AS - Teoria 3

3.2.3. Designing with Services

1 factoria de servei, i per cada servei, un adaptador

Introducció

Patrons que ja sabem:

- Controlador
- Factoria
- Plantilla
- Adaptador
- Expert
- Singleton
- Estratègia

Coses a tenir en compte:

- Identificar
- Rassignar
- Pattern
- Availability and efficiency

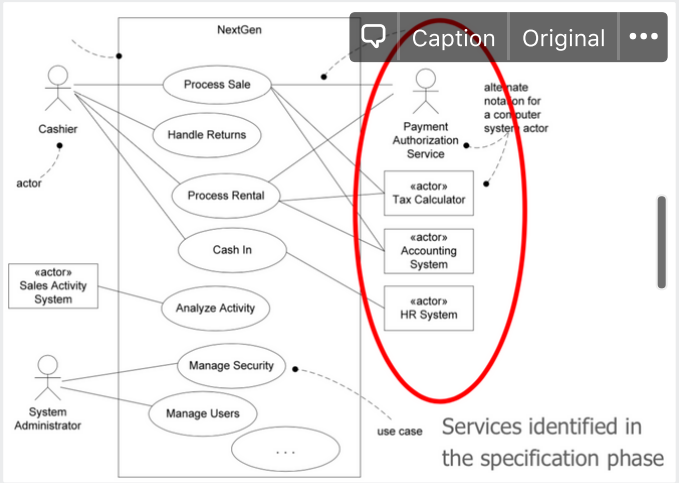
Service Identification

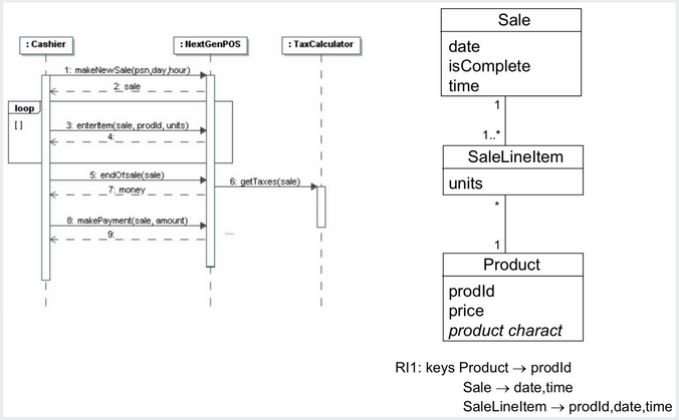
- It is usual to include previously designed components or services in the design of software systems.
- It makes faster and cheaper to design and build new software systems, since the reused components will not only be already designed but also tested.
- Some of these reused services are identified in the specification phase.
    - For instance, in the NextGen POS system, the third-party tax calculator and

inventory control services.

- Other services are identified in the design phase.
- The main reason to identify services in the design phase is to ensure the non-functional requirements (for instance, service availability, efficiency, etc...)
- The way in which our software system will interact with services will depend on the location of the services (local or remote).

Example: A simplified version of NextGen POS system.





context DomainLayer::enterItem(prodId: String, units: Integer)
pre: 1.1 product *prodId* exists
pre: 1.2 units-ok: units>0
post: 2.1 a SalesLineItem instance *sli* was created
2.2 *sli* was associated with the current sale
2.3 *sli.units* became *units*
2.4 *sli* was associated with a product with *prodId*

context DomainLayer::endOfSale(): Money
post: 2.1 the sale became complete
2.2 result= sale price + taxes (the system calls the *getTaxes* operation of the TaxCalculator service with the current sale as a paramete to obtain the taxes)

...

- We identify a service that provides the management of a product catalog. The use of this service may reduce the time and cost of developing our system.
- The operations offered by the service, among others, are the following:

context SvCatàleg::nouProducte(infoP: DTOproducte)
exc 1.1 producte-ja-existeix: existeix un producte amb identificador infoP.idProducte
post 2.1 crea un producte amb identificador infoP.idProducte, preu infoP.preu, i les característiques donades

context SvCatàleg::eliminaProducte(idP: String)
exc 1.1 producte-no-existeix: no existeix cap producte amb identificador idP
post 2.1 incrementa el preu de tots els productes del catàleg amb l'ipc

context SvCatàleg::obtéInfoProducte(idP: String, **out** infoP: DTOproducte)
exc producte-no-existeix: no existeix cap producte amb identificador idP
post 2.1 infoP conté la informació del producte amb identificador idP

DTOProduct
prodId price *product charact*

Reassignment of Responsibilities to Services

PER ACCEDIR A SERVEIS, NECESSITES UN **ADAPTADOR**

- After the identification of new services in the design phase, some responsabilities of the contracts have to be assigned to the services.
- Classes and associations managed by the services must be removed from the class diagram
  - Keeping the information (basically, keys) to obtain the data managed by the services.
  - Adding classes and operations to the class diagram (as a result of applying design patterns) to use and access services.

