Patterns for Domain Layer

Patterns for Domain Layer

- Adapter
- Abstract Factory
- Singleton
- Strategy
- Template Method
- Access to the Data Layer
- References

Adapter Pattern

Adapter Pattern

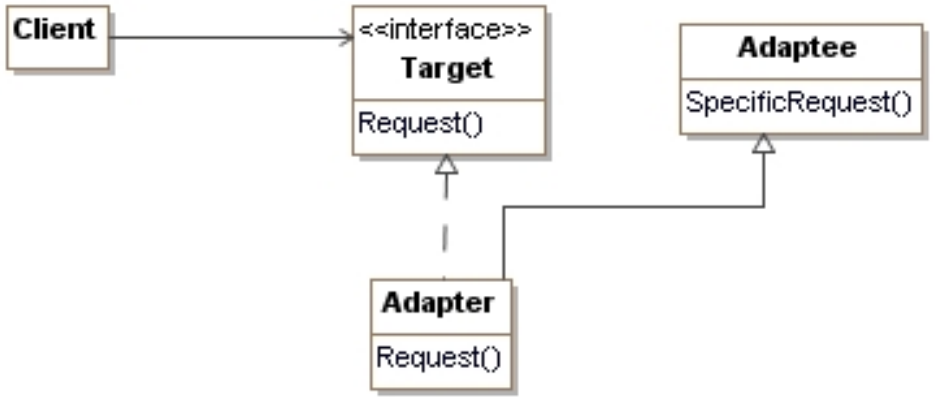
- Overview
- Static View
- Dynamic View
- Example
- References

Overview

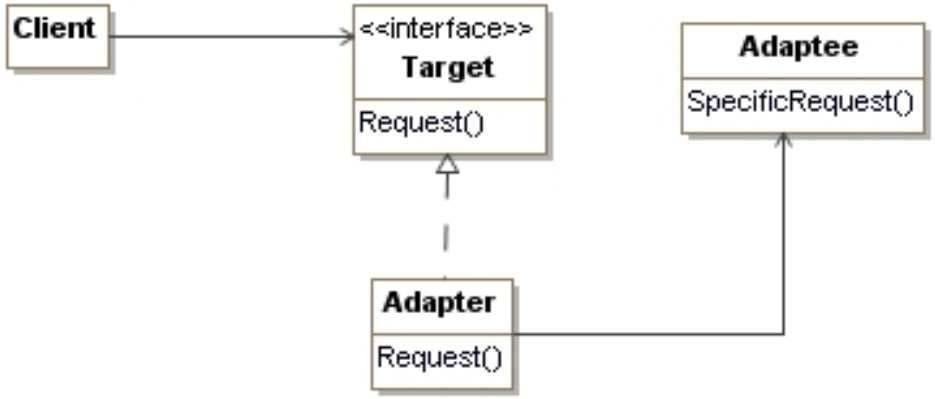
- Context
  - The interface of an existing class (or a set of subclasses) does not match the one needed
- Problem
  - How to resolve incompatible interfaces, or provide a stable interface to similar components with different interfaces?
- Solution
  - Convert the interface of a class (Adaptee class) 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

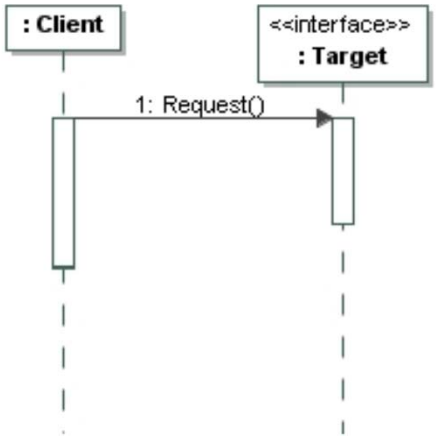


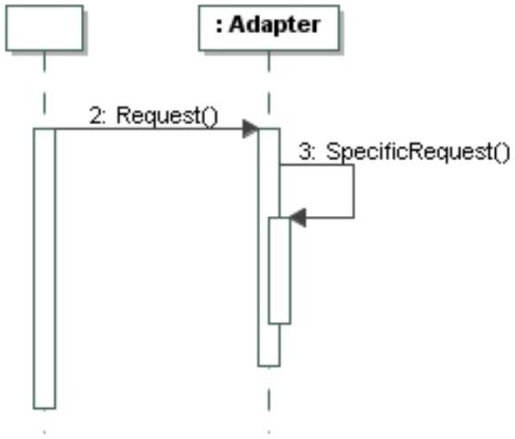
- Object adapter (useful when one adapter adapts more than one adaptee)



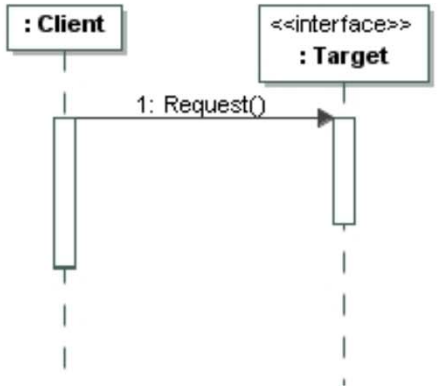
Dynamic View

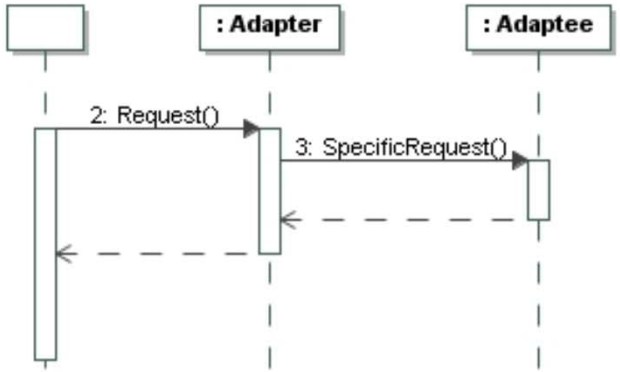
- Class adapter





- Object adapter

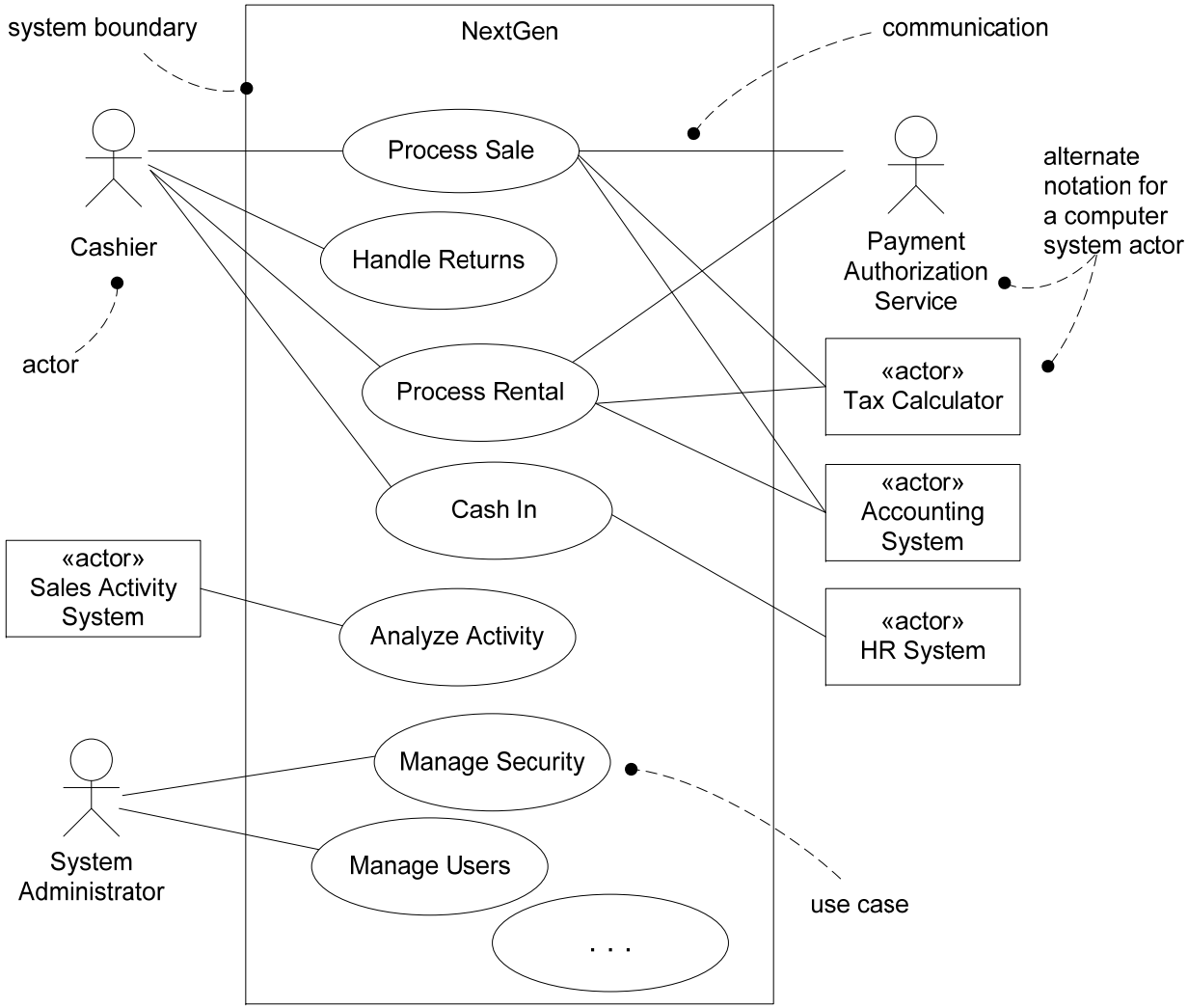




Example

- 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.

Example


system boundary
NextGen
communication
Process Sale
Handle Returns
Process Rental
Cash In
Analyze Activity
Manage Security
Manage Users
. . .
Cashier
actor
Payment Authorization Service
alternate notation for a computer system actor
«actor» Tax Calculator
«actor» Accounting System
«actor» HR System
«actor» Sales Activity System
System Administrator
use case

