SOEN 387
WEB-BASED ENTERPRISE APPLICATIONS DESIGN

TUTORIAL – 11

Patterns of Enterprise Application Architecture

Agenda

- Domain Logic Patterns
  - Transaction Script
    - Example1: TS Pattern
    - How TS pattern is implemented?
    - Example2: TS Pattern
  - Domain Model
    - Example3: Domain model
    - Class diagram of Example3
  - Service Layer
- Data Source Architectural Patterns
  - Table Module
    - Example4: Table module
  - Data Mapper
- Class exercise: Transaction script Vs. Domain model

Transaction Script

Definition: Transaction Script organizes business logic by procedures where each procedure handles a single request from the presentation.

Pros:

- TS works well for small applications that doesn't implement any complex logic
- TS is simple and lightweight domain logic pattern

Cons:

- When used in bigger applications TS has problems like code and functionalities duplication between procedures.
- Refactoring TS to other domain logic patterns is not easy.
- Don't make TS classes dependent on presentation layer classes because dependencies like these makes it harder to share those classes between applications and also testing of TS classes will be very hard this way.

[Example next]

Example1: TS Pattern

Imagine application that provides some functionalities:

- load orders to deliver,
- generate invoices,
- calculate bonuses for customers.

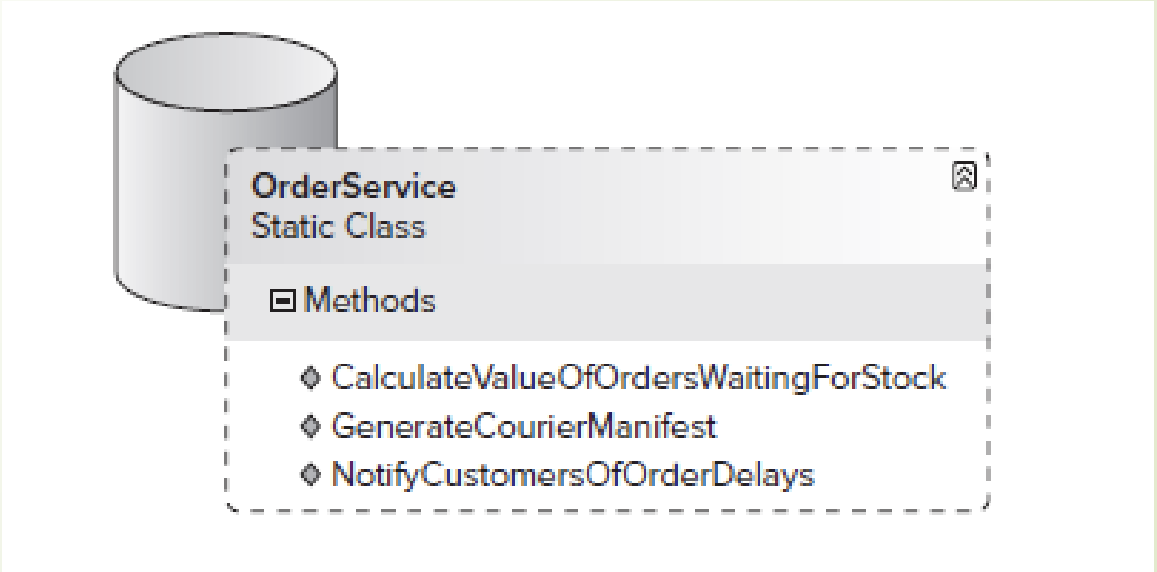
These functionalities can be handled as two transactions: generate invoices, calculate bonuses. We cań create classes to handle invoice generating and bonus calculation but we can also go with one class that provides both methods.

```
class InvoiceGenerator
{
    public void GenerateInvoices()
    {
        load orders to deliver
        iterate through orders
        generate invoice
        mark them ready to deliver
    }
}
```

This example uses one class for all operations. If you have many operations then use multiple classes to organize code better.

How TS pattern is implemented?

