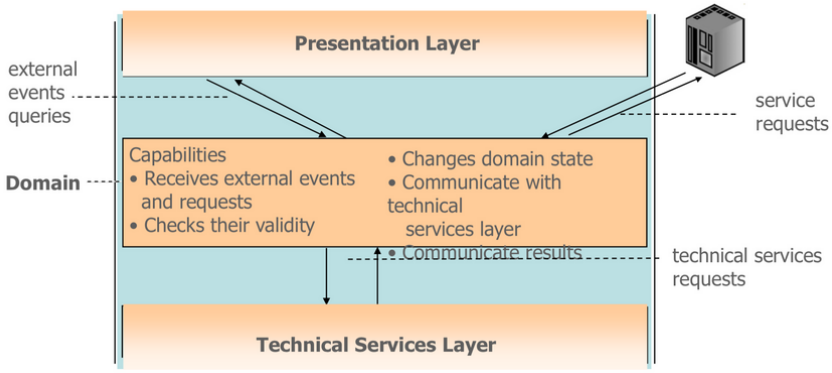
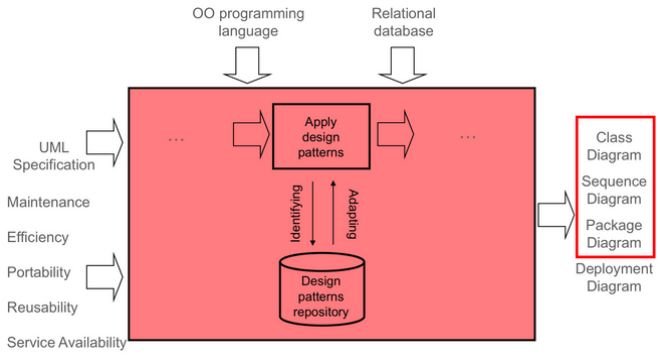
AS - Teoria 3

Unit 3.2.1. Domain Layer Design

Pattern-based Design





Patterns for Domain Layer

Patterns that determine the layers' structure.

Domain layer:

- Great influence in the assignment of responsibilities to layers
- Dominant patterns: Domain Model, Transaction Script
- They determine the services and patterns that are offered by the data layer (Data Mapper, Row Gateway, Active Record)
- Throughout this unit, we assume that Domain Model is applied

General purpose-patterns that may be applied to the domain layer. Proposed by GoF (1995) and adapted by several authors to their own methods, e.g. Larman (2005).

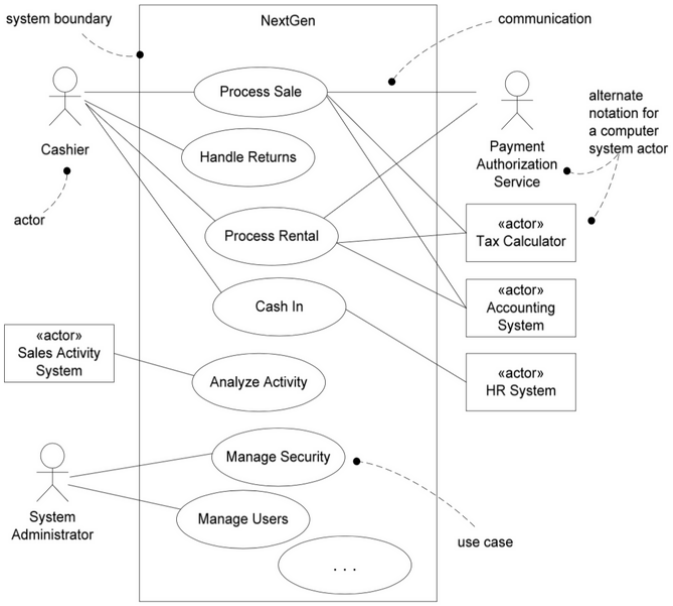
- Controller, Expert, State, Adapter, Abstract Factory, Singleton, Strategy, ...

Case Study

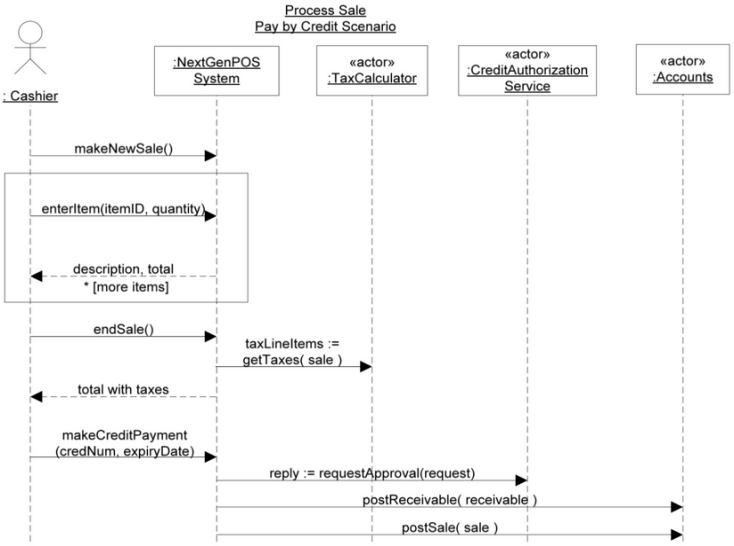
NextGen is a point-of-sale system (POS) used to record sales and handle payments.

- It is typically used in a retail store. It includes hardware components as a computer and bar code scanner; and software to run the system.
- It interfaces to various service applications, such as a third-party tax calculator and inventory control. A POS system must be relatively fault-tolerant; that is even if remote services are temporarily unavailable (such as the inventory system), it must still be capable of capturing sales and handling at least cash payments.
- A POS system increasingly must support multiple and varied client-side terminals and interfaces.