Example

- *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

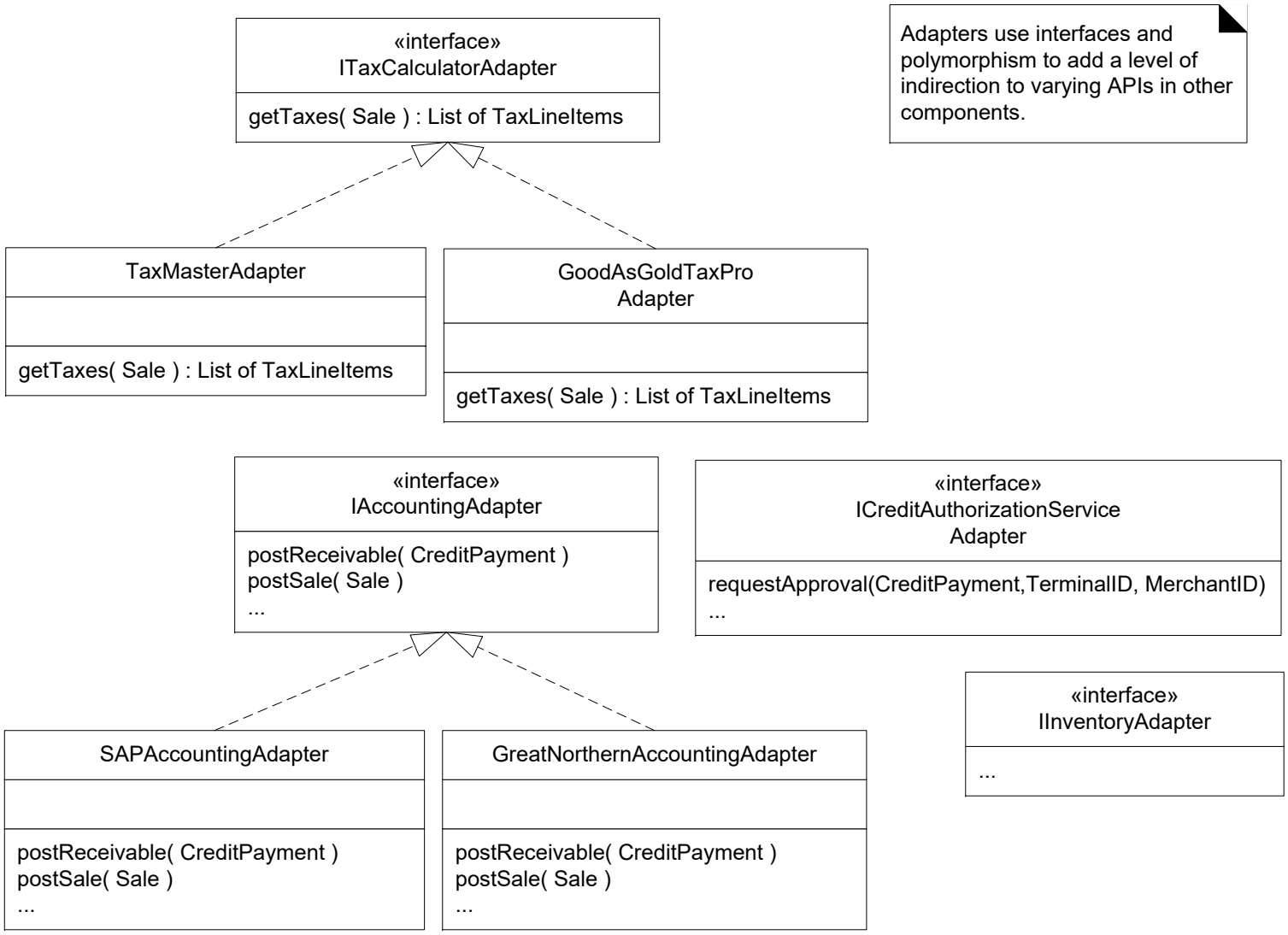
…

post: 2.1 creates an instance of payment

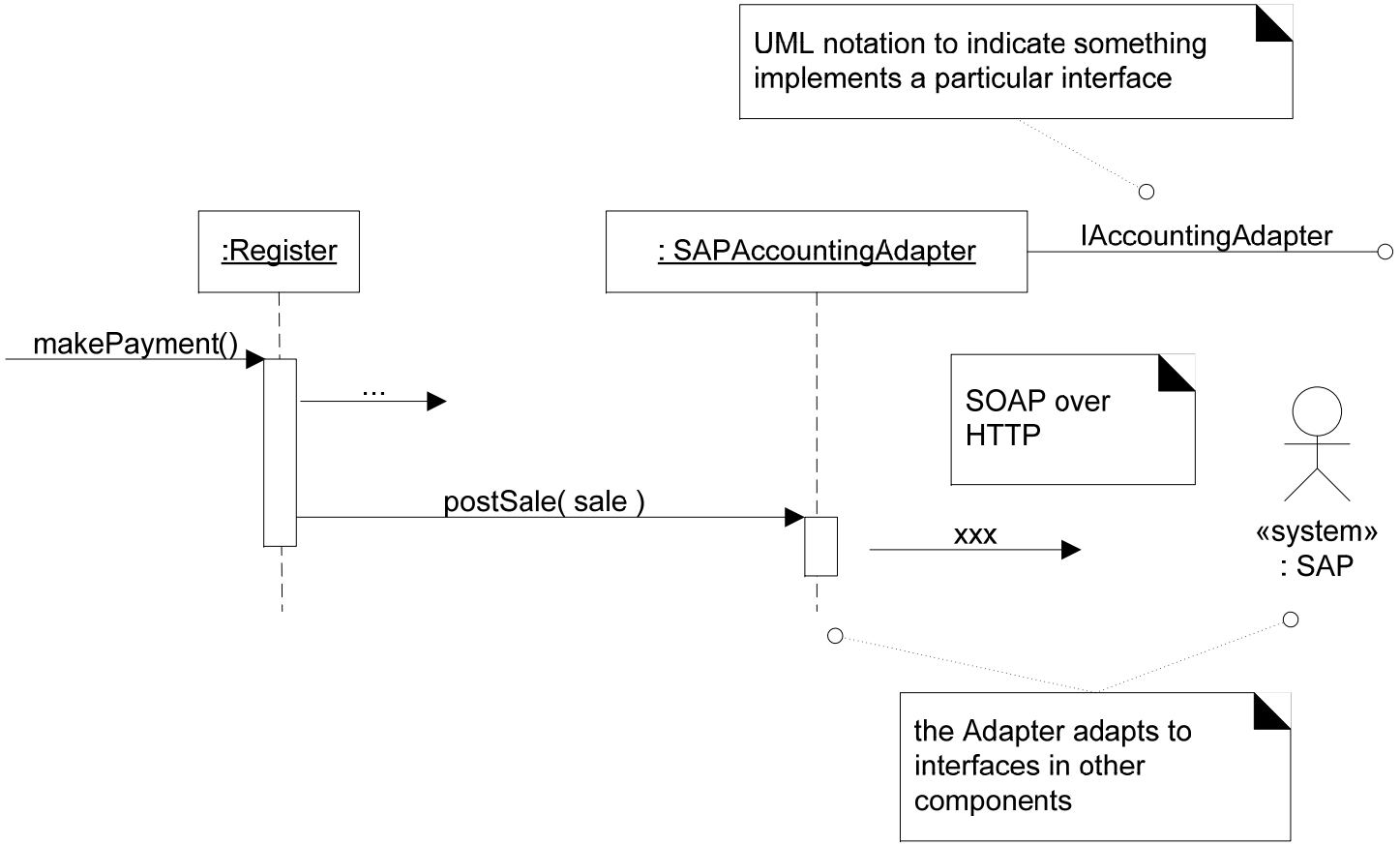
…

post: 2.3 the system calls the *postSale* operation of the *Accounting System* with the current sale as a parameter

Example


«interface»
ITaxCalculatorAdapter
getTaxes(Sale) : List of TaxLineItems
Adapters use interfaces and polymorphism to add a level of indirection to varying APIs in other components.
TaxMasterAdapter
getTaxes(Sale) : List of TaxLineItems
GoodAsGoldTaxPro Adapter
getTaxes(Sale) : List of TaxLineItems
«interface»
IAccountingAdapter
postReceivable(CreditPayment)
postSale(Sale)
...
«interface»
ICreditAuthorizationService Adapter
requestApproval(CreditPayment,TerminalID, MerchantID)
...
«interface»
IInventoryAdapter
...
SAPAccountingAdapter
postReceivable(CreditPayment)
postSale(Sale)
...
GreatNorthernAccountingAdapter
postReceivable(CreditPayment)
postSale(Sale)
...

Example



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

References

- *Design Patterns: Elements of Reusable Object-Oriented Software*
  E. Gamma; R. Helm; R. Johnson; J. Vlissides
  Addison-Wesley, 1995, pp. 139-150.

- *Applying UML and Patterns*
  C. Larman
  Prentice Hall, 2005 (Third edition), ch. 26

Abstract Factory Pattern

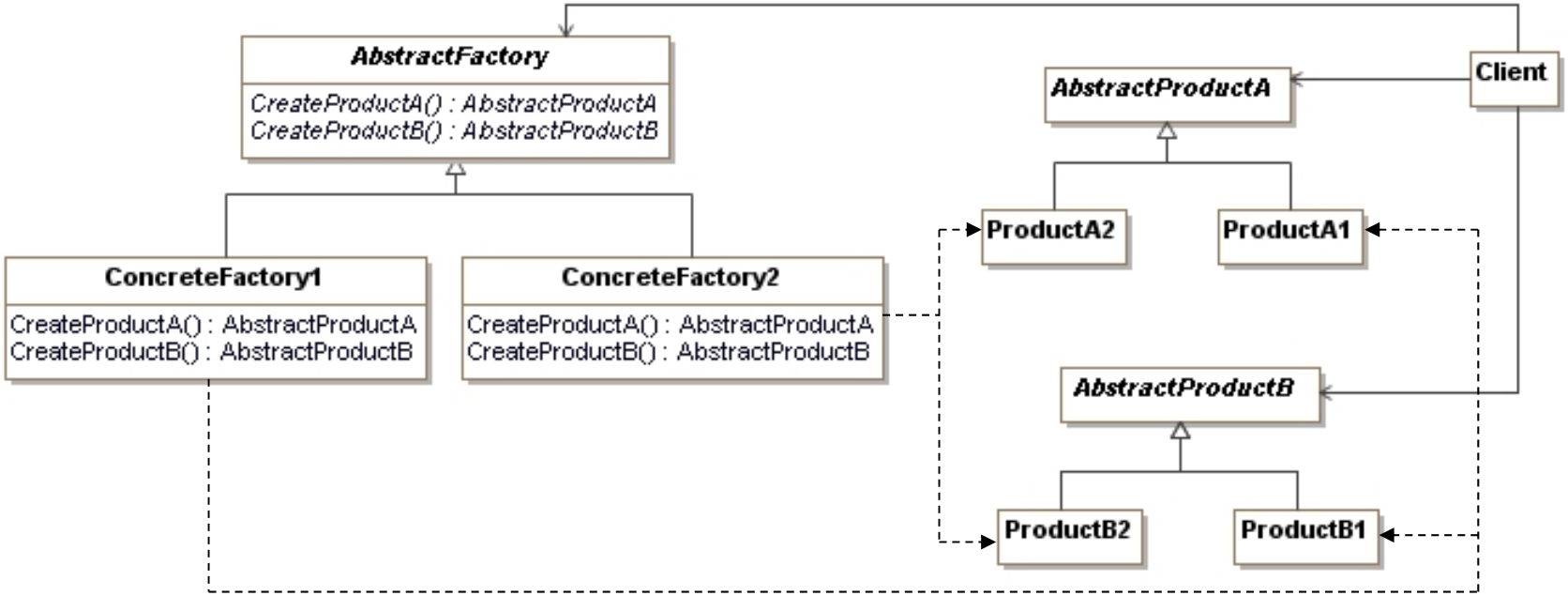
Abstract Factory Pattern

- Overview
- Static View
- Dynamic View
- Simple or Concrete Factory
- Example
- References

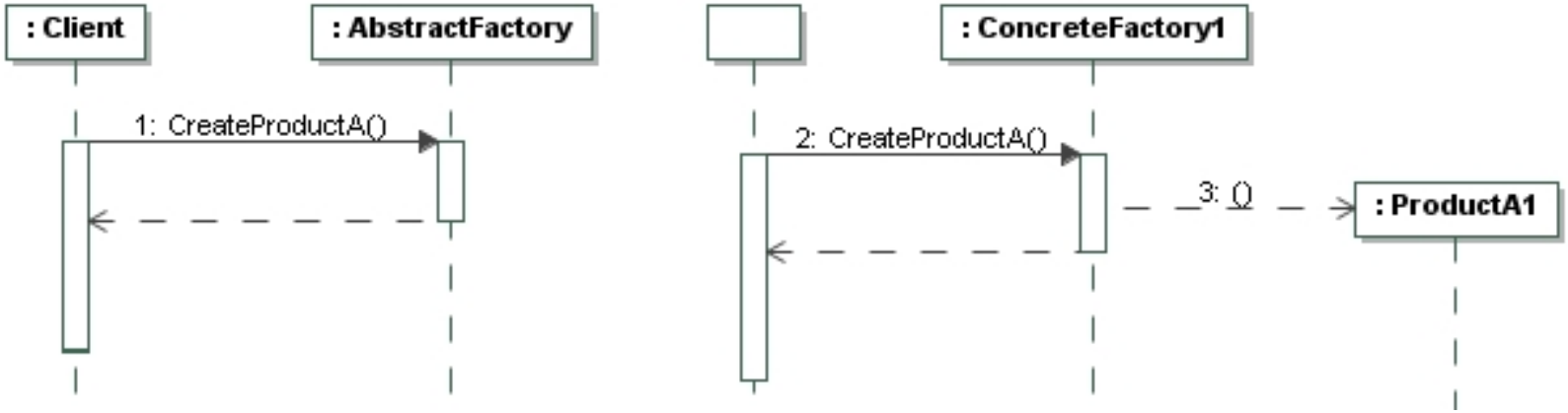
Overview

- 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


AbstractFactory
CreateProductA() : AbstractProductA
CreateProductB() : AbstractProductB
ConcreteFactory1
CreateProductA() : AbstractProductA
CreateProductB() : AbstractProductB
ConcreteFactory2
CreateProductA() : AbstractProductA
CreateProductB() : AbstractProductB
AbstractProductA
ProductA2
ProductA1
AbstractProductB
ProductB2
ProductB1
Client

Dynamic View



The sequence diagrams of the other operations are similar

Simple or Concrete Factory

- Simple or Concrete Factory is not a GoF pattern (introduced by Gamma et al.), but extremely widespread.