Example: Some responsabilities of the contracts have to be assigned to the services

context DomainLayer::enterItem(prodId: String, units: Integer)
pre: 1.1 product *prodId* exists
pre: 1.2 units-ok: units>0
post: 2.1 a SalesLineItem instance *sli* was created
2.2 *sli* was associated with the current sale
2.3 *sli.units* became *units*
2.4 *sli* was associated with a product with *prodId*

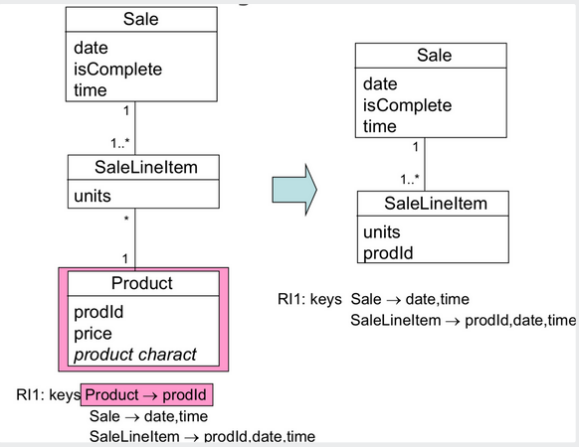
Consider the use of SvCatàleg

context DomainLayer::endOfSale(): Money
post: 2.1 the sale became complete
2.2 result= sale price + taxes

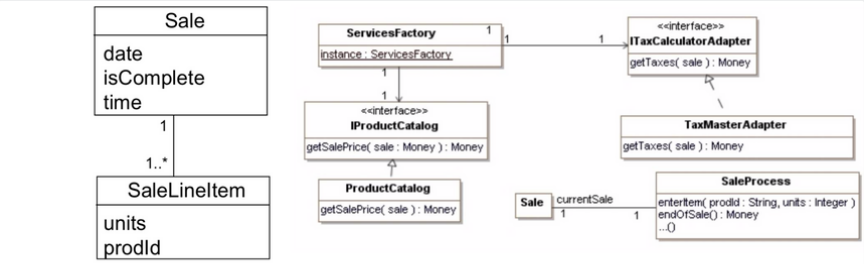
Responsibility assigned to TaxCalculator service (all the details in unit 5)

...

- Modification of diagram

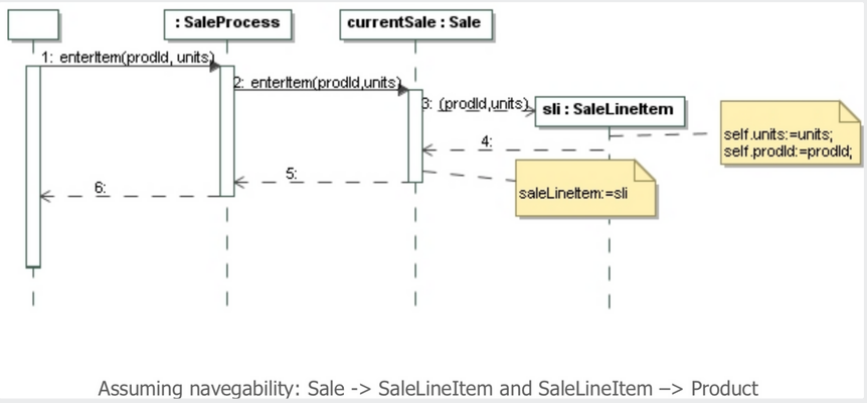


- Applying design patterns to use services (singleton, factory, adapter)