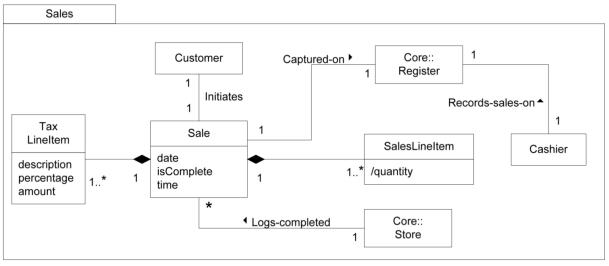
Use Case Diagram

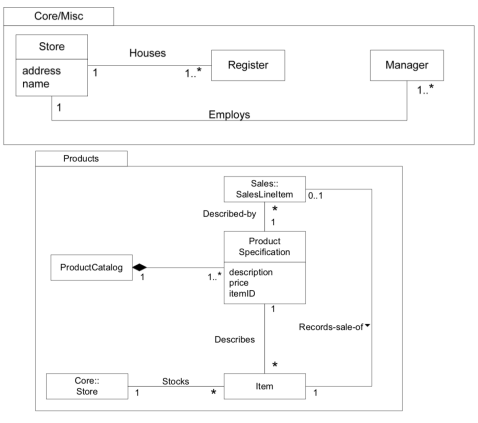


System Sequence Diagram



Conceptual schema (partial)





Unit 3.2.2. Patterns for Domain Layer

Adapter

- Overview

  Context:

  - The interface of an existing class (or a set of subclasses) does not match the one needed

  Problem:

  - How to resolve incompatible interfaces, or provide a stable interface to similar components with different interfaces?

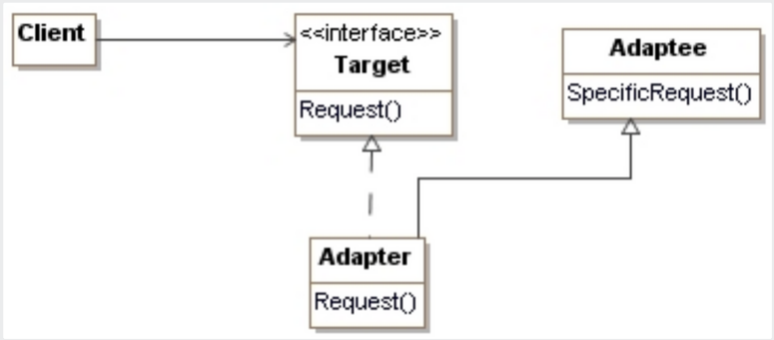
  Solution

  - Convert the interface of a class (Adaptee class) into another interface (Adapter class) clients (Client class) expects.
  - Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.
  - Also known as Wrapper.

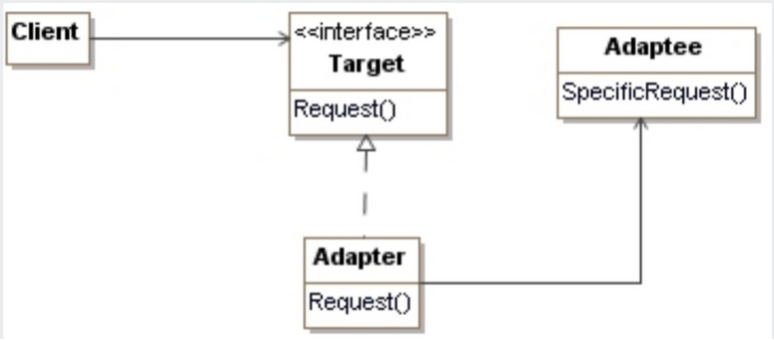
- Static View

  Two options:

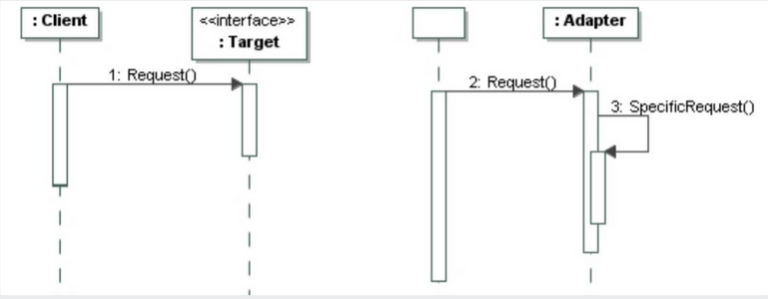
  - Class adapter

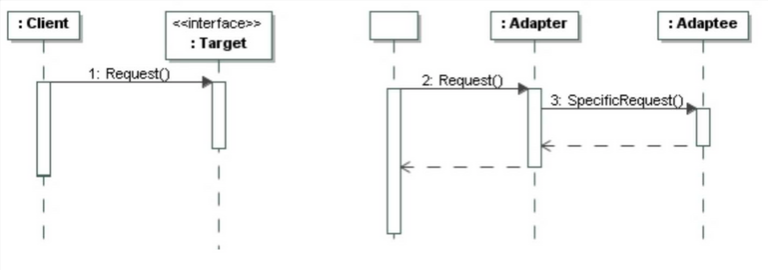
  - Object adapter (useful when one adapter adapts more than one adaptee)