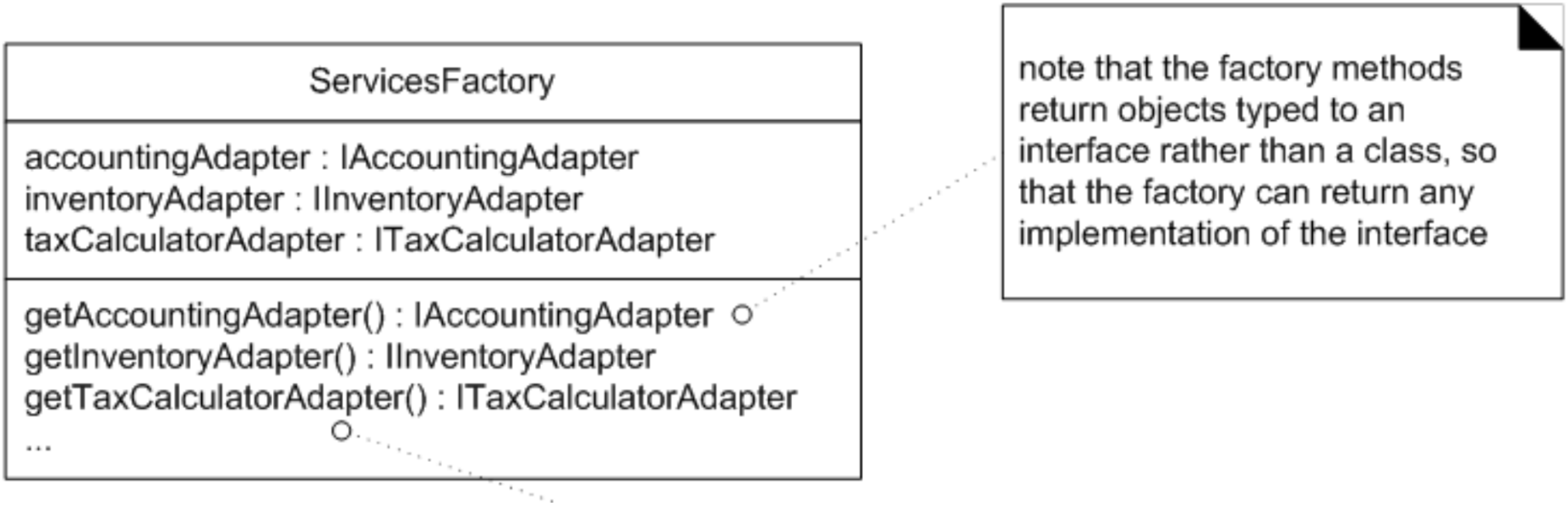
- It is a variation of Abstract Factory Pattern where an object called Factory is the responsible for creating objects with a complex creation logic or for a better cohesion.

Example

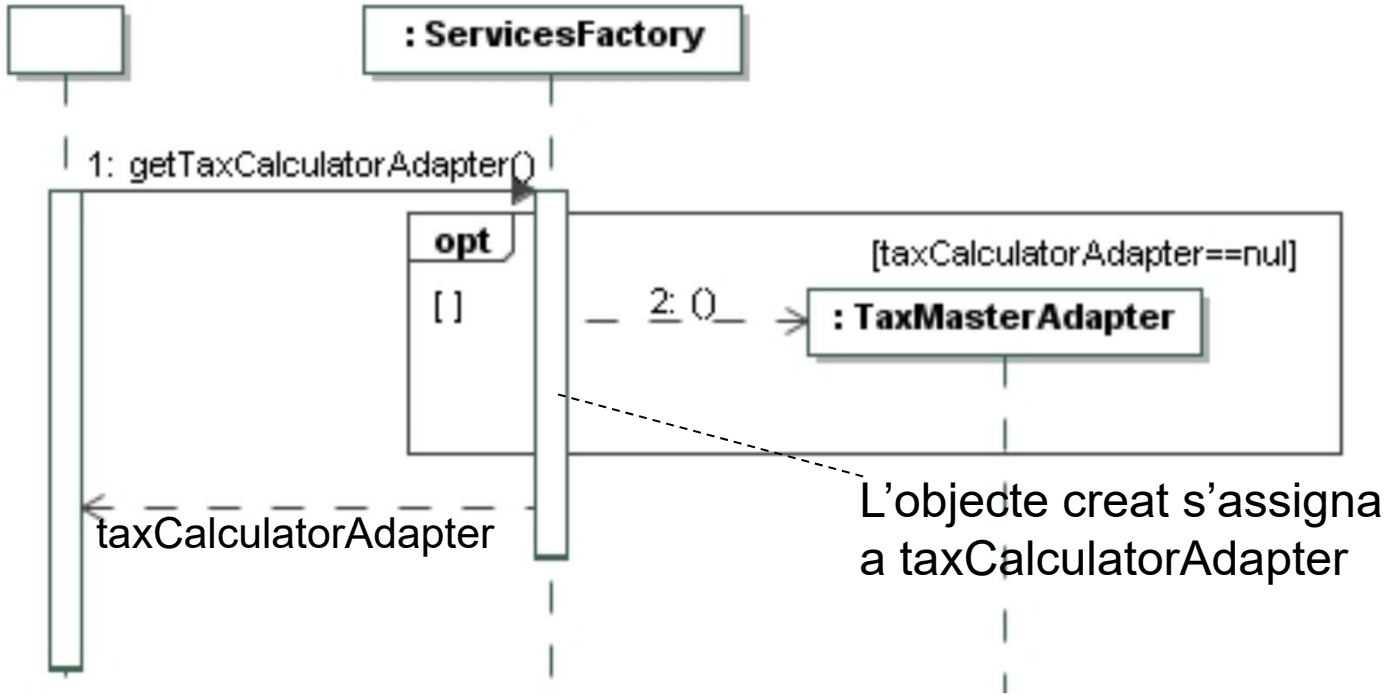
- In the NextGen POS system the use of adapters raises a new problem in the design.
- In the Adapter pattern solution for external services with varying interfaces, who creates the adapters?

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

Example


ServicesFactory
accountingAdapter : IAccountingAdapter
inventoryAdapter : IInventoryAdapter
taxCalculatorAdapter : ITaxCalculatorAdapter
getAccountingAdapter() : IAccountingAdapter
getInventoryAdapter() : IInventoryAdapter
getTaxCalculatorAdapter() : ITaxCalculatorAdapter
...
note that the factory methods return objects typed to an interface rather than a class, so that the factory can return any implementation of the interface

Example


: ServicesFactory
1: getTaxCalculatorAdapter()
opt
[ ]
[taxCalculatorAdapter==nul]
2: ()
: TaxMasterAdapter
taxCalculatorAdapter
L'objecte creat s'assigna
a taxCalculatorAdapter

References

- *Design Patterns: Elements of Reusable Object-Oriented Software*
  E. Gamma; R. Helm; R. Johnson; J. Vlissides
  Addison-Wesley, 1995, pp. 87-96.

- *Applying UML and Patterns*
  C. Larman
  Prentice Hall, 2005 (Third edition), ch. 26

Singleton Pattern

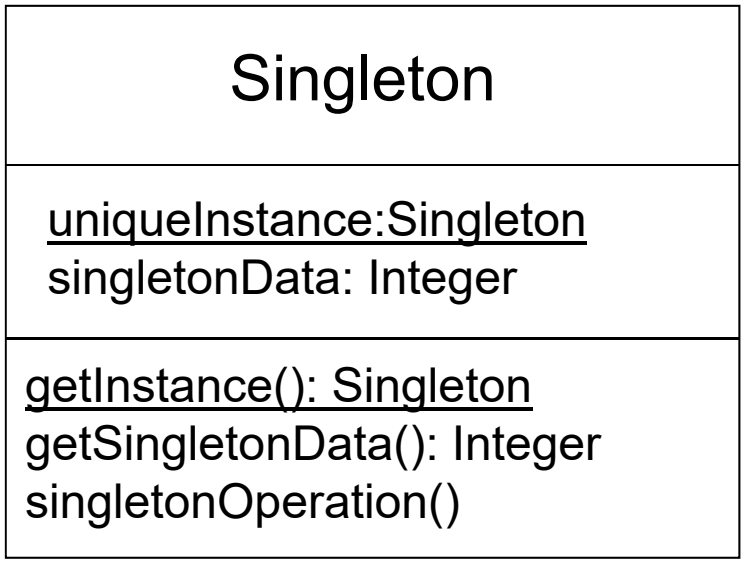
Singleton Pattern

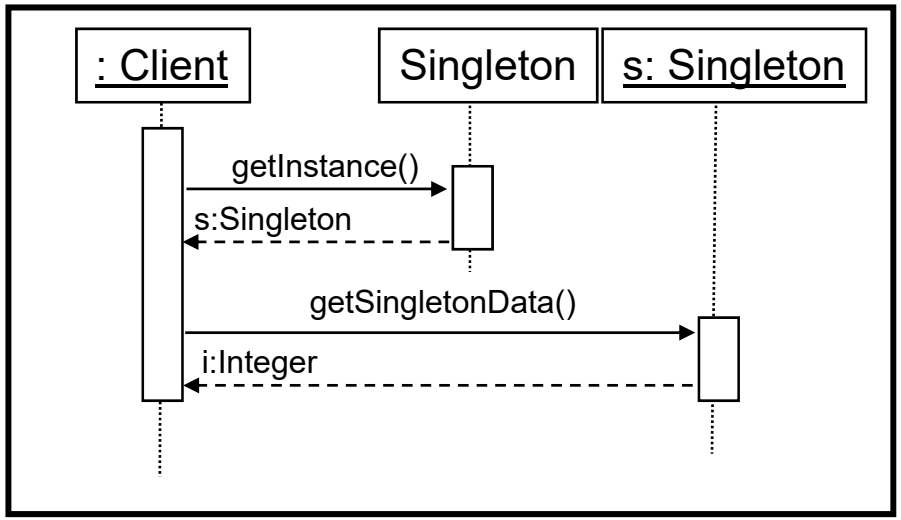
- Overview
- Static View
- Dynamic View
- Example
- References

Overview

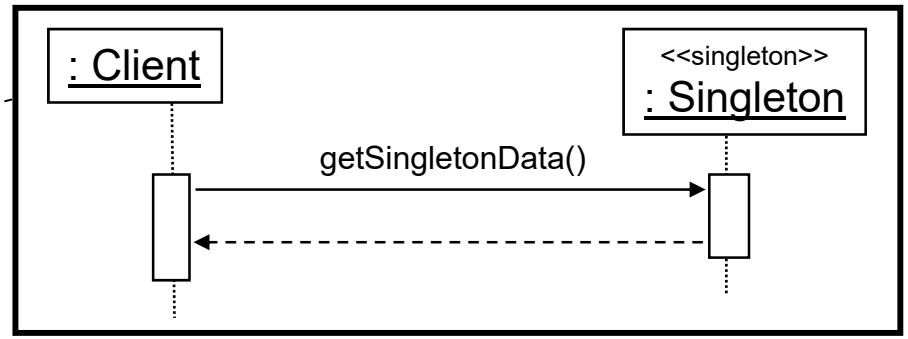
- Context
  - Systems that have classes with exactly one instance that must be accessible
- Problem
  - There must be exactly one instance of a class, and it must be accessible to clients from a well-known access point.
  - A global variable makes an object accessible, but it doesn't keep you from instantiating multiple objects.
- Solution
  - Define a class operation of the class that returns the singleton.