A single static class corresponding to your business transaction will be created and all the workflow(business rules,validation etc) which is required to complete the business transaction will be pertained in that class



Example2: TS Pattern

```
public static class HolidayService
{
    public static bool BookHolidayFor(int employeeId,DateTime
from,DateTime to)
    {
        bool booked = false;
        TimeSpan numberOfDaysRequestedForHolidays = to - from;

        if(numberOfDaysRequestedForHolidays.Days>0)
        {
            int holidaysAvailabile = GetHolidaysRemainingFor(employeeId);

            if(holidaysAvailabile >=
numberOfDaysRequestedForHolidays.Days)
            {
                SubmitHolidayBookingFor(employeeId, from, to);
            }
        }
    }

    private static void SubmitHolidayBookingFor(int employeeId, object from,
object to)
    {
        // implementation
    }
```

```
 private static int GetHolidaysRemainingFor(int employeeId)
    {
        // implementation
    }

public static List<EmployeeDTO> GetAllEmployeesOnLeaveBetween(DateTime
From, DateTime To)
    {
        //  implementation
    }

    public static List<EmployeeDTO> GetAllEmployeesWithHolidayRemaining()
    {
        //  implementation
    }
}

public class EmployeeDTO
{
        //  implementation
}
```

Domain Model

A Domain Model creates a web of interconnected objects, where each object represents some meaningful individual, whether as large as a corporation or as small as a single line on an order form.

Object-oriented way:
Many techniques to handle increasingly complex logic.
Domain Model as opposed to a Transaction Script is essence of the paradigm shift.
It takes time for people new to object models.

Database mapping:
The richer Domain Model, the more complex database mapping.
Usually with Data Mapper.

Example3: Domain model

```
class RevenueRecognition....
{
// Private attributes
private Money amount;
private MfDate date;

public RevenueRecognition(Money amount, MfDate date) {
// Constructor details
                    }
// public methods goes here
};
```

```
class Contract…
{
private List revenueRecognitions = new ArrayList(); //Object creation
public Money recognizedRevenue(MfDate asOf) {
Money result = Money.dollars(0);
Iterator it = revenueRecognitions.iterator();

// Computation of revenue goes here

return result;
};
```

Observation: A common thing you find in domain models is how multiple classes interact to do even the simplest tasks.

Well known OO pattern can be applied. Example below shows Strategy pattern

```
class Product...
private String name;
private RecognitionStrategy recognitionStrategy;
public Product(String name, RecognitionStrategy recognitionStrategy) {
// constructor details
                    }
public static Product newWordProcessor(String name) {
return new Product(name, new CompleteRecognitionStrategy());
}
public static Product newSpreadsheet(String name) {
return new Product(name, new ThreeWayRecognitionStrategy(60, 90)); }
public static Product newDatabase(String name) {
return new Product(name, new ThreeWayRecognitionStrategy(30, 60));
}
class RecognitionStrategy...
abstract void calculateRevenueRecognitions(Contract contract);
class CompleteRecognitionStrategy...
void calculateRevenueRecognitions(Contract contract) {
contract.addRevenueRecognition(new
RevenueRecognition(contract.getRevenue(), contract.getWhenSigned()));
}
```