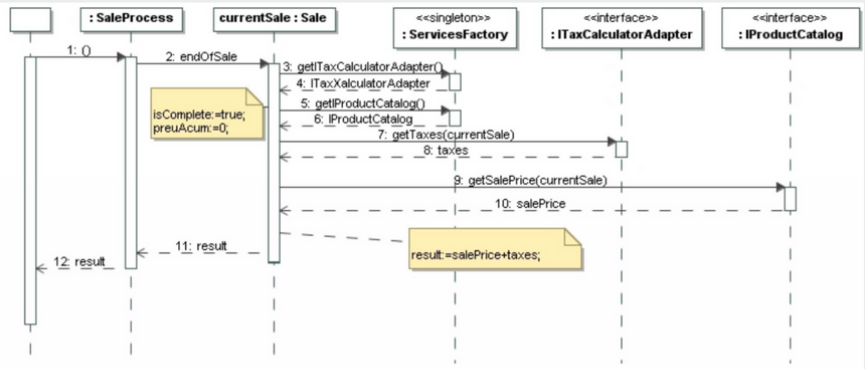


RI1: keys Sale → date,time
SaleLineItem → prodId,date,time

- Sequence diagrams



Assuming navegability: Sale -> SaleLineItem and SaleLineItem –> Product



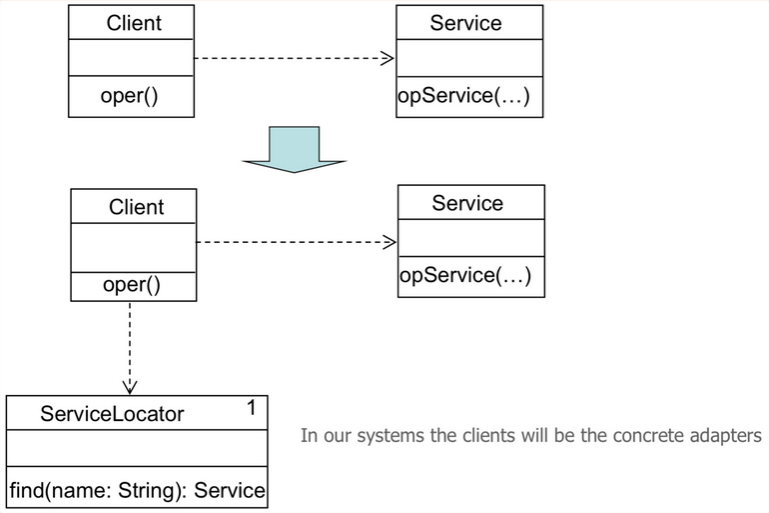
It remains to define the sequence diagrams for accessing services

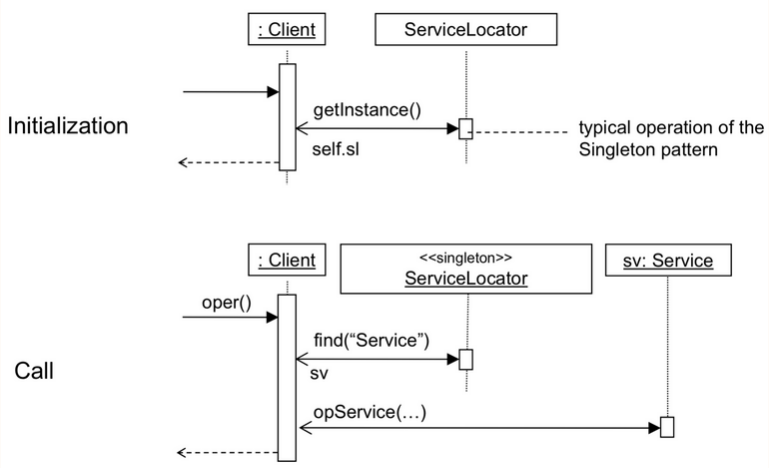
Patterns for Accessing Services

- To access services we must take into account:
  - How to localize services
  - Coupling between the classes of our system and services
  - How to communicate data between our system and services
- Some patterns can be applied to access services
  - Service Locator Pattern
  - Data Transfer Object Pattern

Service Locator (conectar)