Static and Dynamic View





The UML stereotype indicates that visibility to this instance was achieved by the singleton pattern



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

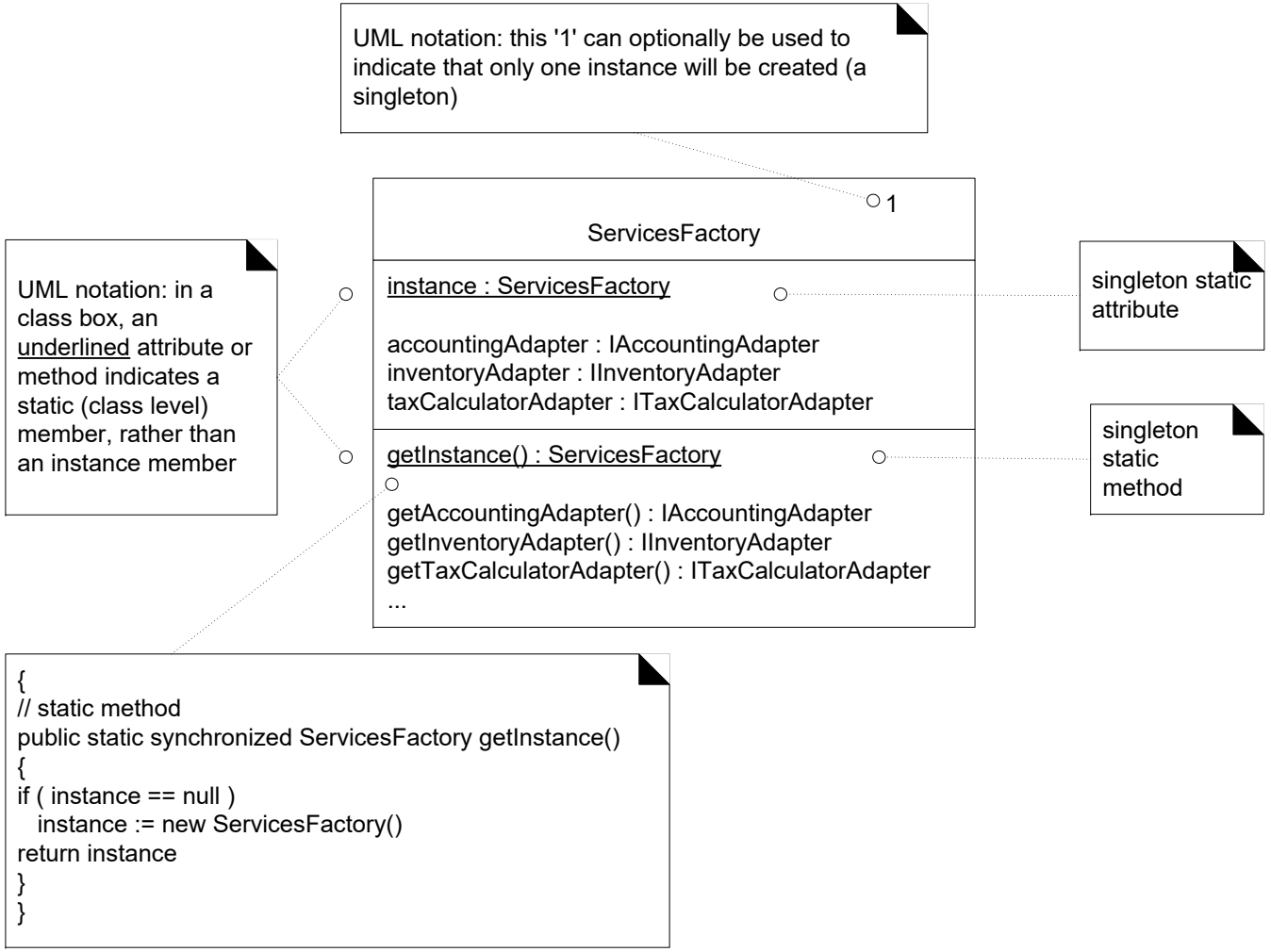
...

post: 2.1 creates an instance of payment

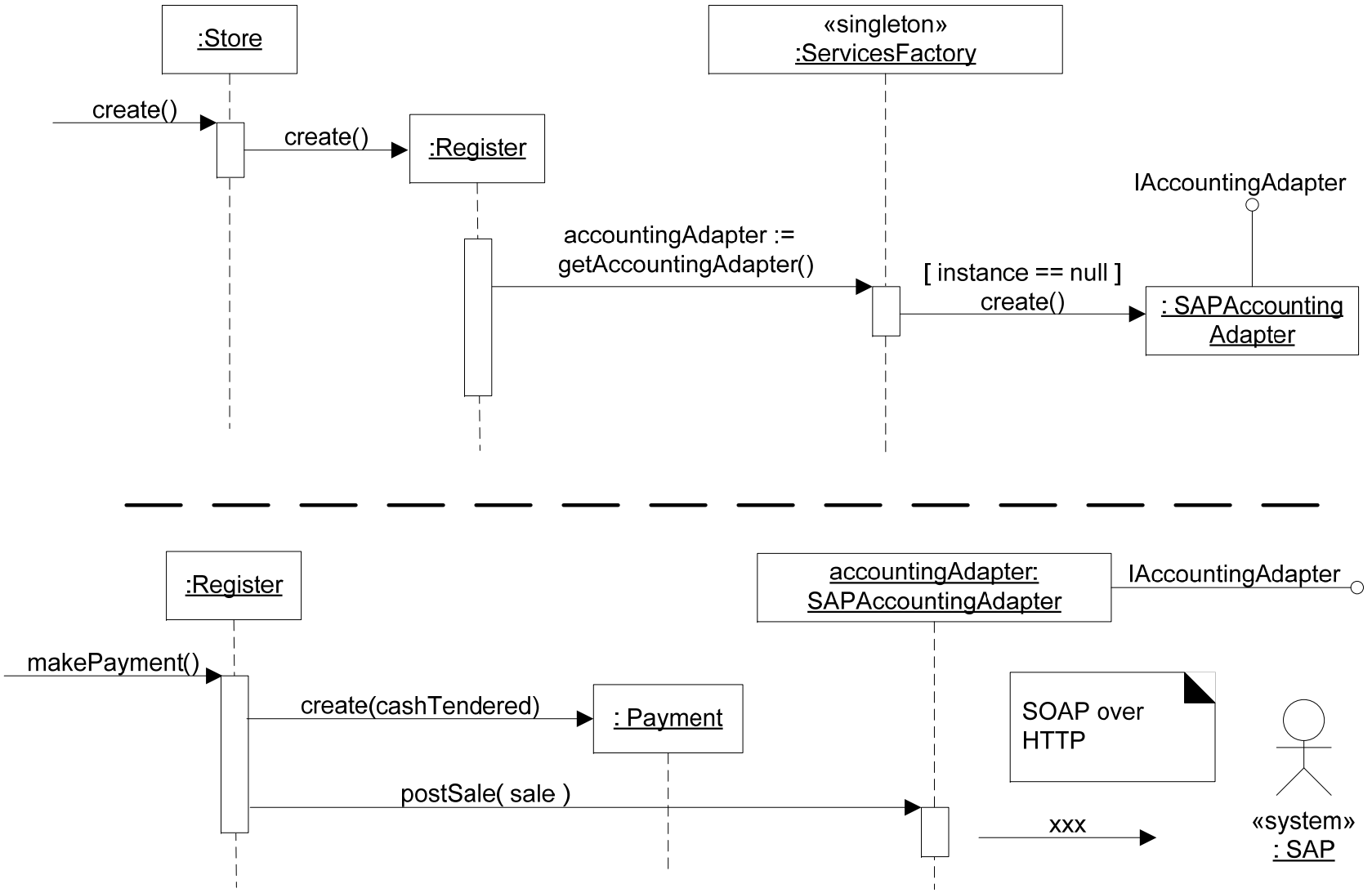
...

post: 2.3 the system call the *postSale* operation of the *Accounting System* with the current sale as a parameter

Example


UML notation: this '1' can optionally be used to indicate that only one instance will be created (a singleton)
1
ServicesFactory
instance : ServicesFactory
accountingAdapter : IAccountingAdapter
inventoryAdapter : IInventoryAdapter
taxCalculatorAdapter : ITaxCalculatorAdapter
getInstance() : ServicesFactory
getAccountingAdapter() : IAccountingAdapter
getInventoryAdapter() : IInventoryAdapter
getTaxCalculatorAdapter() : ITaxCalculatorAdapter
...
singleton static attribute
singleton static method
UML notation: in a class box, an underlined attribute or method indicates a static (class level) member, rather than an instance member
{
// static method
public static synchronized ServicesFactory getInstance()
{
if (instance == null)
instance := new ServicesFactory()
return instance
}
}

Example


:Store
«singleton»
:ServicesFactory
create()
create()
:Register
IAccountingAdapter
accountingAdapter :=
getAccountingAdapter()
[instance == null]
create()
: SAPAccounting
Adapter
:Register
accountingAdapter:
SAPAccountingAdapter
IAccountingAdapter
makePayment()
create(cashTendered)
: Payment
SOAP over
HTTP
postSale(sale)
xxx
«system»
: SAP

References

- *Design Patterns: Elements of Reusable Object-Oriented Software*
  E. Gamma; R. Helm; R. Johnson; J. Vlissides
  Addison-Wesley, 1995, pp. 127-134.

- *Applying UML and Patterns*
  C. Larman
  Prentice Hall, 2005 (Third edition), ch. 26

Strategy Pattern

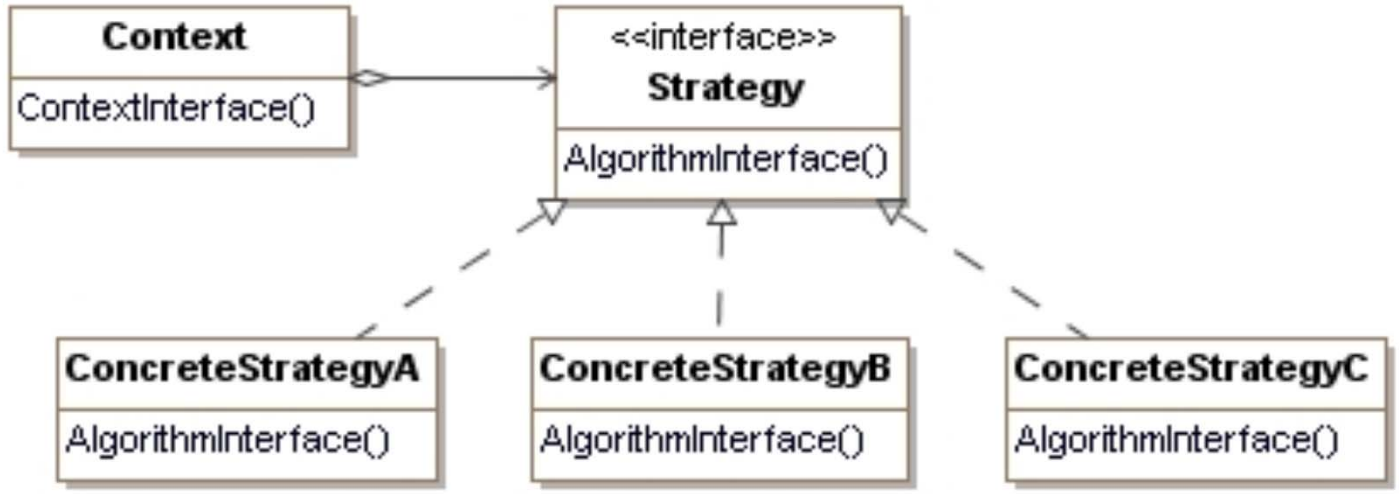
Strategy Pattern

- Overview
- Static View
- Dynamic View
- Example
- References

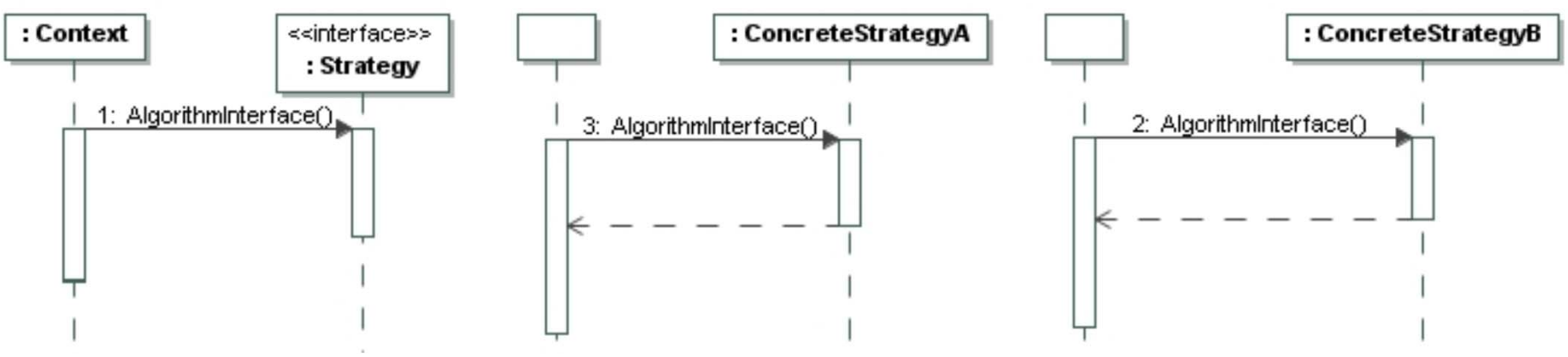
Overview

- 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


Context
ContextInterface()
<<interface>>
Strategy
AlgorithmInterface()
ConcreteStrategyA
AlgorithmInterface()
ConcreteStrategyB
AlgorithmInterface()
ConcreteStrategyC
AlgorithmInterface()

