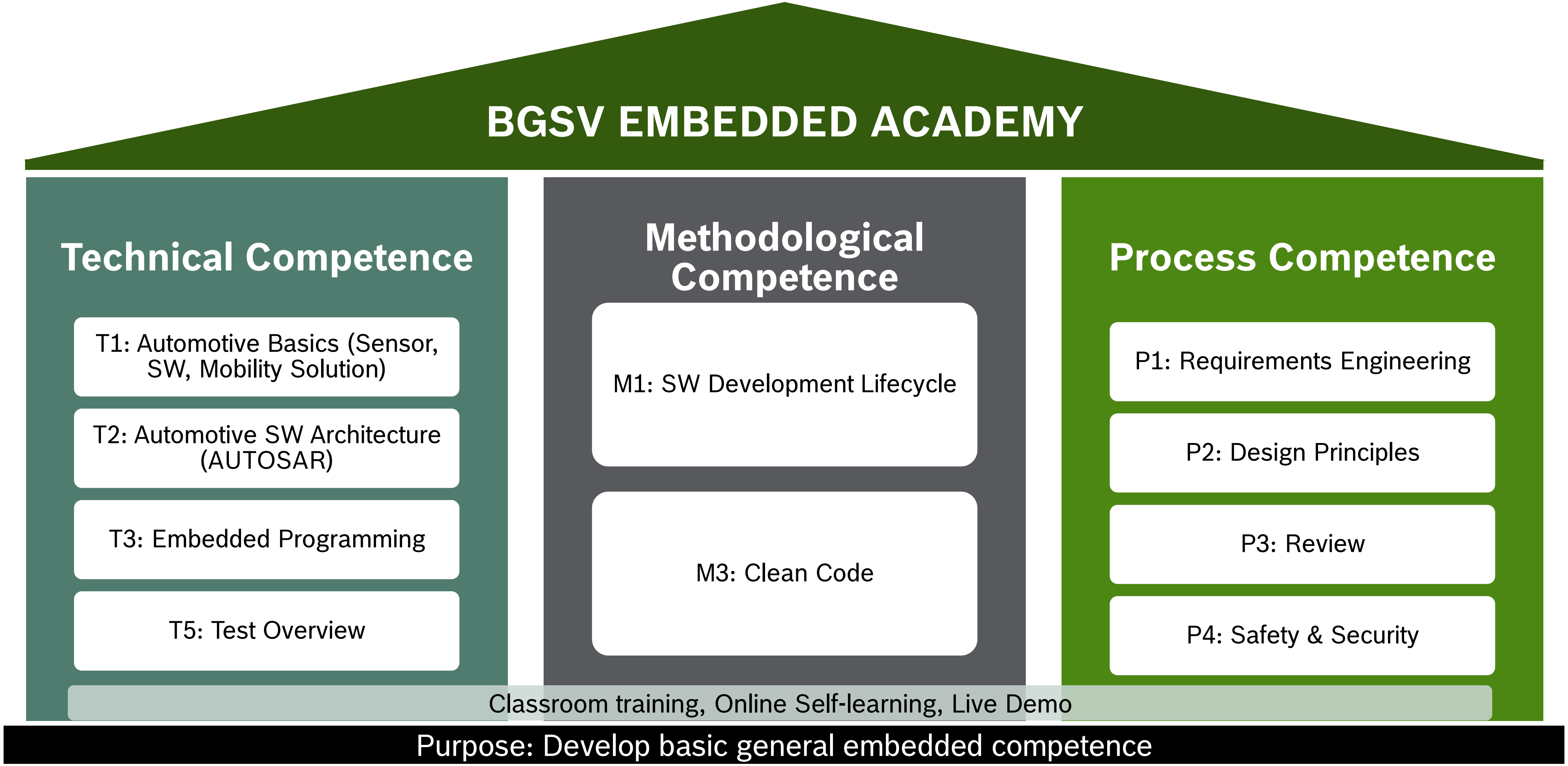
EMBEDDED ACADEMY

★ PEDAL TO THE MEDAL ★

BOSCH

BGSV Embedded Academy (BEA)

Focused Program to Develop Embedded Competence



BGSV EMBEDDED ACADEMY

Technical Competence

- T1: Automotive Basics (Sensor, SW, Mobility Solution)
- T2: Automotive SW Architecture (AUTOSAR)
- T3: Embedded Programming
- T5: Test Overview

Methodological Competence

- M1: SW Development Lifecycle
- M3: Clean Code

Process Competence

- P1: Requirements Engineering
- P2: Design Principles
- P3: Review
- P4: Safety & Security

Classroom training, Online Self-learning, Live Demo

Purpose: Develop basic general embedded competence

Disclaimer

- This slide is a part of BGSV Embedded Academy (BEA) program and only used for BEA training purposes.
- This slide is Bosch Global Software Technology Company Limited’s internal property. All rights reserved, also regarding any disposal, exploitation, reproduction, editing, distribution as well as in the event of applications for industrial property rights.
- This slide has some copyright images and text, which belong to the respective organizations.

P2
Design Principles

Agenda

- Introduction: What and Why do we need Design Principles?
- Selected list of design principles
- High Level Design and Low Level Design
- Static Design and Dynamic Design
- Tools
- Observance

INTRODUCTION

Design Principles

What are Software Design Principles?

- Software Design Principles are not design patterns
- Software Design Principles are not hard rules that are fulfilled or not fulfilled
- Software Design Principles are not unambiguous

Design Principles
What are Software Design Principles?

- Software Design Principles are not design patterns
- As any principle Software Design Principles are kind of universal
  i.e. unlike patterns they do not provide a solution for a concrete task or problem
- Software Design Principles are not hard rules that are fulfilled or not fulfilled
- SW design principles can be regarded recommendations for good software engineering
  They are not hard rules that are fulfilled or not fulfilled (unlike e.g. MISRA)
- Design principles may be contradictory for specific problems
  (e.g. "separation of concerns" vs. "keep it simple and stupid")
- Software Design Principles are not unambiguous
- Universal recommendations may be contradictory.
- The selected set of design principles form a mindset for architectural and detailed design work

Design Principles

What are Software Design Principles?

- As any principle Software Design Principles are kind of **universal**
  i.e. unlike patterns they do not provide a solution for a concrete task or problem

- SW design principles can be regarded **recommendations** for good software engineering
  They are not hard rules that are fulfilled or not fulfilled (unlike e.g. MISRA)

- Design principles **may be contradictory** for specific problems
  (e.g. "separation of concerns" vs. "keep it simple and stupid")

- The selected set of design principles form a **mindset** for architectural and detailed design work

Design Principles
What are Software Design Principles good for?

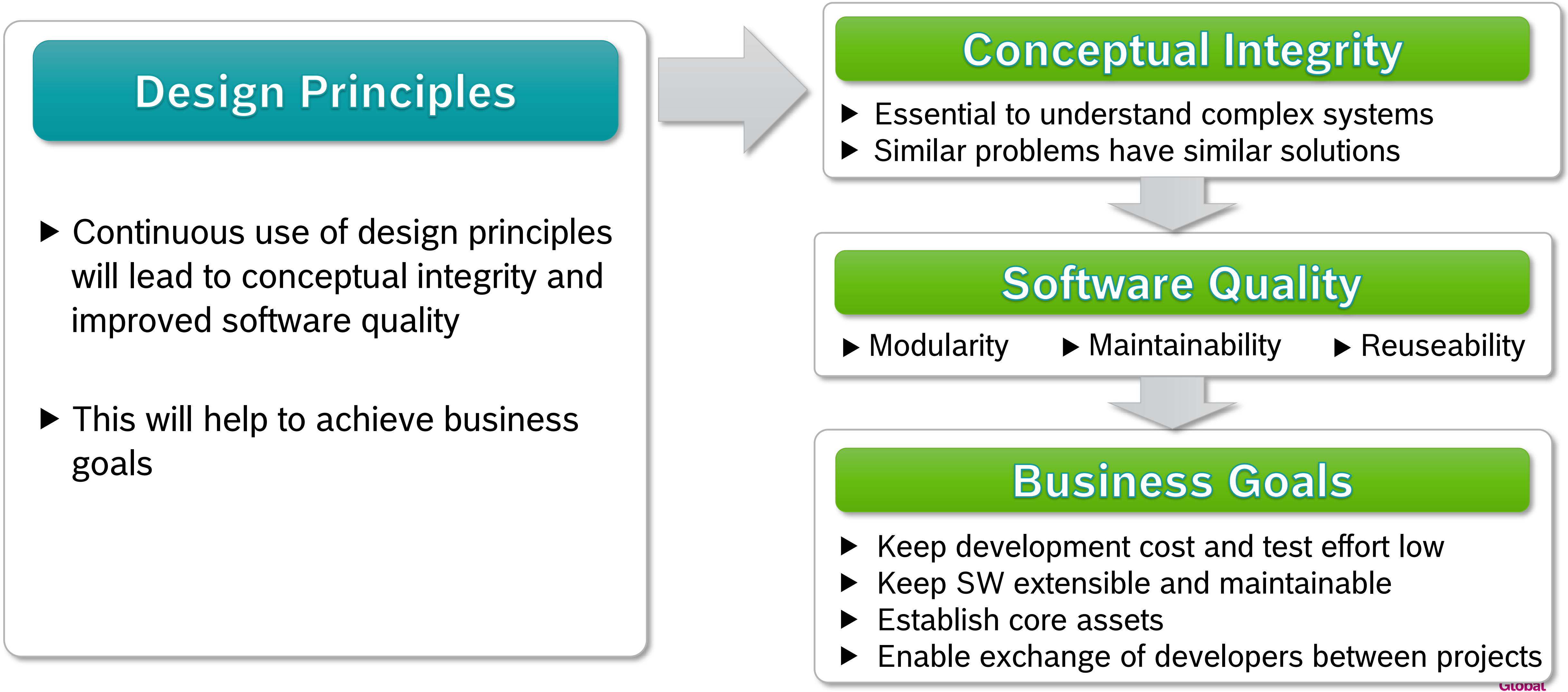
- Design principles lead to Conceptual integrity
  - This is essential to understand complex systems

- Design principles support Software Quality characteristics like
  - Modularity
  - Maintainability
  - Reuseability

- By that, design principles help achieving Business Goals as they help to
  - Keep development cost low
  - Keep test effort low
  - Keep SW extensible and maintainable over a long time
  - Establish core assets
  - Enable easy switch of developers between projects

Design Principles

What are Software Design Principles good for?



Design Principles

- Continuous use of design principles will lead to conceptual integrity and improved software quality
- This will help to achieve business goals

Conceptual Integrity

- Essential to understand complex systems
- Similar problems have similar solutions

Software Quality

- Modularity
- Maintainability
- Reuseability

Business Goals

- Keep development cost and test effort low
- Keep SW extensible and maintainable
- Establish core assets
- Enable exchange of developers between projects

SELECTED LIST OF DESIGN PRINCIPLES

Selected List of Design Principles

The list below shows a selection of commonly known design principles

- Separation of Concerns *–> next slides*
- Don´t Repeat Yourself (DRY) *–> next slides*
- Keep it simple and stupid (KISS) *– Also: Keep it simple, stupid! Don't do fancy things if a simple solution does the job*
- You ain't gonna need it (YAGNI) *– Do not prepare for future changes that might never come*
- "SOLID" is a set of principles commonly applied in (but not restricted to) OOP development
  - **S**ingle responsibility *– Closely related to separation of concerns*
  - **O**pen/closed principle *–> next slides*
  - **L**iskov substitution principle *–> Objects of a superclass shall be replaceable with objects of its subclasses without breaking the application. That requires the objects of your subclasses to behave in the same way as the objects of your superclass*
  - **I**nterface Segregation *–> Keep interfaces as small as possible; provide specific interfaces for different counterparts*
  - **D**ependency inversion *–> next slides*

Selected List of Design Principles
Separation of Concerns (SoC)

What it means

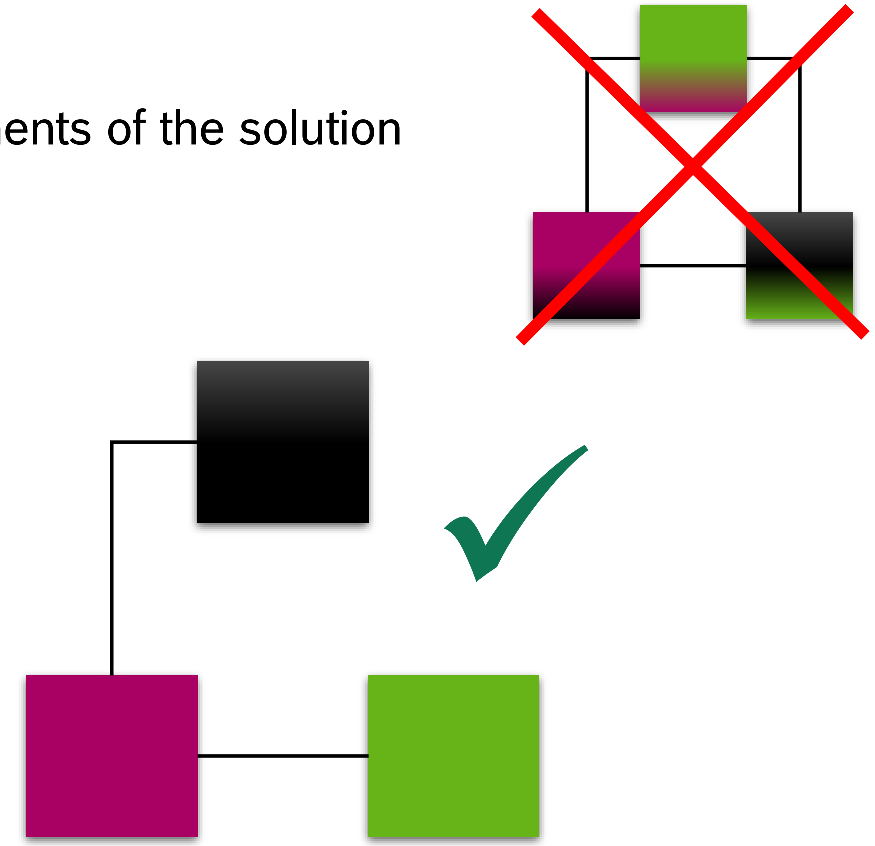
- Different elements of the task shall be represented in different elements of the solution
- Each SW component has a single dedicated responsibility

Why it’s useful

- No hidden functionality
- Maintenance
  - Parts that must be modified because of a change are only concerned with exactly the changed functionality
- Less test effort

Example

- Layer concept of SW reference architecture

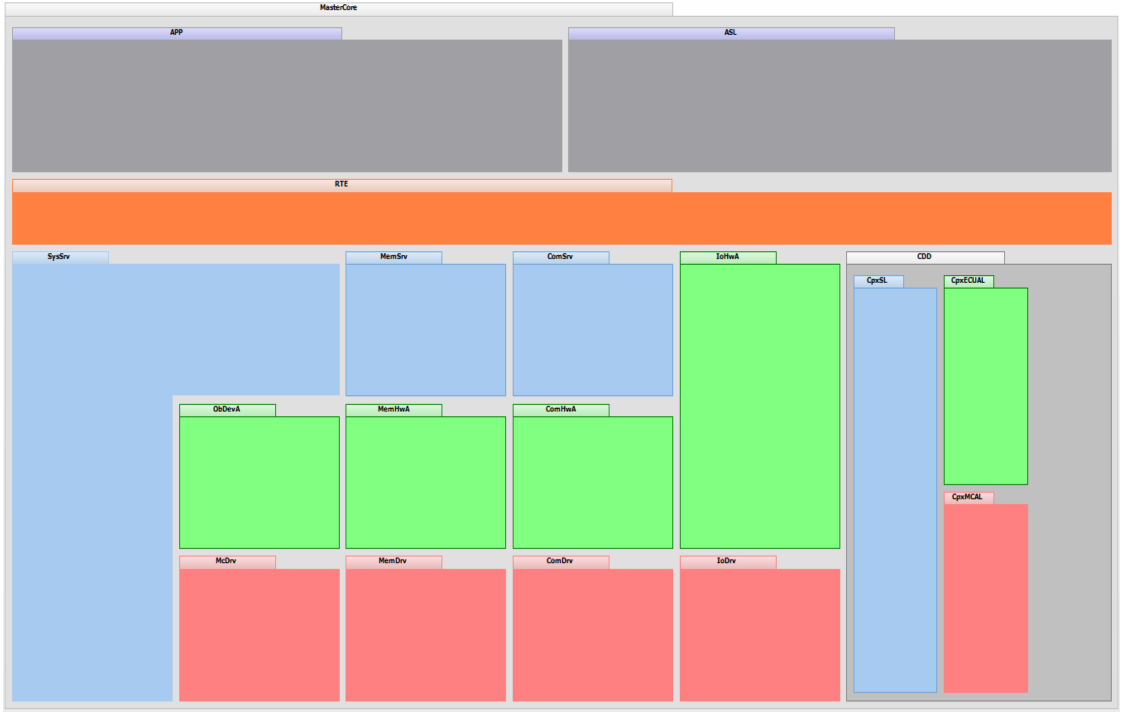


Selected List of Design Principles

Separation of Concerns

Example: Layered SW-Architecture (Strict Layering)