- Dynamic View
  - Class adapter



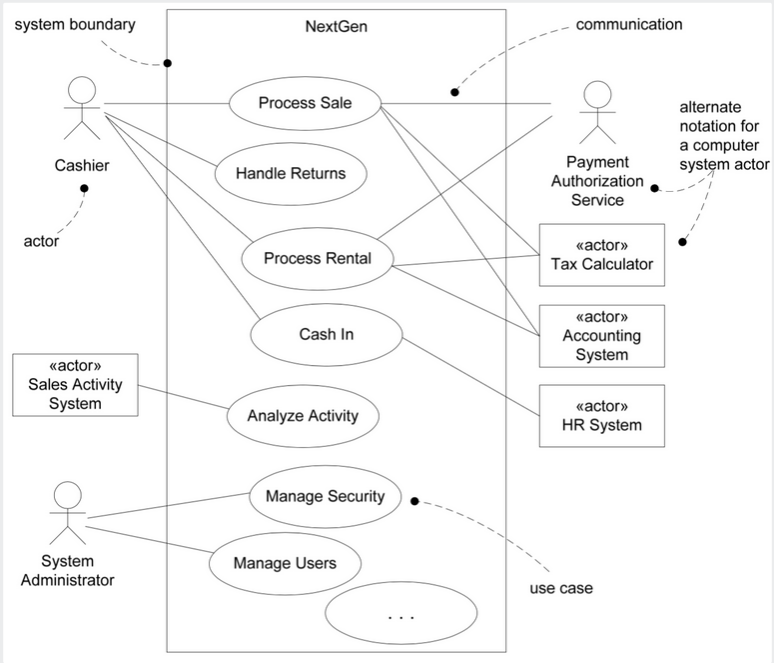
  - Object adapter



- Example

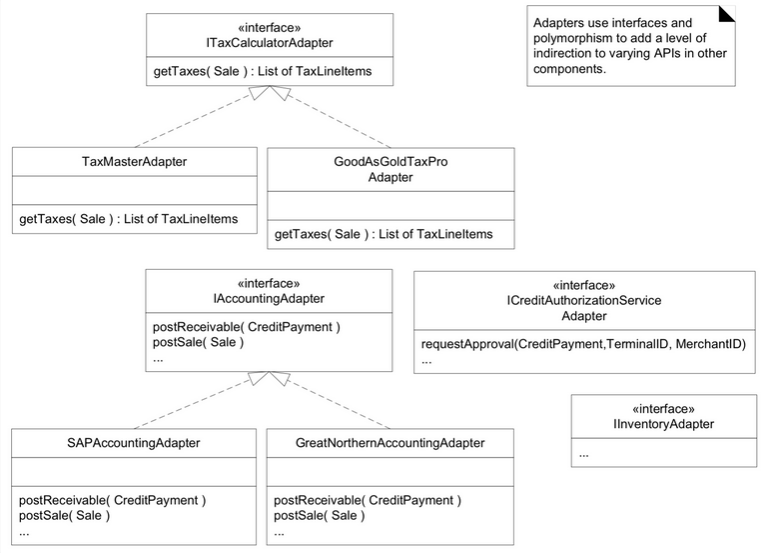
  The NextGen POS system needs to support several kinds of external services, including tax calculators, credit authorization services, inventory services, and account systems, among others. Each has a different API, which cannot be changed.

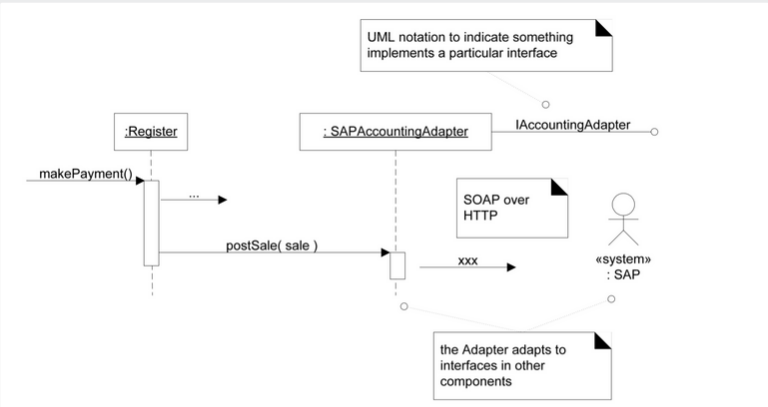
  A solution is to add a level of indirection with objects that adapt the varying external interfaces to a consistent interface used within the software system.



- makePayment contract (partial) of the Domain Layer

context DomainLayer :: makePayment(amount:Money)
-- make the Payment of the current Sale
exc: 1.1: the amount is negative or zero
...
post: 2.1 creates an instance of payment
...
post: 2.3 the system calls the *postSale* operation of the *Accounting System* with the current sale as a parameter





Remember that it is better to use the interface (IAccountingAdapter) instead of a specific object of a class (SAPAccountigAdapter) to avoid the implementation dependency.

Abstract Factory → singleton

- Overview

  Context

  - Systems that create, represent and compose a family of products that should be used together and that we do not want to reveal their implementations.

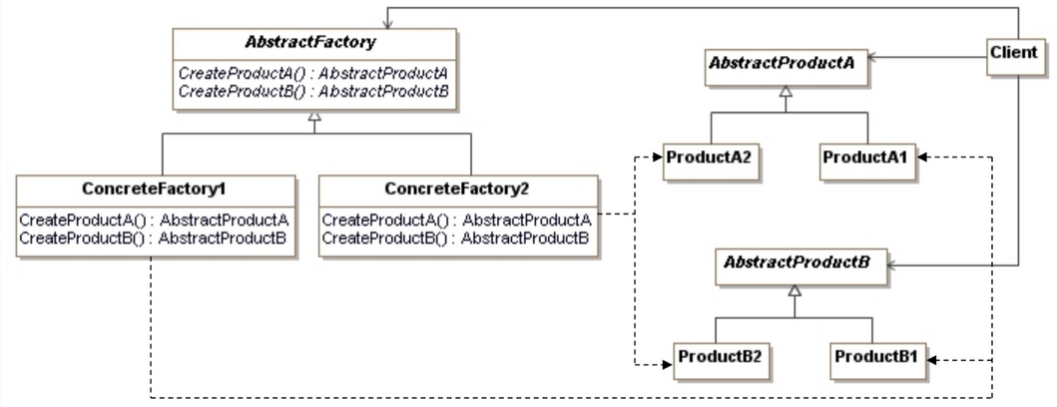
Problem

- Who should be responsible for creating objects when there are special considerations, such as a family of related or dependent objects?

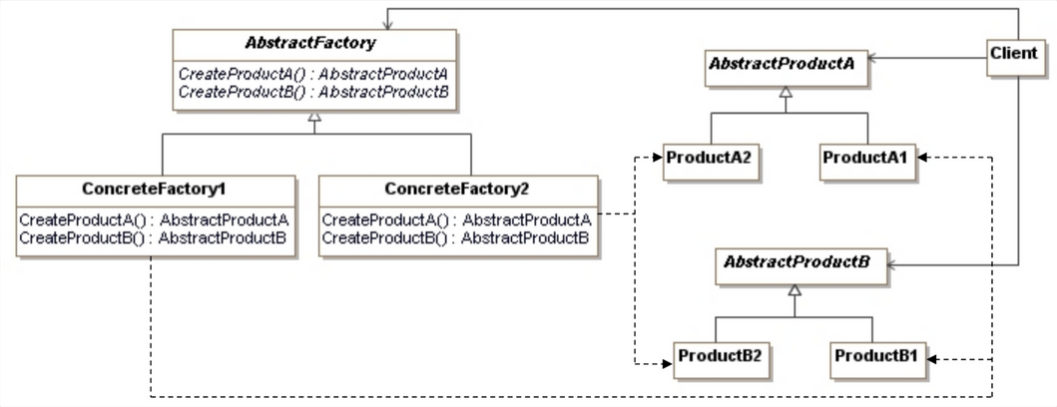
Solution

- Provide an interface for creating families of related or dependent objects without specifying their concrete classes.

