Presenter meets visitor

Lars-Erik Roald,

lars@timpex.no,

twitter : @lroal

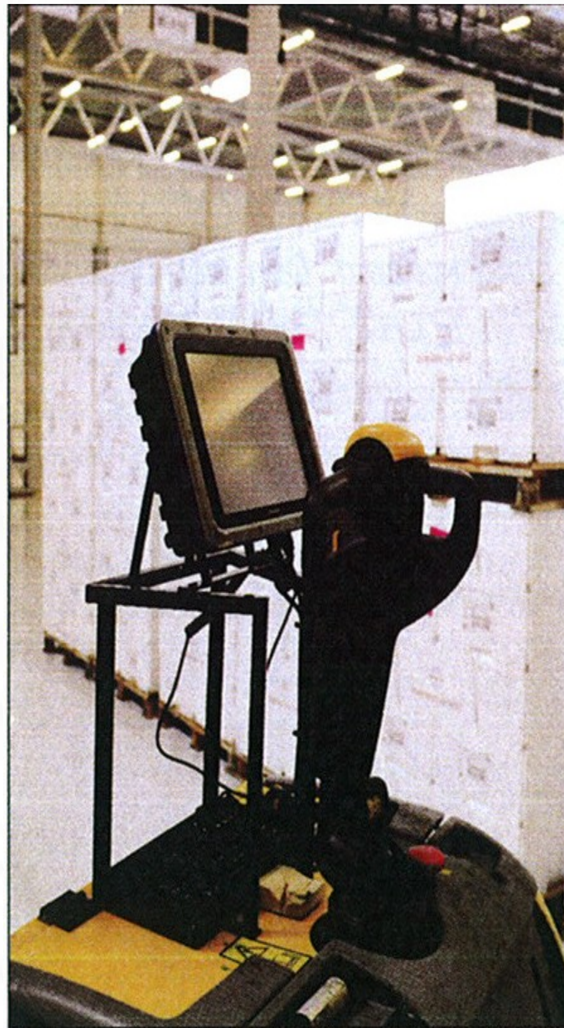
Timpex as

Litt om Timpex as

- Utvikler programvare for transport, spedisjon, terminaltjenester ++
- Etablert i 1987
- Kundemasse: 150 transportører, 40 fiskeeksportører, 20 i handel/industri
- 16 ansatte
- 2 utviklere i Trondheim, fra 2011 blir vi 3
- 5 utviklere på Åndalsnes
- **Vi ønsker flere utviklere i Trondheim. Ta kontakt !**

Presenter and Visitor

- Two powerful patterns
- Even more powerful when combined
- Presenter is easy to understand
- Visitor can make you dizzy the first time
- Visitor uses the true power of OO
- Let's start with an example..
- ..from the real world



The view

Property ETA (Estimated time of arrival)

WorkOrderListWindow

Available Workorders

Work Order	Type	Pallet Count	Gate	Est. Time	Priority	Status	Action
933	Unloading	43	05	27.01 06:00	Normal	In work	Start
922	Loading	32	13	26.01 07:22	Normal	In work	Start

Property ETD (Estimated time of departure)

Pallet Info

Logout

Class InboundWorkOrderDto

Class OutboundWorkOrderDto

How can we solve this ?

- We deal with two different classes here.
- The look similar, but differs by ETD and ETA.
- They have no Property that returns the string «Unloading» / «Loading»
- They have no Property that returns the string «In Work». (These are actually enums.)

Dirty Solution #1: Let the view handle it

- We leave the classes unchanged.
- Let the view do the dirty work

```
public void AddWorkOrdersToGrid(IEnumerable<object> workOrders) //weak typing !
{
   foreach (object workOrder  in workOrders)
   {
      if (workOrder is OutboundWorkOrderDto) //type checking!
      {
         var outboundWorkOrder = (OutboundWorkOrderDto)workOrder;
         AddOutboundWorkOrderToGrid(outboundWorkOrder); //do some really dirty work here
      }
      else if (workOrder is InboundWorkOrderDto) //type checking!

      {
         var inboundWorkOrder = (InboundWorkOrderDto)workOrder;
  AddInboundWorkOrderToGrid(inboundWorkOrder); //do some really dirty work here
 }
   }
}
```

- No strong typing
- If-switch
- Not type-safe
- A good old GUI hack

Dirty solution #2: implement an interface

- Make both WorkOrder classes implement a common interface

```
•public interface IWorkOrderDto
  {
      int WorkOrder { get; }
      string Type { get; }
      int PalletCount { get; }
      string Gate { get; }
      string ETX { get; }
      string Priority { get; }
      string Status { get; }
  }
```

- OutboundWorkOrderDto : IWorkOrderDto
- InboundWorkOrderDto : IWorkOrderDto

Why is this bad ?