- Characteristics:
  - Each SW component is dedicated to a specific layer
  - Layer crossing SW components are not allowed
  - Clear responsibilities of each layer
  - Clear call restrictions between layers
- E.g. separation of actuator control and application
  - actuator can be exchanged without changing the application
    (provided interface and black box behavior are the same
    -> Dependency inversion, Liskov, Interface segregation)

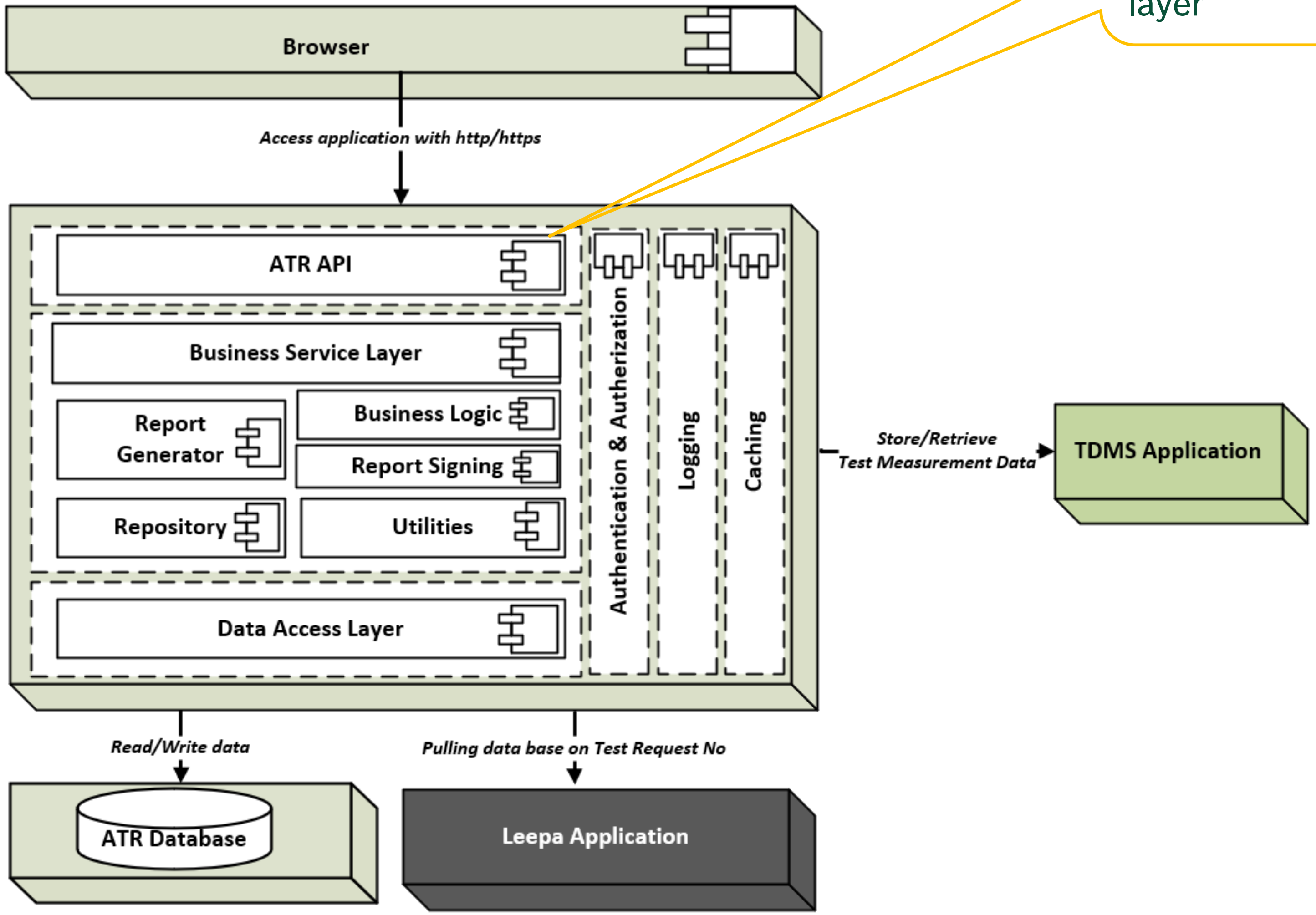


Selected List of Design Principles

Separation of Concerns (SoC)

Example: Multiple layers SW-Architecture implement the Single responsibility

Each layer owns a separated responsibility.
Perform the task belong to that layer



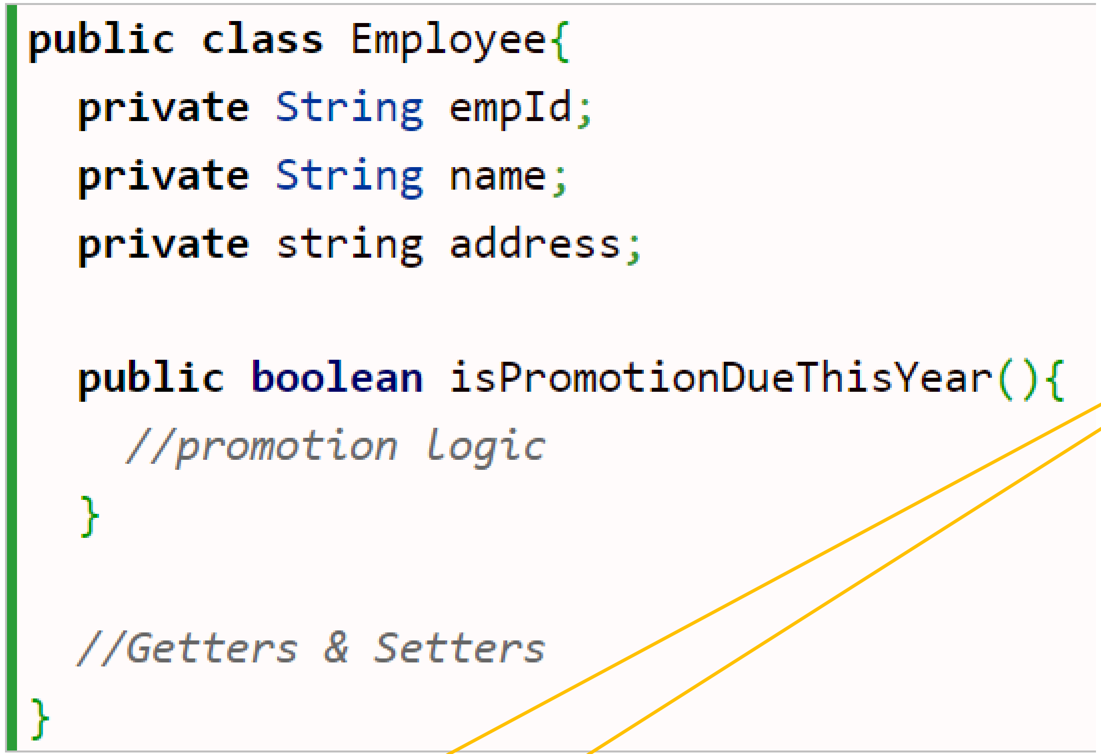
ATR Application building blocks

Selected List of Design Principles

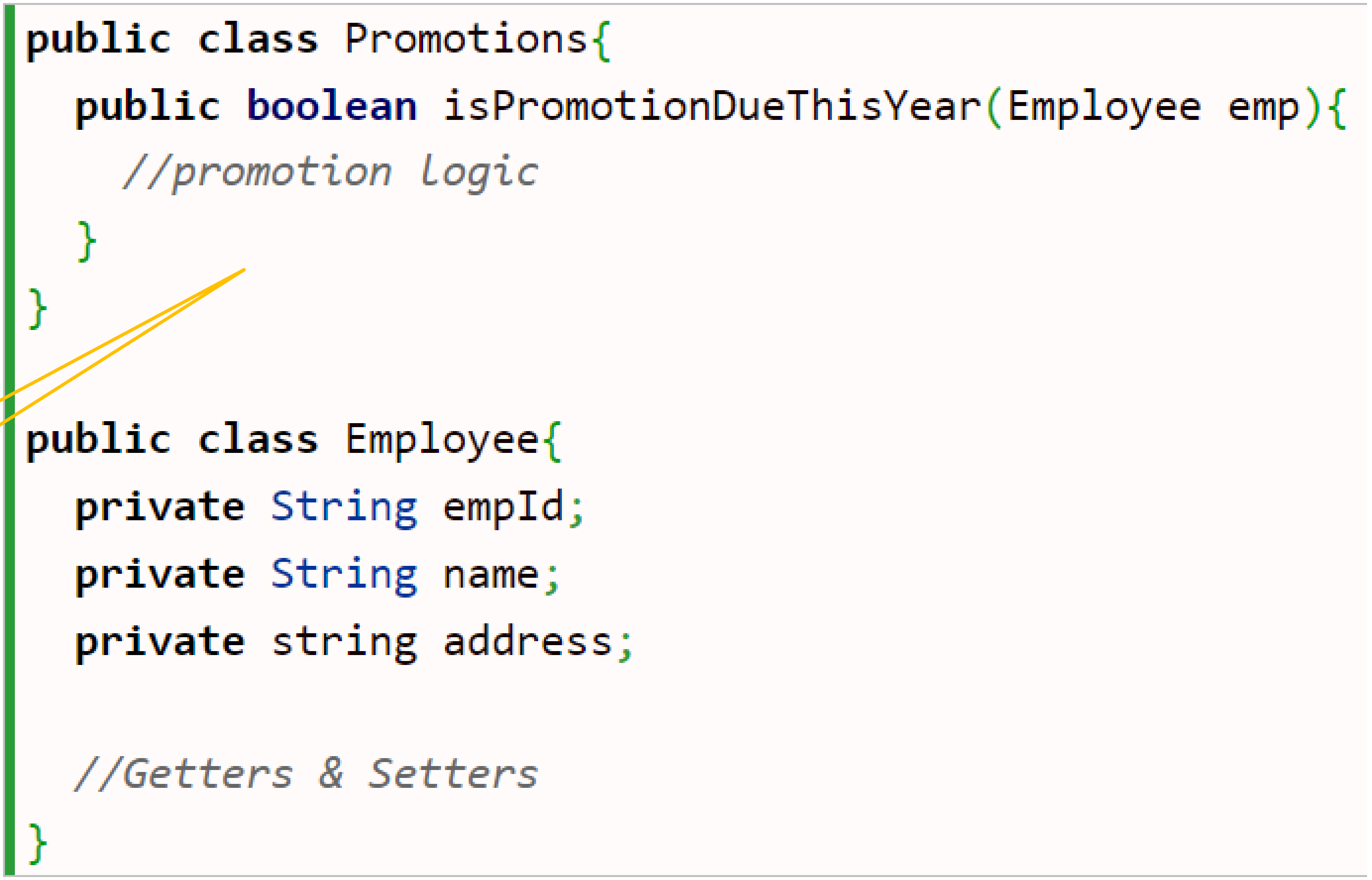
Separation of Concerns (SoC)

The employee class violates the single responsibility principle

Example: Class implement that shows how Single responsibility can be done



```
public class Employee{
  private String empId;
  private String name;
  private string address;

  public boolean isPromotionDueThisYear(){
    //promotion logic
  }

  //Getters & Setters
}
```



```
public class Promotions{
  public boolean isPromotionDueThisYear(Employee emp){
    //promotion logic
  }
}

public class Employee{
  private String empId;
  private String name;
  private string address;

  //Getters & Setters
}
```

Separate Employee and Promotion into separated classes make it easier to modify. Each class would have one single responsibility

Bosch
Global
Software
Technologies
alt_future

Selected List of Design Principles

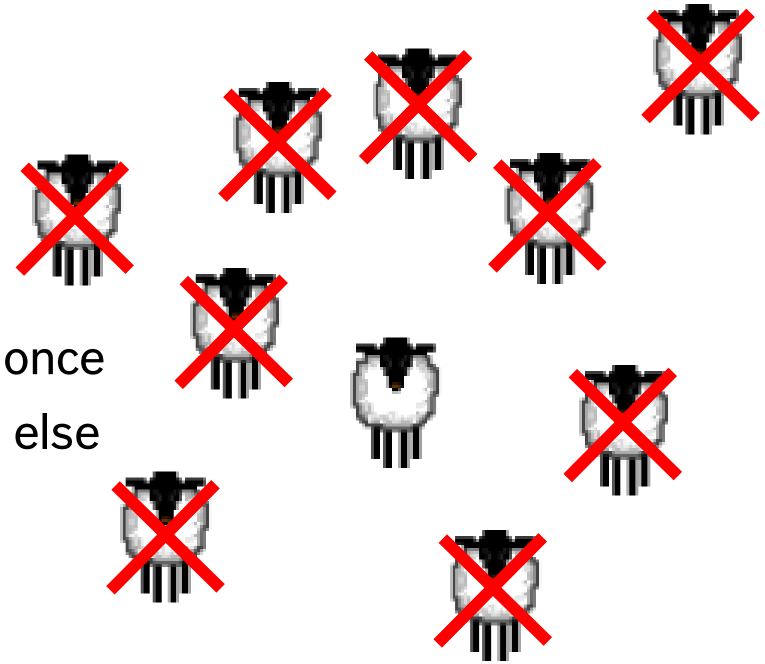
Don´t Repeat Yourself (DRY)

What it means

- Don't implement the same thing several times
- Don't clone and own

Why it’s useful

- Reduced development effort: One problem is only solved once
- Reduced maintenance effort: Changes and fixes have to be applied only once
- Less error prone: Changes in one place can not be forgotten somewhere else



Selected List of Design Principles

Open-Closed Principle

What it means

- Open for extensions: Additional functionality can be added
- Closed for changes: Implemented core functionality is not changed
- Instead of MODIFYING, we should go for EXTENSION.
- Reduced risk of introducing bugs or unwanted side effects