Dynamic View



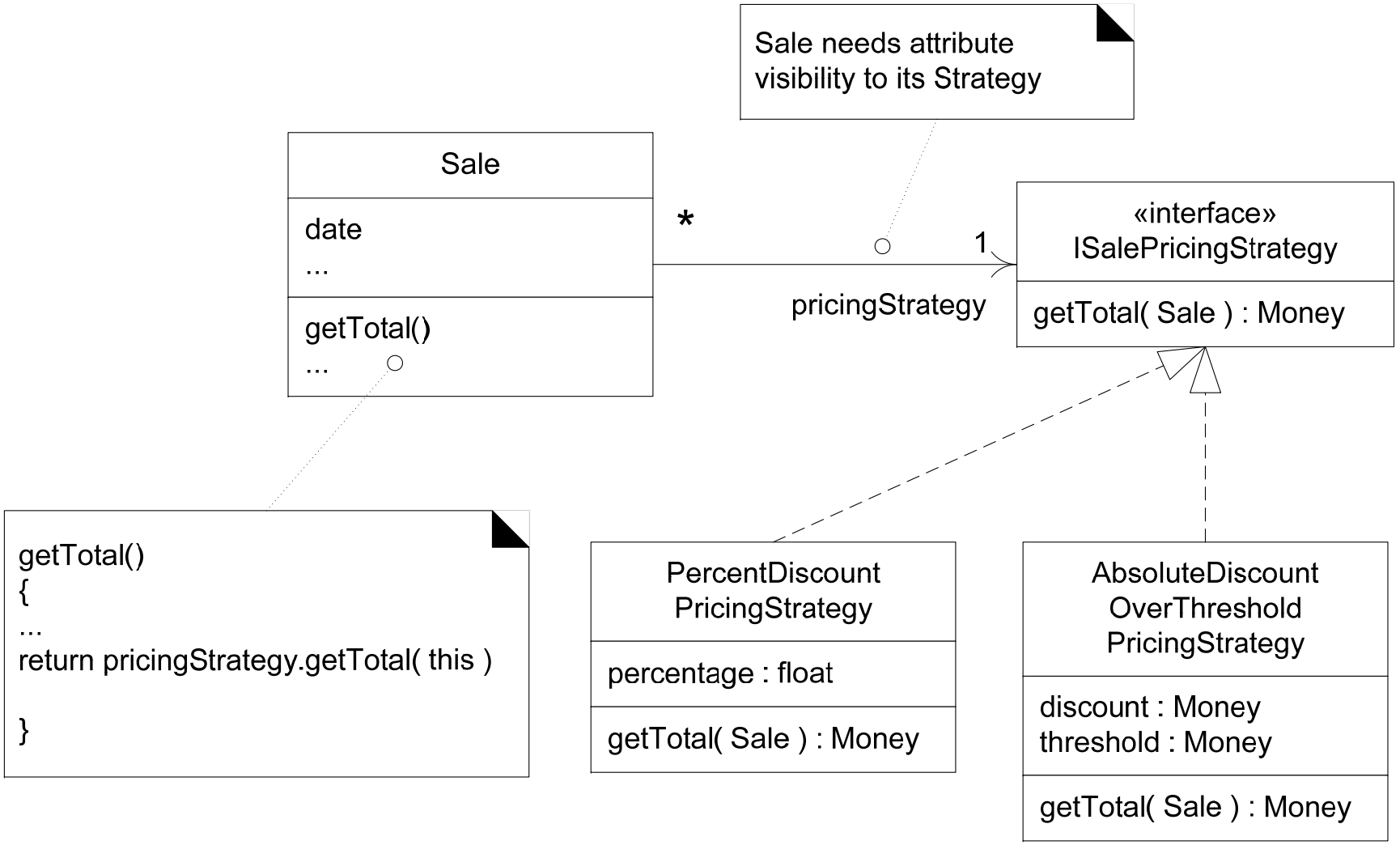
- The *AlgorithmInterface* method is different for each ConcreteStrategy class
- A similar sequence diagram for the ConcreteStrategyC

Example

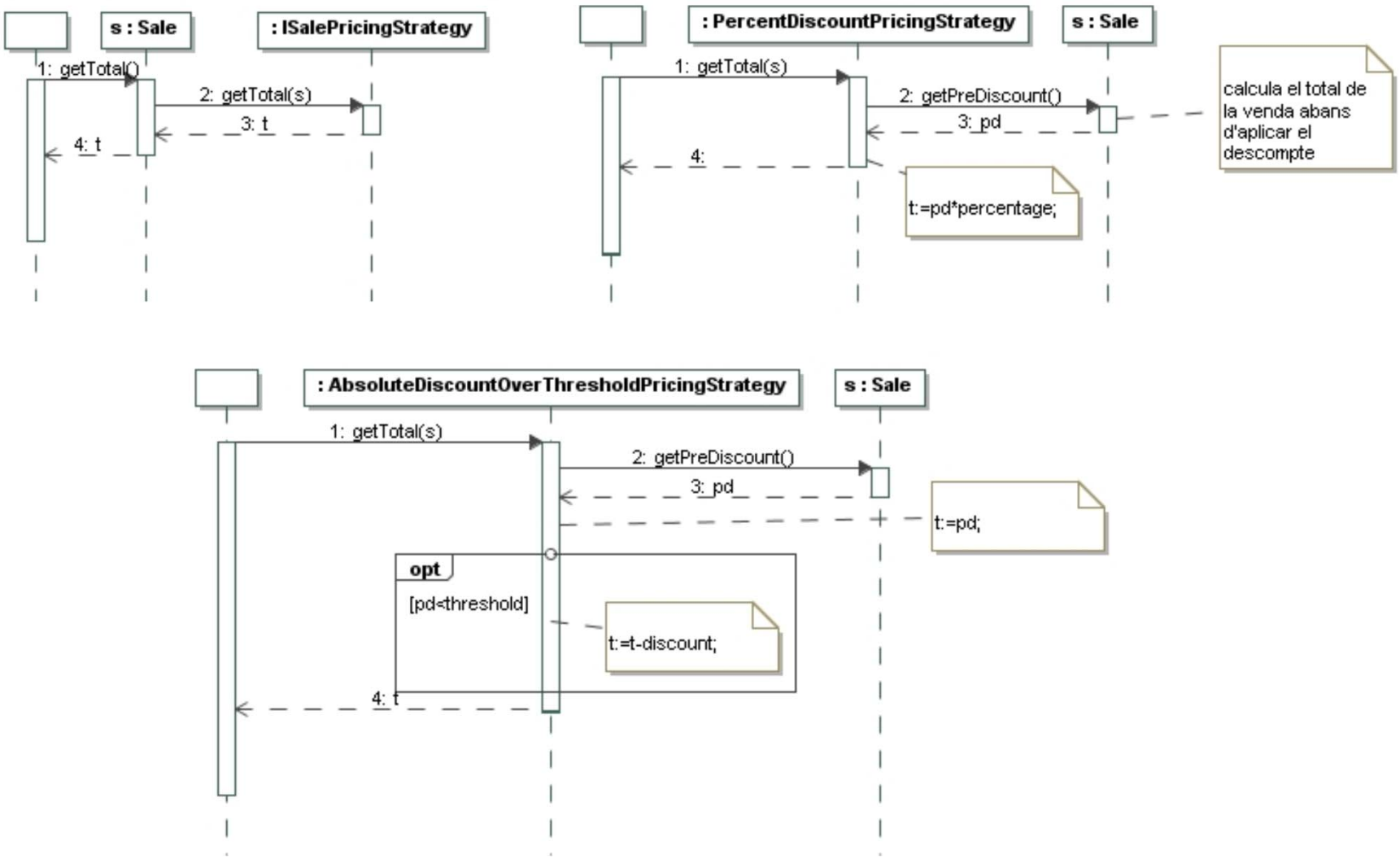
- In the NextGen POS system the pricing strategy for a sale can vary. During one period it may be 10% off all sales, later it may be 10 euros off if sale total is greater than 200 euros, and myriad other variations.

- How do we design for these varying pricing algorithms?

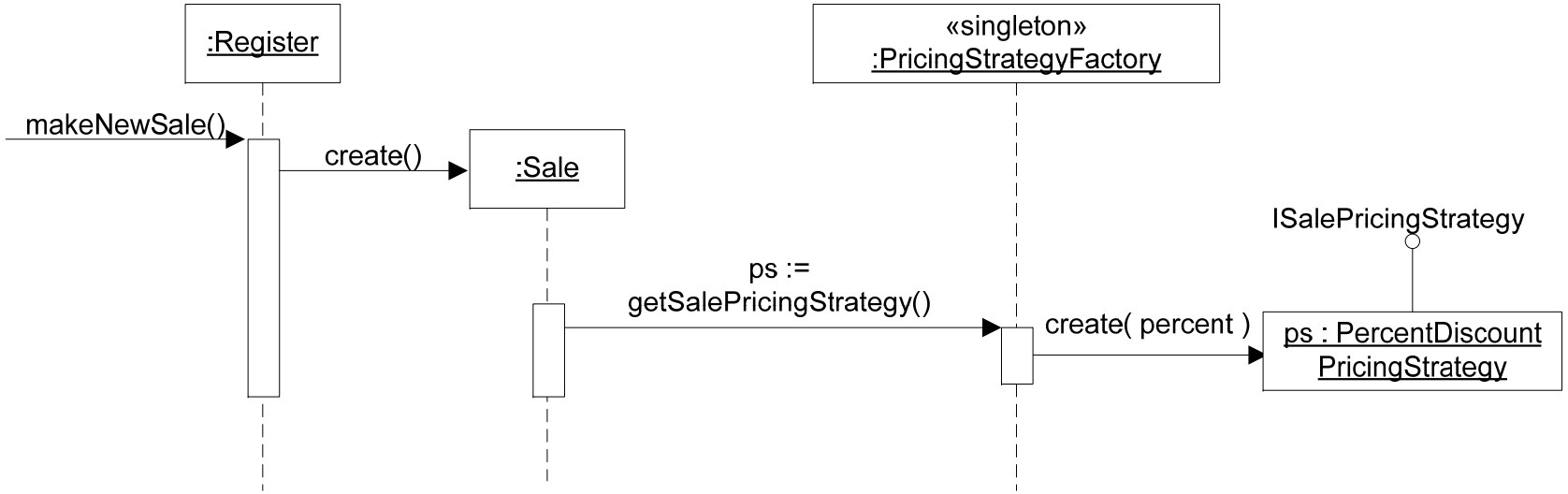
Example


Sale needs attribute visibility to its Strategy
Sale
date
...
getTotal()
...
*
1
pricingStrategy
«interface»
ISalePricingStrategy
getTotal(Sale) : Money
getTotal()
{
...
return pricingStrategy.getTotal(this)
}
PercentDiscount
PricingStrategy
percentage : float
getTotal(Sale) : Money
AbsoluteDiscount
OverThreshold
PricingStrategy
discount : Money
threshold : Money
getTotal(Sale) : Money

Example


s : Sale
: ISalePricingStrategy
1: getTotal()
2: getTotal(s)
3: t
4: t
: PercentDiscountPricingStrategy
s : Sale
1: getTotal(s)
2: getPreDiscount()
3: pd
4:
calcula el total de la venda abans d'aplicar el descompte
t:=pd*percentage;
: AbsoluteDiscountOverThresholdPricingStrategy
s : Sale
1: getTotal(s)
2: getPreDiscount()
3: pd
t:=pd;
opt
[pd<threshold]
t:=t-discount;
4: t

Example


:Register
«singleton»
:PricingStrategyFactory
makeNewSale()
create()
:Sale
ISalePricingStrategy
ps :=
getSalePricingStrategy()
create(percent)
ps : PercentDiscount
PricingStrategy

References

- *Design Patterns: Elements of Reusable Object-Oriented Software*
  E. Gamma; R. Helm; R. Johnson; J. Vlissides
  Addison-Wesley, 1995, pp. 315-324.