- Aren't interfaces supposed to be good ?
  ..making your code loosely coupled.
- OutboundWorkOrderDto / Inb.W.Dto are now responsible for two things:
  - Keeping track of dates, Pallets, Status etc.. (logistics data)
  - Shaping data for the WorkOrderListView
- What happens when another view needs a slightly different data shaping ?

The Single Responsibility Principle (SRP)

A CLASS SHOULD HAVE ONE, AND ONLY ONE, REASON TO CHANGE.

A CLASS SHOULD DO ONLY ONE THING,

AND DO IT WELL.

- Clearly, OutboundWorkOrderDto, has two reasons to change:
  - When the view needs another shape of the data
  - When it needs to modify or extend its logistics data

Solution #3: presenters

- The presenter is responsible for shaping the data for the view.
- OutboundWorkOrderPresenter
- InboundWorkOrderPresenter
- Both must implement a common interface: IWorkOrderPresenter

IWorkOrderPresenter

```
public interface IWorkOrderPresenter
  {
      int WorkOrder { get; }
      string Type { get; }
      int PalletCount { get; }
      string Gate { get; }
      string ETXDescription { get; }
      DateTime? ETX { get; }
      string Priority { get; }
      string Status { get; }
  }
```

- This is actually the same interface as IWorkOrderDto, but OutboundWorkOrderDto / InboundWorkOrderDto don't inherit from it.
- Instead we have separate presenter classes with workorder instances as input

OutboundWorkOrderPresenter

```
public class OutboundWorkOrderPresenter : IWorkOrderPresenter
{
    private OutboundWorkOrderDto _workOrder;

    public void SetWorkOrder(OutboundWorkOrderDto workOrder)
    {
        _workOrder = workOrder;
    }

    public int WorkOrder
    {
        get { return _workOrder.Id; }
    }

    public string Type
    {
        get
        {
            return "Loading";
        }
    }

    public int PalletCount
    {
        get { return _workOrder.OrderLines.Count; }
    }


    public string Gate
    {
        get { return _workOrder.Gate; }
    }

//.......continues on next page
```

...continuing OutboundWorkOrderPresenter ...

```
//.....
        public string ETXDescription
        {
            get
            {
                if (_workOrder.Etd != null)
                    return _workOrder.Etd.Value.ToString("dd.") + _workOrder.Etd.Value.ToString("MM ") +
                           _workOrder.Etd.Value.ToShortTimeString();
                return String.Empty;
            }
        }
        public DateTime? ETX
        {
            get
            {
                return _workOrder.Etd ;
            }
        }

        public string Priority
        {
            get { return _workOrder.Priority.Name;}
        }

        public String Status
        {
            get
            {
                return GetNameForCurrentStatusCode()
            }
        }

        private string GetNameForCurrentStatusCode()
        {
            //implementation not shown here....
            //....
            .
        }
    }
```

InboundWorkOrderPresenter

```
public class InboundWorkOrderPresenter
    : IWorkOrderPresenter
{
    private InboundWorkOrderDto _workOrder;

    public void SetWorkOrder(InboundWorkOrderDto workOrder)
    {
        _workOrder = workOrder;
    }

    public int WorkOrder
    {
        get { return _workOrder.Id; }
    }

    public string Type
    {
        get { return "Unloading"; }
    }

    public int PalletCount
    {
        get { return _workOrder.OrderLines.Count; }
    }

    public string Gate
    {
        get { return _workOrder.Gate; }
    }

//.......continues on next page
```

...continuing InboundWorkOrderPresenter ...

```
//.....

    public string ETXDescription
    {
        get
        {
            if (_workOrder.Eta != null)
                return _workOrder.Eta.Value.ToString("dd.") + _workOrder.Eta.Value.ToString("MM ") +
                       _workOrder.Eta.Value.ToShortTimeString();
            return string.Empty;
        }
    }

    public DateTime? ETX
    {
        get
        {
            return _workOrder.Eta ;
        }
    }

    public string Priority
    {
        get { return _workOrder.Priority.Name; }
    }

    public string Status
    {
        get { return GetStatusNameForWorkOrderStatus(); }
    }

    private string GetStatusNameForWorkOrderStatus()
    {
        //implementation not shown here....
        //....
    }
  }
```