Why it’s useful

- Reduced test effort

Examples

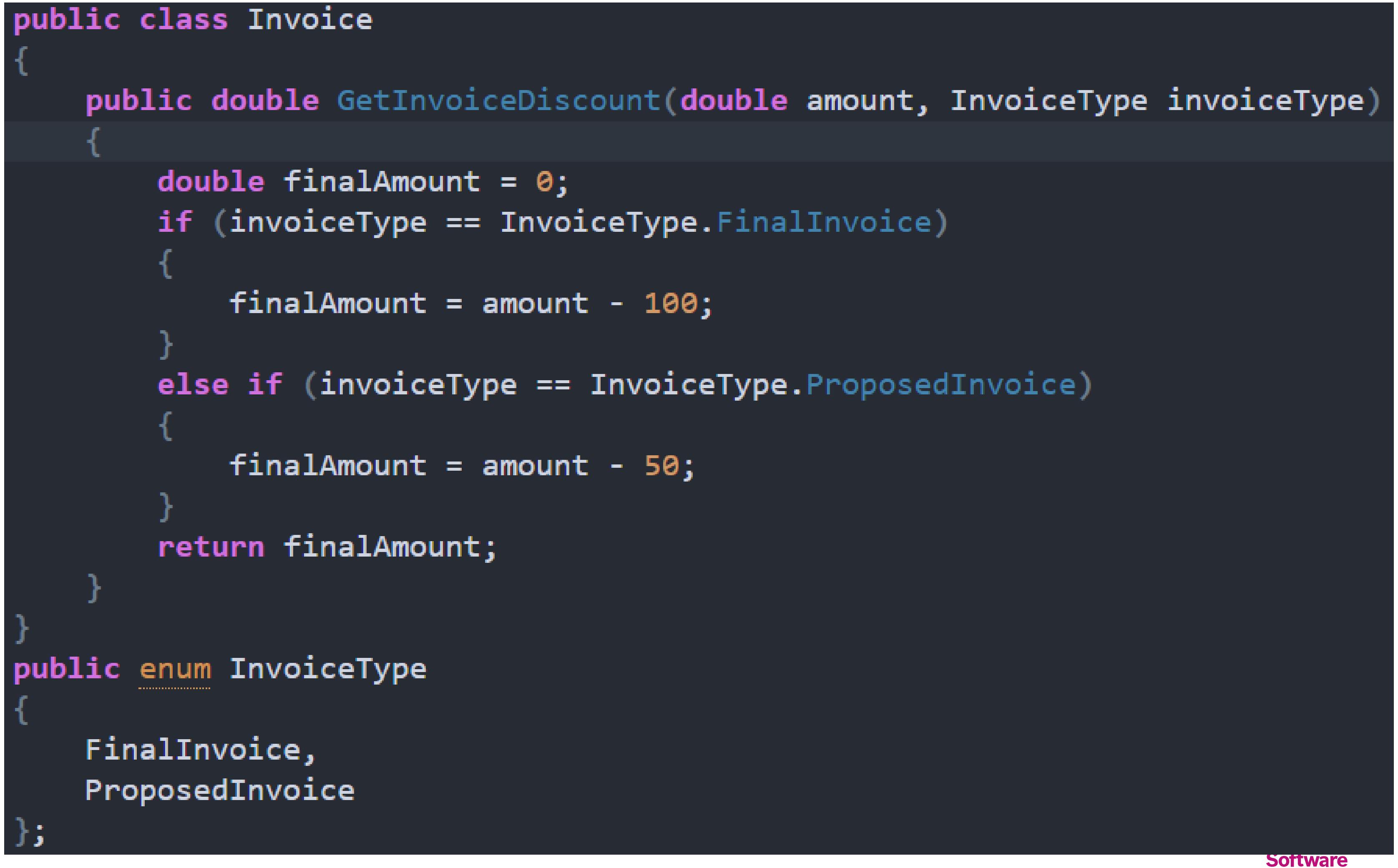
- Separate attributes type of class into class objects.

Selected List of Design Principles

Open-Closed Principle

Example: Open-Closed Principle

- New type of invoice need to modify the GetInvoiceDiscount
- Risks of new bug for other functionality using the same class.



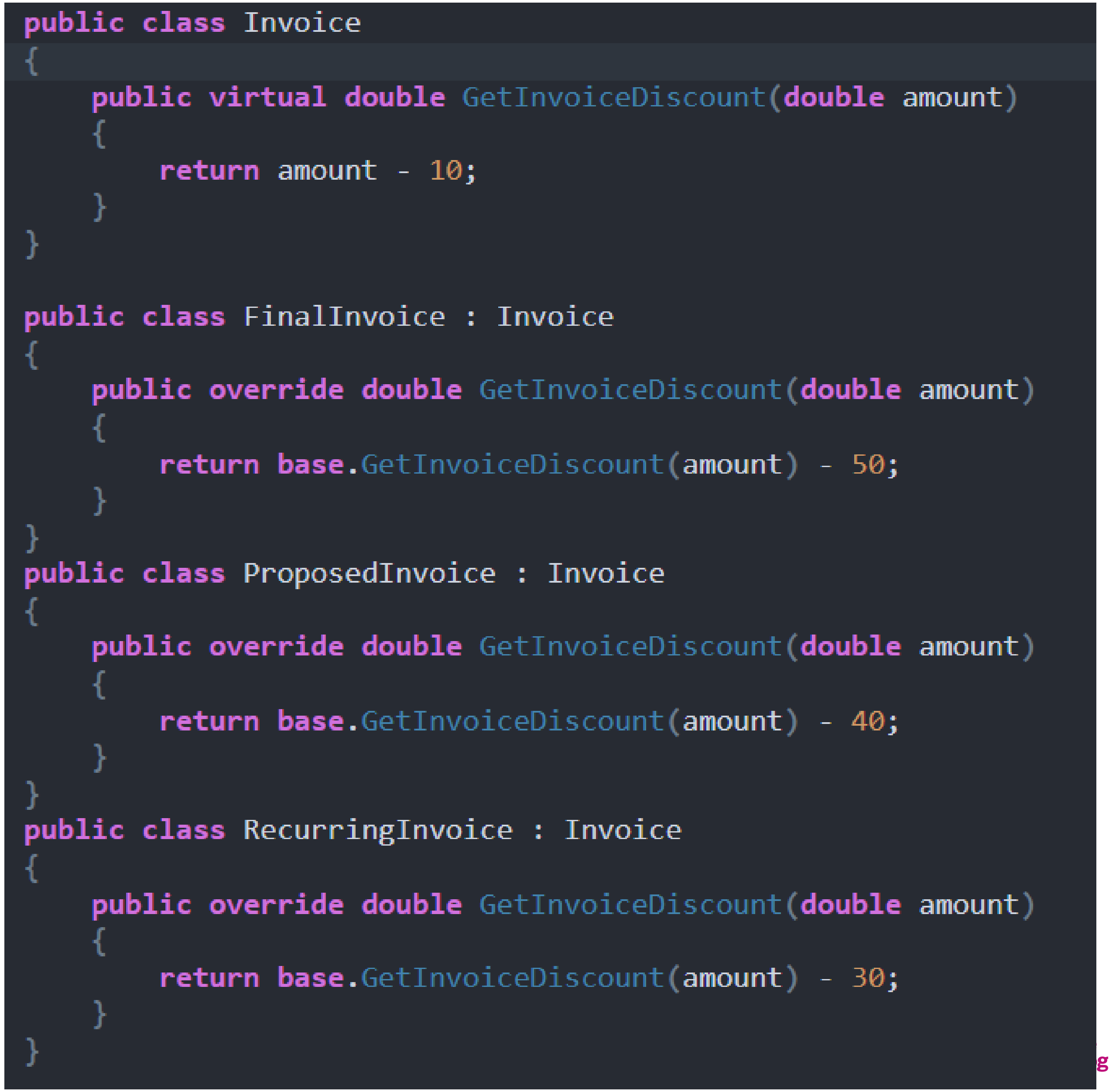
```
public class Invoice
{
    public double GetInvoiceDiscount(double amount, InvoiceType invoiceType)
    {
        double finalAmount = 0;
        if (invoiceType == InvoiceType.FinalInvoice)
        {
            finalAmount = amount - 100;
        }
        else if (invoiceType == InvoiceType.ProposedInvoice)
        {
            finalAmount = amount - 50;
        }
        return finalAmount;
    }
}
public enum InvoiceType
{
    FinalInvoice,
    ProposedInvoice
};
```

Selected List of Design Principles

Open-Closed Principle

Example: Open-Closed Principle

- Making the Invoice class as an interface or abstract class or virtual method
- Get rid of the risks of new bug when it comes to modification logic for a specific invoice type



```
public class Invoice
{
    public virtual double GetInvoiceDiscount(double amount)
    {
        return amount - 10;
    }
}

public class FinalInvoice : Invoice
{
    public override double GetInvoiceDiscount(double amount)
    {
        return base.GetInvoiceDiscount(amount) - 50;
    }
}
public class ProposedInvoice : Invoice
{
    public override double GetInvoiceDiscount(double amount)
    {
        return base.GetInvoiceDiscount(amount) - 40;
    }
}
public class RecurringInvoice : Invoice
{
    public override double GetInvoiceDiscount(double amount)
    {
        return base.GetInvoiceDiscount(amount) - 30;
    }
}
```

Selected List of Design Principles

Liskov's Substitution Principle

What it means

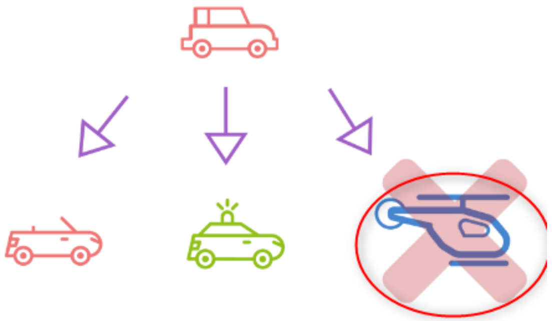
- *Derived or child classes must be substitutable for their base or parent classes*
- Without unexpected behaviour changes

Why it's useful

- Enables you to replace objects of a parent class with objects of a subclass without breaking the application
- Following Liskov Principle leads to following Open Closed Principle

Examples

- Helicopter can never replace a Car



Selected List of Design Principles

Interface Segregation Principle

What it means

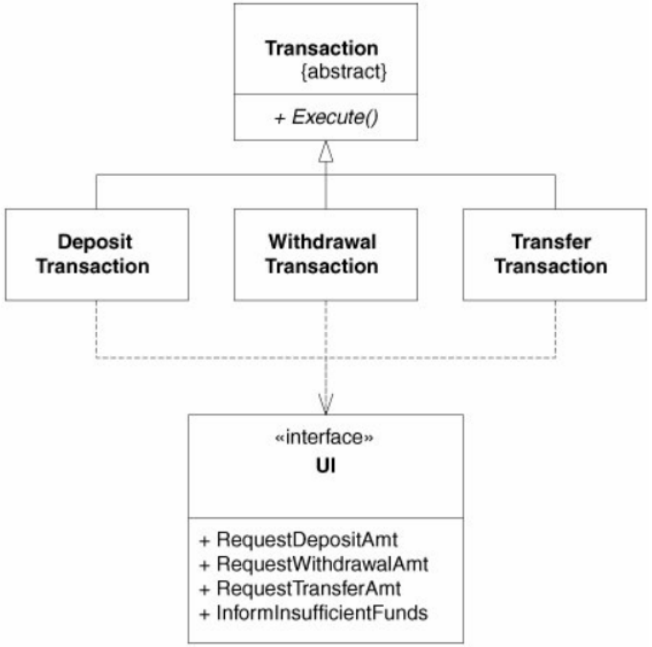
- *Clients should not be forced to depend upon interfaces that they do not use*

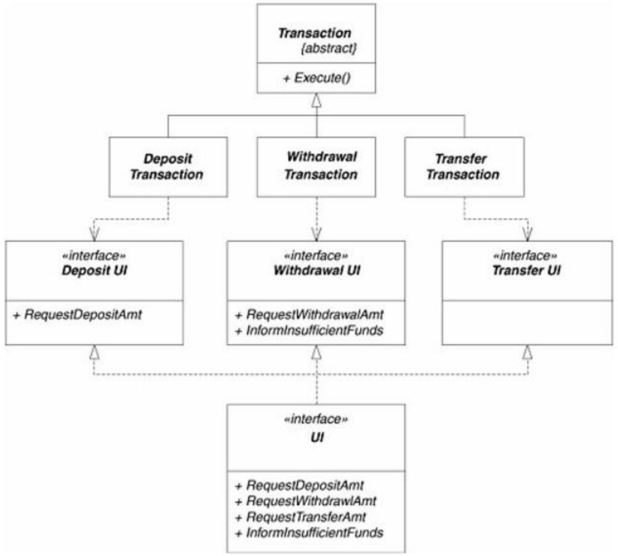
Why it’s useful

- Why is it useful?
  - Allows you to maintain SRP (Single Responsibility Priniciple)
  - Seperate Interfaces for each function can reduce dependency.

Examples

- Examples:
  - ATM Heirarchy





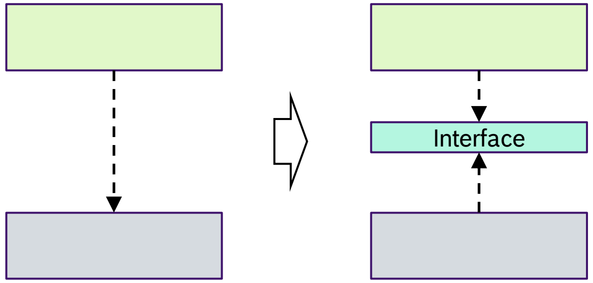
Selected List of Design Principles
Dependency Inversion

What it means

- High-level modules should not depend on low-level modules. Both should depend on abstractions.
- Abstractions should not depend on details. Details should depend on abstractions.

Why it’s useful

- Upper layer module needs not be changed when lower level module is changed as they both depend on the same abstraction (implementation-independent interface definition)



Inversion of Control (IoC)

- IoC is design pattern for implementation of Dependency inversion principle

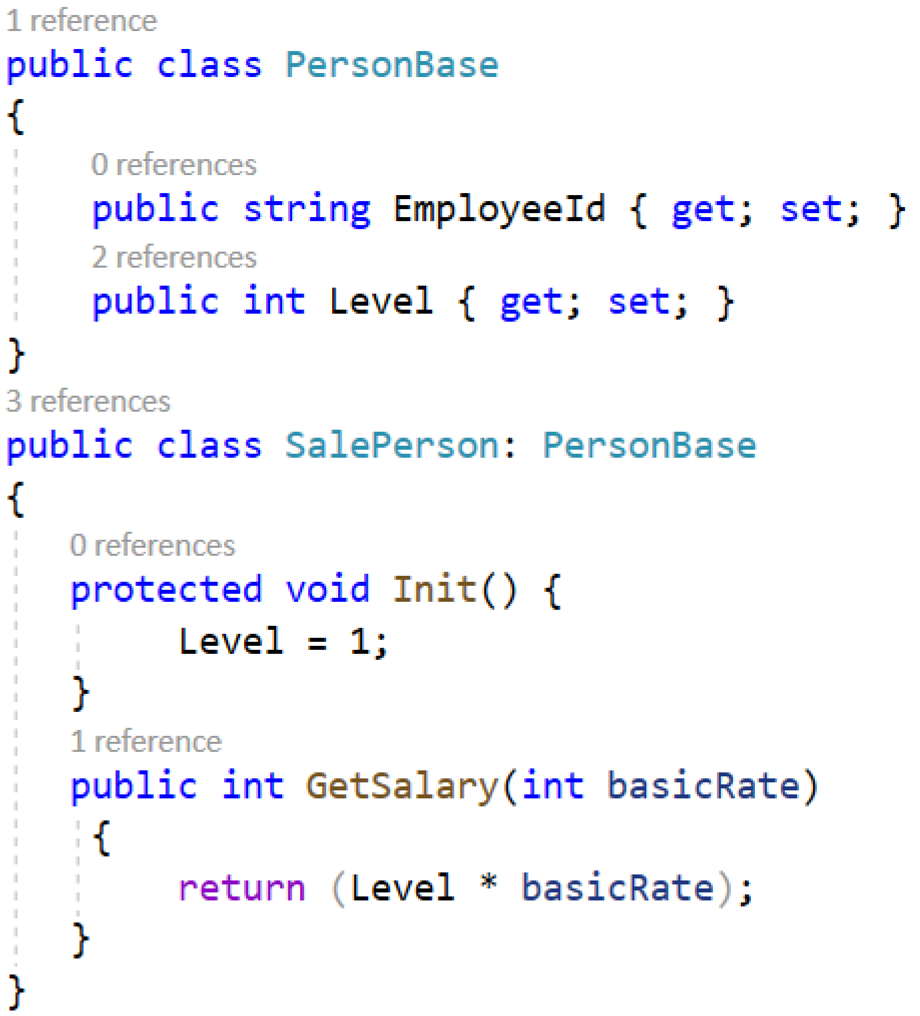
Dependency Injection (DI)

- DI is the design pattern for implementation of Dependency inversion principle. It is a subtype of IoC