- *Applying UML and Patterns*
  C. Larman
  Prentice Hall, 2005 (Third edition), ch. 26

Template Method Pattern

Template Method Pattern

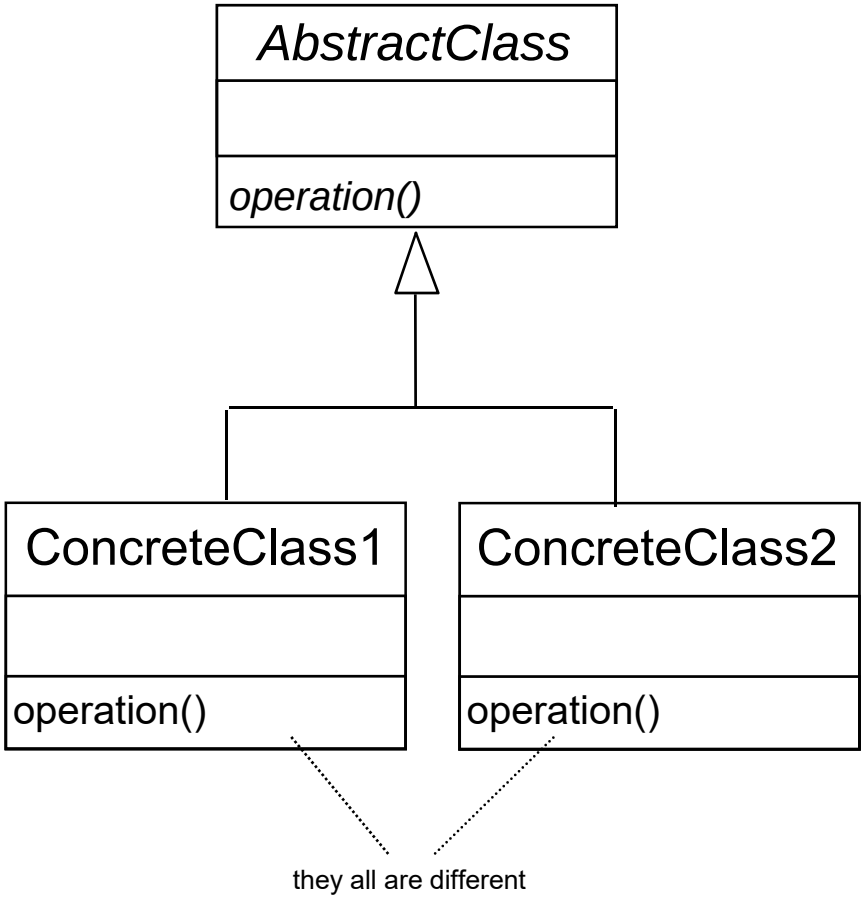
- Overview
- Static View
- Dynamic View
- Example
- References

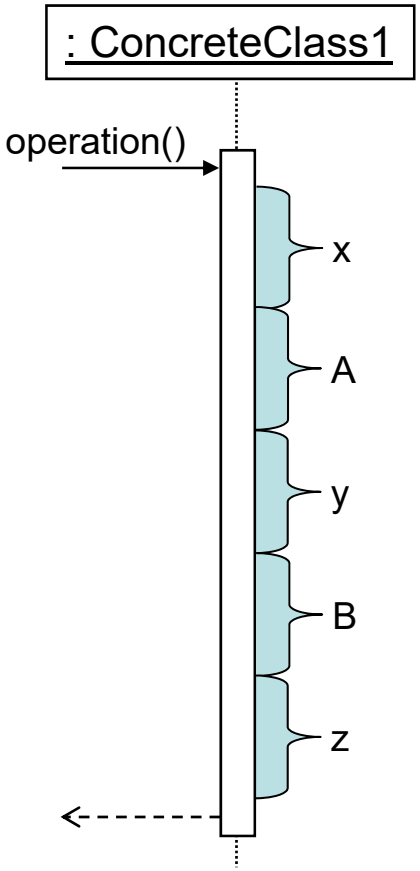
Overview

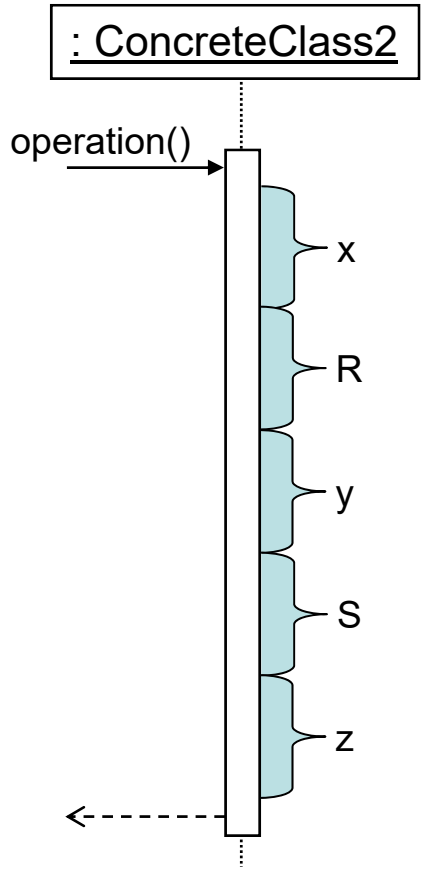
- Context
  - The definition of an operation in a hierarchy has some common behaviour to all subclasses but also some specific behaviour for each of them.
- Problem
  - Replicating the common behaviour in all subclasses requires code duplication and therefore a more costly maintenance.
- Solution
  - 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.