Class diagram of Example3

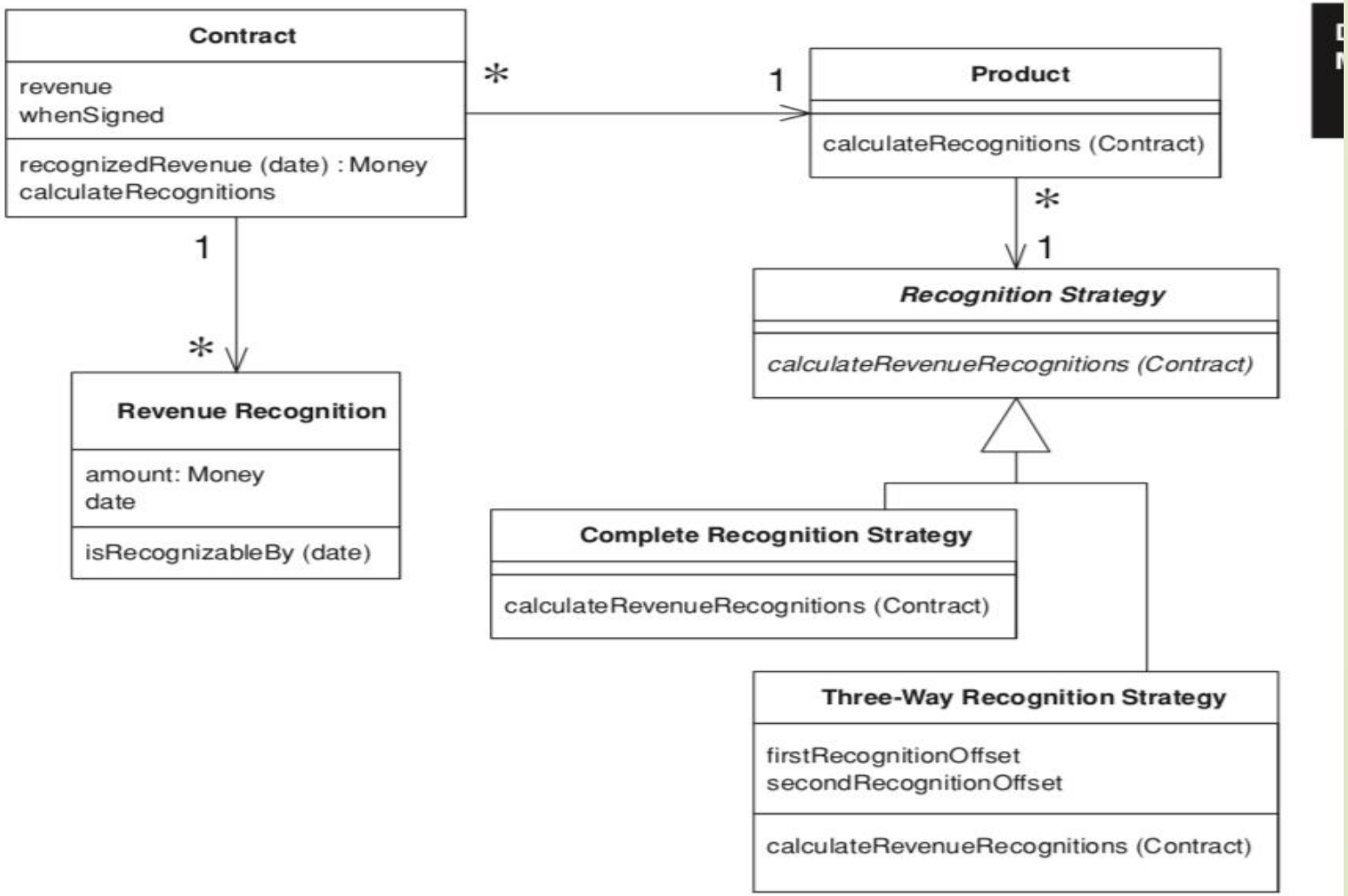
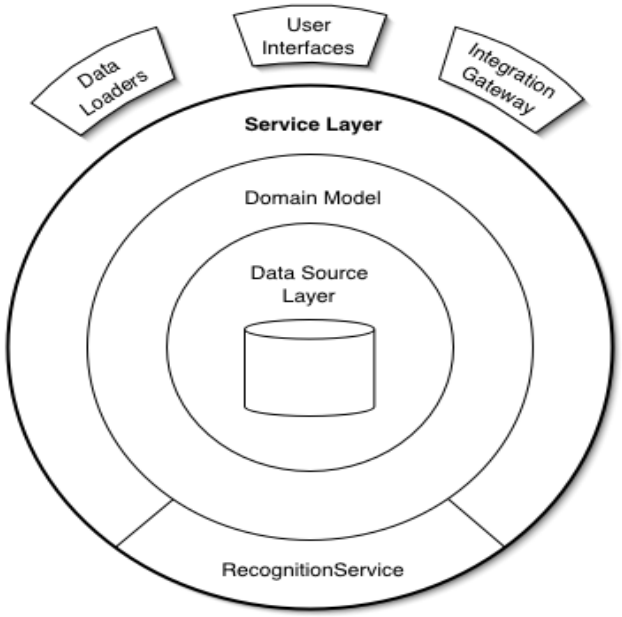


Figure 9.3 *Class diagram of the example classes for a Domain Model.*

Service Layer

The service layer is a common interface to your application logic that different clients like a web interface, a command line tool or a scheduled job can use.



The Service Layer (source: P of EAA catalog)

In its basic form, a service layer provides a set of methods that any other client can use:

```
public class ApplicationService {
                protected EmailGateway getEmailGateway() {
                  //return an instance of EmailGateway
                }
                protected IntegrationGateway getIntegrationGateway()
{
                  //return an instance of IntegrationGateway
                }
              }
              public interface EmailGateway {
                void sendEmailMessage(String toAddress, String
subject, String body);
              }
              public interface IntegrationGateway {
                void publishRevenueRecognitionCalculation(Contract
contract);
              }
```

```
public class RecognitionService extends ApplicationService
{
  public void calculateRevenueRecognitions(long
contractNumber) {
                Contract contract =
Contract.readForUpdate(contractNumber);
                contract.calculateRecognitions();
                getEmailGateway().sendEmailMessage(
                  contract.getAdministratorEmailAddress(),
                  "RE: Contract #" + contractNumber,
                  contract + " has had revenue recognitions
calculated.");

getIntegrationGateway().publishRevenueRecognitionCalculation(
contract);
              }
              public Money recognizedRevenue(long
contractNumber, Date asOf) {
                return
Contract.read(contractNumber).recognizedRevenue(asOf);
              }
            }
```

The service layer methods itself then implement the application logic and make calls to the databases or models. It does not have to be a class but can also be a set of functions that are publicly exposed.