Transformation

- Assume we have service that returns both InboundWorkorderDtos and OutboundWorkOrderDtos
- *IEnumerable<object> GetWorkOrders()*
- We need to **transform** this to *IEnumerable<IWorkOrderPresenter>*
- Dealing with *object means casting...*
- *Let's change the signature of the service...but first..*

Revised interface for dto's

```
public interface IWorkOrderDto
   {
       int WorkOrder { get; }
       string Type { get; }
       int PalletCount { get; }
       string Gate { get; }
       string ETX { get; }
       string Priority { get; }
       string Status { get; }
   }
```

REMOVE PROPERTIES
RELATED TO DATA SHAPING

```
public interface IWorkOrderDto
   {
       int Id { get; set; }
       DateTime? DoneDate { get; set; }
       string Warehouse { get; set; }
       ..
       ..
   }
```

ONLY LOGISTICS DATA.
NO DATA SHAPING .

- InboundWorkorderDto : IWorkOrderDto
- OutboundWorkorderDto : IWorkOrderDto
- We change the signature of the service:
- *IEnumerable<IWorkOrderDto>GetWorkOrders()*

Using the transformer

```
IEnumerable<IWorkOrderDto> workOrders = workOrderService.GetWorkOrders();
var transformer = new WorkOrdersToPresentersTransformer();
IList<IWorkOrderPresenter> presenters = transformer.Transform(workOrders);
```

- How do we implement the transformer ?

Transformer implementation

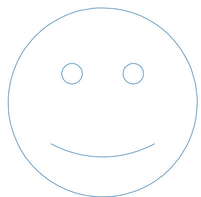
```
public class WorkOrdersToPresentersTransformer
{
    private IInboundWorkOrderPresenterFactory_inboundPresenterFactory.;
    private IOutboundWorkOrderPresenterFactory_outboundPresenterFactory.;
    private IList<IWorkOrderPresenter> _presenters;
   …
   ...constructor injection of factories not shown here
   ..
    public IList<IWorkOrderPresenter> Transform(IEnumerable<IWorkOrderDto> workOrders)
    {
        _presenters = new List<IWorkOrderPresenter>();
        foreach (var workOrder in workOrders)
        {
            if (workOrder is OutboundWorkOrderDto)
            {
                var order = (OutboundWorkOrderDto) workOrder;
                var presenter = _outboundPresenterFactory.Create(order);
                _presenters.Add(presenter);
            }
            if (workOrder is InboundWorkOrderDto)
            {
                var order = (InboundWorkOrderDto)workOrder;
                var presenter = _inboundPresenterFactory.Create(order );
                _presenters.Add(presenter);
            }
        }
        OrderByETX();
        return _presenters;
    }

    private void OrderByETX()
    {
        var orderedPresenters = _presenters.OrderBy(x => x.ETX).ToList();
        _presenters = orderedPresenters;
    }
}
```

Visitor pattern

- WorkOrdersToPresentersTransformer is the **Visitor**
- IWorkOrderDto is the one being visited (Visitable)

I want to **visit** you in order to know your true type.
I don't want to guess anymore.

Sure. I will **accept** your visit as long as you are an IWorkOrderDtoVisitor



WorkOrdersToPresentersTransformer
(Visitor)

IWorkOrderDto
(Visitable)

IWorkOrderVisitor

```
public interface IWorkOrderVisitor
{
    void Visit(OutboundWorkOrderDto outboundWorkOrder);
    void Visit(InboundWorkOrderDto inboundWorkOrder);
}
```

Revised IWorkOrderDto

```
public interface IWorkOrderDto
{
    int Id { get; set; }
    DateTime? DoneDate { get; set; }
    string Warehouse { get; set; }

    void Accept(IWorkOrderVisitor workOrderVisitor);
}
```

The WorkOrderDto's

```
public  class OutboundWorkOrderDto : IWorkOrderDto
{
    ...
    .....
    public void Accept(IWorkOrderVisitor workOrderVisitor)
    {
        workOrderVisitor.Visit(this);
    }
    ....
    .....
}


public  class InboundWorkOrderDto : IWorkOrderDto
{
    ...
    .....
    public void Accept(IWorkOrderVisitor workOrderVisitor)
    {
        workOrderVisitor.Visit(this);
    }
    ....
    .....
}
```