Overview

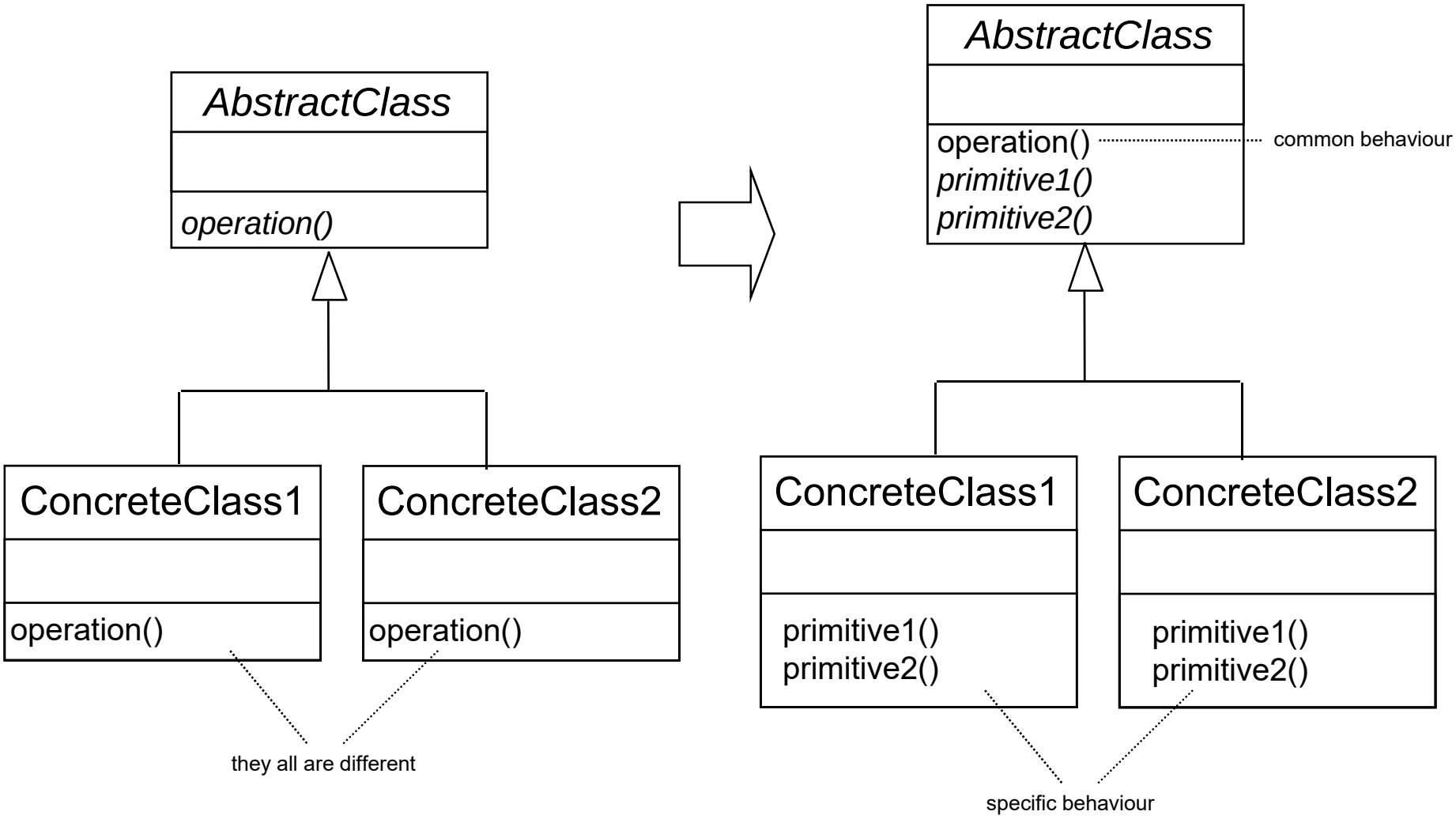
- Context: Starting situation



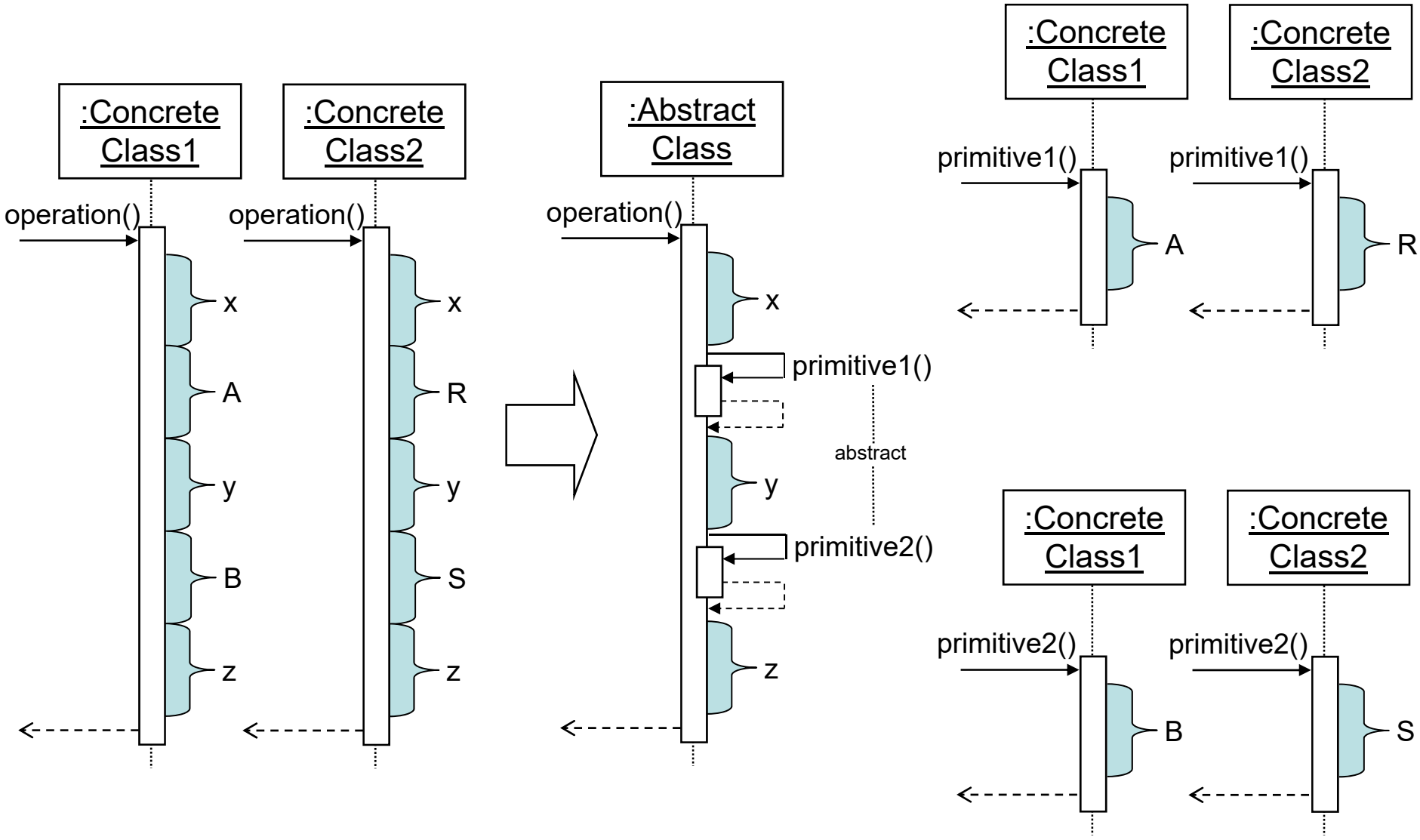




Static View

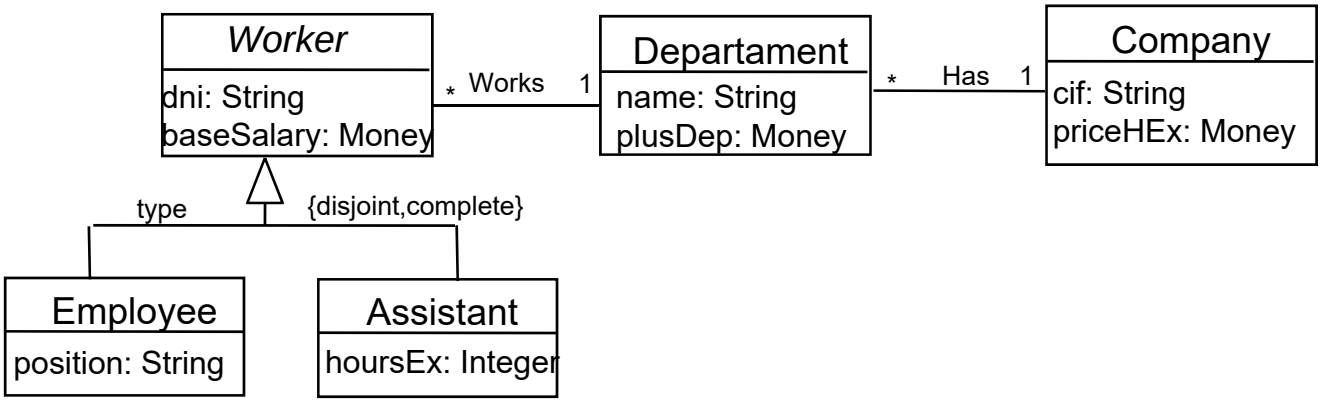

AbstractClass
operation()
ConcreteClass1
operation()
ConcreteClass2
operation()
they all are different
AbstractClass
operation()
primitive1()
primitive2()
common behaviour
ConcreteClass1
primitive1()
primitive2()
ConcreteClass2
primitive1()
primitive2()
specific behaviour

Dynamic View



Example

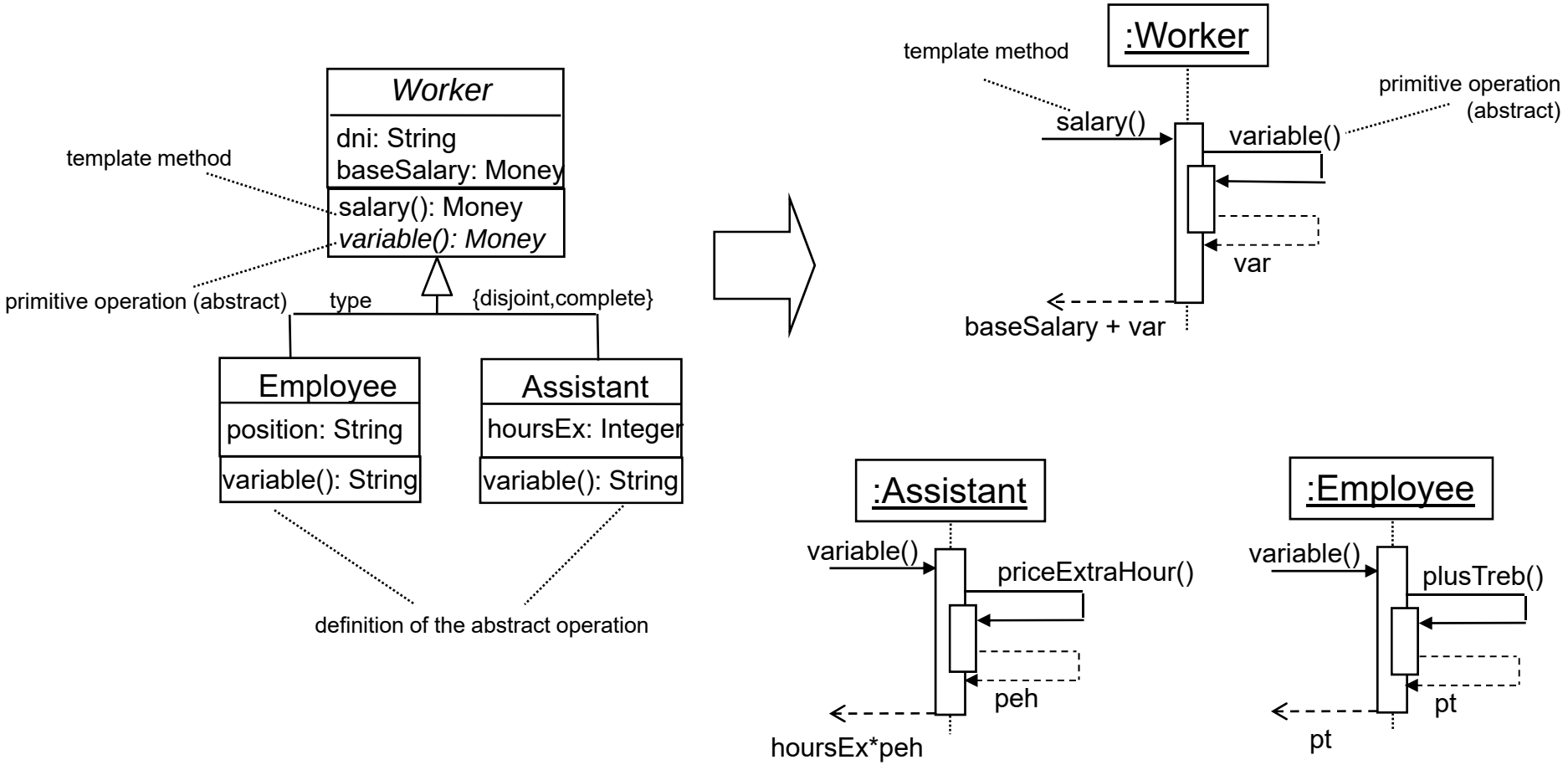
- Specification data conceptual model:



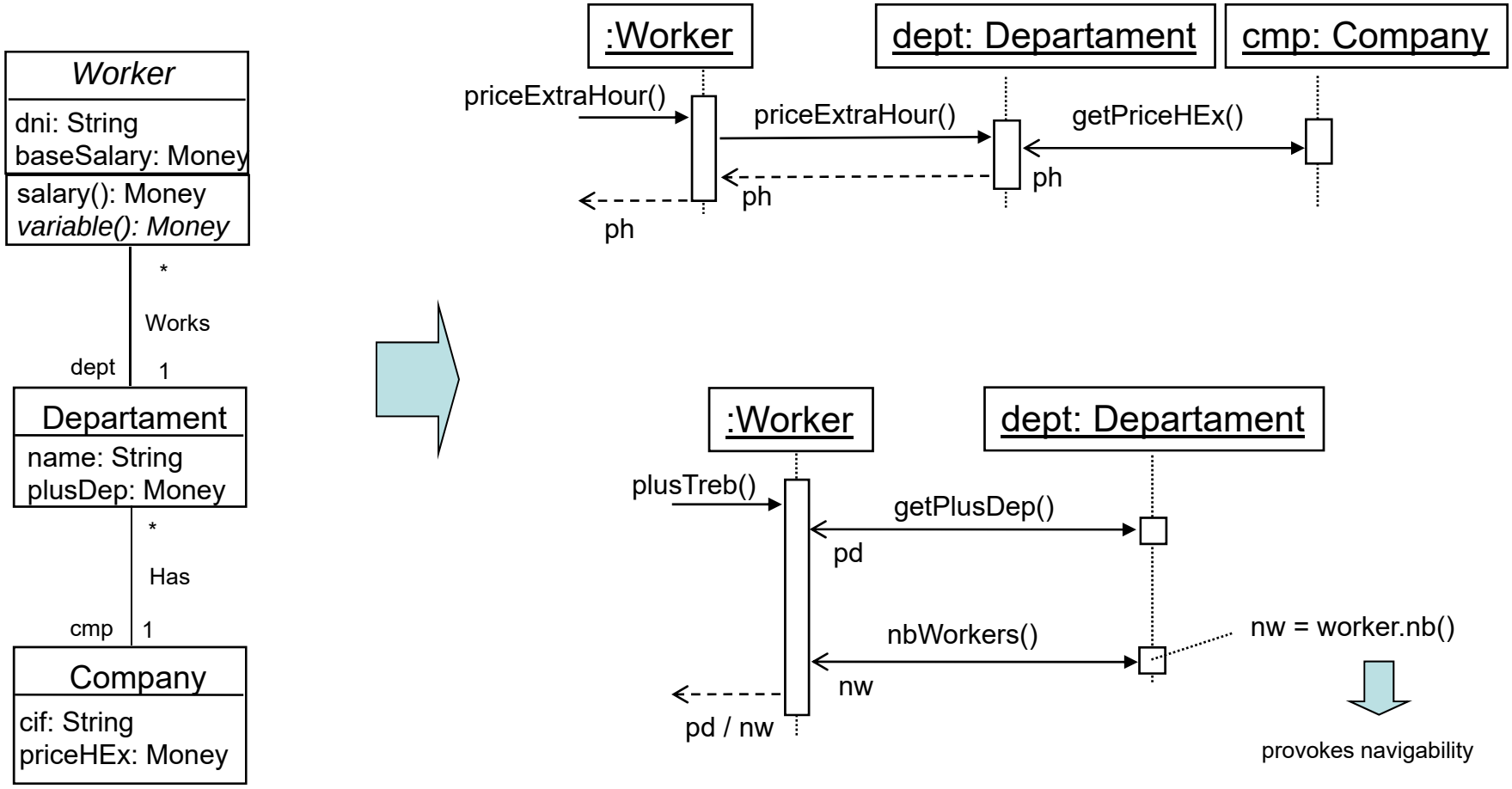
Textual Integrity Constraints:
Identifiers: (Worker, dni); (Company, cif)
There cannot exist two departments with the same name in the same company

- 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*

Example


Worker
dni: String
baseSalary: Money
salary(): Money
variable(): Money
template method
primitive operation (abstract)
type
{disjoint,complete}
Employee
position: String
variable(): String
Assistant
hoursEx: Integer
variable(): String
definition of the abstract operation
:Worker
template method
salary()
variable()
primitive operation (abstract)
var
baseSalary + var
:Assistant
variable()
priceExtraHour()
peh
hoursEx*peh
:Employee
variable()
plusTreb()
pt
pt

Example


Worker
dni: String
baseSalary: Money
salary(): Money
variable(): Money
*
Works
dept
1
Departament
name: String
plusDep: Money
*
Has
cmp
1
Company
cif: String
priceHEx: Money
:Worker
dept: Departament
cmp: Company
priceExtraHour()
priceExtraHour()
getPriceHEx()
ph
ph
ph
:Worker
dept: Departament
plusTreb()
getPlusDep()
pd
nbWorkers()
nw
pd / nw
nw = worker.nb()
provokes navigability

References

- *Design Patterns: Elements of Reusable Object-Oriented Software*
  E. Gamma; R. Helm; R. Johnson; J. Vlissides
  Addison-Wesley, 1995, pp. 325-330.

