition/object-oriented-programming-OOP

## Classes and Objects

- **Class (codeunit): Definition of the data and available procedures inside an object**

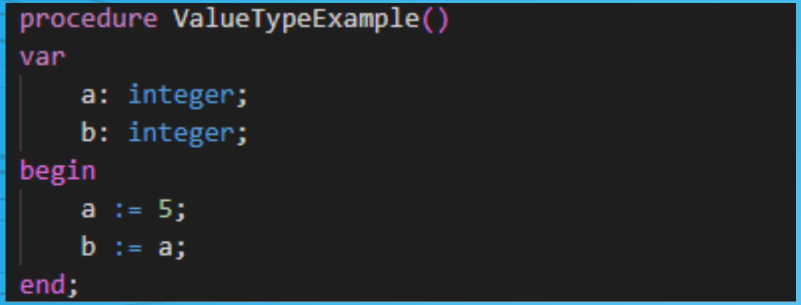
- **Object: instances of classes**
  - Consisting of properties and procedures

## Value and reference types

- **Value type: the value is stored directly in an variable**
  - Numbers
  - Char
  - Byte
  - More or less every single type

- **Reference type: the variable only includes a reference to the instance**

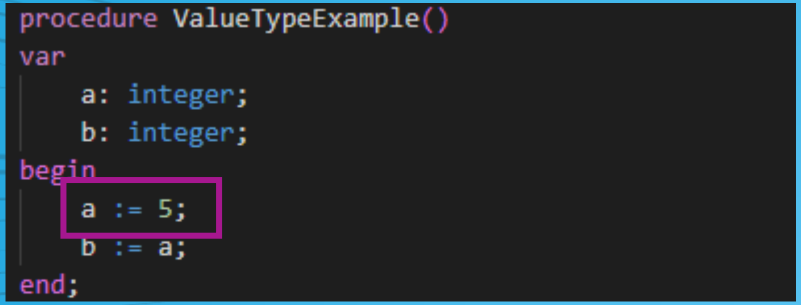
## Behaviour of value types



```
procedure ValueTypeExample()
var
    a: integer;
    b: integer;
begin
    a := 5;
    b := a;
end;
```

| Variablename | Value |
|---|---|
| a | |
| b | |

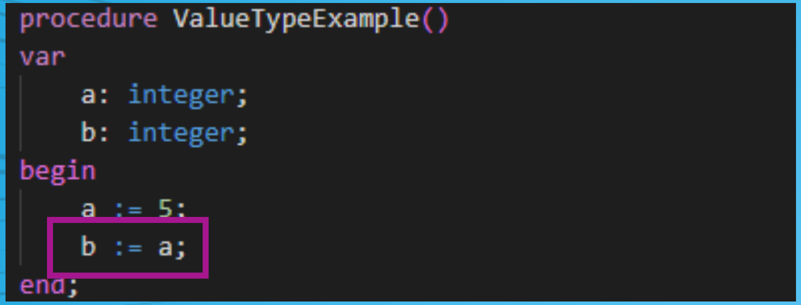
# Behaviour of value types



```
procedure ValueTypeExample()
var
    a: integer;
    b: integer;
begin
    a := 5;
    b := a;
end;
```