Revised Transformer

```
public class WorkOrdersToPresentersTransformer : IWorkOrderVisitor
{
  private IInboundWorkOrderPresenterFactory_inboundPresenterFactory.;
  private IOutboundWorkOrderPresenterFactory_outboundPresenterFactory.;
  private IList<IWorkOrderPresenter> _presenters;

  //..constructor with presenterFactories not shown here in order get all code in one page

  public IList<IWorkOrderPresenter> Transform(IEnumerable<IWorkOrderDto> workOrders)
   {
      _presenters = new List<IWorkOrderPresenter>();
      foreach (var workOrder in workOrders)
      {
         workOrder.Accept(this);
      }
      OrderByETX();
      return _presenters;
   }

   public void Visit(InboundWorkOrderDto inboundWorkOrder)
   {
      var presenter = _inboundPresenterFactory.Create( inboundWorkOrder);
      _presenters.Add(presenter);
   }

   public void Visit(OutboundWorkOrderDto outboundWorkOrder)
   {
      var presenter = _outboundPresenterFactory.Create( outboundWorkOrder );
       _presenters.Add(presenter);
   }
   private void OrderByETX()
   {
      var orderedPresenters = _presenters.OrderBy(x => x.ETX).ToList();
      _presenters = orderedPresenters;
   }
}
```

Advantages

- Look mama, no ifs !
- True OOD
- Type-safe, strong typing.
- When introducing a new WorkOrder type...
  - ..inheriting from IWorkOrderDto ensures that it accepts the transformer (visitor)
  - ..add a Visit(..) method to the IWorkOrderVisitor (or else you will get a compile error!)
  - ..then the transformer must decide how to deal with the new WorkOrder type by implementing Visit(«<theNewWorkOrderClass>» workOrder)

Are we done yet ?

WorkOrderListWindow

Available Workorders

Work Order	Type	Pallet Count	Gate	Est. Time	Priority	Status	Action
933	Unloading	43	05	27.01 06:00	Normal	In work	Start
922	Loading	32	13	26.01 07:22	Normal	In work	Start

Pallet Info Logout

Are there still any code smells ?

- The IWorkOrderVisitor, InboundWorkOrderDto and OutboundWorkOrderDto must be in the same assembly or we will get cyclic references.
- The IWorkOrderVisitor is depending on concrete dtos – makes testing harder.

Dto interfaces (marker)

```
public  interface IOutboundWorkOrderDto : IWorkOrderDto
{
}

public  interface IInboundWorkOrderDto : IWorkOrderDto
{
}

public  interface IWorkOrderDto
{
    public void Accept(IWorkOrderVisitor workOrderVisitor);
}
```

Revised WorkOrderDto's

```
public  class OutboundWorkOrderDto : IOutboundWorkOrderDto
{
    ...
    .....
    public void Accept(IWorkOrderVisitor workOrderVisitor)
    {
        workOrderVisitor.Visit(this);
    }
    ....
    .....
}


public  class InboundWorkOrderDto : IInboundWorkOrderDto
{
    ...
    .....
    public void Accept(IWorkOrderVisitor workOrderVisitor)
    {
        workOrderVisitor.Visit(this);
    }
    ....
    .....
}
```

Dto implementations

```
public class OutboundWorkOrderDto : IOutboundWorkOrderDto
{
…
    public void Accept(IWorkOrderVisitor visitor);
    {
        visitor.Visit(this);
    }
…
}
```

```
public class InboundWorkOrderDto : IInboundWorkOrderDto
{
…
    public void Accept(IWorkOrderVisitor visitor);
    {
        visitor.Visit(this);
    }
…
}
```

Visitor interface

```
public interface IWorkOrderVisitor
    {
        void Visit(IOutboundWorkOrderDto outboundWorkOrder);
        void Visit(IInboundWorkOrderDto inboundWorkOrder);
    }
```