- Static View



- Dynamic View



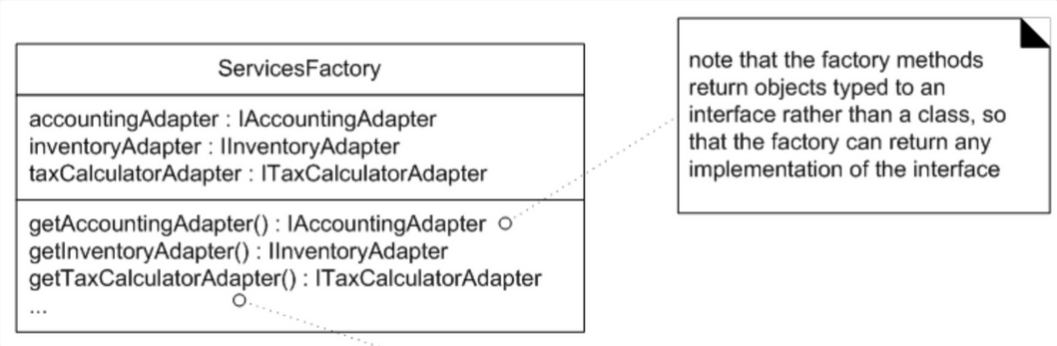
- Simple or Concrete Factory
  - Simple or Concrete Factory is not a GoF pattern (introduced by Gamma et al.), but extremely widespread.
  - It is a variation of Abstract Factory Pattern where an object called Factory is the responsible for creating objects with a complex creation logic or for a better cohesion.

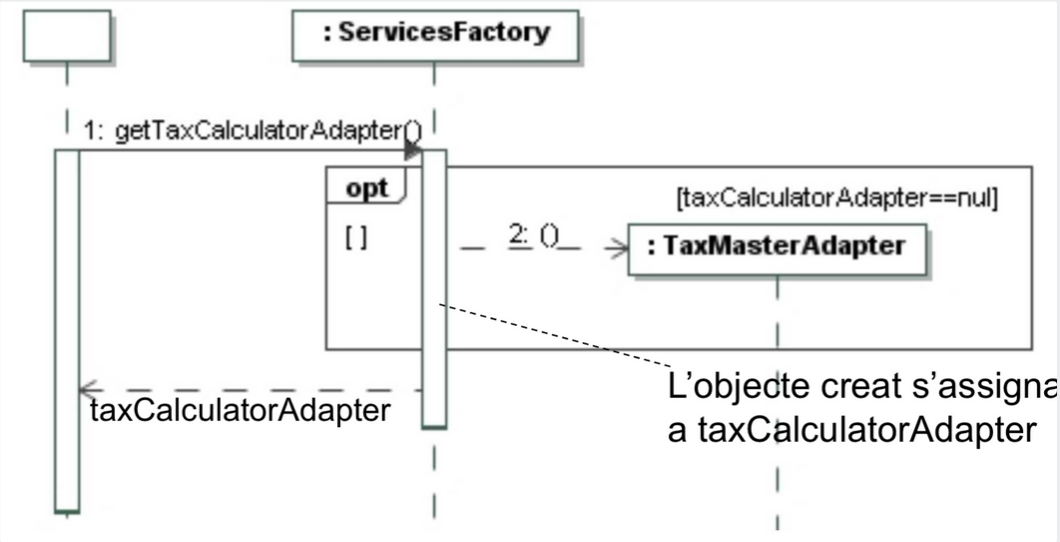
- Example

  In the NextGen POS system the use of adapters raises a new problem in the design.

  In the Adapter pattern solution for external services with varying interfaces, who creates the adapters?

  context DomainLayer :: makePayment(amount:Money)
  -- make the Payment of the current Sale
  exc: 1.1: the amount is negative or zero
  ...
  post: 2.1 creates an instance of payment
  ...
  post: 2.3 the system calls the *postSale* operation of the *Accounting System* with the current sale as a parameter





Singleton

- Overview

  Context

  - Systems that have classes with exactly one instance that must be accessible

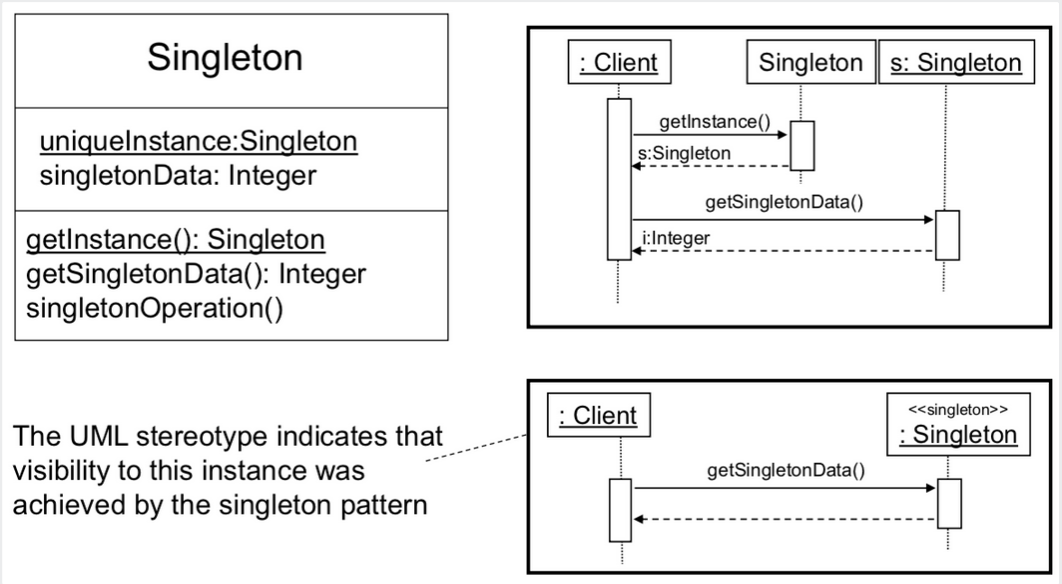
  Problem

- There must be exactly one instance of a class, and it must be accessible to clients from a well-known access point.
- A global variable makes an object accessible, but it doesn't keep you from instantiating multiple objects.

Solution

- Define a class operation of the class that returns the singleton.