Selected List of Design Principles

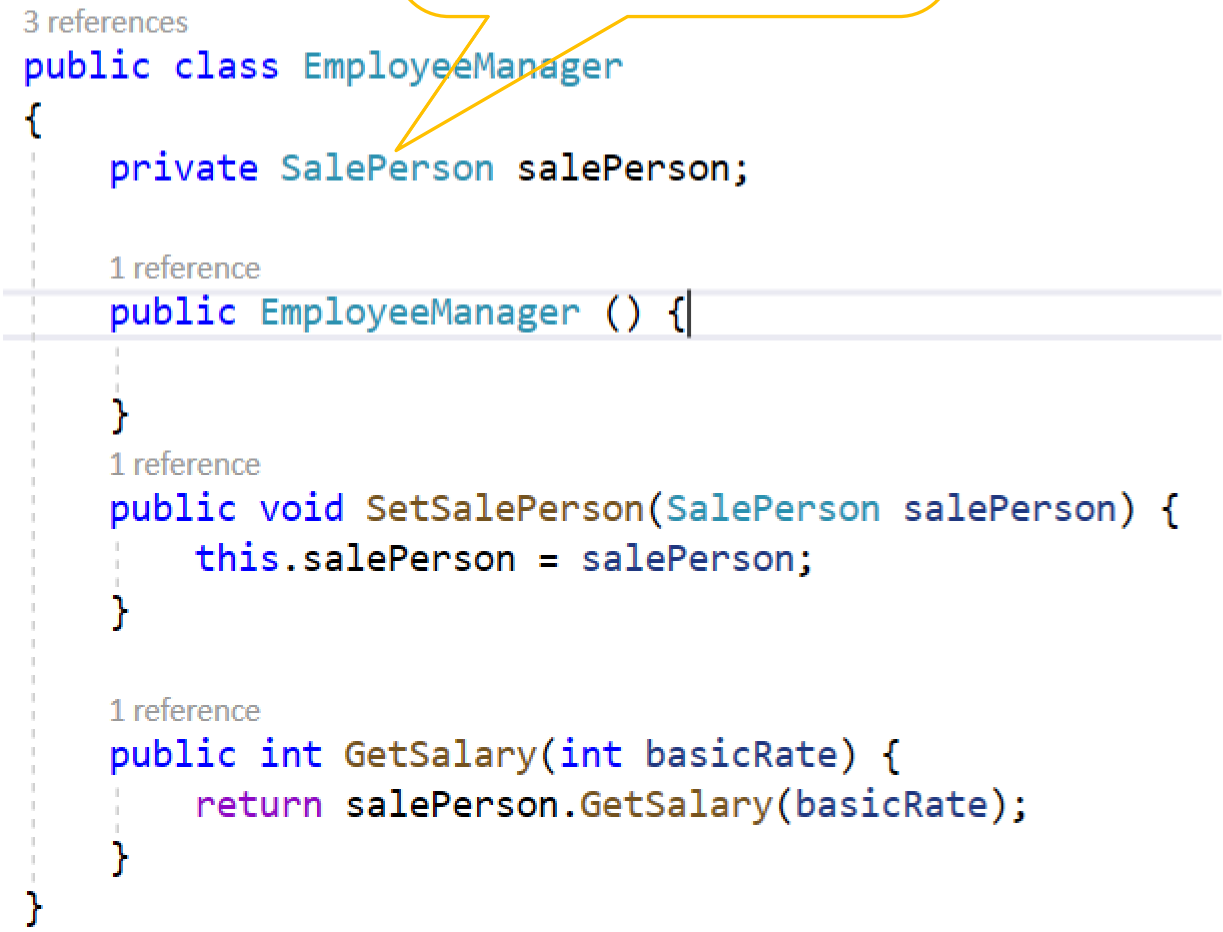
Dependency Inversion

Example: Inversion of control



```
1 reference
public class PersonBase
{
    0 references
    public string EmployeeId { get; set; }
    2 references
    public int Level { get; set; }
}
3 references
public class SalePerson: PersonBase
{
   0 references
   protected void Init() {
        Level = 1;
   }
   1 reference
   public int GetSalary(int basicRate)
    {
        return (Level * basicRate);
   }
}
```

High level employeeManager class completely depends on SalePerson class. With dependency inversion these highly dependable class should me removed.



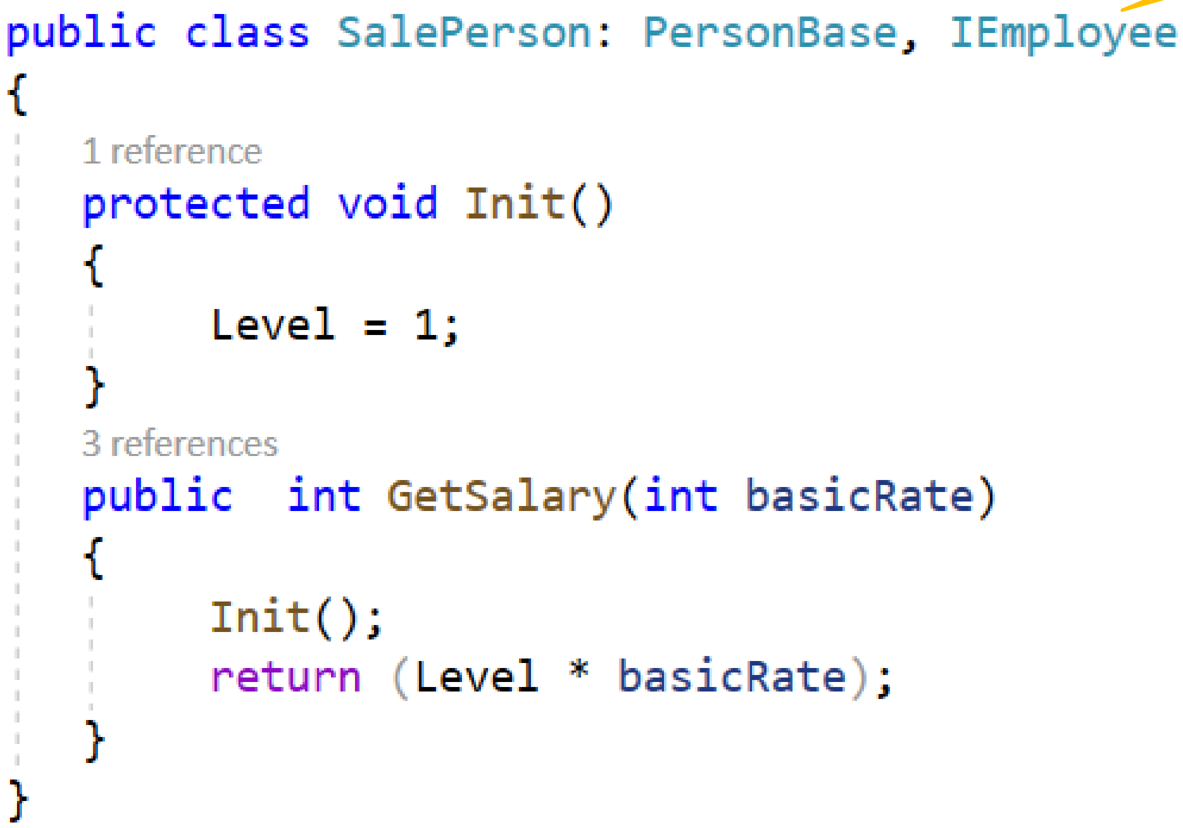
```
3 references
public class EmployeeManager
{
    private SalePerson salePerson;

    1 reference
    public EmployeeManager () {

    }
    1 reference
    public void SetSalePerson(SalePerson salePerson) {
        this.salePerson = salePerson;
    }

    1 reference
    public int GetSalary(int basicRate) {
        return salePerson.GetSalary(basicRate);
    }
}
```

Selected List of Design Principles

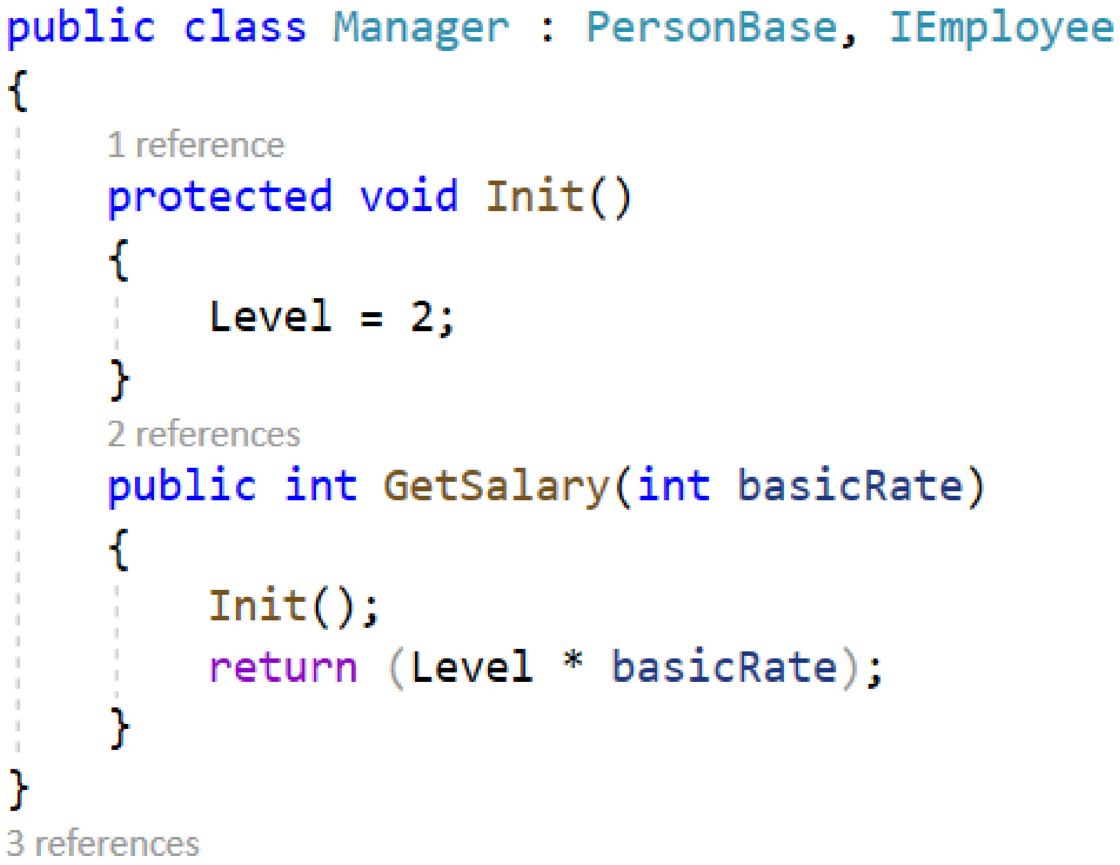
Dependency Inversion

Example: Inversion of control

One interface is introduced for high level class to communicate with low level classes which have detailed implementation



```
public class SalePerson: PersonBase, IEmployee
{
    1 reference
    protected void Init()
    {
        Level = 1;
    }
    3 references
    public  int GetSalary(int basicRate)
    {
        Init();
        return (Level * basicRate);
    }
}
```



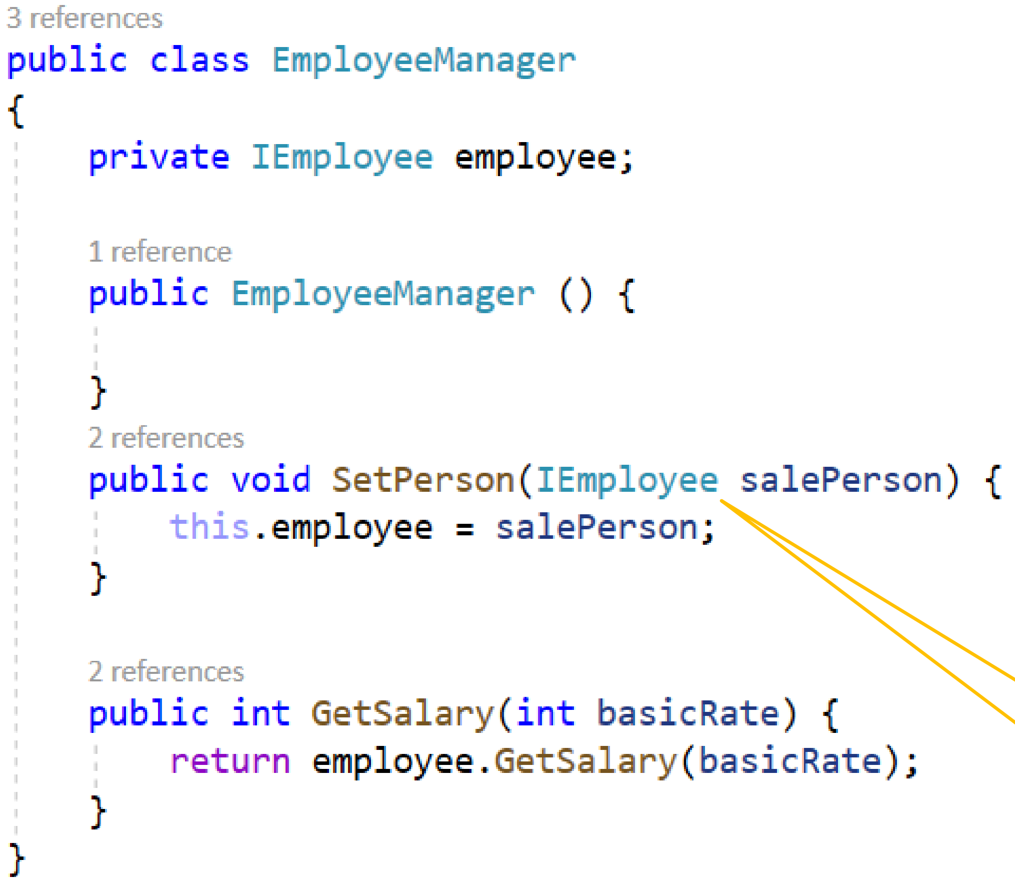
```
public class Manager : PersonBase, IEmployee
{
    1 reference
    protected void Init()
    {
        Level = 2;
    }
    2 references
    public int GetSalary(int basicRate)
    {
        Init();
        return (Level * basicRate);
    }
}
3 references
```