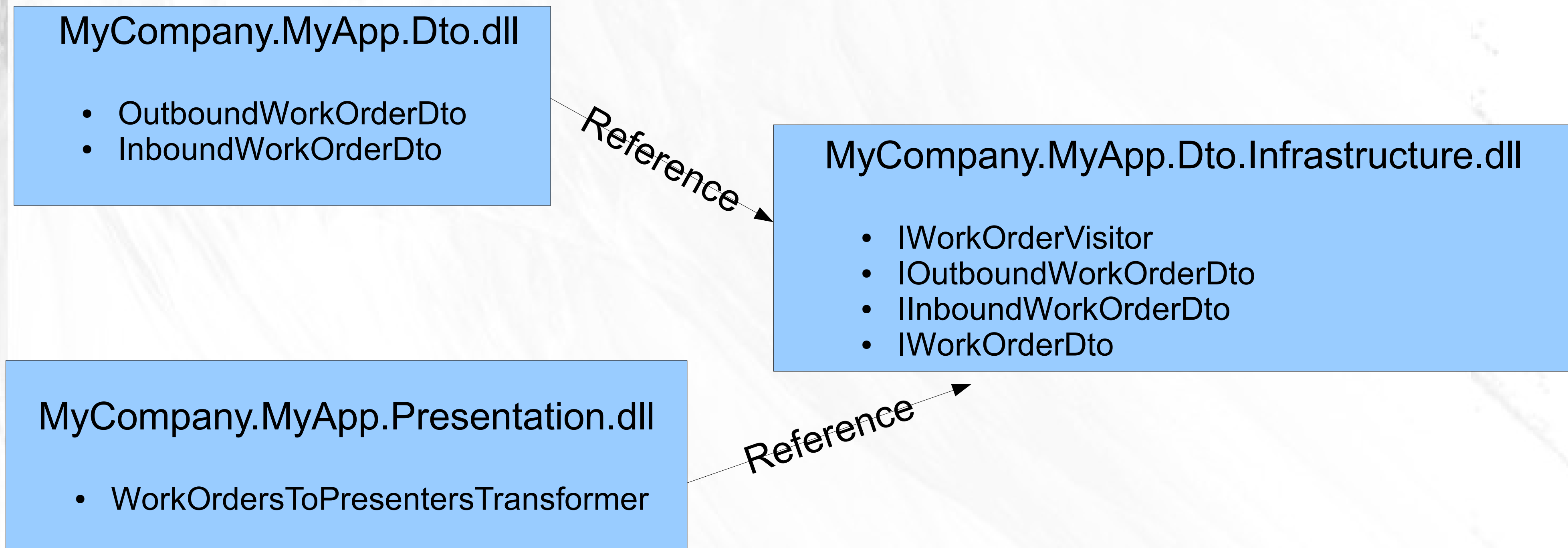
Introduce a transformer interface

```
public interface IWorkOrdersToPresentersTransformer
{
    public IList<IWorkOrderPresenter> Transform(IEnumerable<IWorkOrderDto> workOrders)
}
```

Transformer implementation

```
public class WorkOrdersToPresentersTransformer : IWorkOrdersToPresentersTransformer , IWorkOrderVisitor
{
  private IInboundWorkOrderPresenterFactory_inboundPresenterFactory.;
  private IOutboundWorkOrderPresenterFactory_outboundPresenterFactory.;
  private IList<IWorkOrderPresenter> _presenters;

  //..constructor with presenterFactories not shown here

  public IList<IWorkOrderPresenter> Transform(IEnumerable<IWorkOrderDto> workOrders)
   {
      _presenters = new List<IWorkOrderPresenter>();
      foreach (var workOrder in workOrders)
      {
         workOrder.Accept(this);
      }
      OrderByETX();
      return _presenters;
   }

   public void Visit(IInboundWorkOrderDto inboundWorkOrder)
   {
      var presenter = _inboundPresenterFactory.Create( inboundWorkOrder);
      _presenters.Add(presenter);
   }

   public void Visit(IOutboundWorkOrderDto outboundWorkOrder)
   {
      var presenter = _outboundPresenterFactory.Create( outboundWorkOrder );
       _presenters.Add(presenter);
   }
   private void OrderByETX()
   {
      var orderedPresenters = _presenters.OrderBy(x => x.ETX).ToList();
      _presenters = orderedPresenters;
   }
}
```

Organizing the assemblies



Done

WorkOrderListWindow

Available Workorders

Work Order	Type	Pallet Count	Gate	Est. Time	Priority	Status	Action
933	Unloading	43	05	27.01 06:00	Normal	In work	Start
922	Loading	32	13	26.01 07:22	Normal	In work	Start

Pallet Info Logout