- Static View and Dynamic View

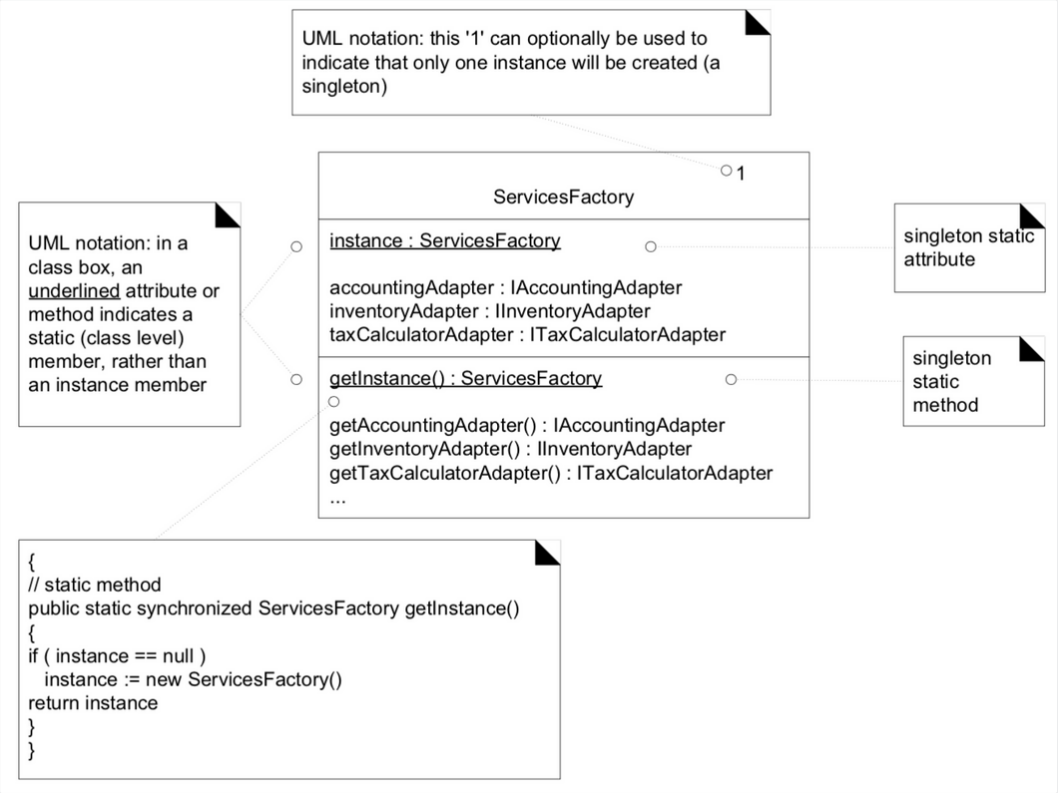


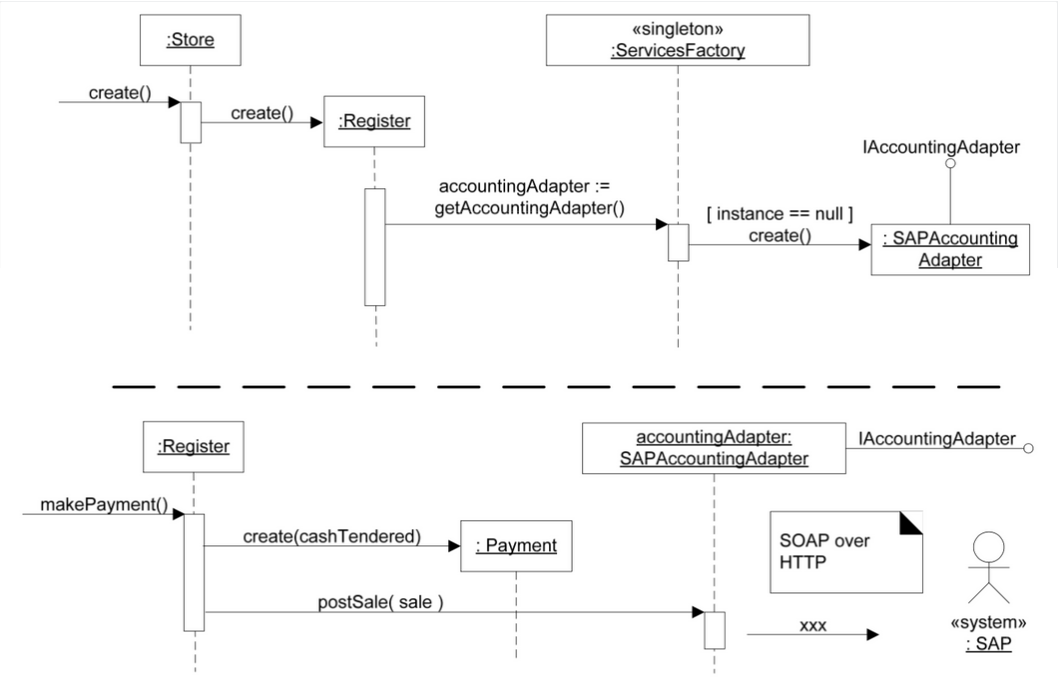
- Example

  In the NextGen POS system the use of service factory raises a new problem in the design.

  Who creates the factory itself, and how is it accessed?

context DomainLayer :: makePayment(amount:Money)
-- make the Payment of the current Sale
exc: 1.1: the amount is negative or zero
...
post: 2.1 creates an instance of payment
...
post: 2.3 the system calls the *postSale* operation of the *Accounting System* with the current sale as a parameter





Strategy

- Overview

  Context

  - Systems that have related classes differing only in their behavior

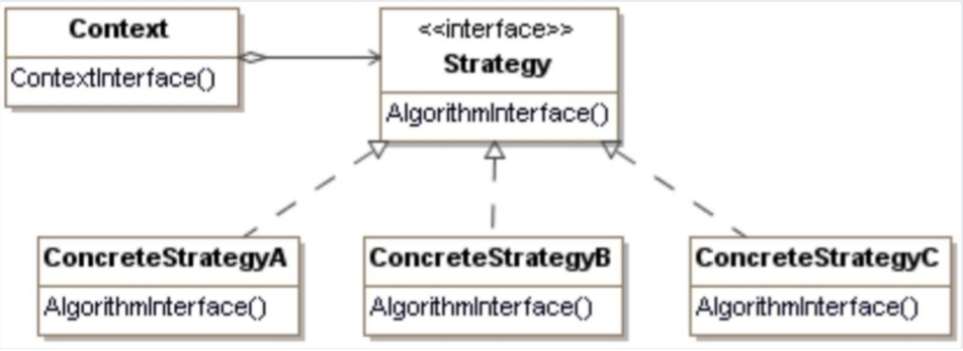
  Problem

  - How to design for varying, but related, algorithms or policies?
  - How to design for the ability to change these algorithms or policies?
  - Including these algorithms in the clients makes them bigger, harder to maintain and extend.