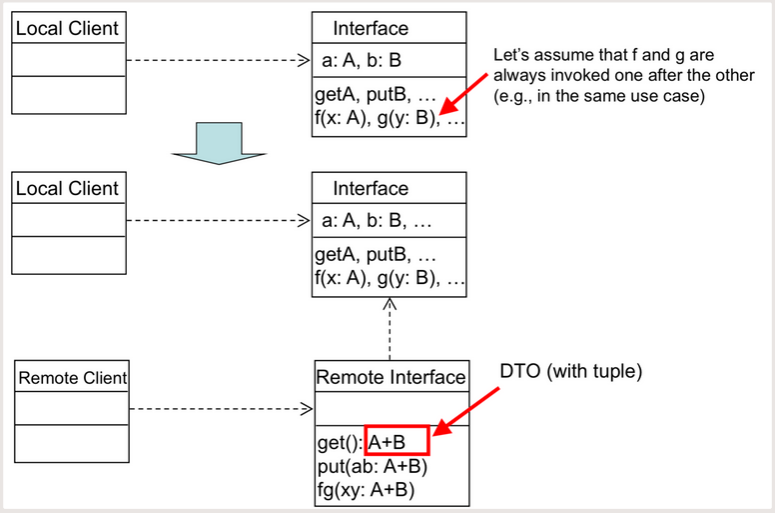
- Context
  - Systems that have to access remote services
- Problem
  - The service localization implies complex interfaces and network operations
  - The process can consume many resources
  - The clients always access the same way
- Solution
  - We define a singleton *class*, Service Locator, that centralizes the localization process
    - Being a singleton, it can be created when initiating the session or application
  - The client only communicates with this class, using the service's logic name
  - Service Locator implements strategies (e.g., caching) to optimize resources and avoid redundant distributed accesses
- Images

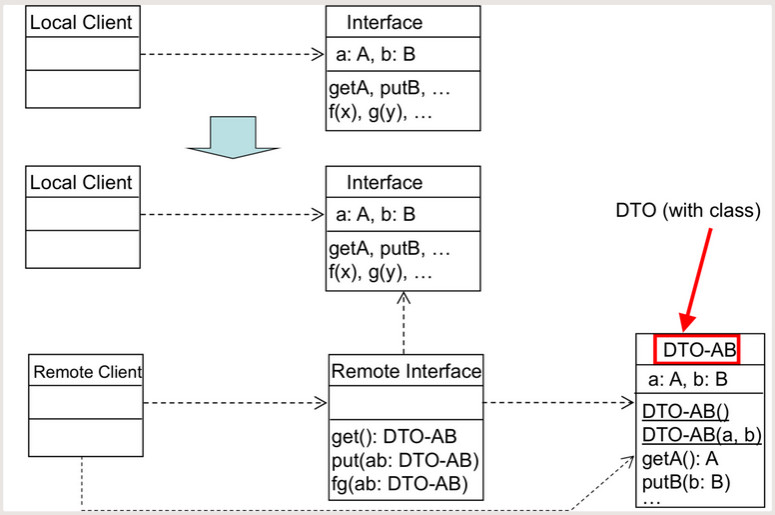




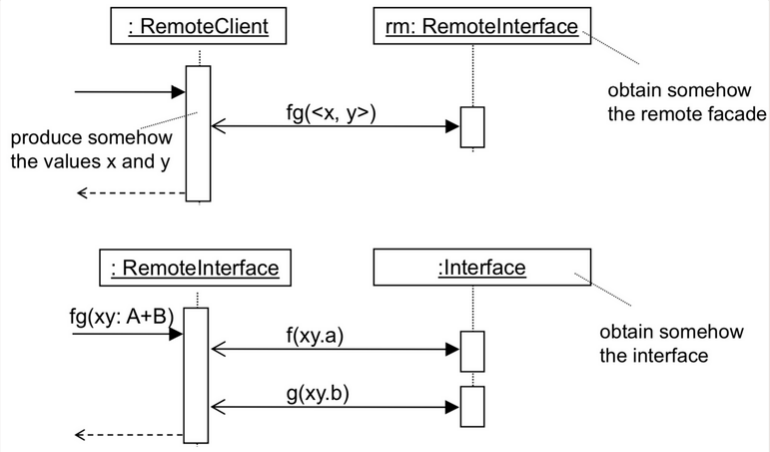
Data Transfer Object

- Context
  - Systems need to communicate data to services in an efficient manner
- Problem
  - To minimize the number of calls in the system, each call has to carry more information in the parameters and in the return value
  - The resulting list of data may be long
- Solution
  - We define (use) a new data group that contains all the data that has to be passed as parameter or result. The objects that are these kind of groups are called Data Transfer Objects (DTO)
  - This grouping can be defined as a class (DTO class) with getter and setter operations, or else as a tuple, if the underlying language supports this construction
- Images