- *Applying UML and Patterns.*
  C. Larman
  Prentice Hall, 2005 (3rd edition), chap. 38.11

Access to the Data Layer

Access to the Data Layer

- Combining Patterns to Access to the Data Layer
- Example
- References

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.

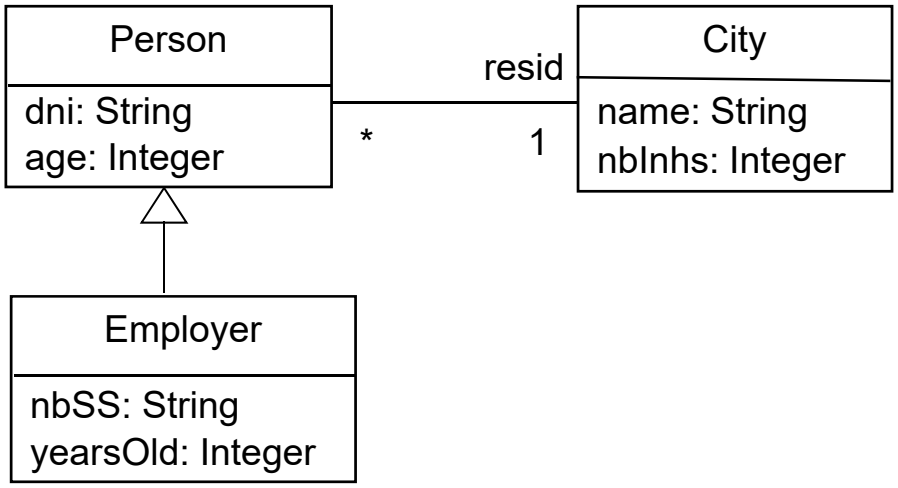
- The Data Layer must be ready to change the implementation of the operations to access and (in case of Transaction Script, modify data)

Combining Patterns to Access to the Data Layer

- For each persistent class of the Domain model that has one or more identifiers, an adapter should be defined.

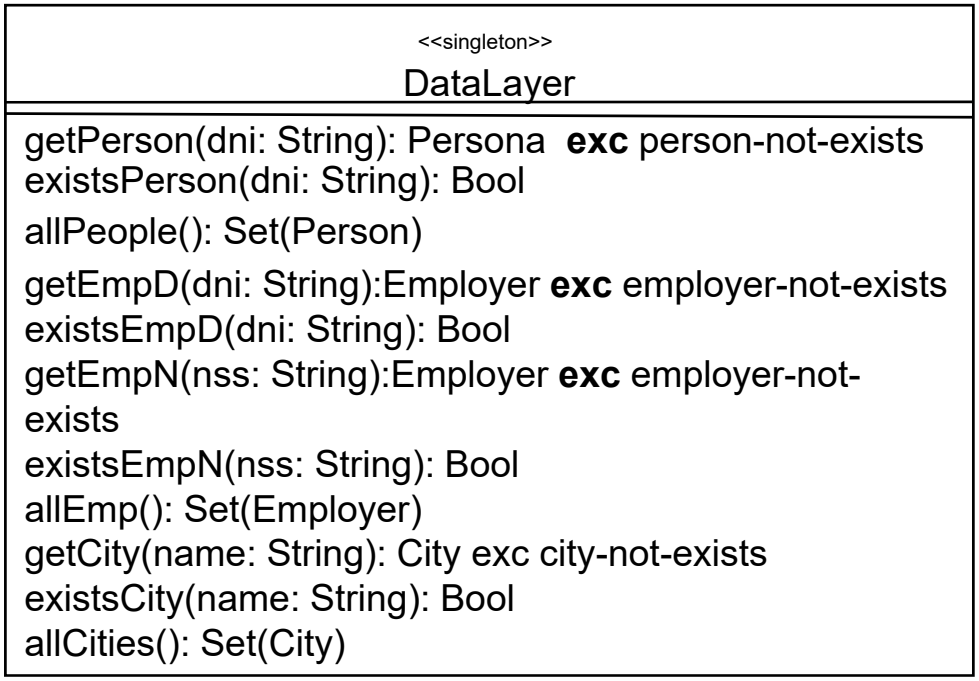
- A factory (singleton) to create all the adapters should be defined.

Example



Integrity constraints:

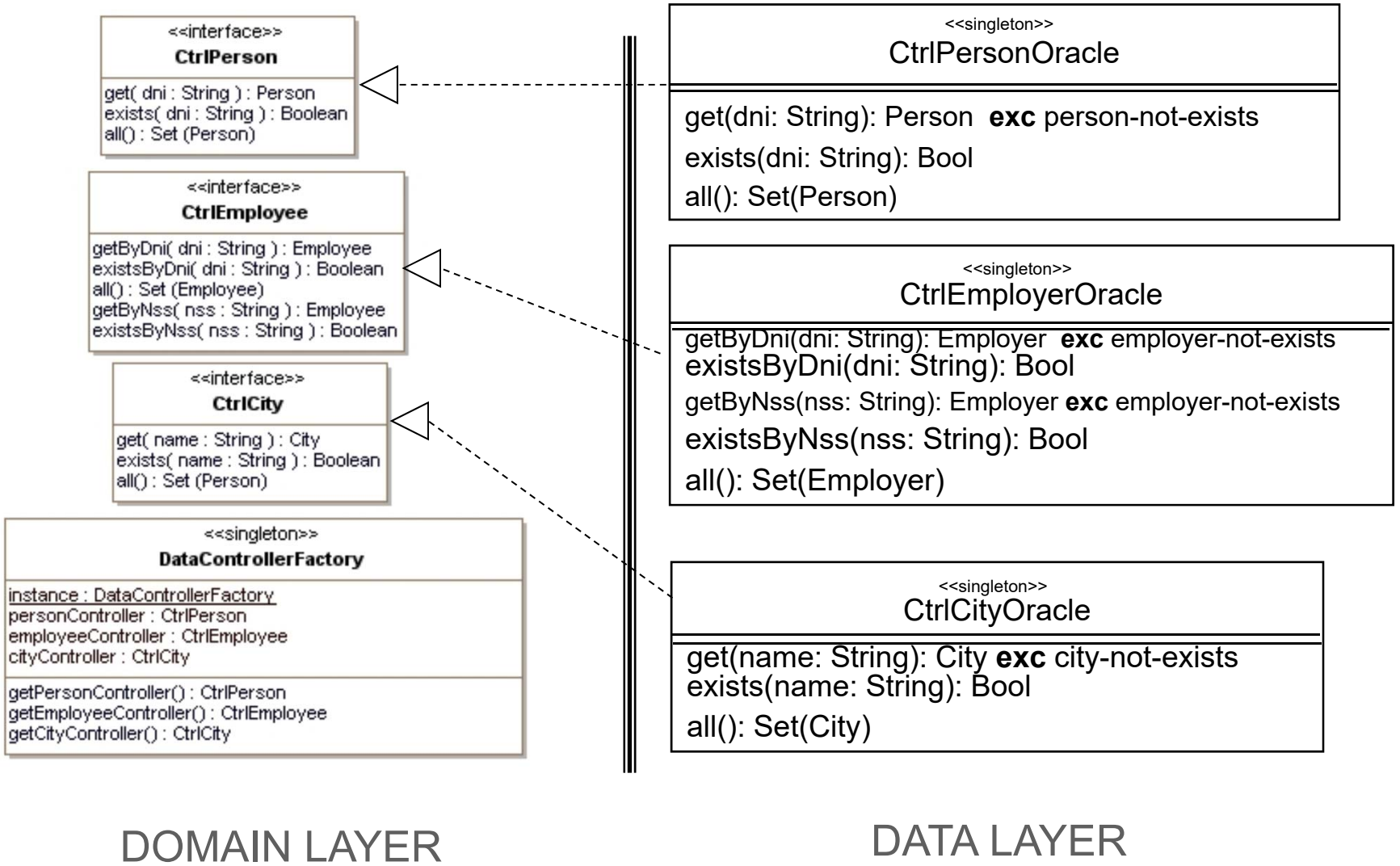
1. Key constraints:
   Person → dni, Employer → nbSS
   City → name



DOMAIN LAYER

DATA LAYER

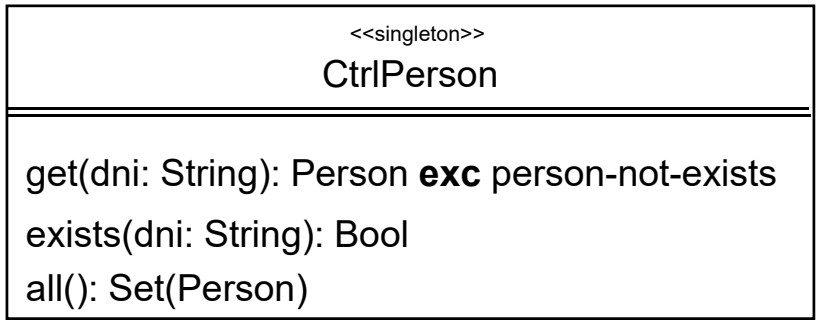
Example



DOMAIN LAYER

DATA LAYER

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

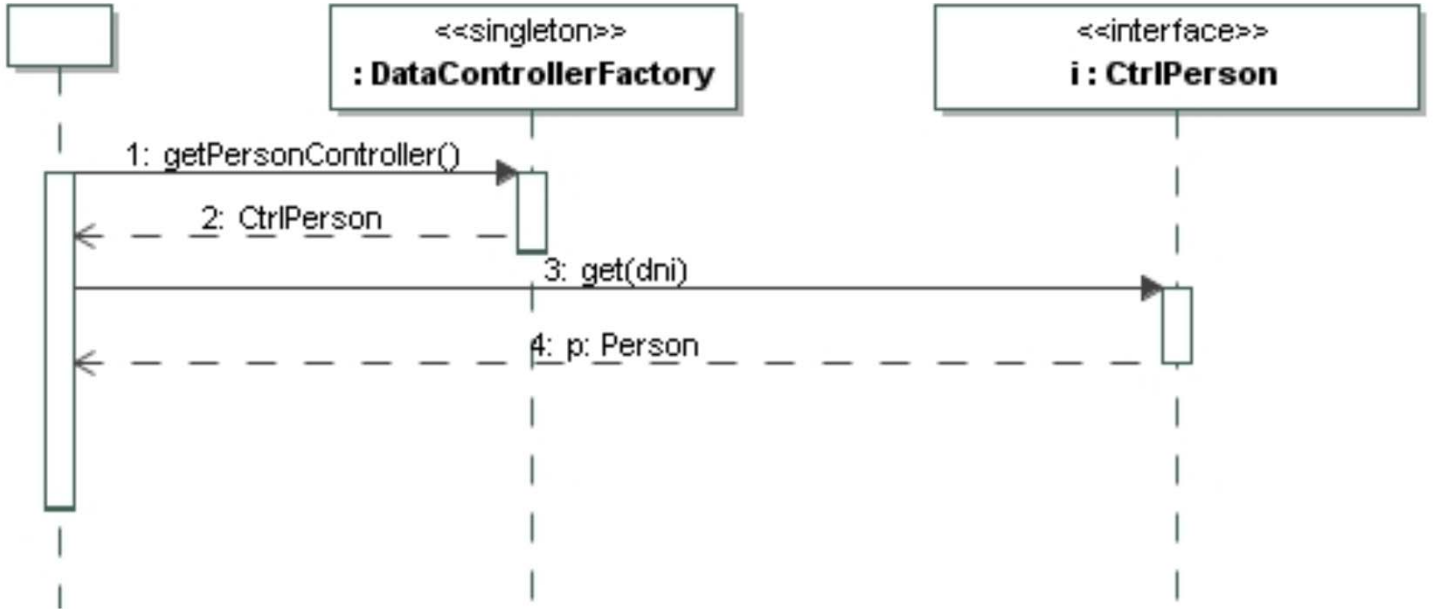
post: 2.1 returns true if the person with *dni* exists, false otherwise

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

post: 2.1 returns all the Persons existing in the system

Example

- Sequence diagram to obtain a Person by its dni.



References

- *Design Patterns: Elements of Reusable Object-Oriented Software*
  E. Gamma; R. Helm; R. Johnson; J. Vlissides
  Addison-Wesley, 1995, pp. 87-96, 127-138, 185-194.

- *Applying UML and Patterns*
  C. Larman
  Prentice Hall, 2005 (Third edition).