Bosch Global Software Technologies alt_future

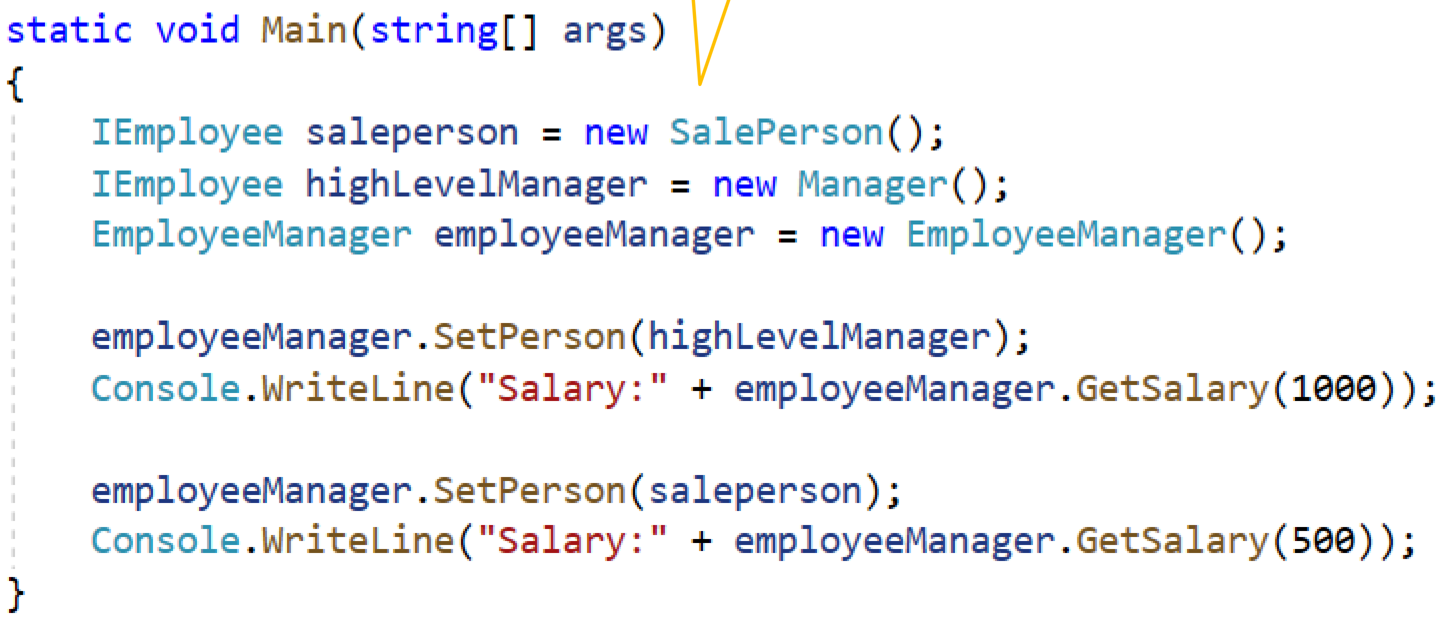
Selected List of Design Principles

Dependency Inversion

Example: Inversion of control



```
3 references
public class EmployeeManager
{
    private IEmployee employee;

    1 reference
    public EmployeeManager () {

    }
    2 references
    public void SetPerson(IEmployee salePerson) {
        this.employee = salePerson;
    }

    2 references
    public int GetSalary(int basicRate) {
        return employee.GetSalary(basicRate);
    }
}
```



```
static void Main(string[] args)
{
    IEmployee saleperson = new SalePerson();
    IEmployee highLevelManager = new Manager();
    EmployeeManager employeeManager = new EmployeeManager();

    employeeManager.SetPerson(highLevelManager);
    Console.WriteLine("Salary:" + employeeManager.GetSalary(1000));

    employeeManager.SetPerson(saleperson);
    Console.WriteLine("Salary:" + employeeManager.GetSalary(500));
}
```

Now it is pretty easy for the client to use these classes

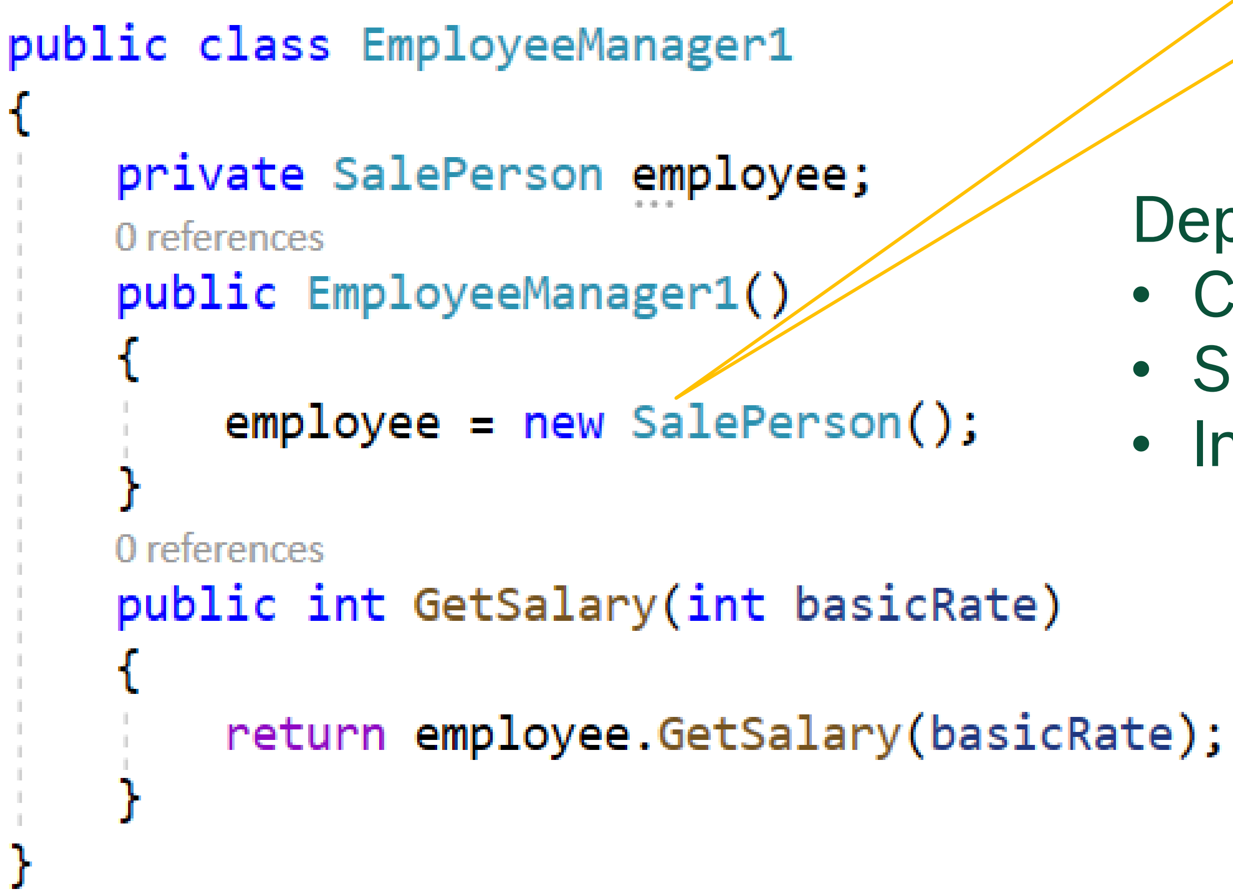
Communication interface

Selected List of Design Principles

Dependency Inversion

SalePerson instance is created in the constructor and cannot be change at runtime

Example: Dependency injection



```
public class EmployeeManager1
{
    private SalePerson employee;
    0 references
    public EmployeeManager1()
    {
        employee = new SalePerson();
    }
    0 references
    public int GetSalary(int basicRate)
    {
        return employee.GetSalary(basicRate);
    }
}
```

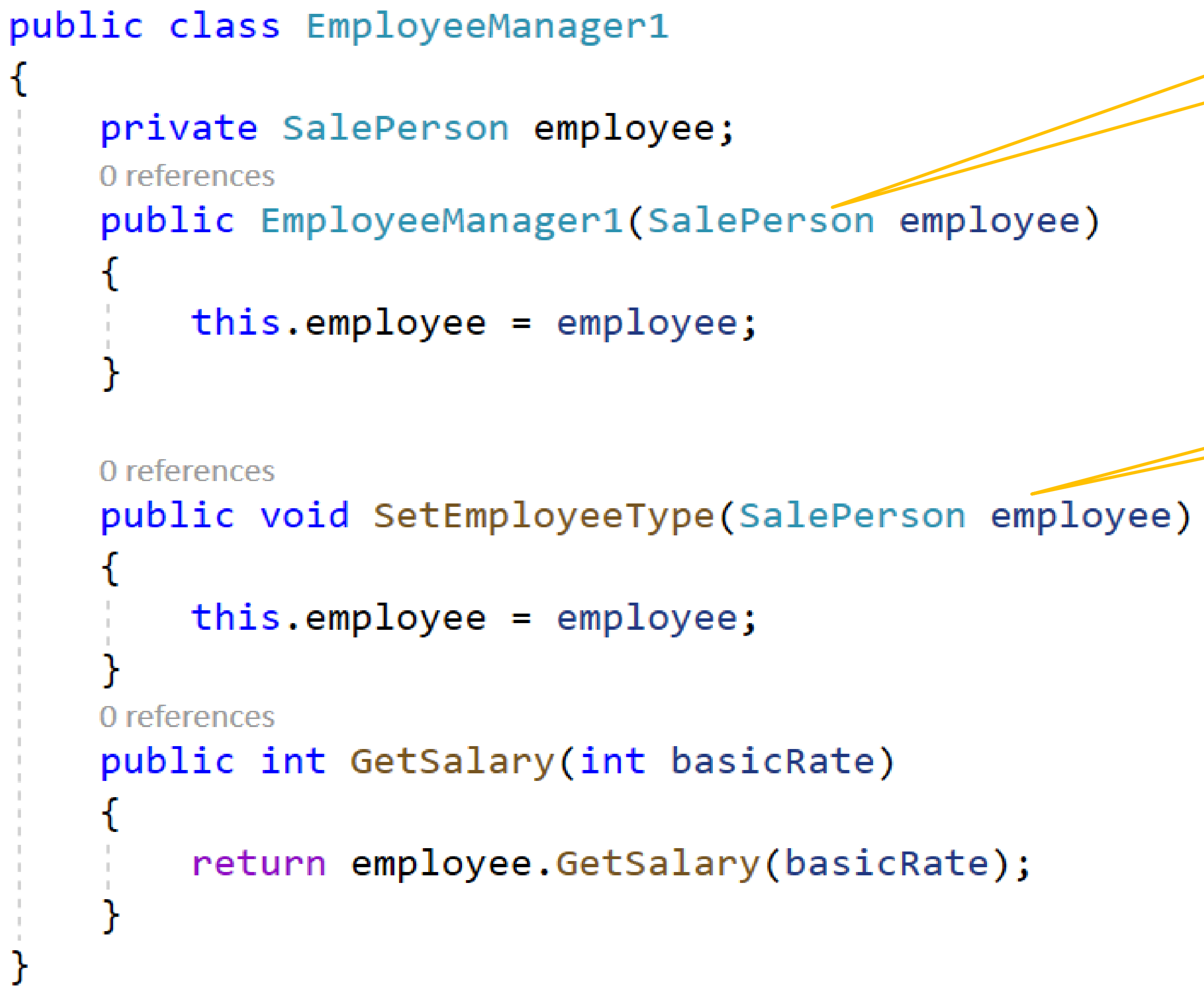
Dependency injection can be implemented with

- Constructor
- Setter
- Interface

Selected List of Design Principles

Dependency Inversion

Example: Dependency injection



```
public class EmployeeManager1
{
    private SalePerson employee;
    0 references
    public EmployeeManager1(SalePerson employee)
    {
        this.employee = employee;
    }

    0 references
    public void SetEmployeeType(SalePerson employee)
    {
        this.employee = employee;
    }
    0 references
    public int GetSalary(int basicRate)
    {
        return employee.GetSalary(basicRate);
    }
}
```

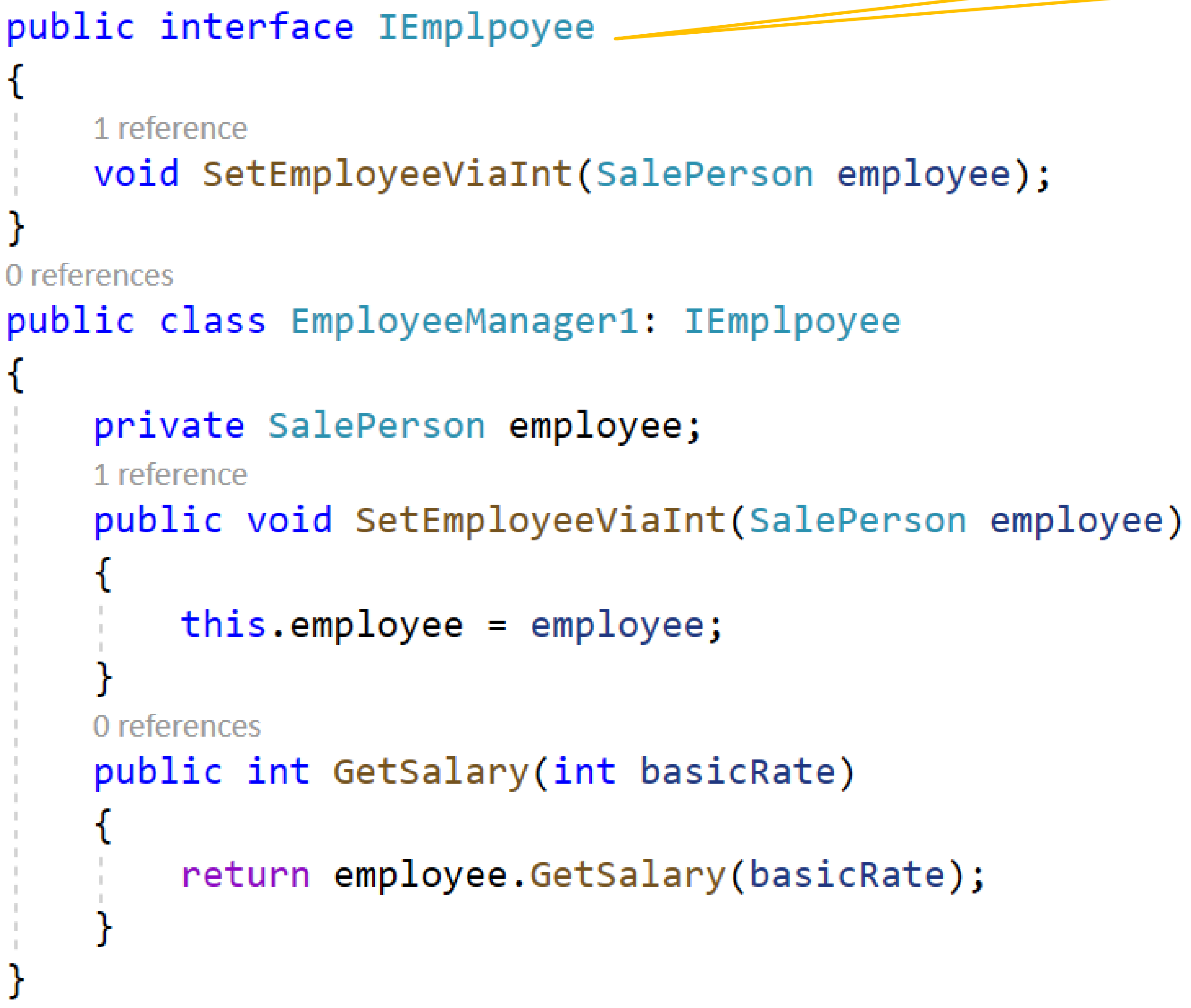
Injected via constructor

Injected via Setter

Selected List of Design Principles

Dependency Inversion

Example: Dependency injection



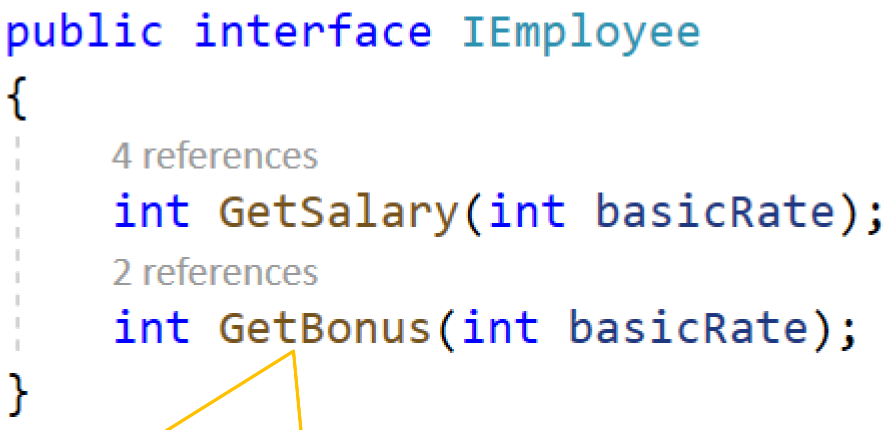
```
public interface IEmplpoyee
{
    1 reference
    void SetEmployeeViaInt(SalePerson employee);
}
0 references
public class EmployeeManager1: IEmplpoyee
{
    private SalePerson employee;
    1 reference
    public void SetEmployeeViaInt(SalePerson employee)
    {
        this.employee = employee;
    }
    0 references
    public int GetSalary(int basicRate)
    {
        return employee.GetSalary(basicRate);
    }
}
```

Injected via Interface

Selected List of Design Principles

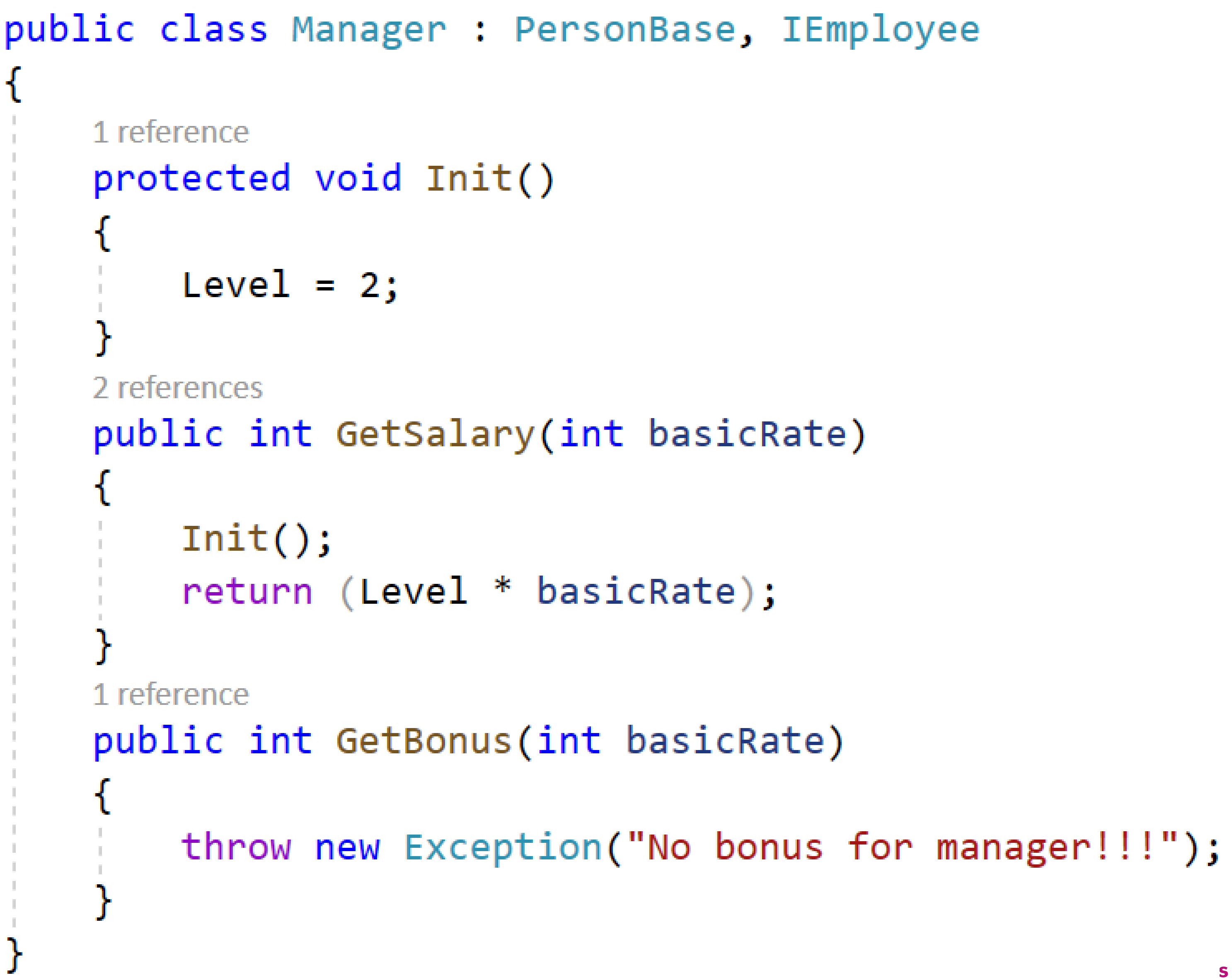
Dependency Inversion

Example: **L**iskov substitution principle



```
public interface IEmployee
{
        4 references
        int GetSalary(int basicRate);
        2 references
        int GetBonus(int basicRate);
}
```

One more method is brought into the application named GetBonus

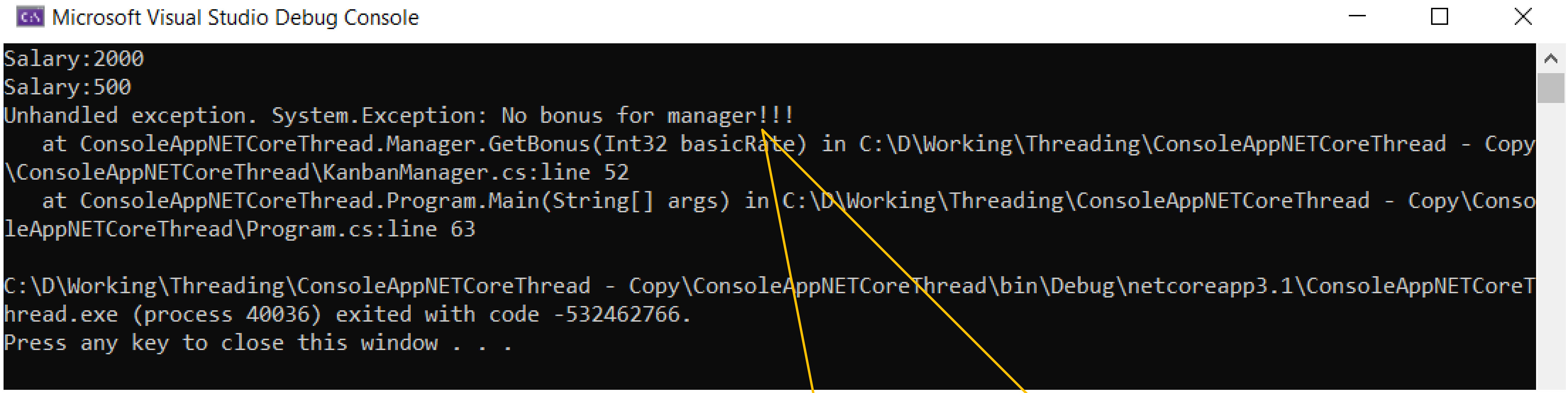


```
public class Manager : PersonBase, IEmployee
{
        1 reference
        protected void Init()
        {
            Level = 2;
        }
        2 references
        public int GetSalary(int basicRate)
        {
            Init();
            return (Level * basicRate);
        }
        1 reference
        public int GetBonus(int basicRate)
        {
            throw new Exception("No bonus for manager!!!");
        }
}
```

Selected List of Design Principles

Dependency Inversion

Example: Liskov substitution principle



```
Salary:2000
Salary:500
Unhandled exception. System.Exception: No bonus for manager!!!
   at ConsoleAppNETCoreThread.Manager.GetBonus(Int32 basicRate) in C:\D\Working\Threading\ConsoleAppNETCoreThread - Copy
\ConsoleAppNETCoreThread\KanbanManager.cs:line 52
   at ConsoleAppNETCoreThread.Program.Main(String[] args) in C:\D\Working\Threading\ConsoleAppNETCoreThread - Copy\Conso
leAppNETCoreThread\Program.cs:line 63

C:\D\Working\Threading\ConsoleAppNETCoreThread - Copy\ConsoleAppNETCoreThread\bin\Debug\netcoreapp3.1\ConsoleAppNETCoreT
hread.exe (process 40036) exited with code -532462766.
Press any key to close this window . . .
```

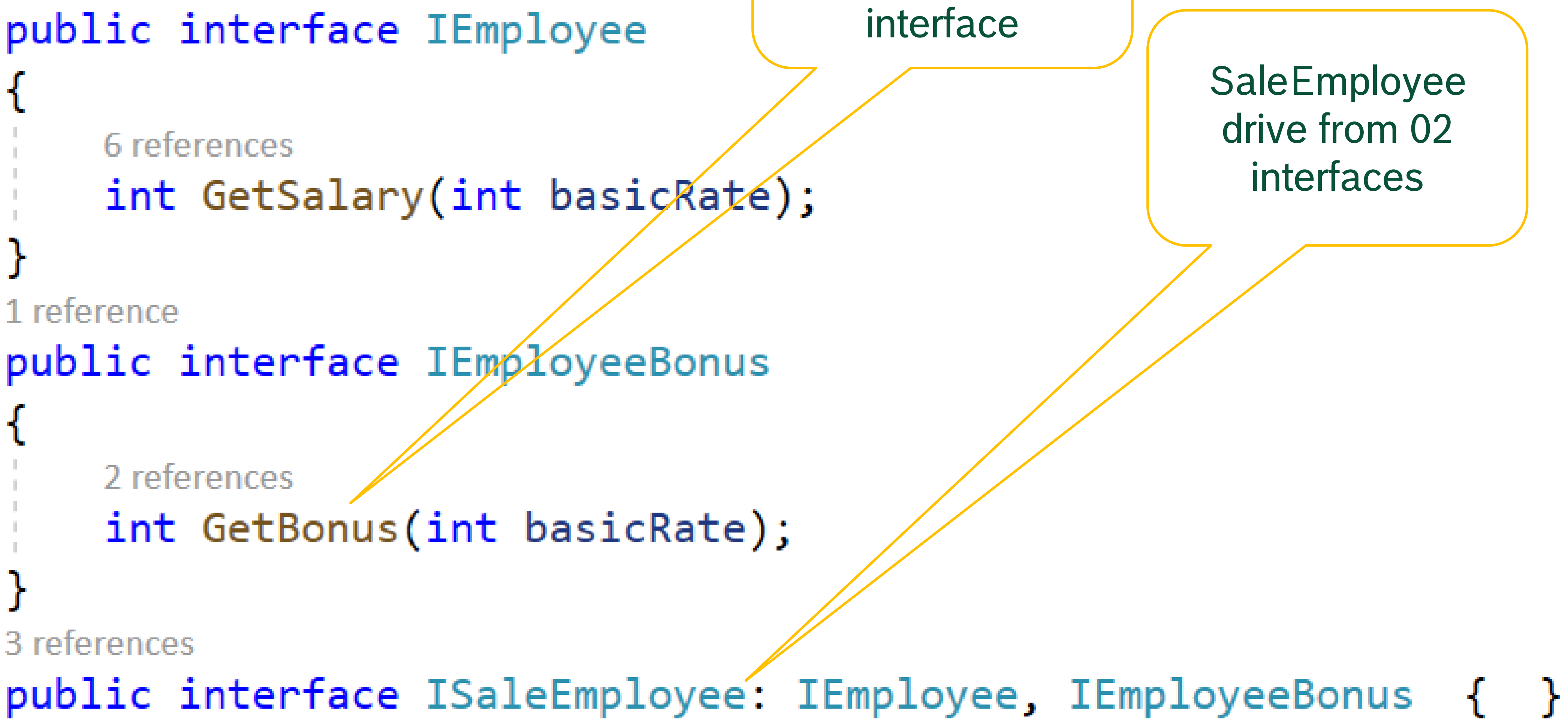
Problem created when calling GetBonus method due to Managers are not allowed to have Bonus. This violates Liskov substitution principle

Bosch Global Software Technologies
alt_future

Selected List of Design Principles

Dependency Inversion

Example: Liskov substitution principle



```
public interface IEmployee
{
    6 references
    int GetSalary(int basicRate);
}
1 reference
public interface IEmployeeBonus
{
    2 references
    int GetBonus(int basicRate);
}
3 references
public interface ISaleEmployee: IEmployee, IEmployeeBonus  {  }
```

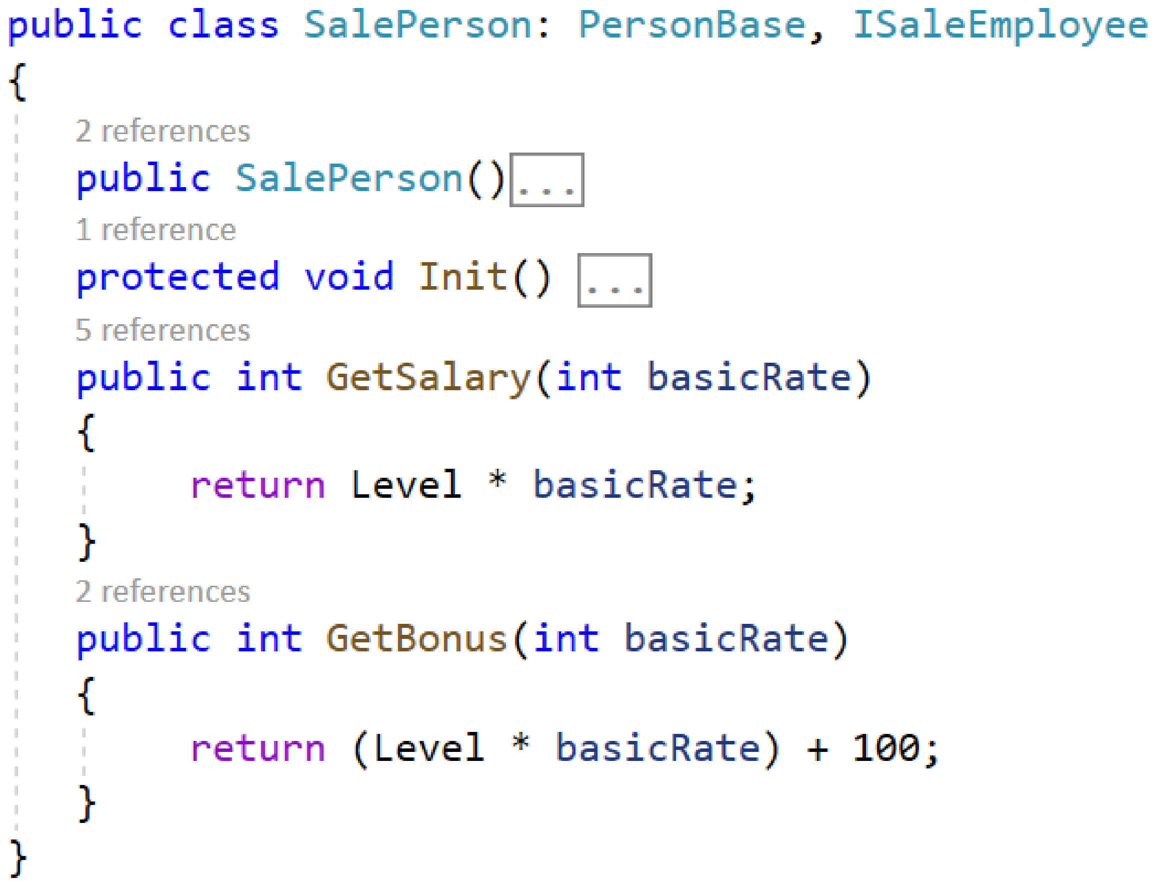
GetBonus method is now separated into other interface

SaleEmployee drive from 02 interfaces

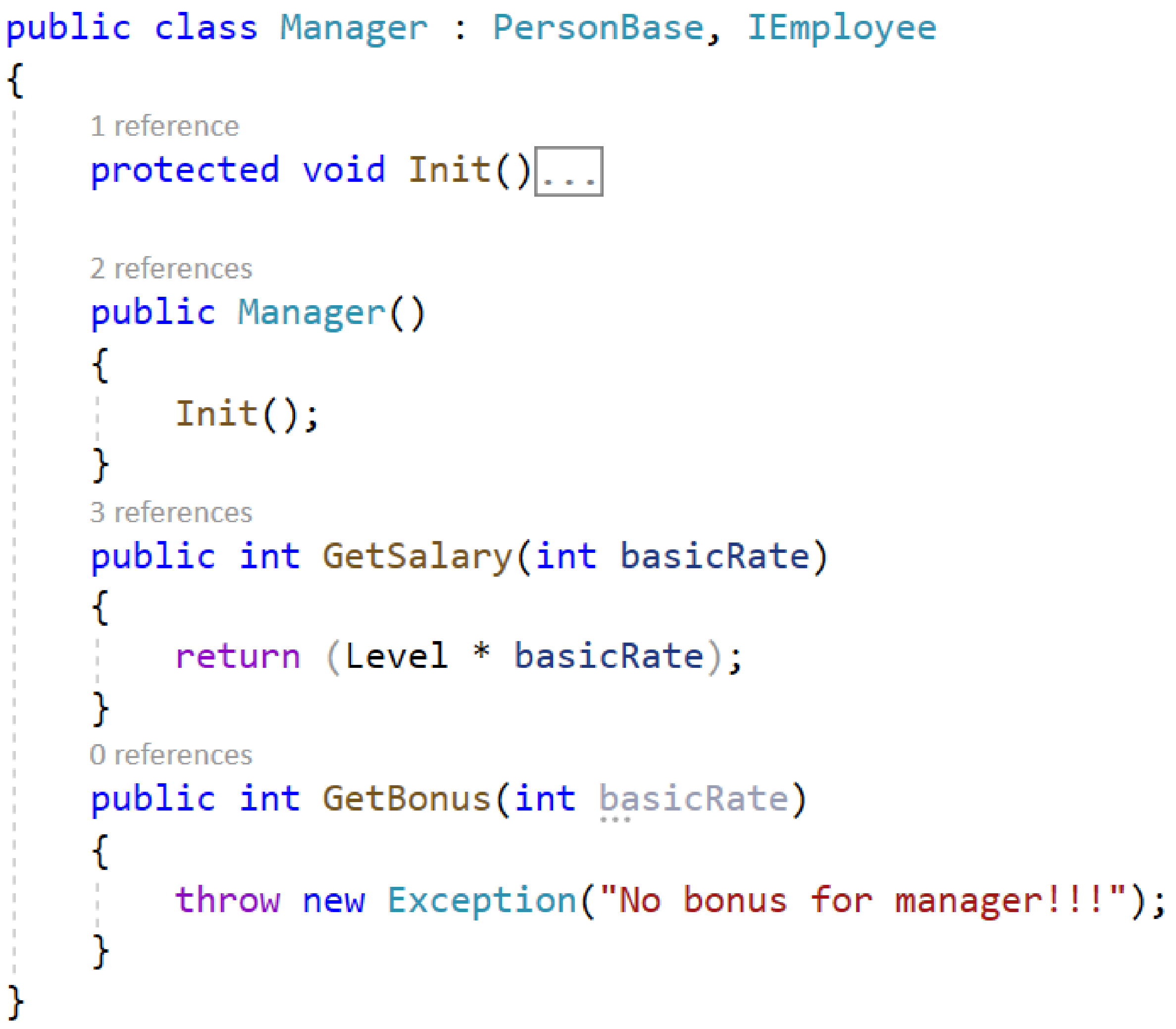
Selected List of Design Principles

Dependency Inversion

Example: Liskov substitution principle



```
public class SalePerson: PersonBase, ISaleEmployee
{
    2 references
    public SalePerson()...
    1 reference
    protected void Init() ...
    5 references
    public int GetSalary(int basicRate)
    {
        return Level * basicRate;
    }
    2 references
    public int GetBonus(int basicRate)
    {
        return (Level * basicRate) + 100;
    }
}
```



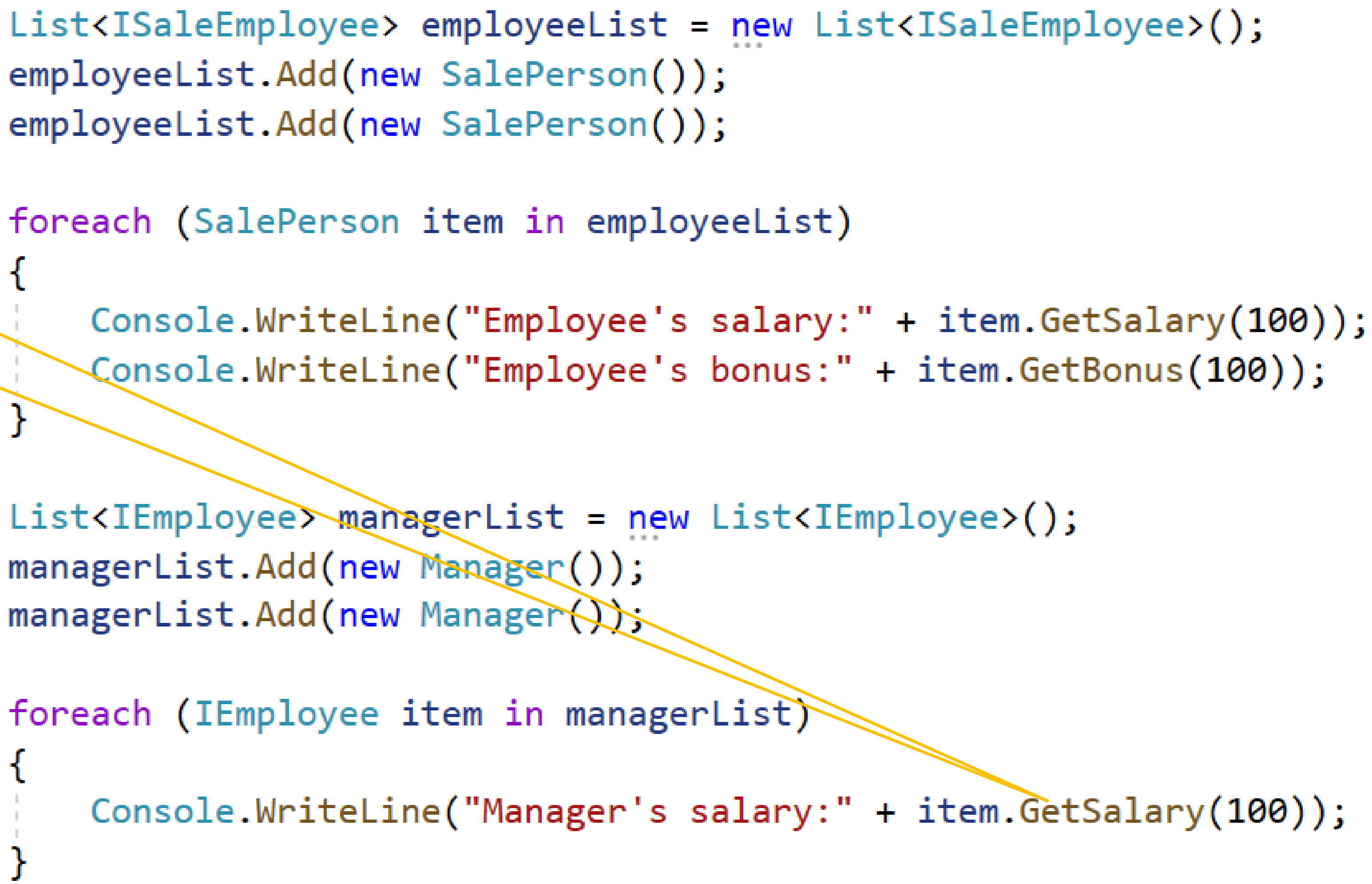
```
public class Manager : PersonBase, IEmployee
{
    1 reference
    protected void Init()...

    2 references
    public Manager()
    {
        Init();
    }
    3 references
    public int GetSalary(int basicRate)
    {
        return (Level * basicRate);
    }
    0 references
    public int GetBonus(int basicRate)
    {
        throw new Exception("No bonus for manager!!!");
    }
}
```

Selected List of Design Principles

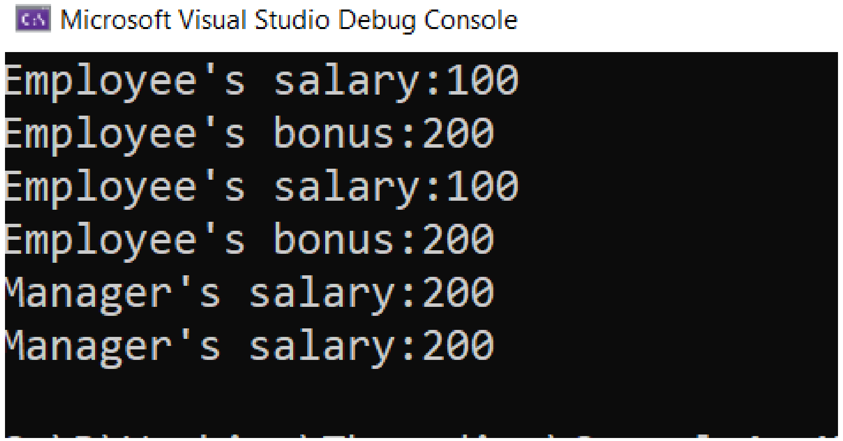
Dependency Inversion

Example: Liskov substitution principle

Run time unexpected error is eliminated. We know that this method is not applicable for manager while compiling the code



```
List<ISaleEmployee> employeeList = new List<ISaleEmployee>();
employeeList.Add(new SalePerson());
employeeList.Add(new SalePerson());

foreach (SalePerson item in employeeList)
{
    Console.WriteLine("Employee's salary:" + item.GetSalary(100));
    Console.WriteLine("Employee's bonus:" + item.GetBonus(100));
}

List<IEmployee> managerList = new List<IEmployee>();
managerList.Add(new Manager());
managerList.Add(new Manager());

foreach (IEmployee item in managerList)
{
    Console.WriteLine("Manager's salary:" + item.GetSalary(100));
}
```



```
Employee's salary:100
Employee's bonus:200
Employee's salary:100
Employee's bonus:200
Manager's salary:200
Manager's salary:200
```

Selected List of Design Principles

Dependency Inversion

Small Exercise in IoT domain

- Device Connectivity:
  - Suppose we have three different device types need to connect to, get data as well as send command to: LG washing machine, Dien Quang buld, Sony TV
  - Each of device can receive Start and Stop commands
  - Each of device has the status returning the device status
  - Each device has different way to connect to it

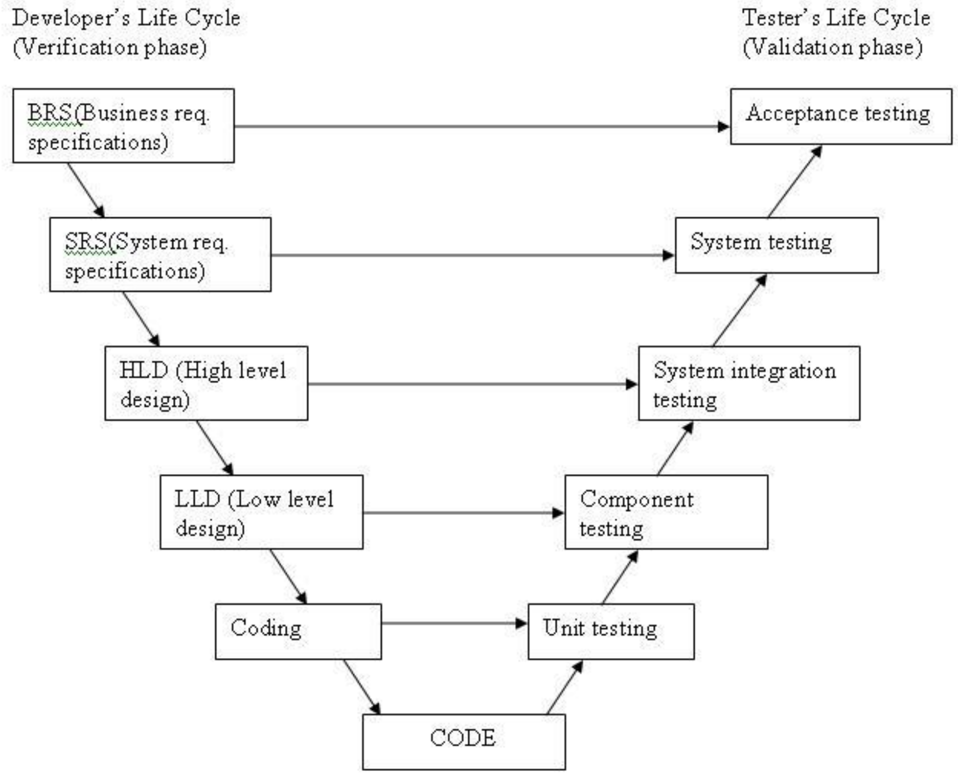
Write small application demostrating the device connectivity by applying the SOLID principles.

High Level and Low Level Design

- High Level Design (System Level)
- Overall system design - covering the system architecture and database design
- Relation between various modules and functions of the system.

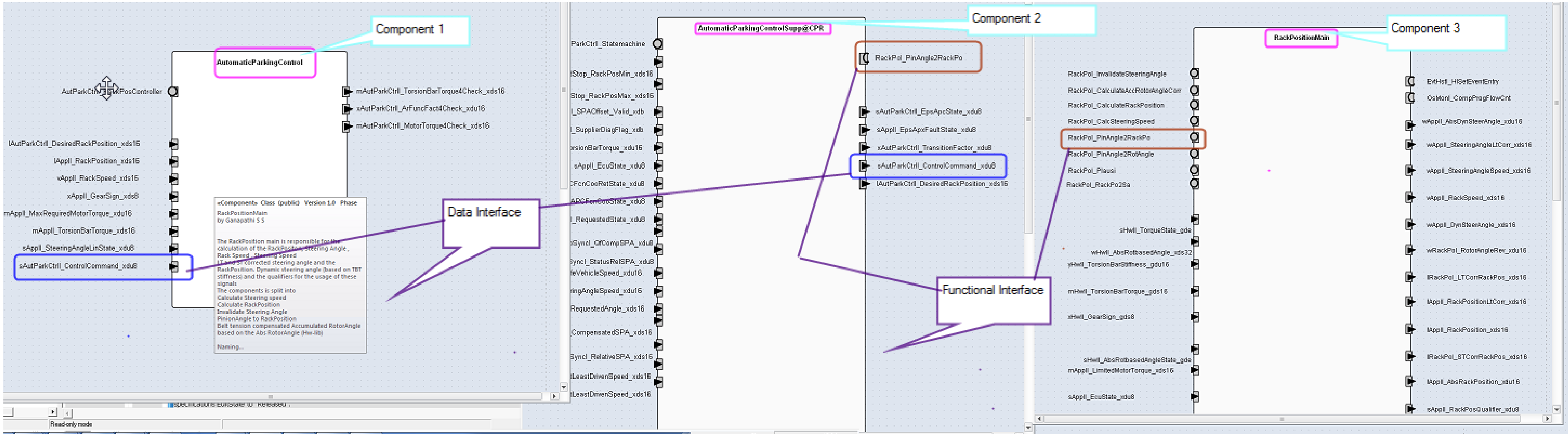
- Low Level Design (Component Level)
- Detailing the HLD.
- Defines the actual logic for each and every component of the system.

- Architecture Diagrams (Functionality Level)
- Static Design (Functional Diagram)
- Dynamic Design (Timing Diagram)



High Level and Low Level Design

High Level Design



«Component» Class (public) Version 1.0 Phase

RackPositionMain
by Ganapathi S S

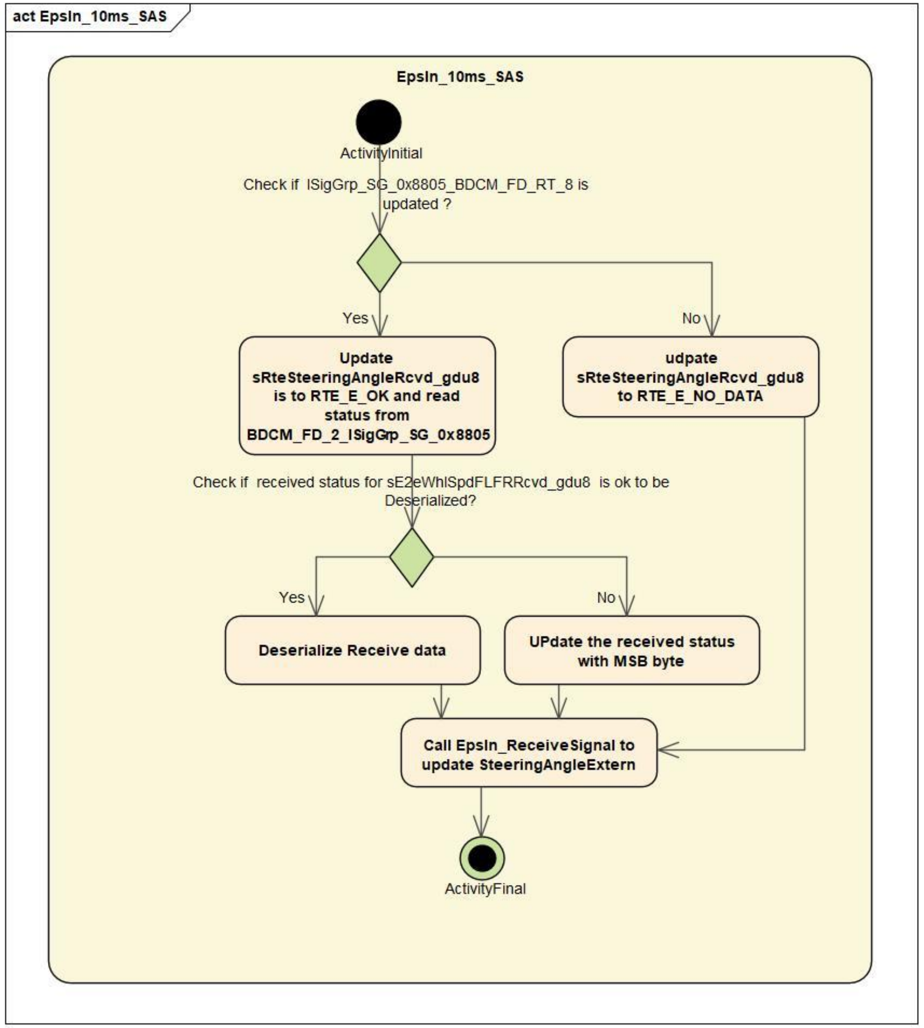
The RackPosition main is responsible for the calculation of the RackPosition, Steering Angle , Rack Speed , Steering speed
LT and ST corrected steering angle and the RackPosition. Dynamic steering angle (based on TBT stiffness) and the qualifiers for the usage of these signals
The components is split into
Calculate Steering speed
Calculate RackPosition
Invalidate Steering Angle
PinionAngle to RackPosition
Belt tension compensated Accumulated RotorAngle based on the Abs RotorAngle (Hw-lib)

Naming...

Component 1
AutomaticParkingControl
AutParkCtrl_RackPosController
mAutParkCtrl_TorsionBarTorque4Check_xds16
xAutParkCtrl_ArFuncFact4Check_xdu16
mAutParkCtrl_MotorTorque4Check_xds16
lAutParkCtrl_DesiredRackPosition_xds16
lAppl_RackPosition_xds16
vAppl_RackSpeed_xds16
xAppl_GearSign_xds8
mAppl_MaxRequiredMotorTorque_xdu16
mAppl_TorsionBarTorque_xds16
sAppl_SteeringAngleLinState_xdu8
sAutParkCtrl_ControlCommand_xdu8
Data Interface
Component 2
AutomaticParkingControlSupp@CPR
ParkCtrl_Statemachine
tStop_RackPosMin_xds16
Stop_RackPosMax_xds16
l_SPAOffset_Valid_xdb
l_SupplierDiagFlag_xdb
rsionBarTorque_xdu16
sAppl_EcuState_xdu8
CFcnCooRatState_xdu8
APCFcnCooState_xdu8
l_RequestedState_xdu8
xSyncl_OfCompSPA_xdu8
Syncl_StatusRatSPA_xdu8
feVehicleSpeed_xdu16
ringAngleSpeed_xdu16
RequestedAngle_xds16
CompensatedSPA_xds16
Syncl_RelativeSPA_xds16
tLeastDrivenSpeed_xds16
tLeastDrivenSpeed_xds16
RackPol_PinAngle2RackPo
sAutParkCtrl_EpsApcState_xdu8
sAppl_EpsApxFaultState_xdu8
xAutParkCtrl_TransitionFactor_xdu8
sAutParkCtrl_ControlCommand_xdu8
lAutParkCtrl_DesiredRackPosition_xds16
Functional Interface
Component 3
RackPositionMain
RackPol_InvalidateSteeringAngle
RackPol_CalculateAccRotorAngleCorr
RackPol_CalculateRackPosition
RackPol_CalcSteeringSpeed
RackPol_PinAngle2RackPo
RackPol_PinAngle2RotAngle
RackPol_Plausi
RackPol_RackPo2Sa
sHwil_TorqueState_gds
wHwil_AbsRotbasedAngle_xds32
yHwil_TorsionBarStiffness_gdu16
mHwil_TorsionBarTorque_gds16
xHwil_GearSign_gds8
sHwil_AbsRotbasedAngleState_gds
mAppl_LimitedMotorTorque_xds16
sAppl_EcuState_xdu8
EvtHstl_HISetEventEntry
OsMonl_CompProgFlowCnt
wAppl_AbsDynSteerAngle_xdu16
wAppl_SteeringAngleLtCorr_xds16
vAppl_SteeringAngleSpeed_xds16
vAppl_RackSpeed_xds16
wAppl_DynSteerAngle_xds16
wRackPol_RotorAngleRev_xdu16
lRackPol_LTCorrRackPos_xds16
lAppl_RackPositionLtCorr_xds16
lAppl_RackPosition_xds16
lRackPol_STCorrRackPos_xds16
lAppl_AbsRackPosition_xdu16
sAppl_RackPosQualifier_xdu8
Specifications EditState 10 Released
Read-only mode

High Level and Low Level Design

Low Level Design



act EpsIn_10ms_SAS

EpsIn_10ms_SAS

ActivityInitial

Check if ISigGrp_SG_0x8805_BDCM_FD_RT_8 is updated ?

Yes

No

Update sRteSteeringAngleRcvd_gdu8 is to RTE_E_OK and read status from BDCM_FD_2_ISigGrp_SG_0x8805

udpate sRteSteeringAngleRcvd_gdu8 to RTE_E_NO_DATA

Check if received status for sE2eWhlSpdFLFRRcvd_gdu8 is ok to be Deserialized?

Yes

No

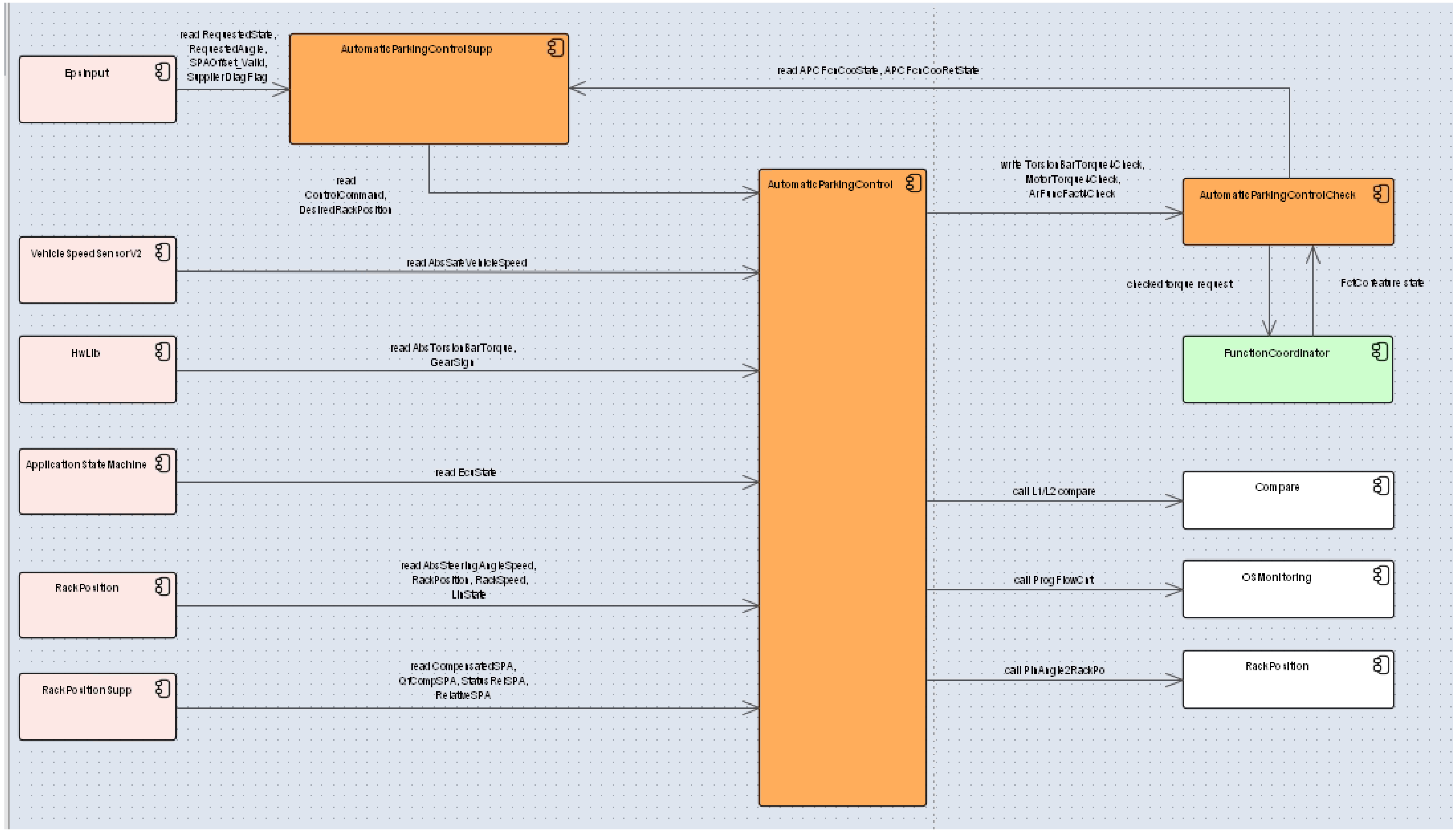
Deserialize Receive data

UPdate the received status with MSB byte

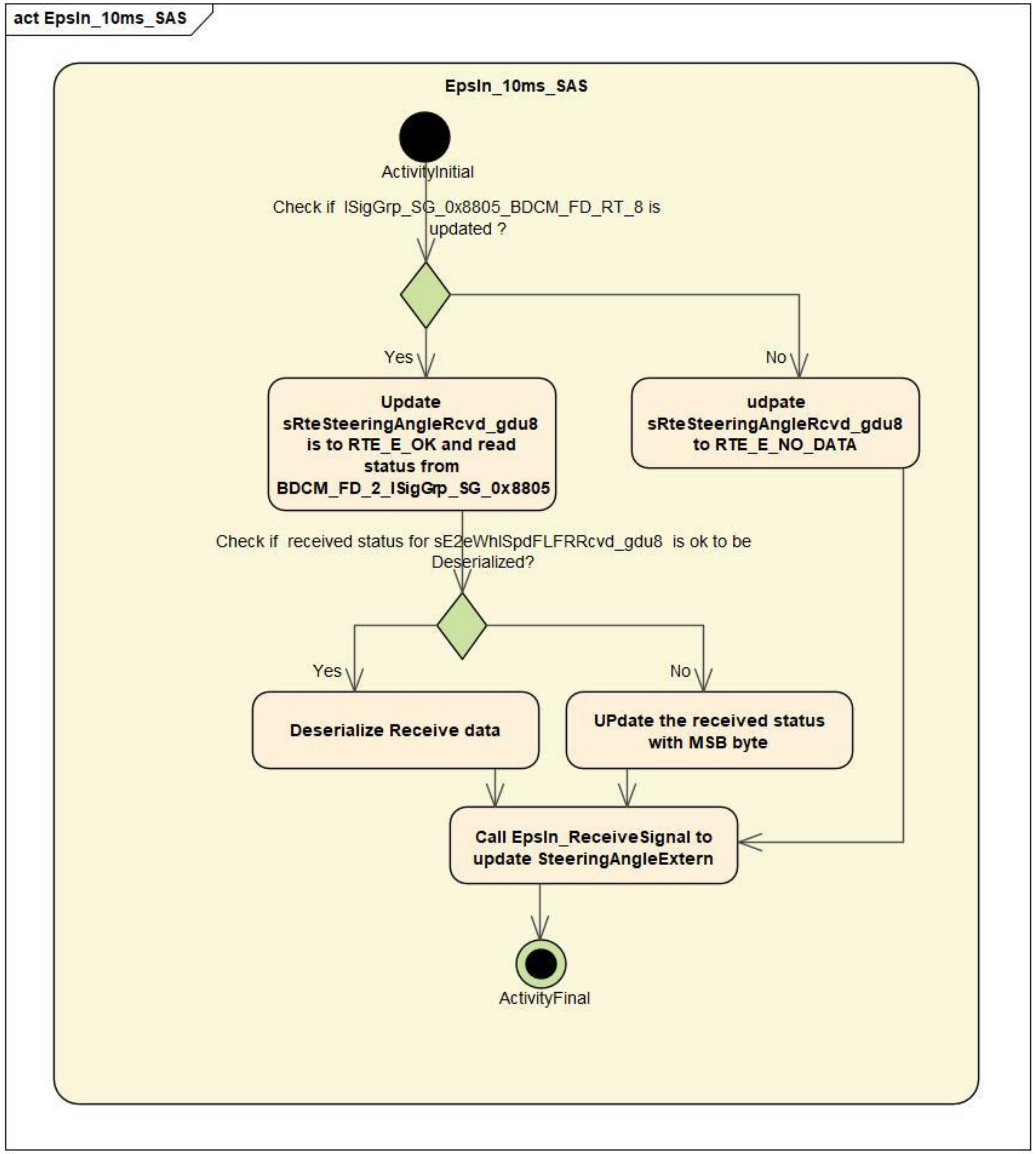
Call EpsIn_ReceiveSignal to update SteeringAngleExtern

ActivityFinal

Static Design


EpsInput
read RequestedState, RequestedAngle, SPAOffset_Valid, SupplierDiagFlag
AutomaticParkingControlSupp
read APC FctCooState, APC FctCooRetState
read ControlCommand, DesiredRackPosition
AutomaticParkingControl
write TorsionBarTorque4Check, MotorTorque4Check, ArFuncFact4Check
AutomaticParkingControlCheck
VehicleSpeedSensorV2
read AbsSafeVehicleSpeed
checked torque request
FctCo feature state
HwLib
read AbsTorsionBarTorque, GearSign
FunctionCoordinator
ApplicationStateMachine
read EcuState
call L1/L2 compare
Compare
read AbsSteeringAngleSpeed, RackPosition, RackSpeed, LinState
RackPosition
call ProgFlowCnt
OSMonitoring
RackPositionSupp
read CompensatedSPA, QfCompSPA, StatusRelSPA, RelativeSPA
call PinAngle2RackPo
RackPosition

Dynamic Design



act EpsIn_10ms_SAS

EpsIn_10ms_SAS

ActivityInitial

Check if ISigGrp_SG_0x8805_BDCM_FD_RT_8 is updated ?

Yes

No

Update sRteSteeringAngleRcvd_gdu8 is to RTE_E_OK and read status from BDCM_FD_2_ISigGrp_SG_0x8805

udpate sRteSteeringAngleRcvd_gdu8 to RTE_E_NO_DATA

Check if received status for sE2eWhlSpdFLFRRcvd_gdu8 is ok to be Deserialized?

Yes

No

Deserialize Receive data

UPdate the received status with MSB byte

Call EpsIn_ReceiveSignal to update SteeringAngleExtern

ActivityFinal

Tools

- Enterprise Architect
- Rhapsody
- Microsoft Visio
- Draw.io
- PowerPoint

OBSERVANCE

Observance Of Design Principles

Why should Software Design Principles be observed?

- They motivate the strategic approach for the design of a software system
- They can support business goals and identified quality goals (architectural drivers)
- They can help make design decisions
- Used consequently, they lead to conceptual integrity
- Violations indicate possible need of improvement / refactoring

Observance of Design Principles

Reviews and static checks can help. But in the end...

☞ ☞ ☞ IT’S YOUR RESPONSIBILITY TO OBSERVE THE ☜ ☜ ☜
SOFTWARE DESIGN PRINCIPLES!

- Consider Separation of Concerns during Feature Tree development
- If you observe a violation of design principles, clarify the issue with a SW architect
- Especially following the *separation of concerns* and *DRY* principles heavily depends on you!

Thank you!

Bosch
Global
Software
Technologies
alt_future