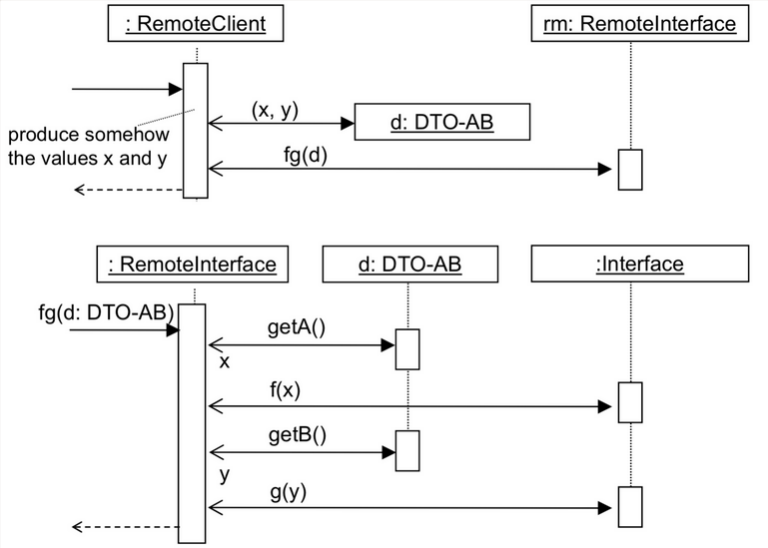




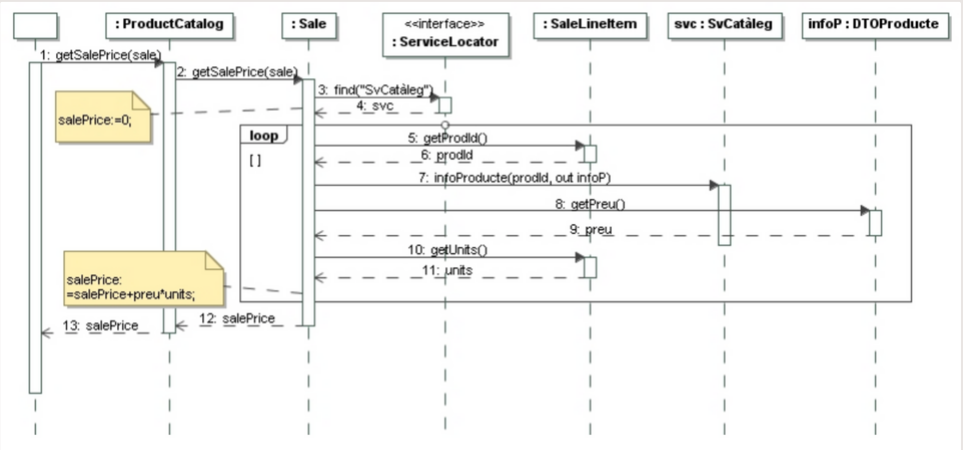
- DTO with Tuple



- DTO with Class

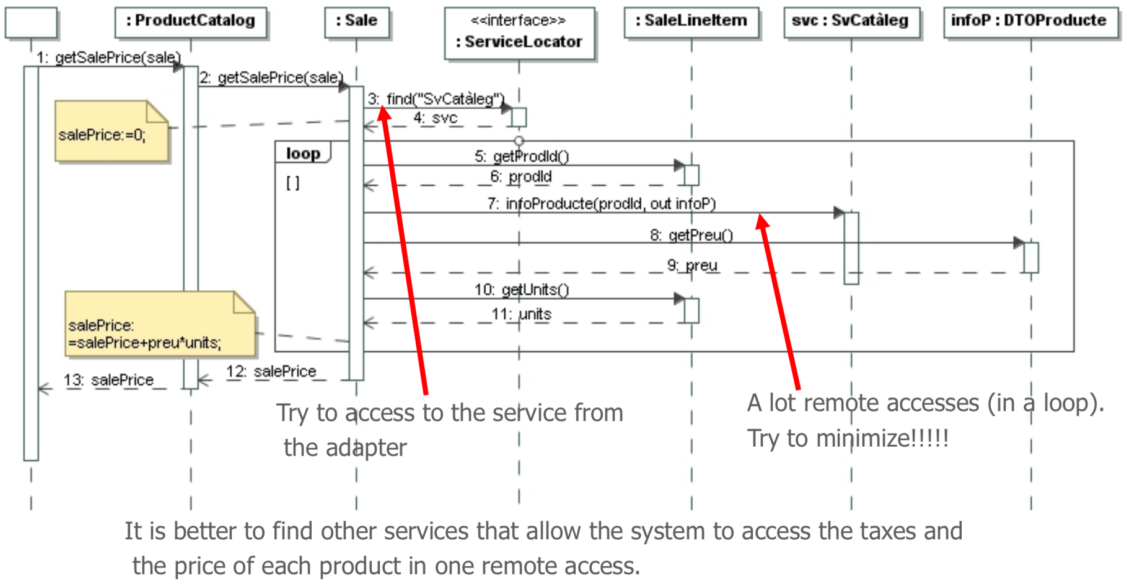


- **Example**



Ensuring Service Availability and Efficiency

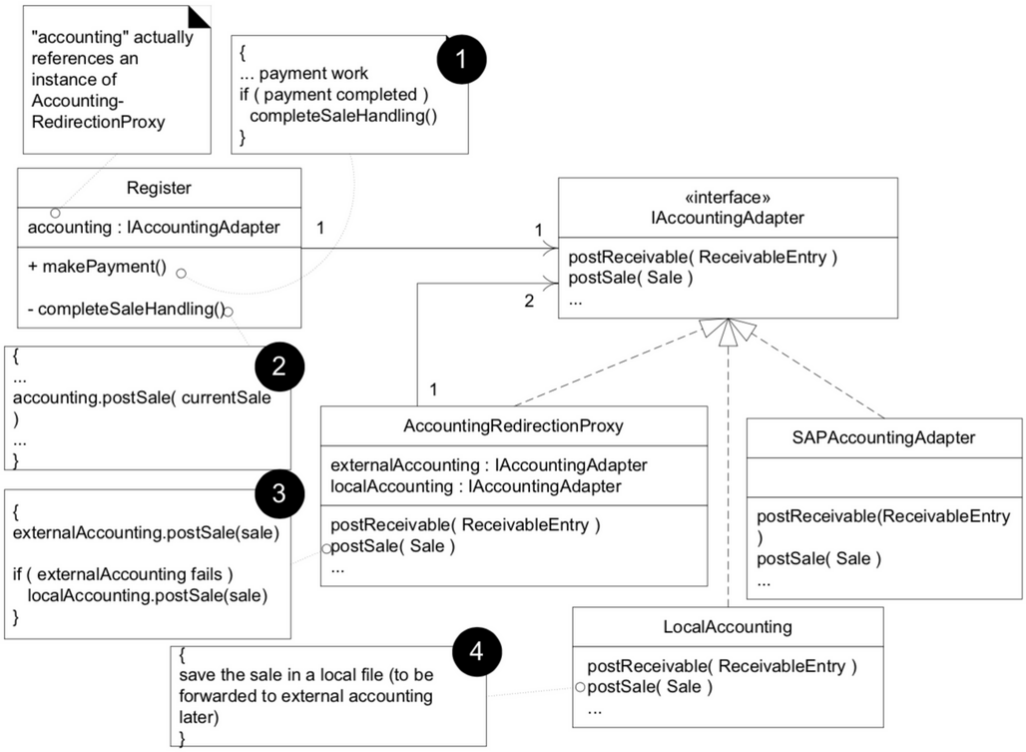
- Design patterns and principles may be applied to satisfy non-functional requirements
- Some of the design patterns studied until now help to ensure some non-functional requirements. For instance, adapter and service locator patterns provide maintenance.
- Now, we focus on efficiency and service availability non-functional requirements
- Efficiency may be achieved in different ways.
    - Applying the adapter and factory pattern to design a cache mechanism to avoid multiple accesses to a remote service.
    - We ensure service efficiency minimizing the remote accesses to services in the following way:
        - We use services that minimize remote accesses and discard the ones that generate multiple remote accesses
        - We use services that allows us to pass the data in a grouped manner (use of the DTO pattern)
- Review of the getSalePrice operation of ProductCatalog



Ensuring Service Availability

The systems that we design require a high service availability.

- There is two options to ensure service availability:
  - Using services with a service level agreement that impose a high availability of the service.
  - Design a solution that ensures a high availability of the service.
- There are lots of ways to design solutions with high service availabity. For instance, one option could be ensuring service availability using the proxy pattern in the following way:
  - The proxy registers the service and its backup.
  - If the service fails, the proxy redirects the request to the backup service.
- Example:



- Service Locator
- [DTO]