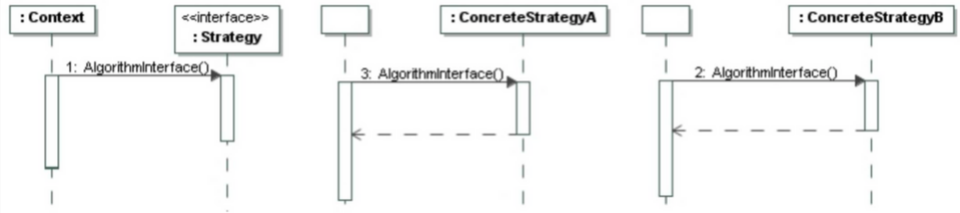
  Solution

  - Define classes that encapsulate different algorithms. An algorithm that is encapsulated in this way is called a strategy.

- Static View



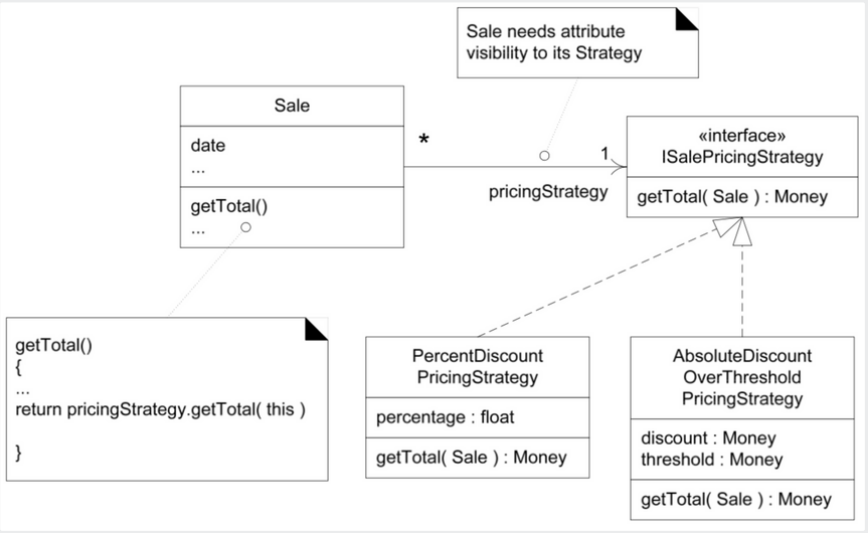
- Dynamic View
  - The AlgorithmInterface method is different for each ConcreteStrategy class.
  - A similar sequence diagram for the ConcreteStrategyC

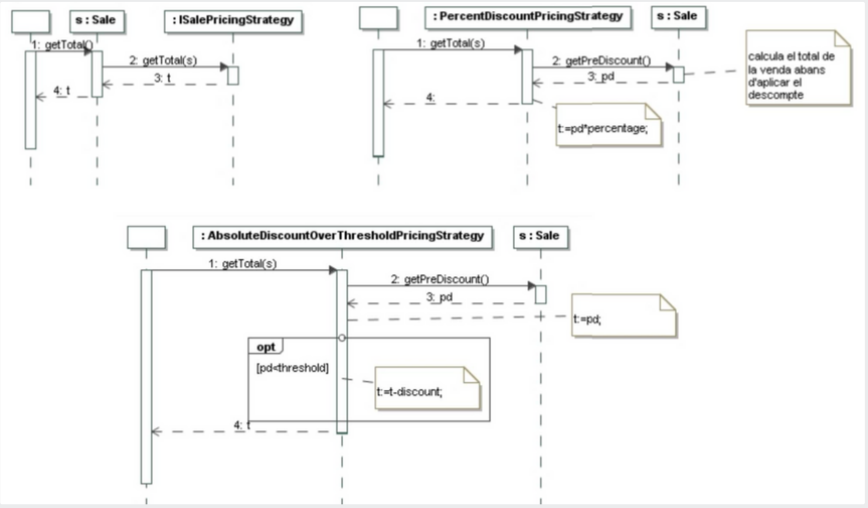


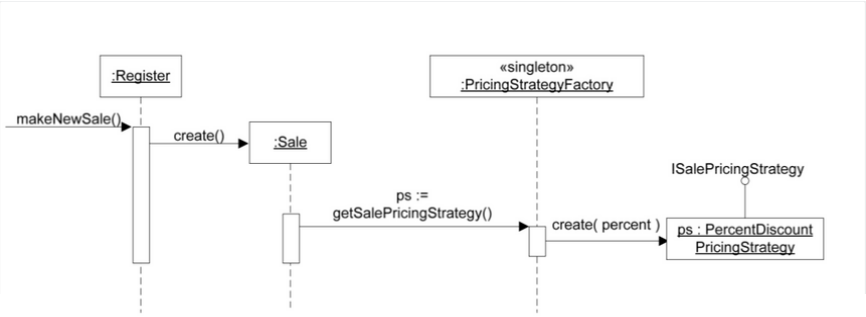
- Example

  In the NextGen POS system the pricing strategy for a sale can vary. During one period it may be 10% off all sales, later it may be 10 euros off if sale total is greater than 200 euros, and myriad other variations.

  How do we design for these varying pricing algorithms?







Template Method

- Overview

  Context

  - The definition of an operation in a hierarchy has some common behaviour to all subclasses but also some specific behaviour for each of them.