Table Module

One of the problems with Domain Model is the interface with relational databases. A Table Module organizes domain logic with one class per table in the data-base, and a single instance of a class contains the various procedures that will act on the data.

Pros:

Many GUI environments are built to work on a results of a SQL query organized in a *Record Set*.

Middle ground between a *Transaction Script* and a *Domain Model*

Table Module provides more structure and makes it easier to find and remove duplication.

Cons:

Inheritance, strategies, and other OO patterns can't be used.

Example4: Table Module

```
class Contract...
public void CalculateRecognitions (long contractID) {
DataRow contractRow = this[contractID];
Decimal amount = (Decimal)contractRow["amount"];
RevenueRecognition rr = new RevenueRecognition (table.DataSet);
Product prod = new Product(table.DataSet);
long prodID = GetProductId(contractID);

// logic goes here

} else throw new Exception("invalid product id");
}
private Decimal[] allocate(Decimal amount, int by) {
Decimal lowResult = amount / by;
lowResult = Decimal.Round(lowResult,2);

// logic goes here

return results;
}
```

inserting a new revenue recognition record.

```
public long Insert (long contractID, Decimal amount, DateTime date)
{
        DataRow newRow = table.NewRow();
        long id = GetNextID();
        newRow["ID"] = id;
        newRow["contractID"] = contractID;
        newRow["amount"] = amount;
        newRow["date"]= String.Format("{0:s}", date);
        table.Rows.Add(newRow);
        return id;
}
```



```
class RevenueRecognition...
public Decimal RecognizedRevenue2 (long contractID, DateTime asOf) {
String filter = String.Format("ContractID = {0} AND date <= #{1:d}#", contractID,asOf);
String computeExpression = "sum(amount)";
Object sum = table.Compute(computeExpression, filter);
return (sum is System.DBNull) ? 0 : (Decimal) sum;
}
```

Query application

Data Mapper

A layer of Mappers that moves data between objects and a database while keeping them independent of each other and the mapper itself.

Mapper means an object that sets up a communication between two independent objects.

- when you want to decouple data objects from DB access layer
- when you want to write multiple data retrieval/persistence implementation

A simple case would have a Person and Person Mapper class. To load a person from the database, a client would call a find method on the mapper.The mapper uses an Identity Map pattern to see if the person is already loaded; if not, it loads it.

Step 1: Create Student domain class.

```
public final class Student implements Serializable {
                    private int studentId;
                    private String name;
                    private char grade;

                    // Constructor details goes here

                    // Getters and Setters goes here

                    @Override
                    public String toString() {
                     return "Student [studentId=" + studentId + ", name="
+ name + ", grade=" + grade + "]";
                    }
                   }
```

Step 2 : Create StudentDataMapper interface - Interface lists out the possible behaviour for all possible student mappers.

```
public interface StudentDataMapper {

                    Optional<Student> find(int studentId);

                    void insert(Student student) throws DataMapperException;

                    void update(Student student) throws DataMapperException;

                    void delete(Student student) throws DataMapperException;
                   }
```

Step 3: Implementation of Actions on Students Data.

```
public final class StudentDataMapperImpl implements StudentDataMapper {
                    /* Compare with existing students */
                    /* Return empty value */

@Override
public void update(Student studentToBeUpdated) throws
DataMapperException {
                     /* Check with existing students */
        /* Get the index of student in list *
        /* Update the student in list */
        /* Throw user error after wrapping in a runtime exception */
                    }

@Override
public void insert(Student studentToBeInserted) throws
DataMapperException {

                    /* Check with existing students */
                    /* Add student in list */
                    /* Throw user error after wrapping in a runtime exception */
                    /* Check with existing students */
                    /* Delete the student from list */
                    /* Throw user error after wrapping in a runtime exception */
                   }
}
```

Link for reference

Class exercise: Transaction Vs. Domain model

Scenario: Your task is to transfer money between two bank accounts using the above two approaches.

While using Transcript model the logic is written in the implementation of MoneyTransferService, where Account is only the data structures of getters and setters.

Additionally, if using domain-driven development (DDD), the entity Account contains behaviors and business logic such as methods of debit ( ) and credit ( ) in addition to account attributes. You may think of using different Strategy pattern for one interface. however, the domain service MoneyTransferServiceDomainModelImpl only needs to call domain objects to complete the business logic.

Refer to this Link for better understanding