| Variablename | Value |
|---|---|
| a | 5 |
| b | |

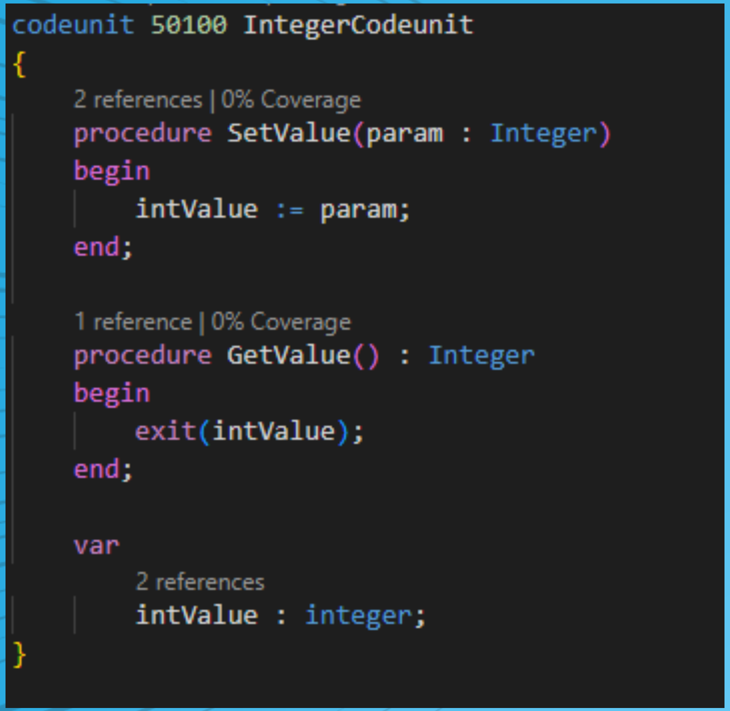
# Behaviour of value types



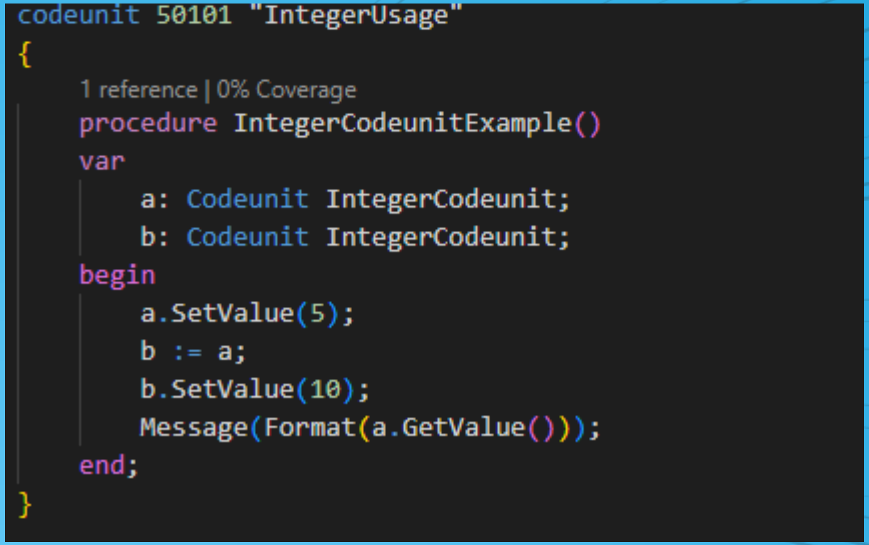
```
procedure ValueTypeExample()
var
    a: integer;
    b: integer;
begin
    a := 5;
    b := a;
end;
```

| Variablename | Value |
|---|---|
| a | 5 |
| b | 5 |

## Behaviour of reference types

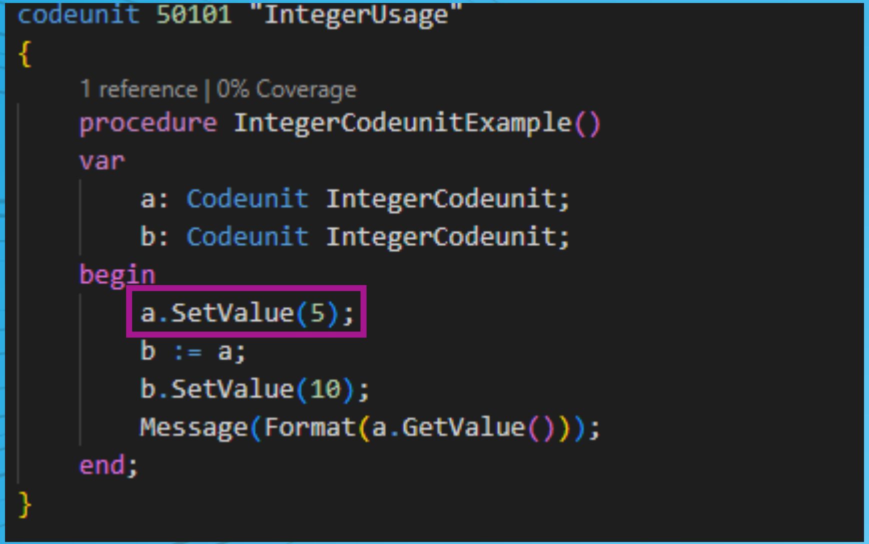


```
codeunit 50100 IntegerCodeunit
{
    2 references | 0% Coverage
    procedure SetValue(param : Integer)
    begin
        intValue := param;
    end;

    1 reference | 0% Coverage
    procedure GetValue() : Integer
    begin
        exit(intValue);
    end;

    var
        2 references
        intValue : integer;
}
```



```
codeunit 50101 "IntegerUsage"
{
    1 reference | 0% Coverage
    procedure IntegerCodeunitExample()
    var
        a: Codeunit IntegerCodeunit;
        b: Codeunit IntegerCodeunit;
    begin
        a.SetValue(5);
        b := a;
        b.SetValue(10);
        Message(Format(a.GetValue()));
    end;
}
```

## Behaviour of reference types


```
codeunit 50101 "IntegerUsage"
{
    1 reference | 0% Coverage
    procedure IntegerCodeunitExample()
    var
        a: Codeunit IntegerCodeunit;
        b: Codeunit IntegerCodeunit;
    begin
        a.SetValue(5);
        b := a;
        b.SetValue(10);
        Message(Format(a.GetValue()));
    end;
}
```

| Variablename | Value |
|---|---|
| a | Instance of a |
| b | Instance of b |
| … | |
| Instance of a | |
| intValue | 5 |
| … | |
| Instance of b | |
| intValue | |

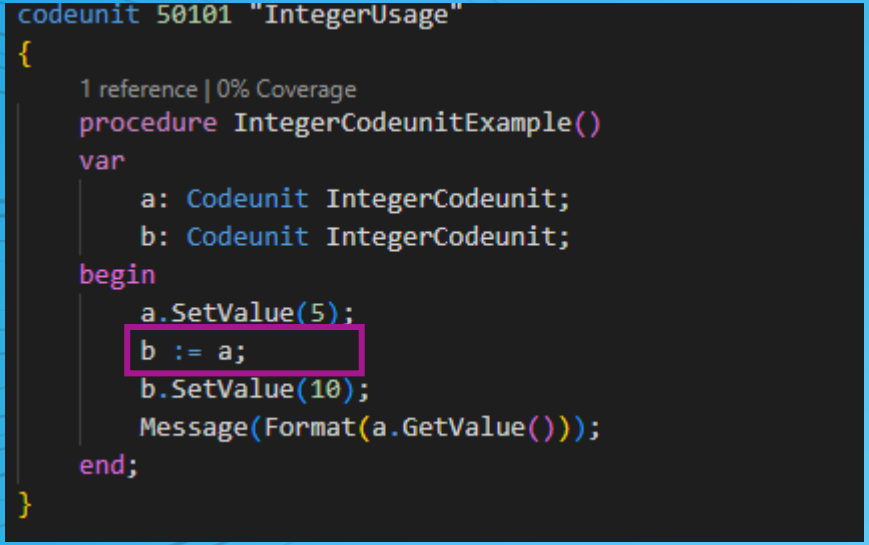
Microsoft

INNOVIA CONSULTING

NETSTOCK

POWERED BY DUG DYNAMICSCON VIRTUAL

# Behaviour of reference types

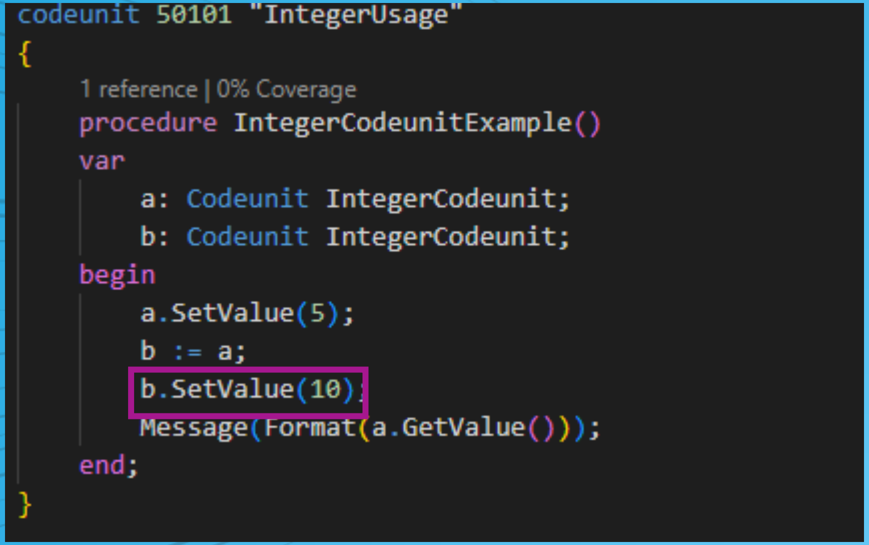


```
codeunit 50101 "IntegerUsage"
{
    1 reference | 0% Coverage
    procedure IntegerCodeunitExample()
    var
        a: Codeunit IntegerCodeunit;
        b: Codeunit IntegerCodeunit;
    begin
        a.SetValue(5);
        b := a;
        b.SetValue(10);
        Message(Format(a.GetValue()));
    end;
}
```

| Variablename | Value |
|---|---|
| a | Instance of a |
| b | Instance of a |
| … | |
| Instance of a | |
| intValue | 5 |
| … | |
| Instance of b | |
| intValue | |

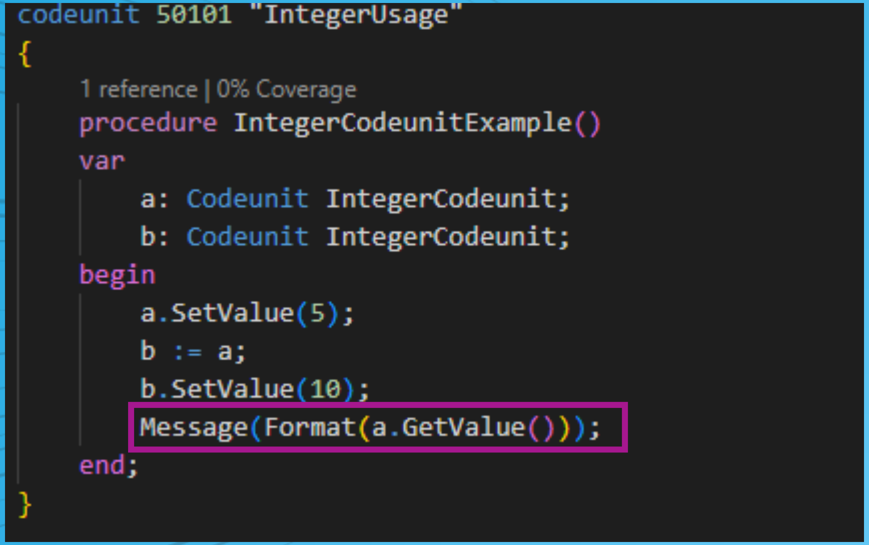
Microsoft   Innovia Consulting   NETSTOCK   POWERED BY DUG DYNAMICSCON VIRTUAL

## Behaviour of reference types

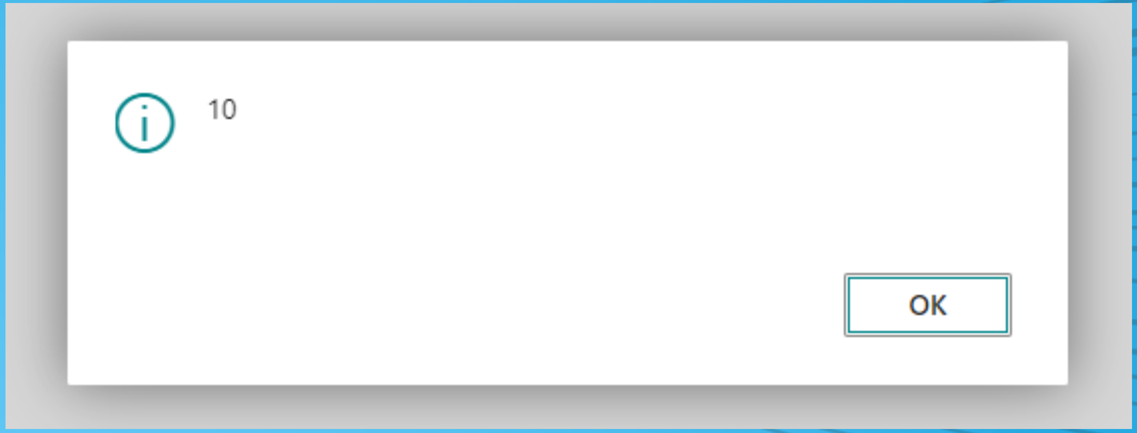

```
codeunit 50101 "IntegerUsage"
{
    1 reference | 0% Coverage
    procedure IntegerCodeunitExample()
    var
        a: Codeunit IntegerCodeunit;
        b: Codeunit IntegerCodeunit;
    begin
        a.SetValue(5);
        b := a;
        b.SetValue(10);
        Message(Format(a.GetValue()));
    end;
}
```

| Variablename | Value |
|---|---|
| a | Instance of a |
| b | Instance of a |
| … | |
| Instance of a | |
| intValue | 10 |

## Behaviour of reference types



```
codeunit 50101 "IntegerUsage"
{
    1 reference | 0% Coverage
    procedure IntegerCodeunitExample()
    var
        a: Codeunit IntegerCodeunit;
        b: Codeunit IntegerCodeunit;
    begin
        a.SetValue(5);
        b := a;
        b.SetValue(10);
        Message(Format(a.GetValue()));
    end;
}
```



```
10
```

Microsoft

innovia CONSULTING

NETSTOCK

POWERED BY DUG DYNAMICSCON VIRTUAL

## 2 Facts about AL

- It is not object oriented

- Uses .net as underlaying technology

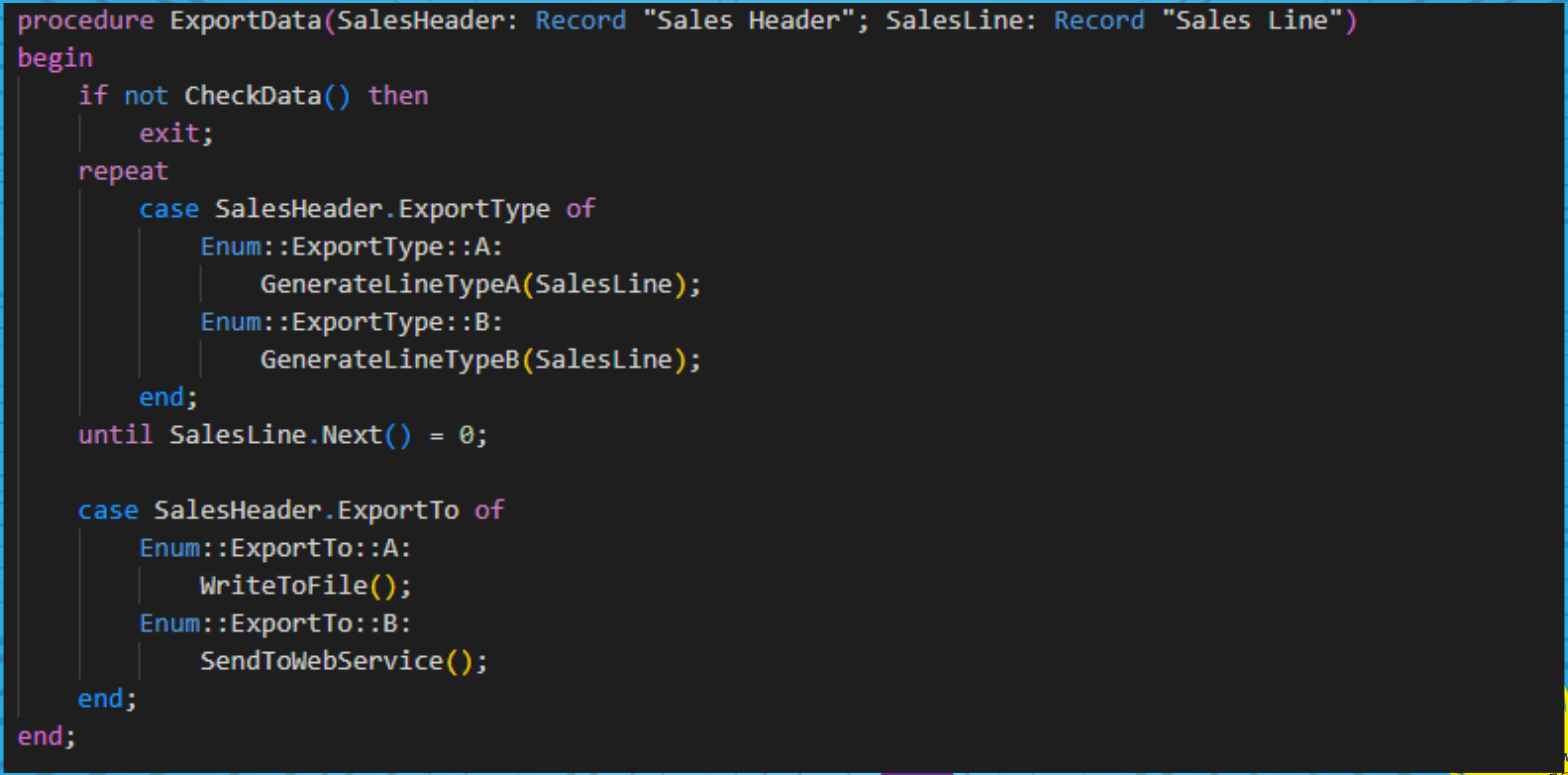
## SOLID Principle

| | |
|---|---|
| S | ingle Responsibility Principle |
| O | pen Closed Principle |
| L | iskov Substitution Principle |
| I | nterface Segregation Principle |
| D | ependency Inversion Principle |

## Template method pattern

- Used to solve similar problems in a similar way
- Uses a template method without actual implementation of the problem
- The implementation is separated in a second codeunit
- An interface defines how the implementation looks
- Could also be used to improve testability of your app

## Bad code example



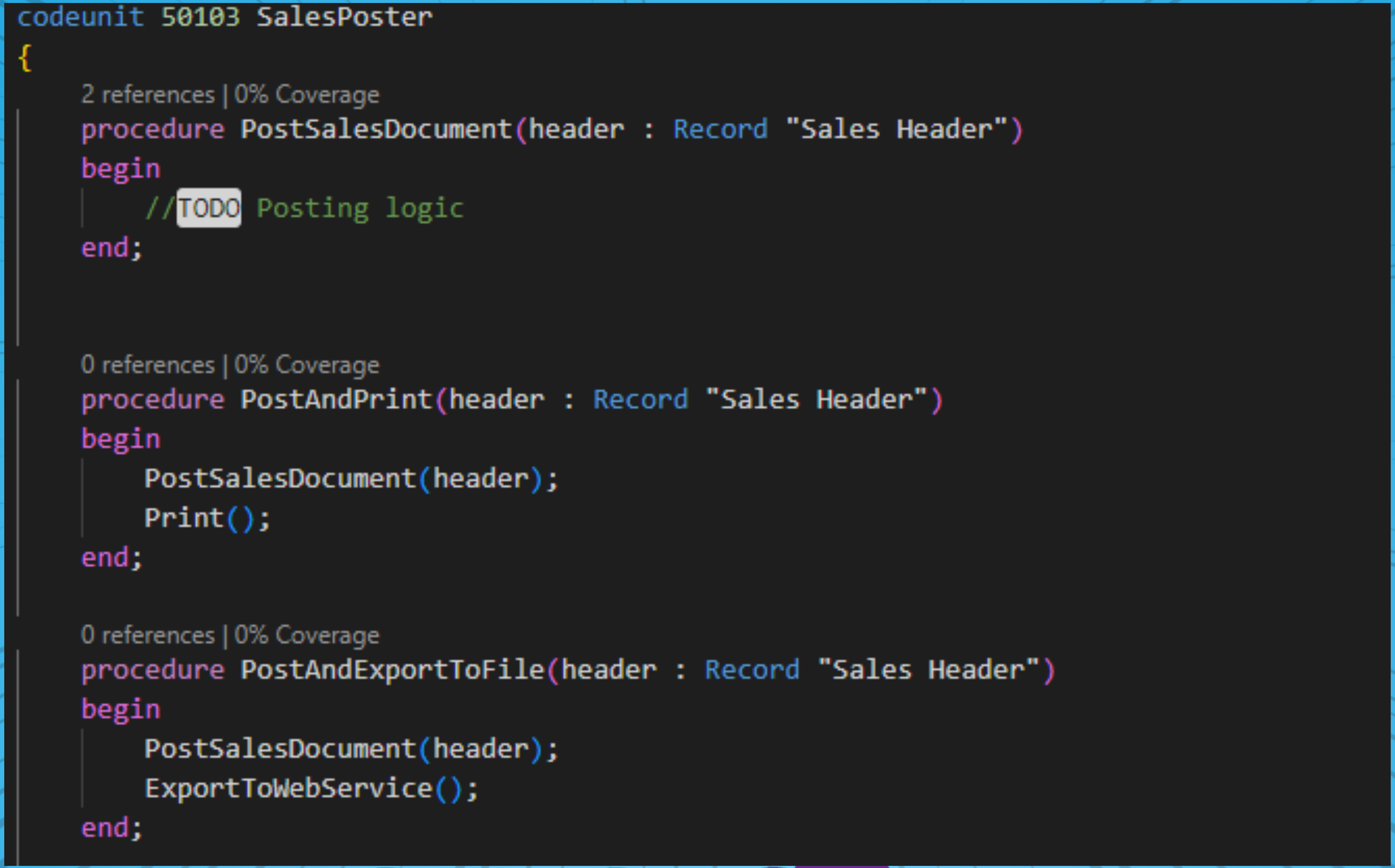
```
procedure ExportData(SalesHeader: Record "Sales Header"; SalesLine: Record "Sales Line")
begin
    if not CheckData() then
        exit;
    repeat
        case SalesHeader.ExportType of
            Enum::ExportType::A:
                GenerateLineTypeA(SalesLine);
            Enum::ExportType::B:
                GenerateLineTypeB(SalesLine);
        end;
    until SalesLine.Next() = 0;

    case SalesHeader.ExportTo of
        Enum::ExportTo::A:
            WriteToFile();
        Enum::ExportTo::B:
            SendToWebService();
    end;
end;
```

## Hands on

## Command Pattern

- Behavioral design pattern
- Generate components which are calling code without knowing which code
- Write flexible code
- Contains at least 3 Elements
  - Interface "ICommand" which defines how a command looks
  - A command codeunit implementing the interface command
  - A commander codeunit invokes the command

## Bad code example



```
codeunit 50103 SalesPoster
{
    2 references | 0% Coverage
    procedure PostSalesDocument(header : Record "Sales Header")
    begin
        //TODO Posting logic
    end;


    0 references | 0% Coverage
    procedure PostAndPrint(header : Record "Sales Header")
    begin
        PostSalesDocument(header);
        Print();
    end;

    0 references | 0% Coverage
    procedure PostAndExportToFile(header : Record "Sales Header")
    begin
        PostSalesDocument(header);
        ExportToWebService();
    end;
```

## Hands on

## Command Queue Patterns

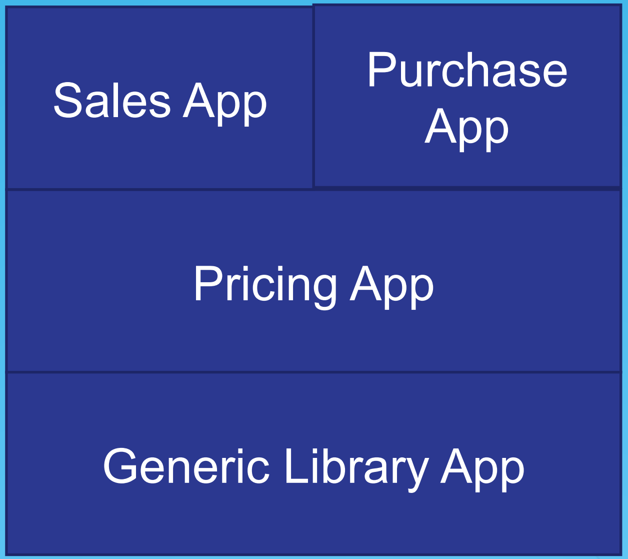
- Extension to the command pattern
- Commands are stored in a queue and executed in sequence
- Used to control the flow of a process
- Combines single tasks to a process

## Hands on

## Dependency Inversion Principle

- Reducing dependencies between objects and apps
- Using Interfaces to connect objects and apps

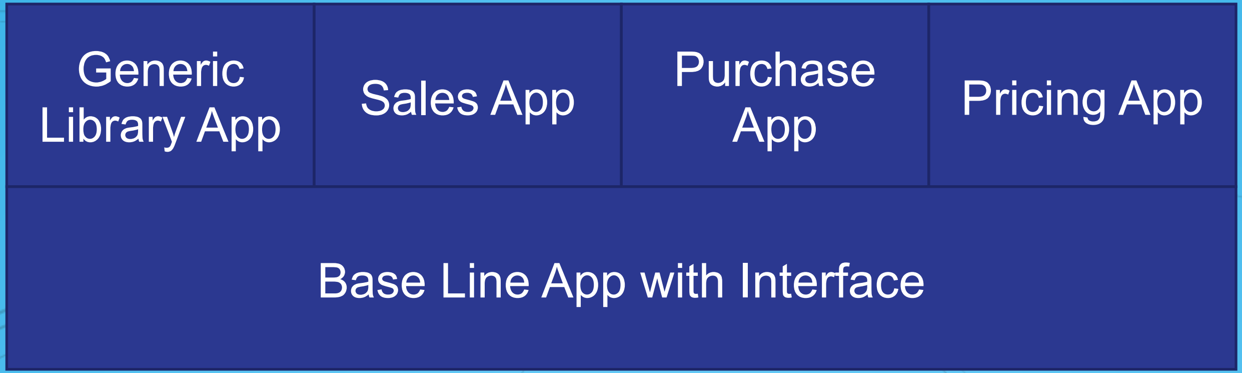
## Dependency Inversion Principle



## Dependency Inversion Principle

- Implizit dependency from Printing App to pricing app
- When Pricing app is updated, sales, purchase and printing app need to be reinstalled

# Dependency Inversion Principle



## Dependency Inversion Principle

- Base Line Includes no code just interfaces
- Update should be very rare
- New Apps could easily be added with out adding complexity to the dependency tree

## Hands on

## Addional Informations

- https://alguidelines.dev
- https://clean-code-developer.com/
- https://springframework.guru/gang-of-four-design-patterns/

# Q & A