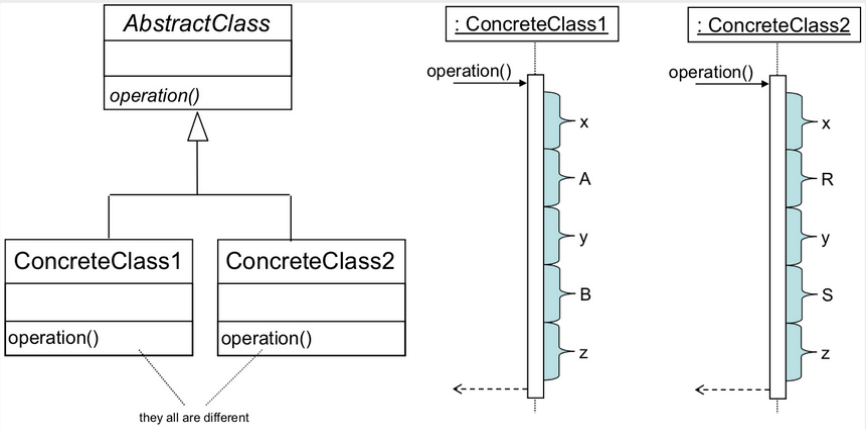
  Problem

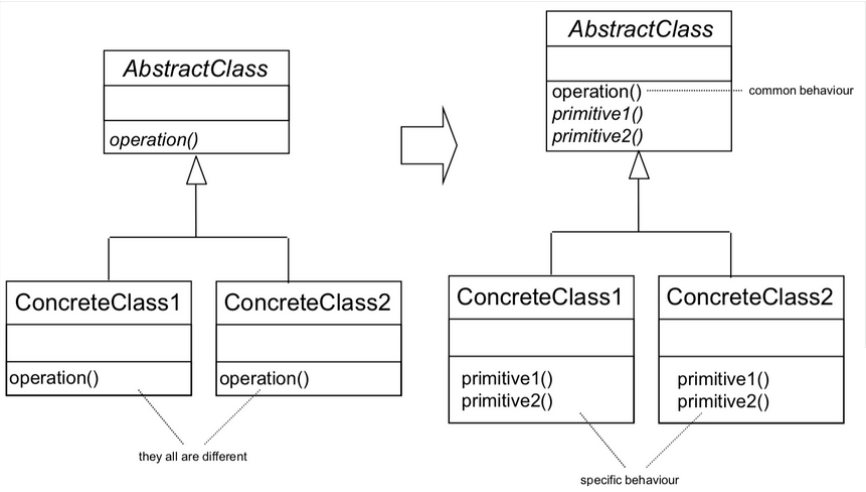
  - Replicating the common behaviour in all subclasses requires code duplication and therefore a more costly maintenance.

  Solution

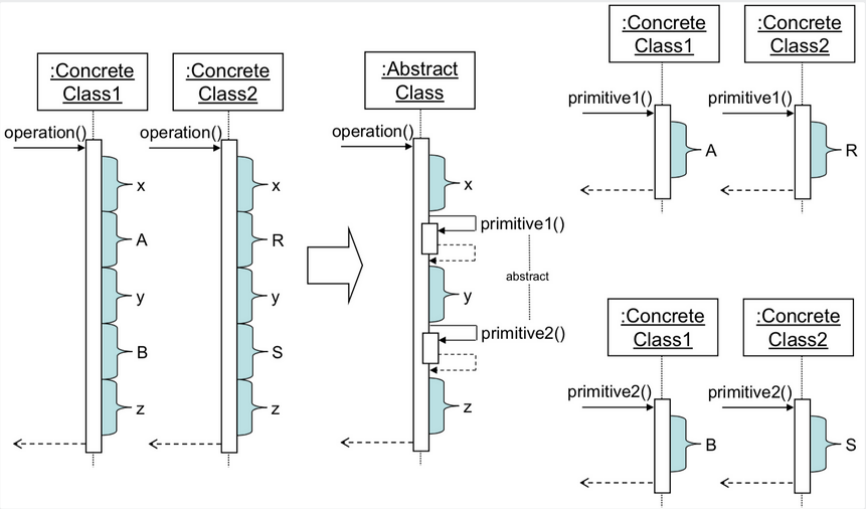
  - To define the algorithm (the operation)in the superclass, invoking abstract operations (with their signature defined in the superclass) that are implemented as methods in the subclasses.
    - The concrete operation is called template
    - The new operations are called primitives
  - The operation at the superclass defines the common behaviour whilst the abstract operations identify the specific behaviour, that is described in each subclass.

- Static View

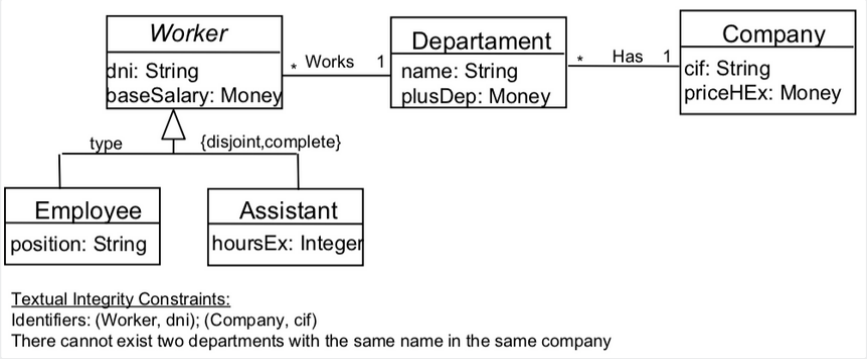




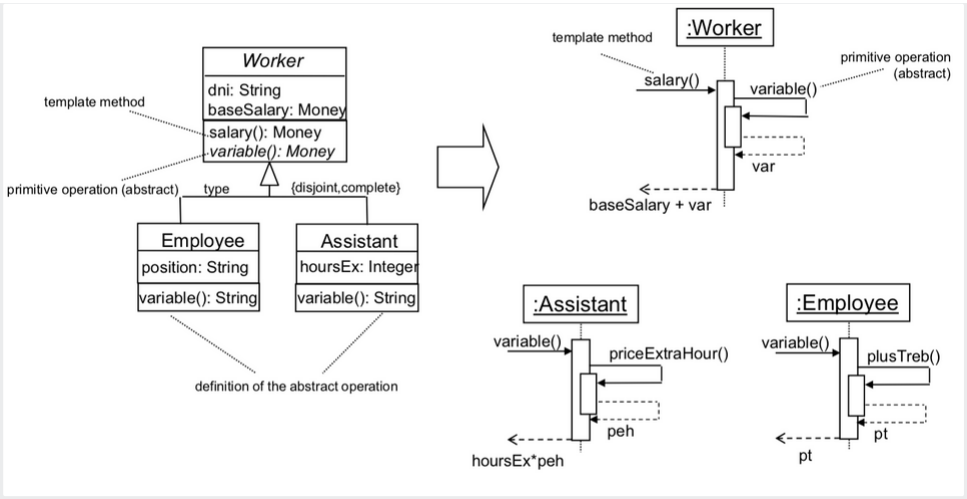
- Dynamic View

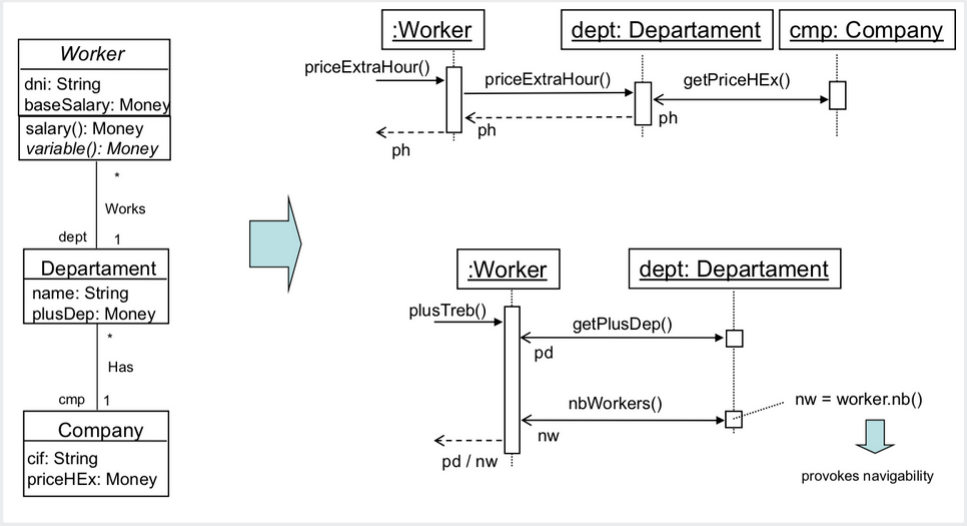


- Example
  - Specification data conceptual model:



  - We want to design an operation in class Worker to compute the salary that has to be paid to employees working in different companies:
    - salary of Assistant = base salary of a Worker + hoursEx * priceHEx
    - salary of Employee = base salary of a Worker + plusDep / number of Workers

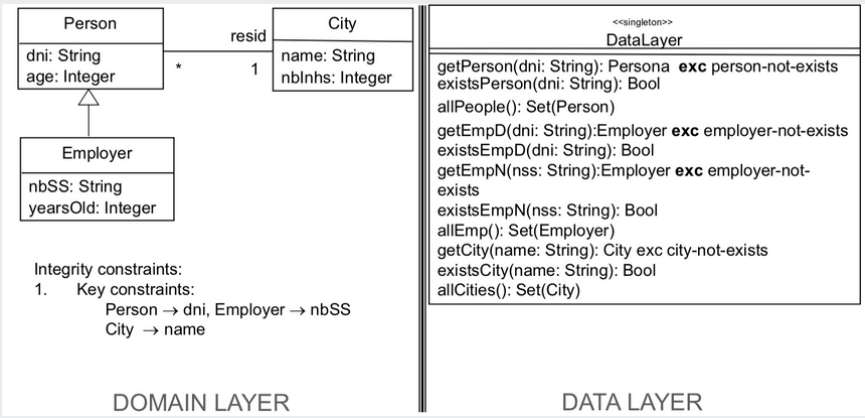


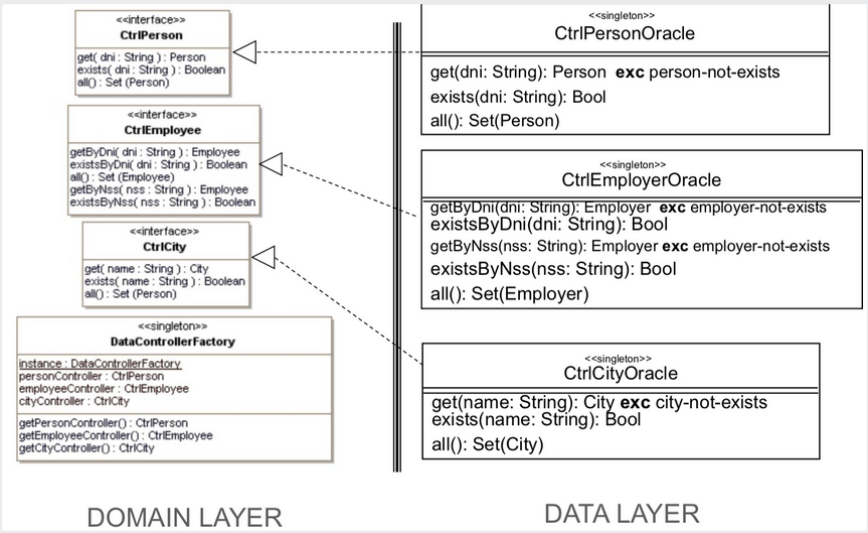


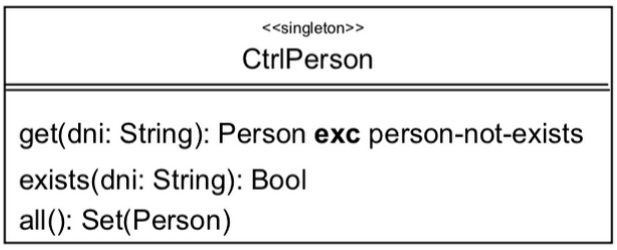
Access to the Data Layer

- Combining Patterns to Access to the Data Layer
  - Throughout domain layer design unit, we assume Domain Model just with operations of individual access and obtaining all the elements.
  - The Data Layer must be ready to change the implementation of the operations to access and (in case of Transaction Script, modify data)
  - For each persistent class of the Domain model that has one or more identifiers, an adapter should be defined.
  - A factory (singleton) to create all the adapters should be defined.

- Example







context CtrlPerson::get(dni: String): Person

exc: person-not-exists: there is no *Person* identified by *dni*

post: 2.1 returns person with *dni*

context CtrlPerson::exists(dni: String): Bool

post: 2.1 returns true if the person with *dni* exists, false otherwise

context CtrlPerson::all(): Set(Person)

post: 2.1 returns all the Persons existing in the system

- Sequence diagram to obtain a Person by its dni: