

Patterns for Domain Layer

Patterns for Domain Layer

- Adapter
- Abstract Factory
- Singleton
- Strategy
- State
- Template Method
- References

Adapter Pattern

Adapter Pattern

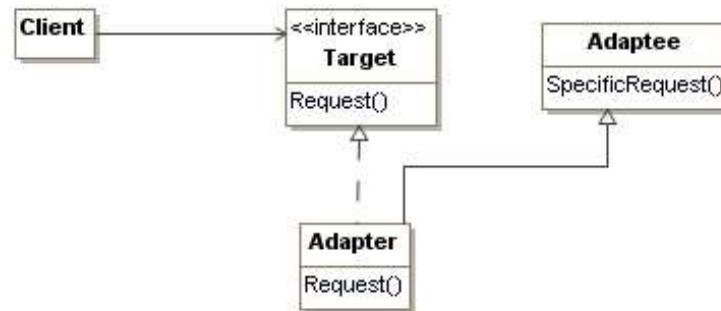
- Overview
- Static View
- Dynamic View
- Example
- References

Overview

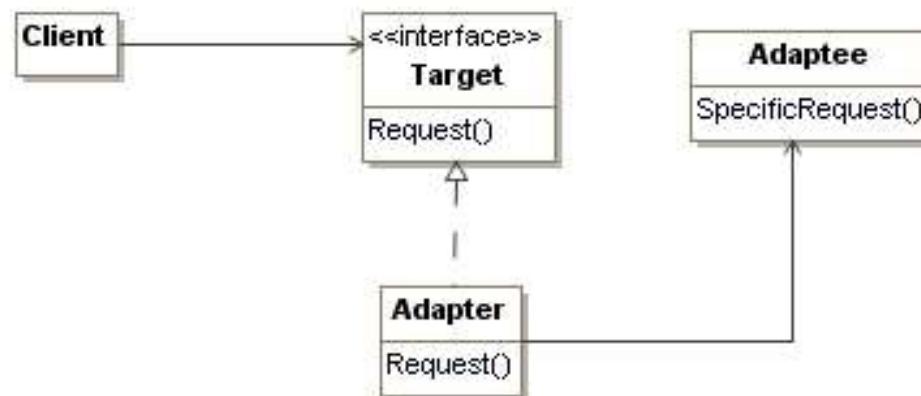
- Context
 - The interface of an existing class (or a set of subclasses) does not match the one needed
- Problem
 - How to resolve incompatible interfaces, or provide a stable interface to similar components with different interfaces?
- Solution
 - Convert the interface of a class (Adaptee class) into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

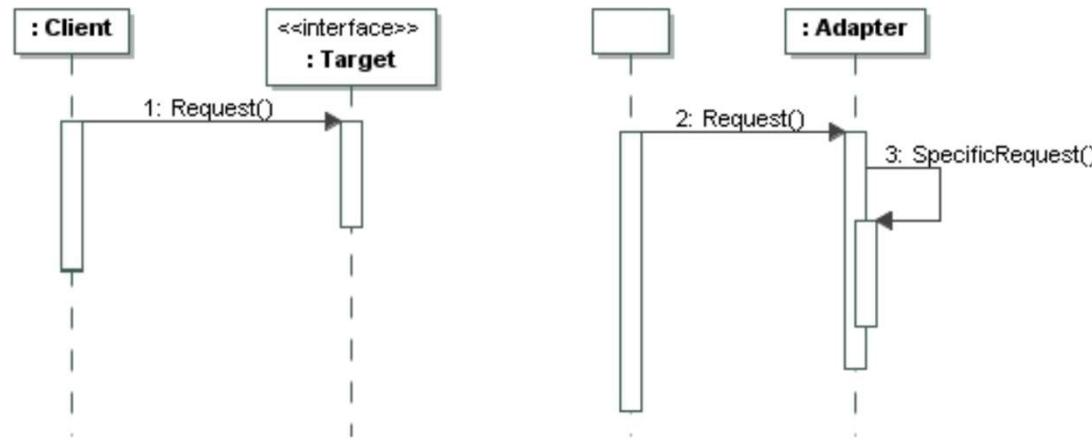


- Object adapter (useful when one adapter adapts more than one adaptee)

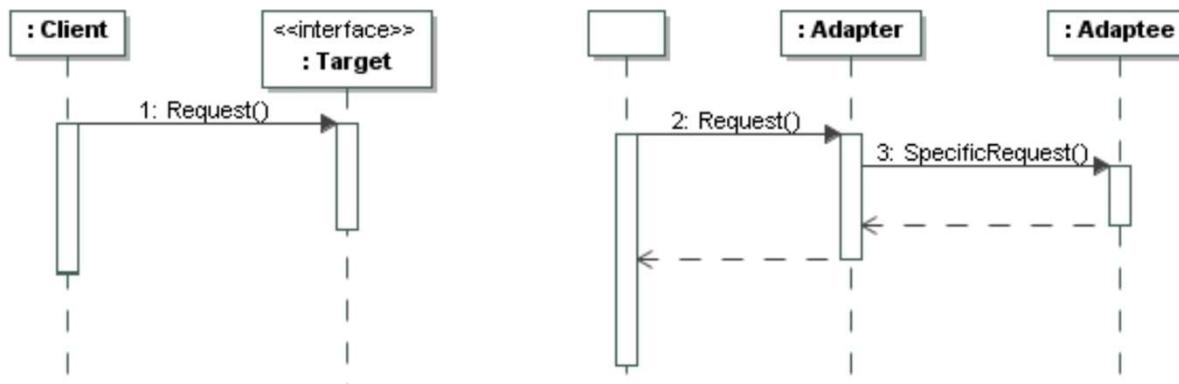


Dynamic View

- Class adapter



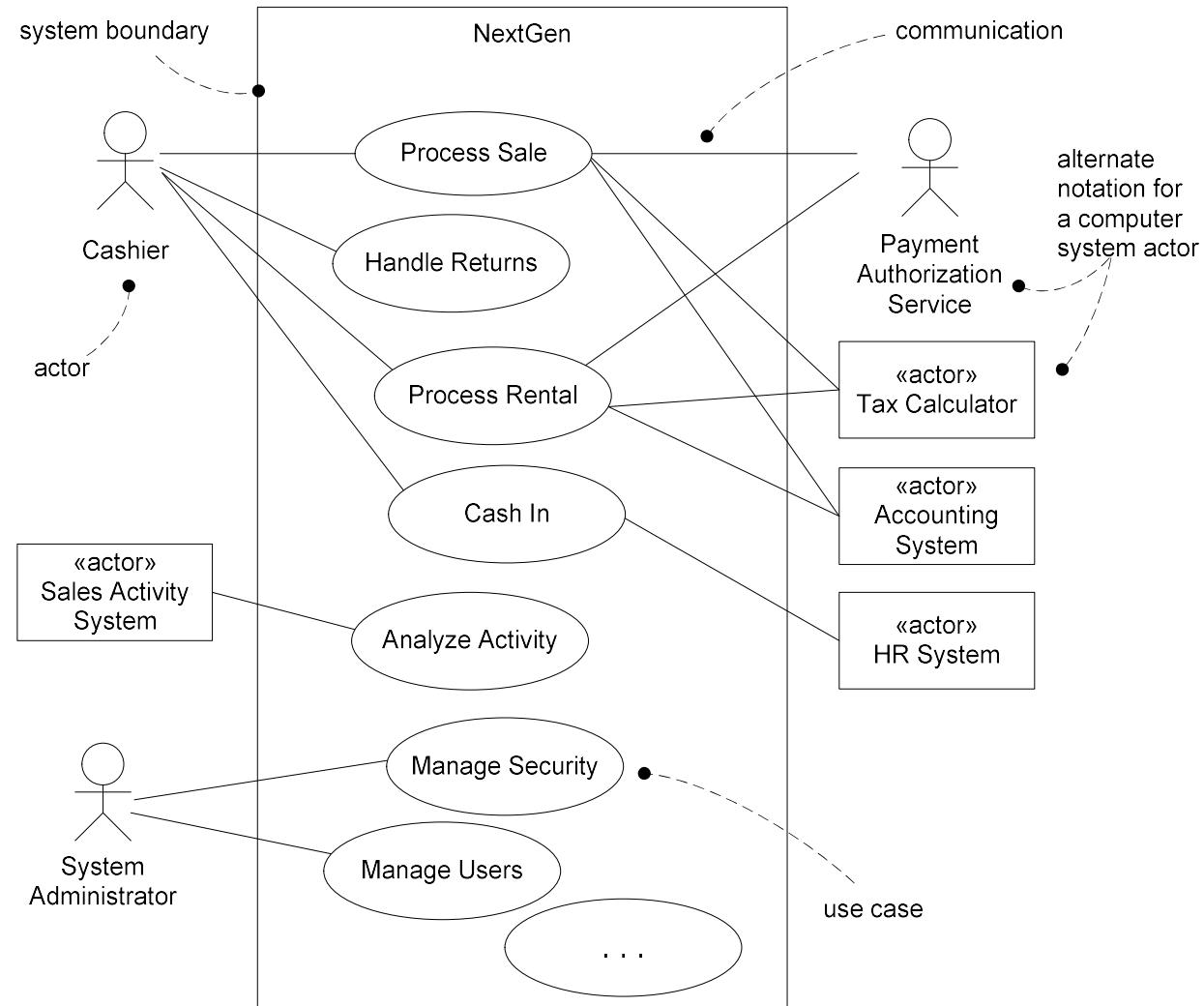
- Object adapter



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

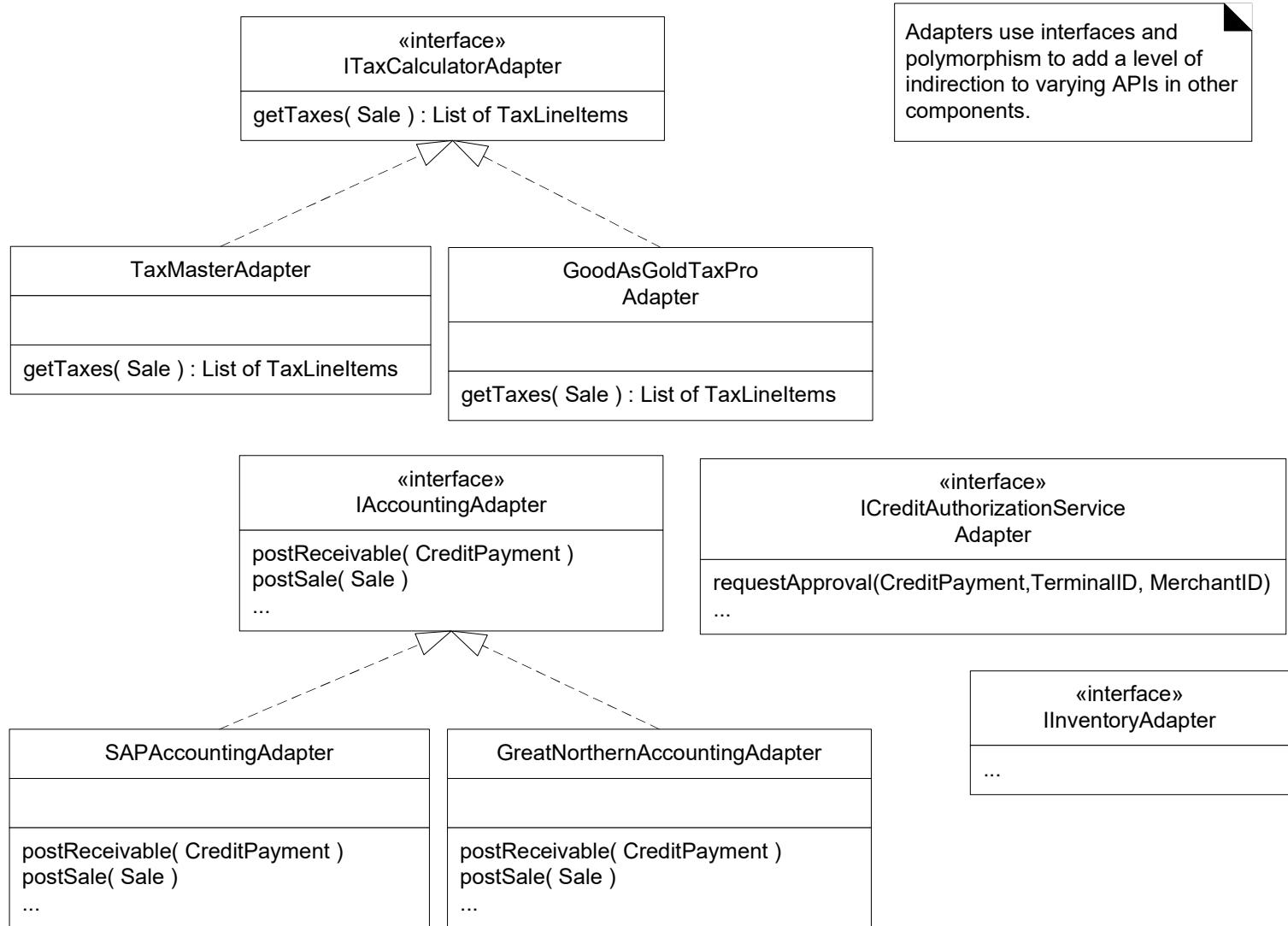


Example

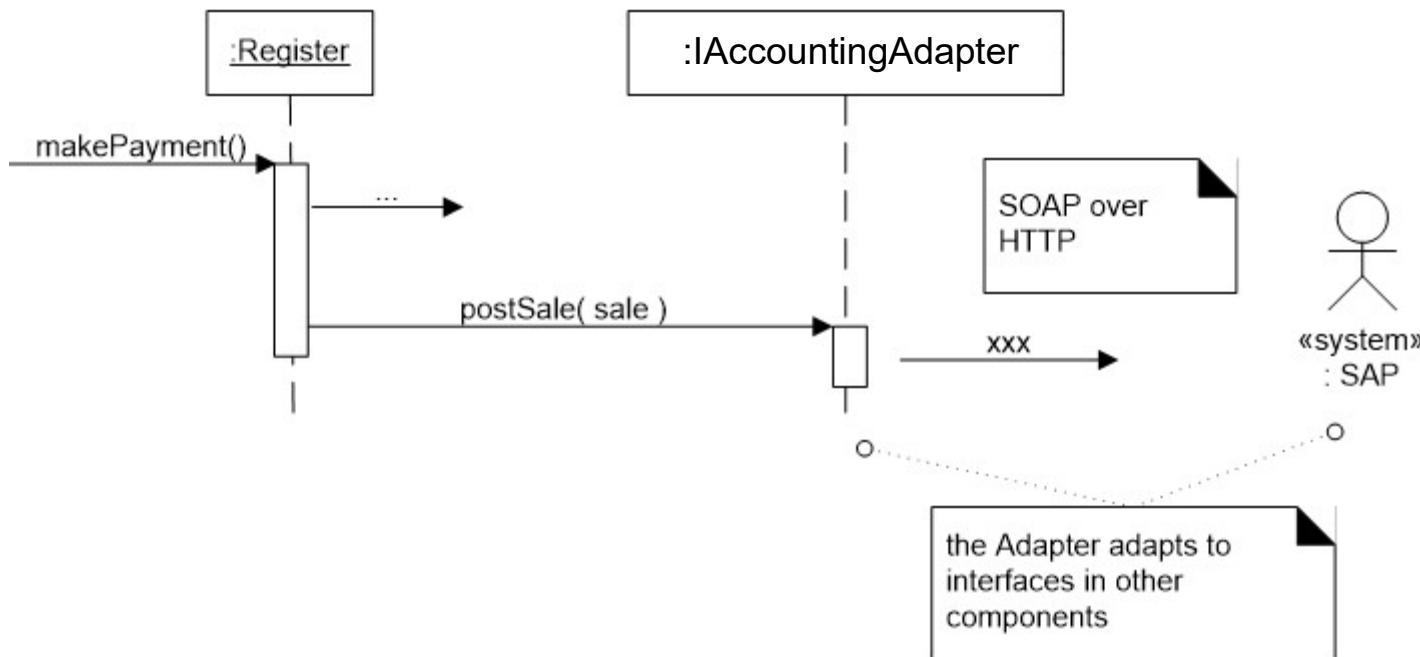
- *makePayment* contract (partial) of the Domain Layer

```
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



Example



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
- <https://www.youtube.com/watch?v=2PKQtcJjYvc&t=1212s>
- <https://refactoring.guru/es/design-patterns/adapter>

Abstract Factory Pattern

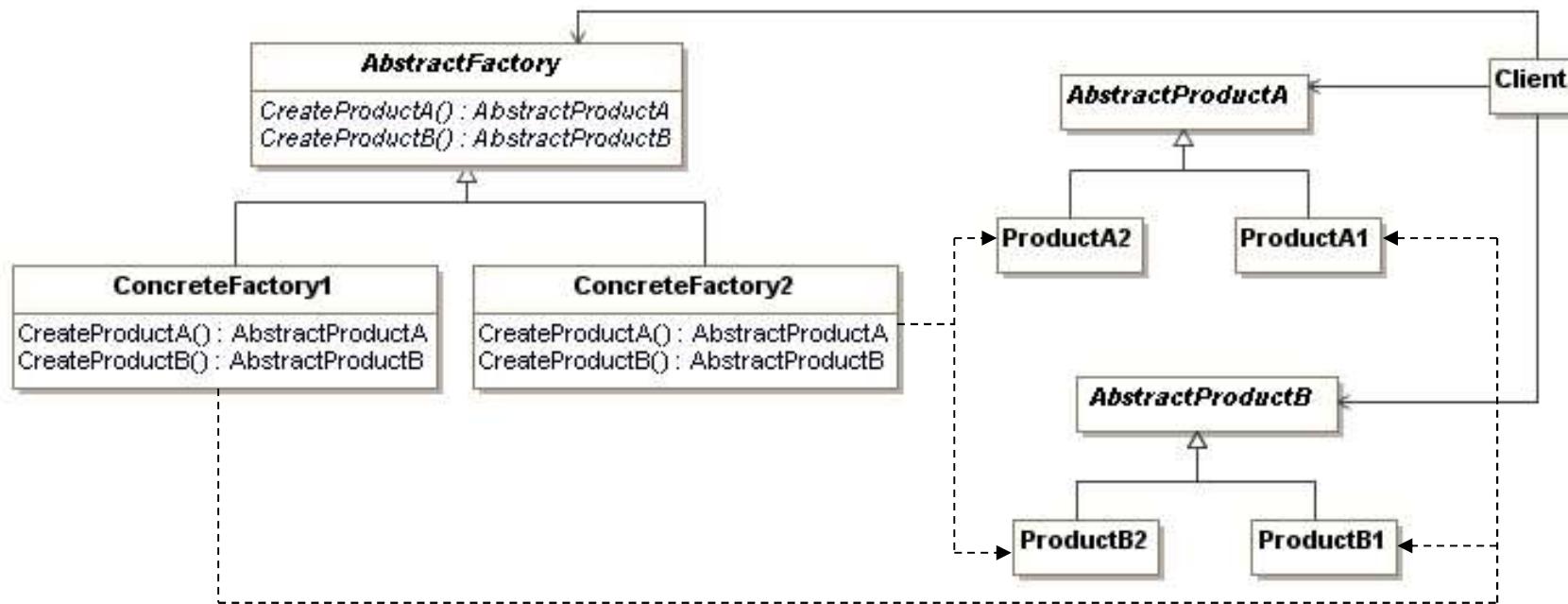
Abstract Factory Pattern

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

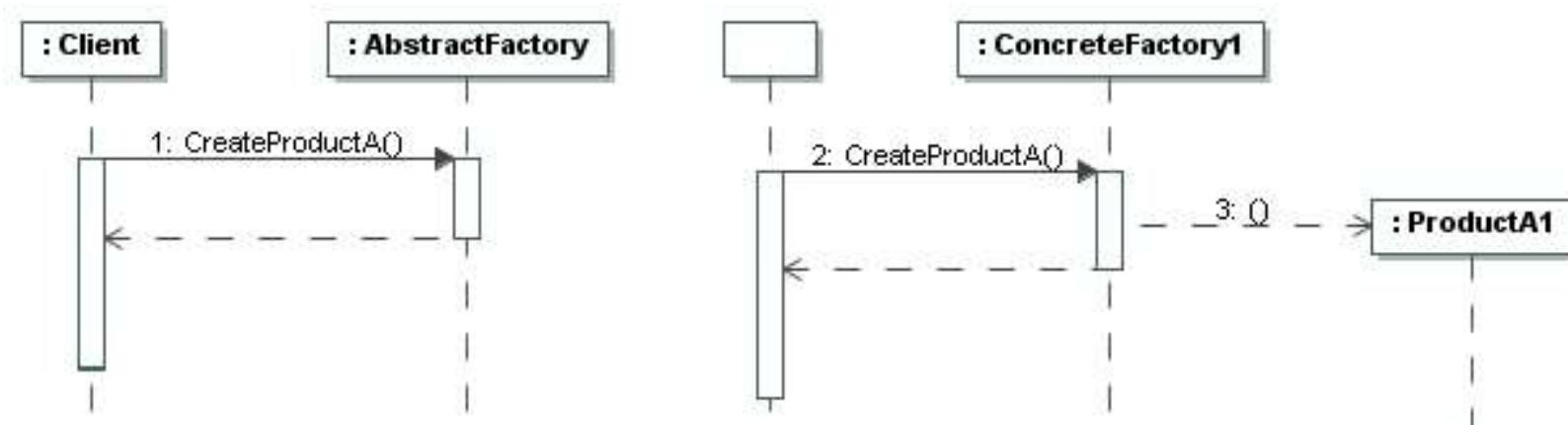
Overview

- 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

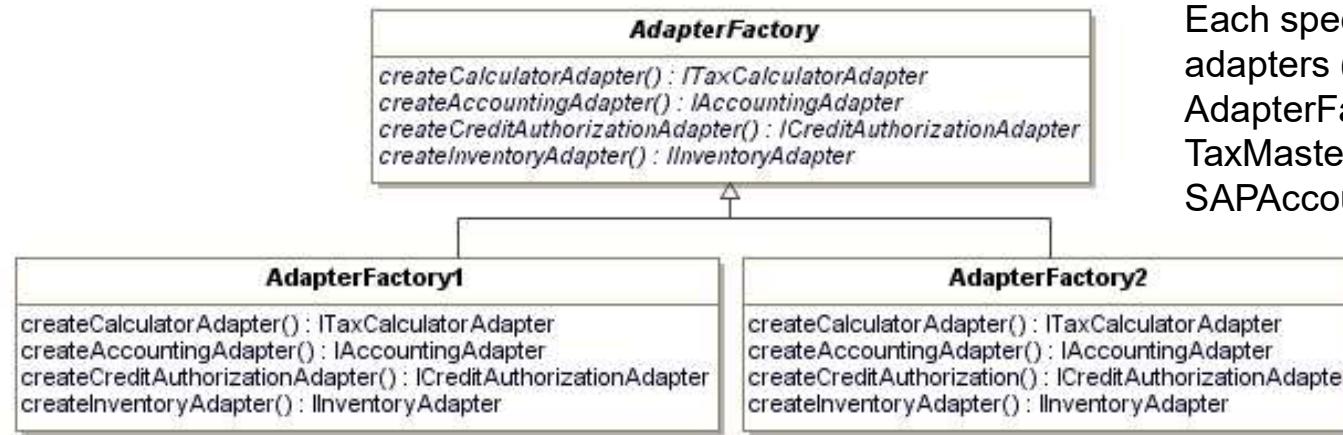


Dynamic View

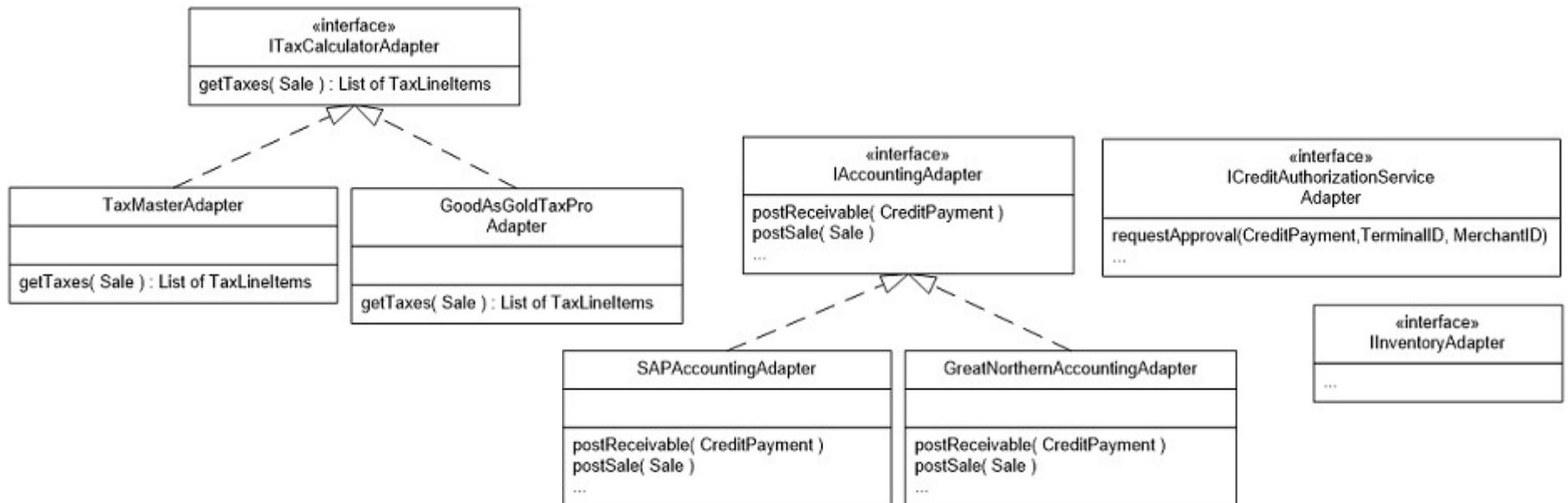


The sequence diagrams of the other operations are similar

Example



Each specific adapter creates a set of adapters (objects). For example, **AdapterFactory1** creates a **TaxMasterAdapter** object, a **SAPAccountingAdapter** object, ...

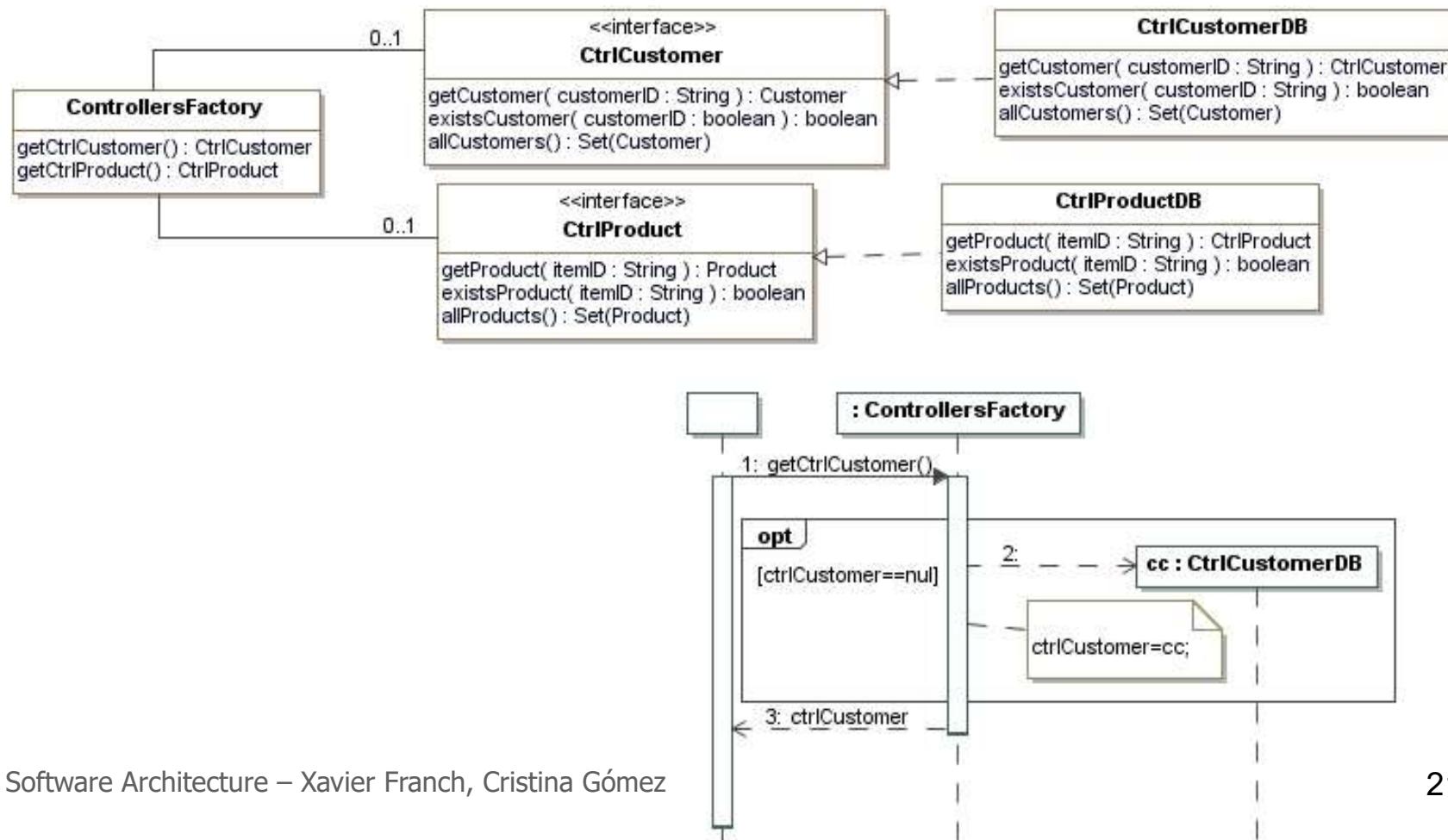


Simple or Concrete Factory

- 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 the access to data controllers raises a new problem in the design.
- Who creates those controllers?



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
- <https://www.youtube.com/watch?v=v-GiuMmsXj4>
- <https://refactoring.guru/es/design-patterns/abstract-factory>

Singleton Pattern

Singleton Pattern

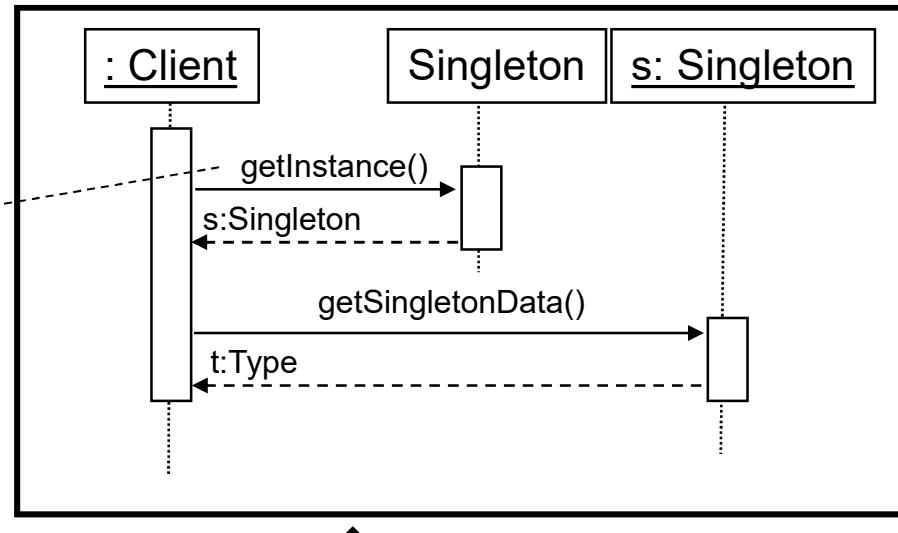
- Overview
- Static View
- Dynamic View
- Example
- References

Overview

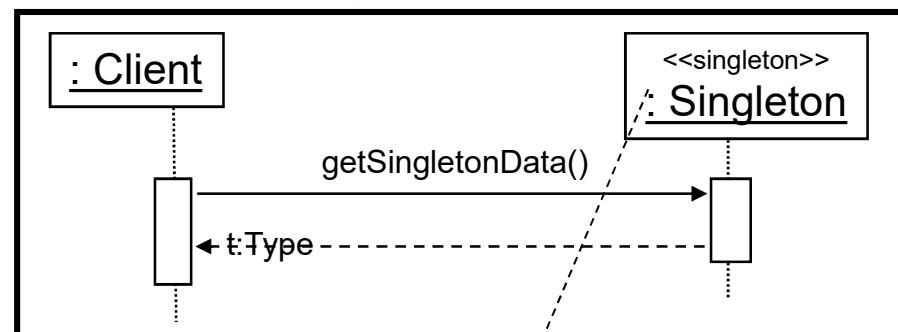
- Context
 - Systems that have classes with exactly one instance that must be accessible
- Problem
 - There must be exactly one instance of a class, and it must be accessible to clients from a well-known access point.
 - A global variable makes an object accessible, but it doesn't keep you from instantiating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

Static and Dynamic View

```
//static method  
public static Singleton getInstance()  
{  
    if (instance==null)  
        instance = new Singleton();  
    return instance;  
}
```



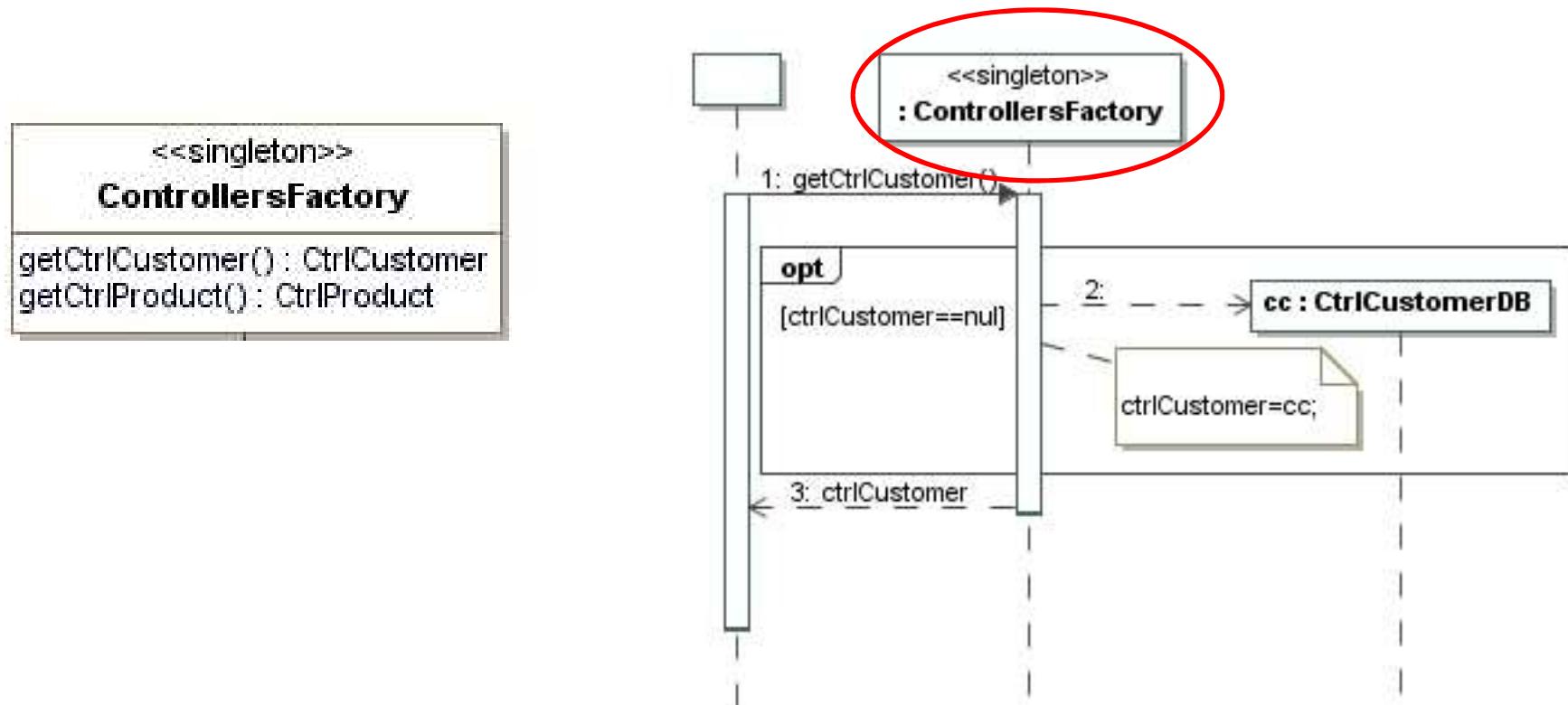
↔ Equivalent



The UML stereotype indicates that visibility to this instance was achieved by the singleton pattern (using the `getInstance()` operation)

Example

- In the NextGen POS system the use of factory to access to data controllers raises a new problem in the design.
- Who creates the factory itself, and how is it accessed?



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
- https://www.youtube.com/watch?v=hUE_j6q0LTQ
- <https://refactoring.guru/es/design-patterns/singleton>

Strategy Pattern

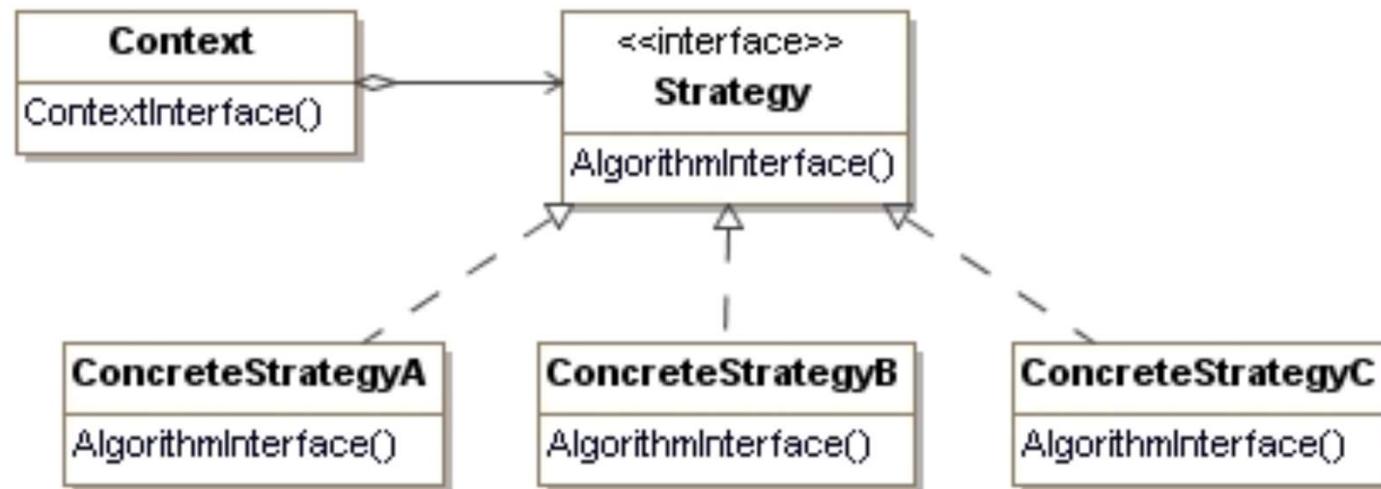
Strategy Pattern

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

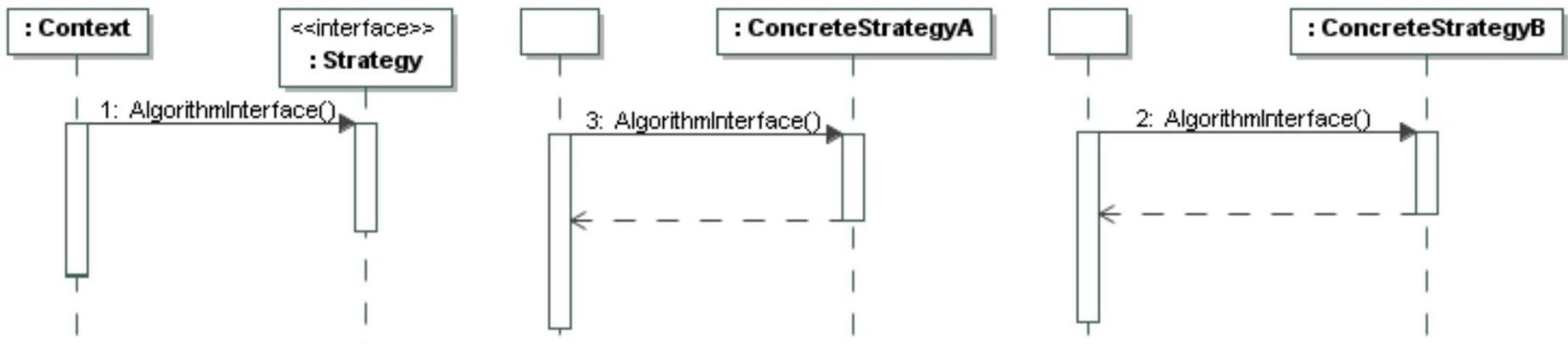
Overview

- 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



Dynamic View

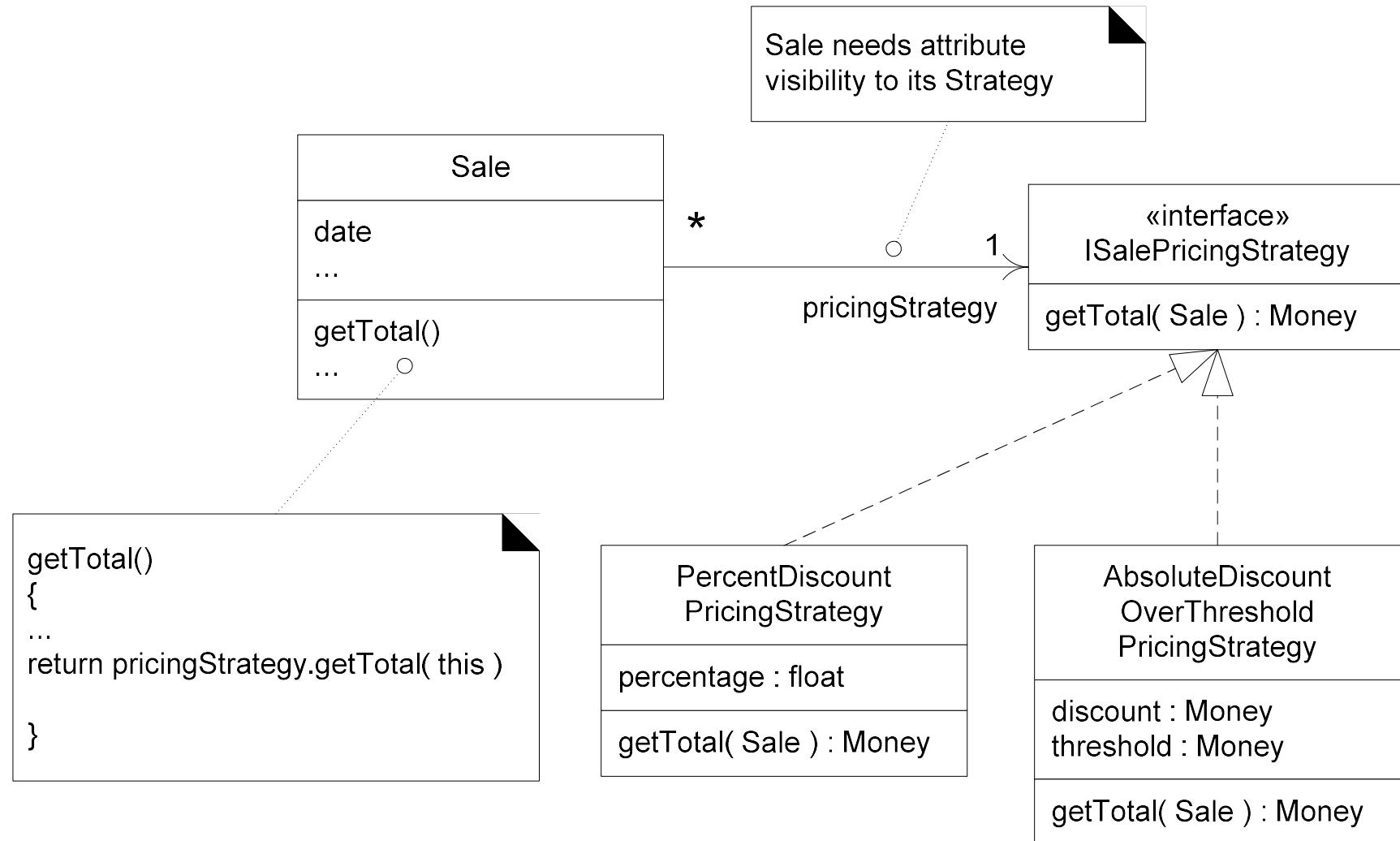


- The *AlgorithmInterface* method is different for each *ConcreteStrategy* class
 - A similar sequence diagram for the *ConcreteStrategyC*

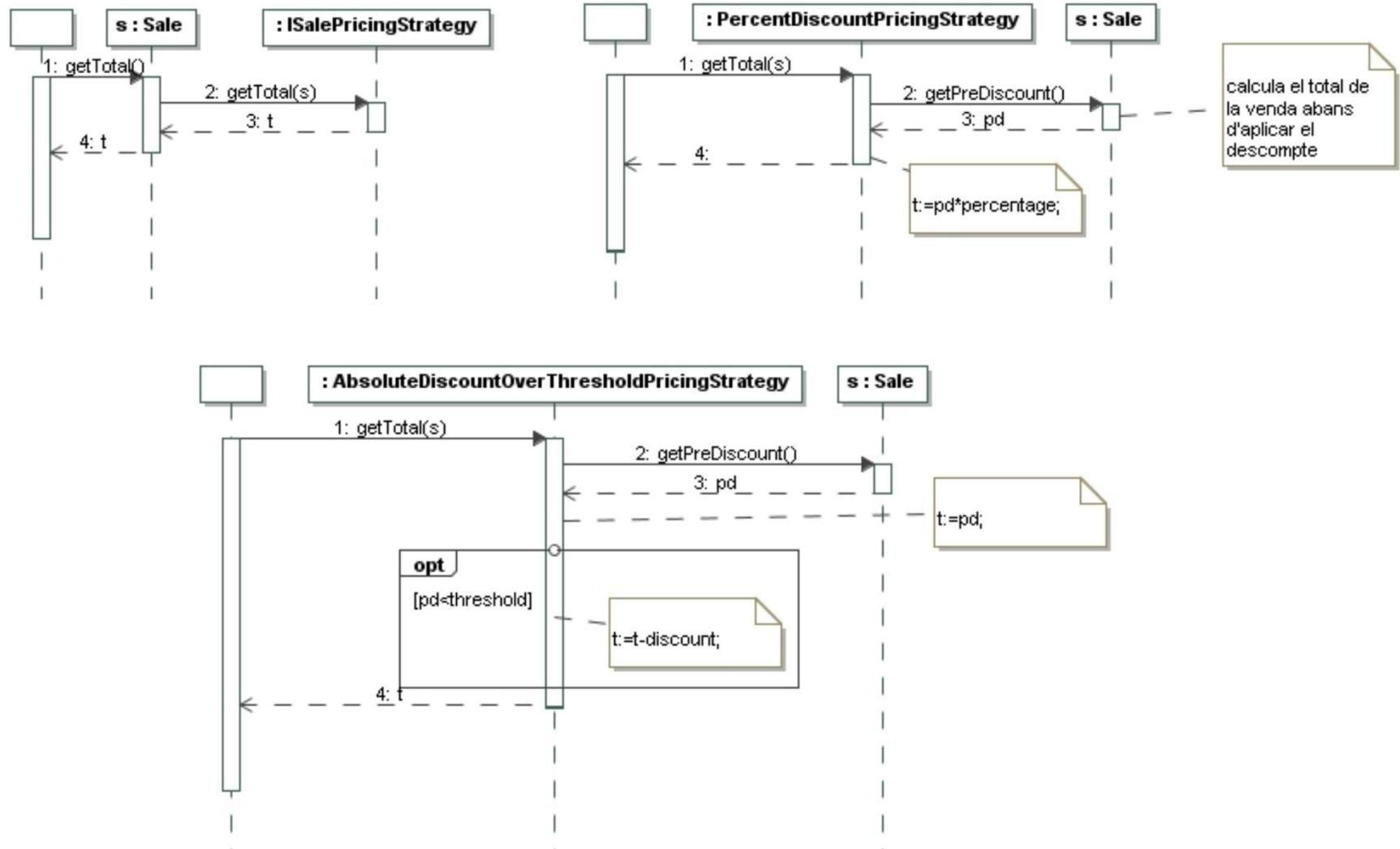
Example

- 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



Example



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
- <https://www.youtube.com/watch?v=v9ejT8FO-7I>
- <https://refactoring.guru/es/design-patterns/strategy>

State Pattern

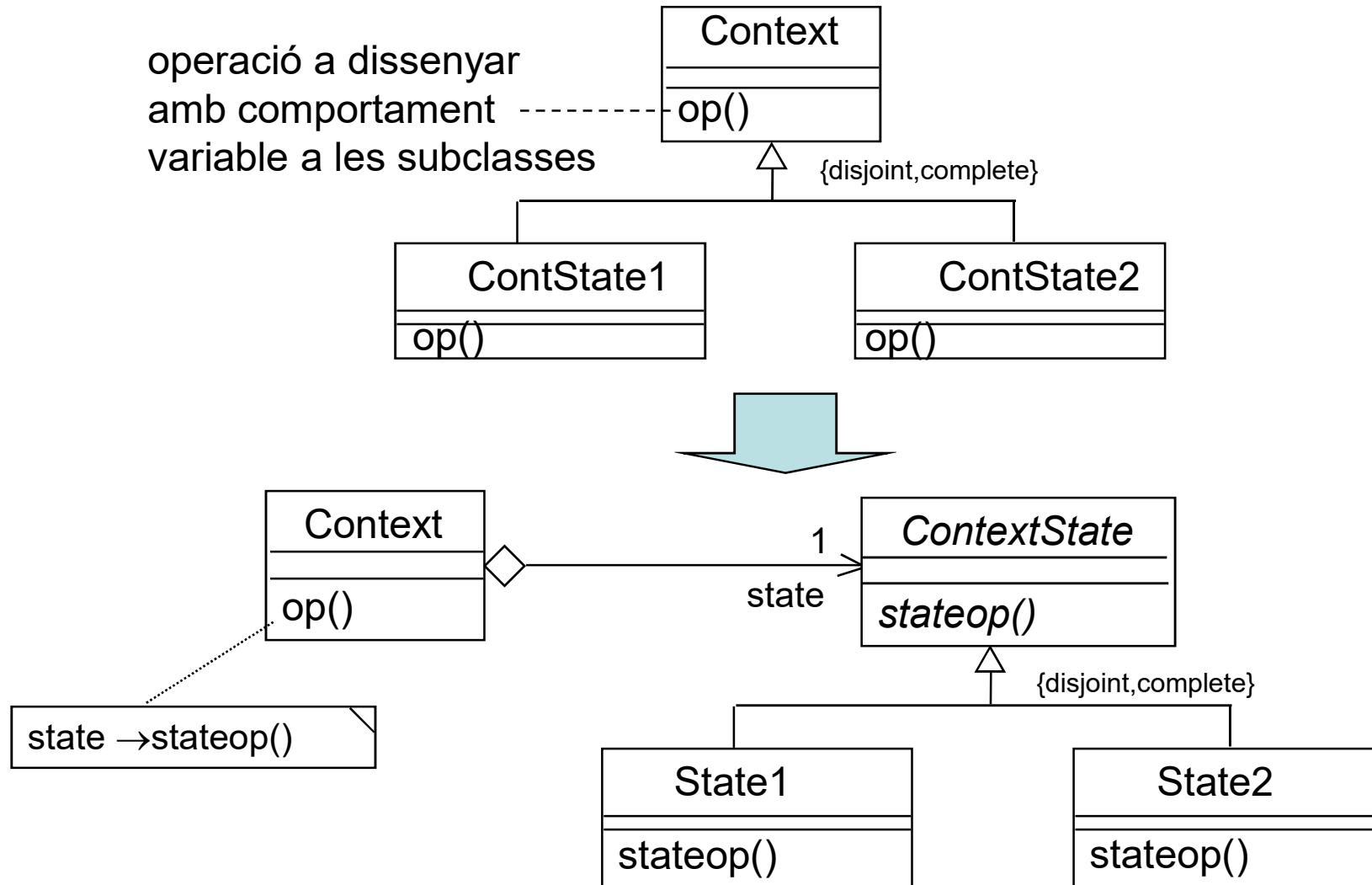
State Pattern

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

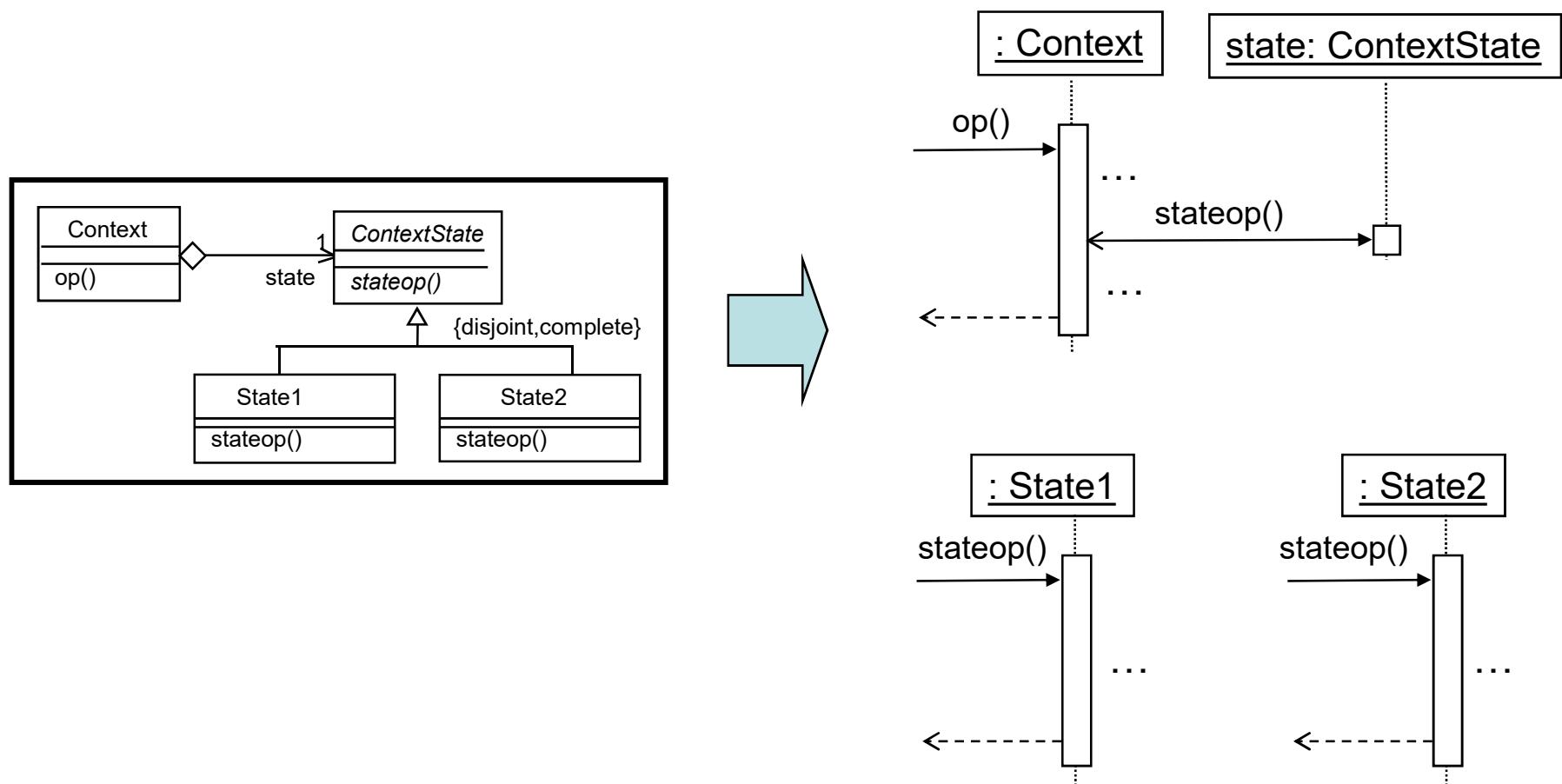
Overview

- Context
 - The definition of an operation in a hierarchy has different behavior for each subclass and the behavior of an object depends on the state of the object in execution time (and may vary over time).
- Problem
 - The technology does not allow an object to change its state in execution time. The use of conditional structures to deal with the behavior of an object depending on its state adds complexity and code duplication.
- Solution
 - Create a class for each state of a context object. Changing the state of an object is performed by changing the association between the context and its state.
 - Using polymorphic operations, assign methods to class states to define the behavior of the context object. When the context object receives a message (depending on its state), it resends the message to the state object.

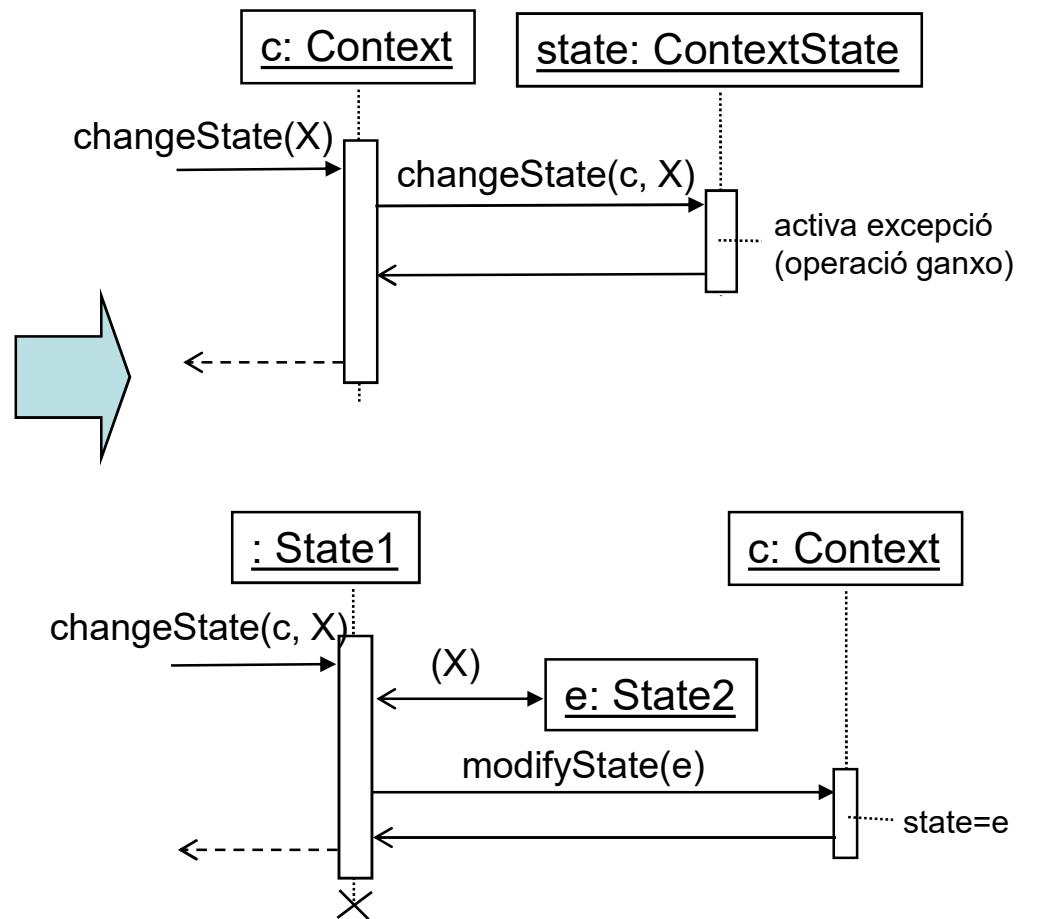
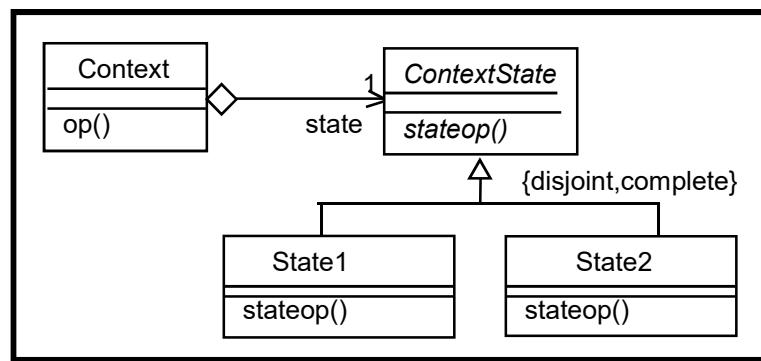
Static View



Dynamic View

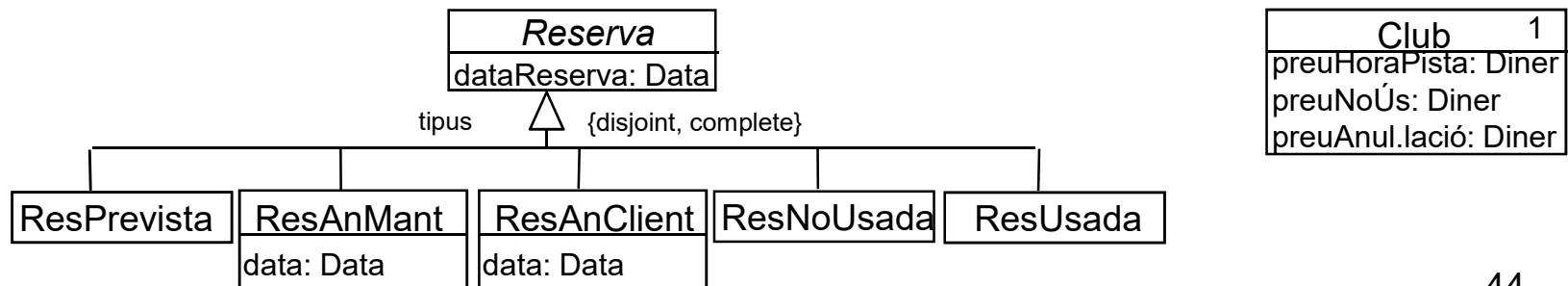


Dynamic View

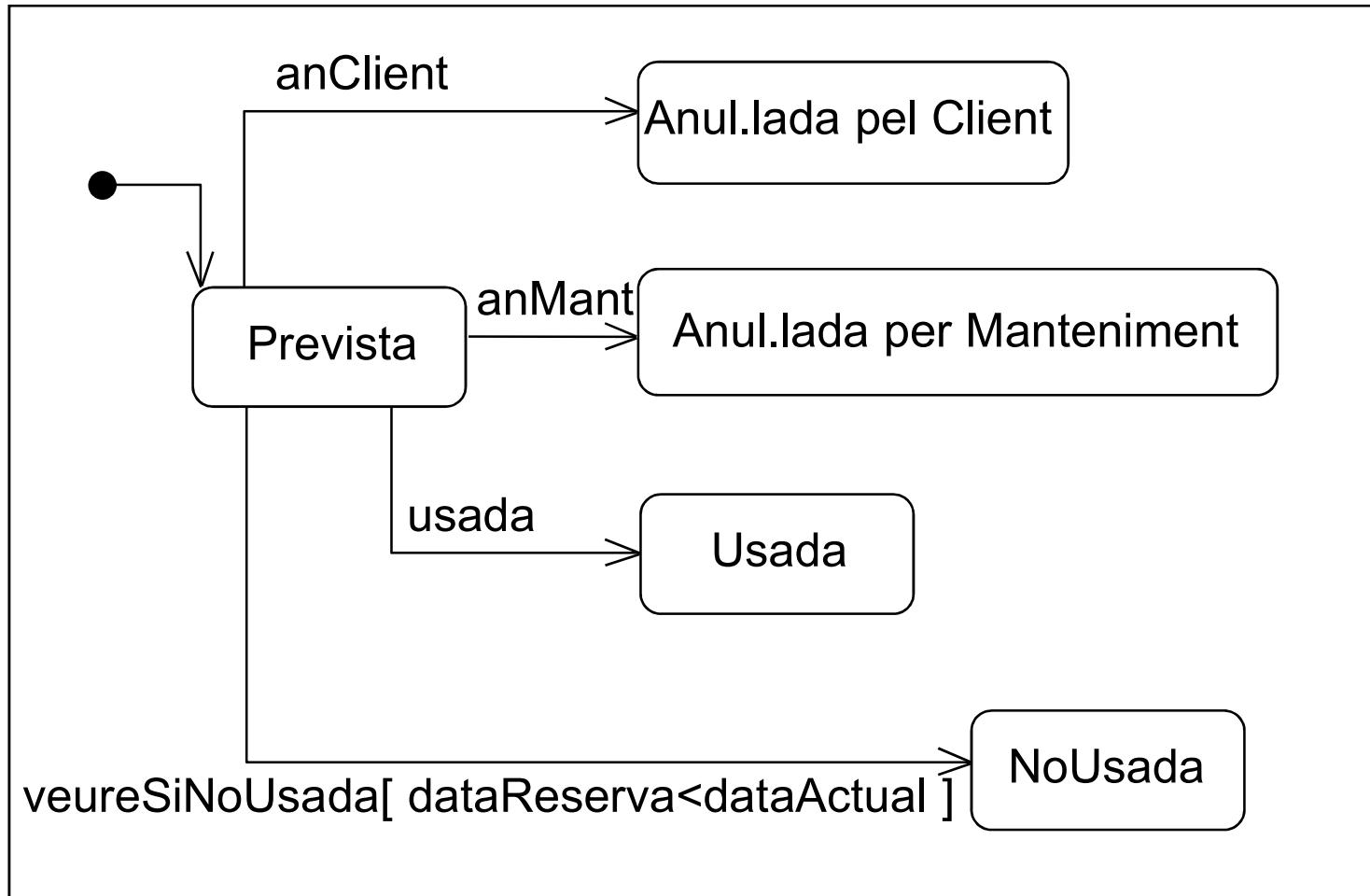


Example

- We want to design two operations to:
 - Cancel a tennis court reservation. The operation changes a scheduled reservation to a canceled reservation due to maintenance.
 - To calculate the cost of the reservation. The cost of a reservation is:
 - 0 if the reservation is pending or canceled.
 - Price per hour if the reservation is used.
 - Price per no use if the reservation has not been used.
 - Canceled price if the reservation has been canceled by the client.
- Esquema conceptual de l'especificació:

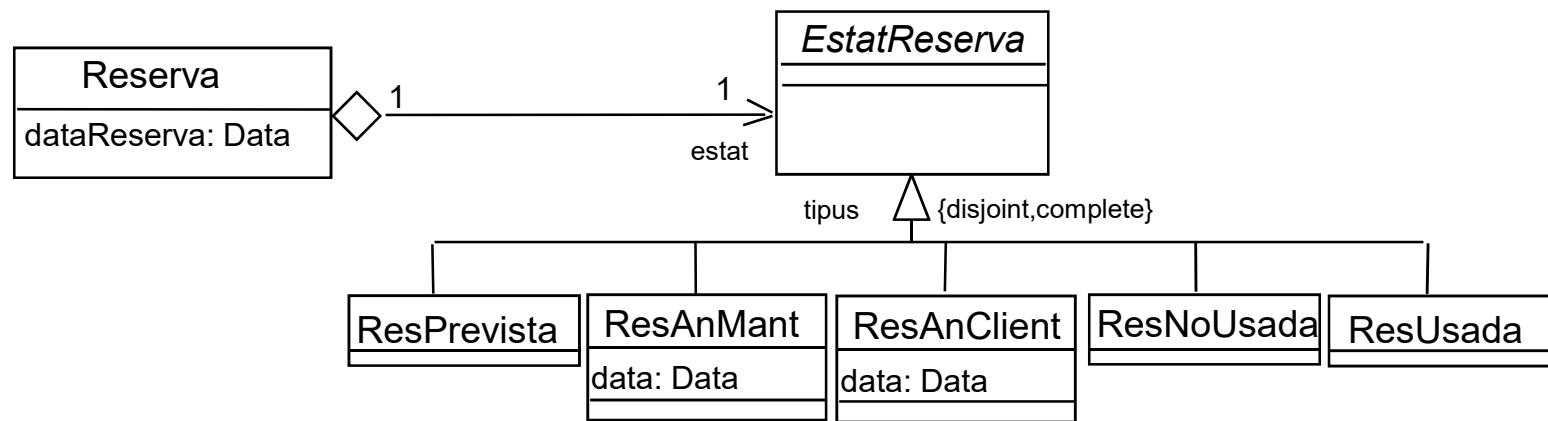


Example



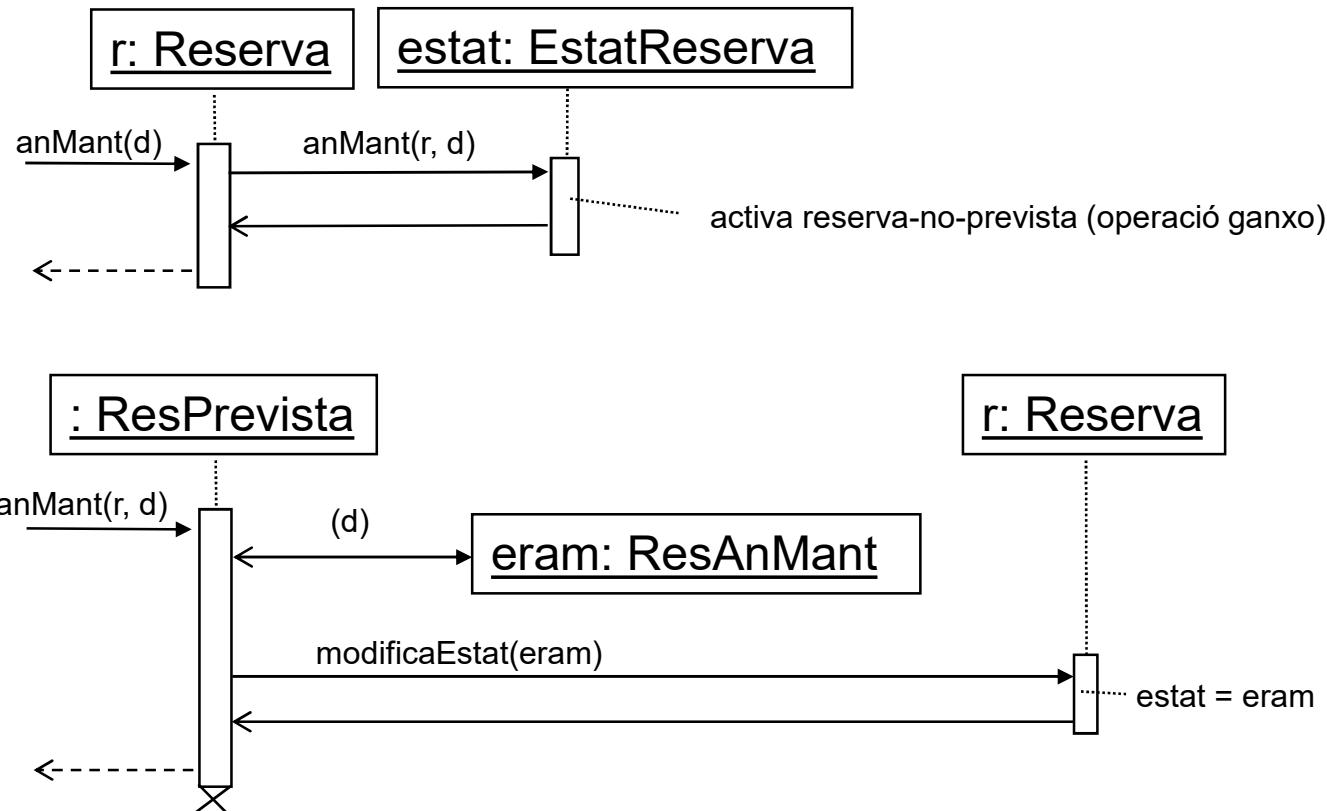
Example

- Applying State Pattern (Static View):



Example

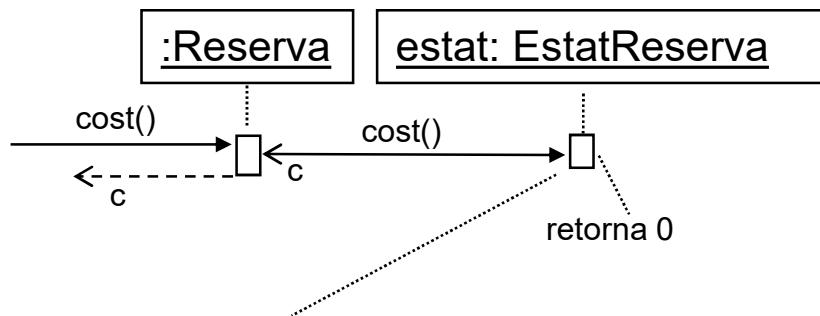
- Operation design: anMant(d:Data)



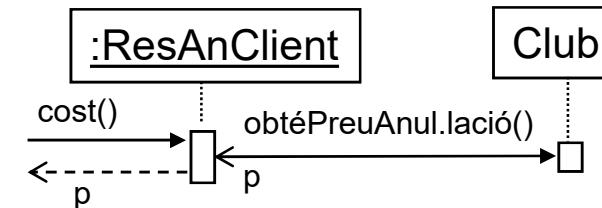
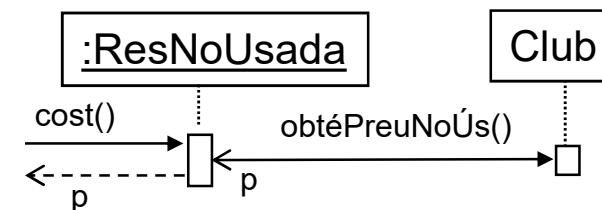
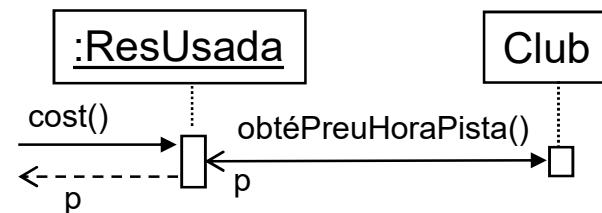
The operacions `anClient`, `usada` and `noUsada` are similars.

Example

- Operation design: `cost()`

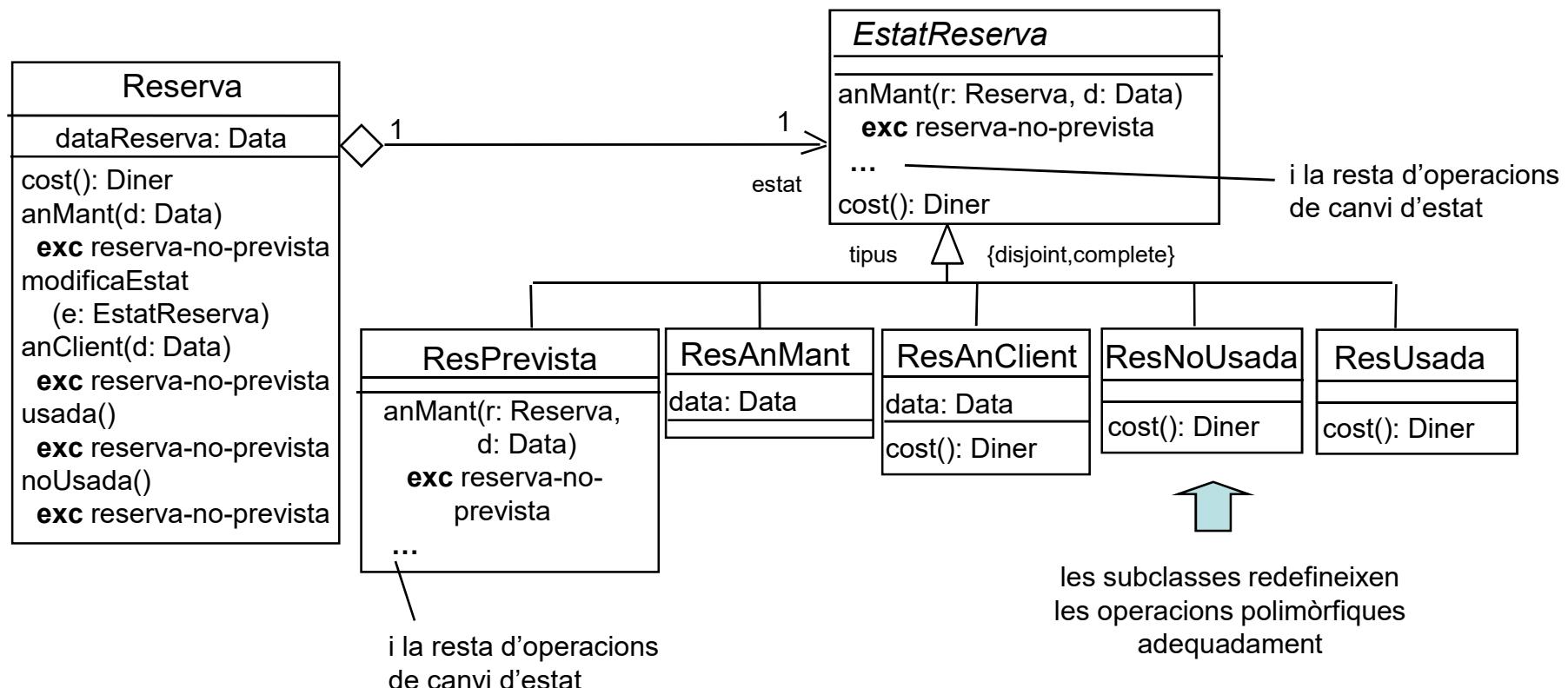


operació ganxo que es heretada per totes les subclasses i redefinida a ResUsada, ResNoUsada i ResAnClient.



Example

- Resulting class diagram



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
- <https://www.youtube.com/watch?v=N12L5D78MAA>
- <https://refactoring.guru/es/design-patterns/state>

Template Method Pattern

Template Method Pattern

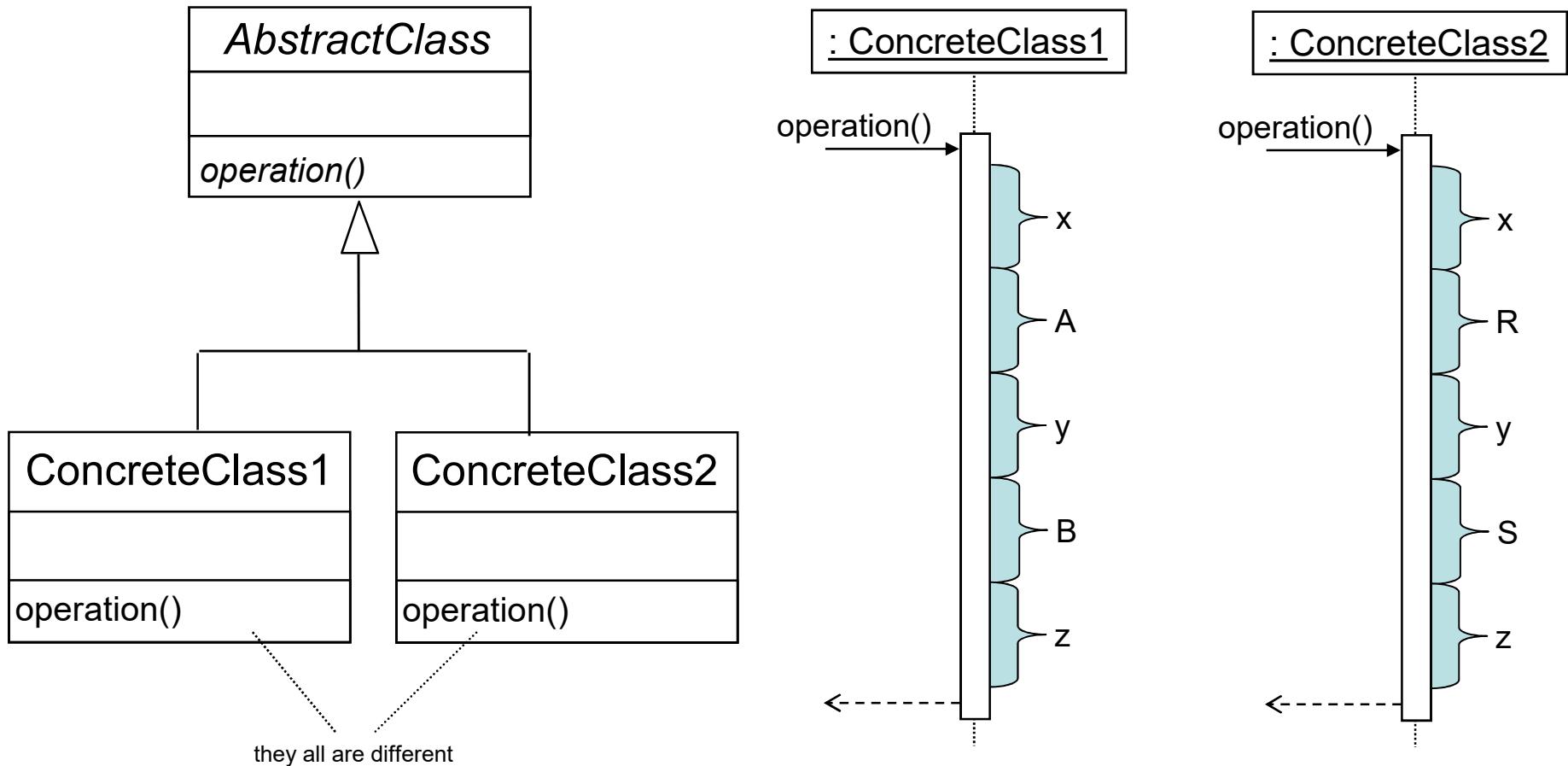
- Overview
- Static View
- Dynamic View
- Example
- References

Overview

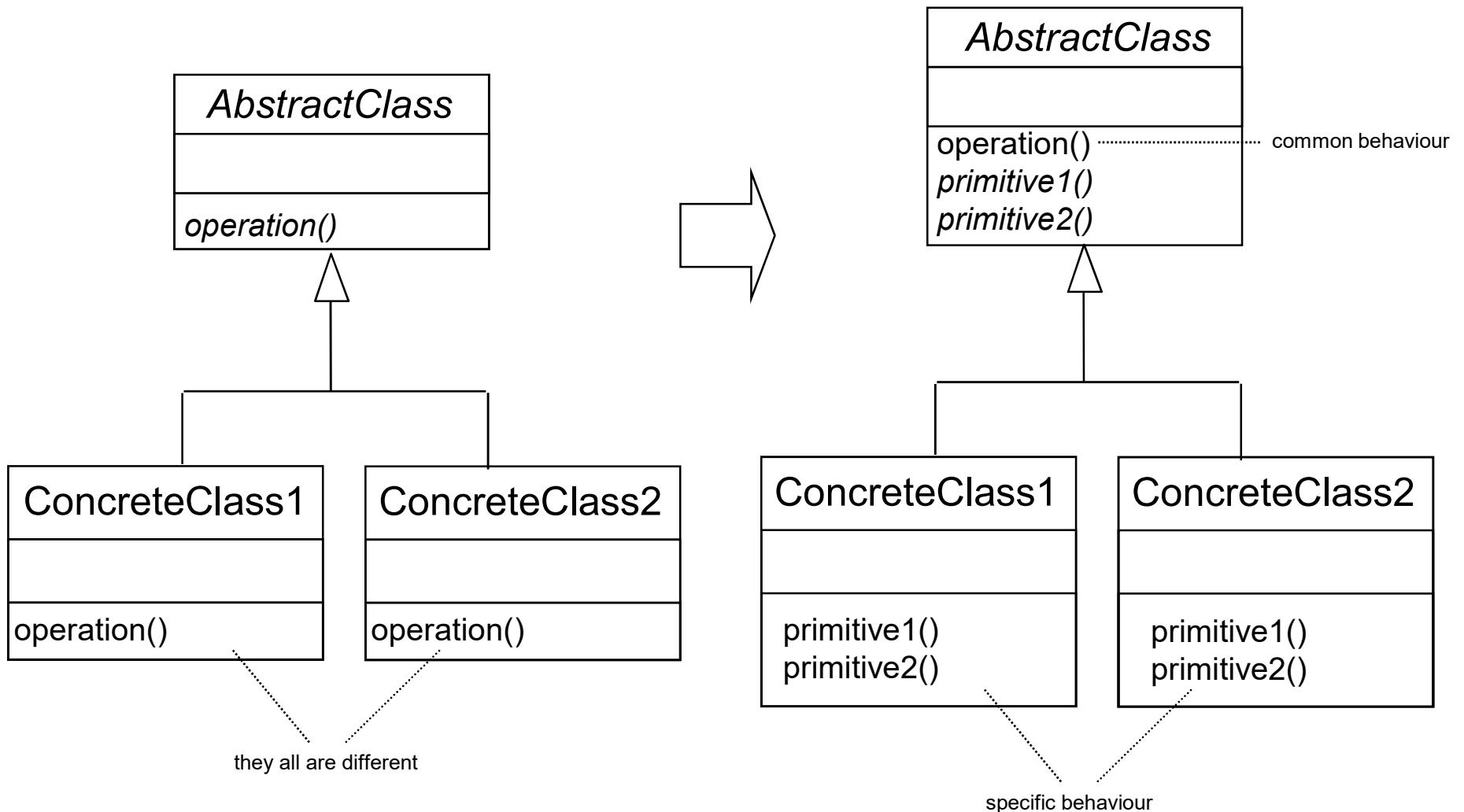
- Context
 - The definition of an operation in a hierarchy has some common behaviour to all subclasses but also some specific behaviour for each of them.
- Problem
 - Replicating the common behaviour in all subclasses requires code duplication and therefore a more costly maintenance.
- Solution
 - To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

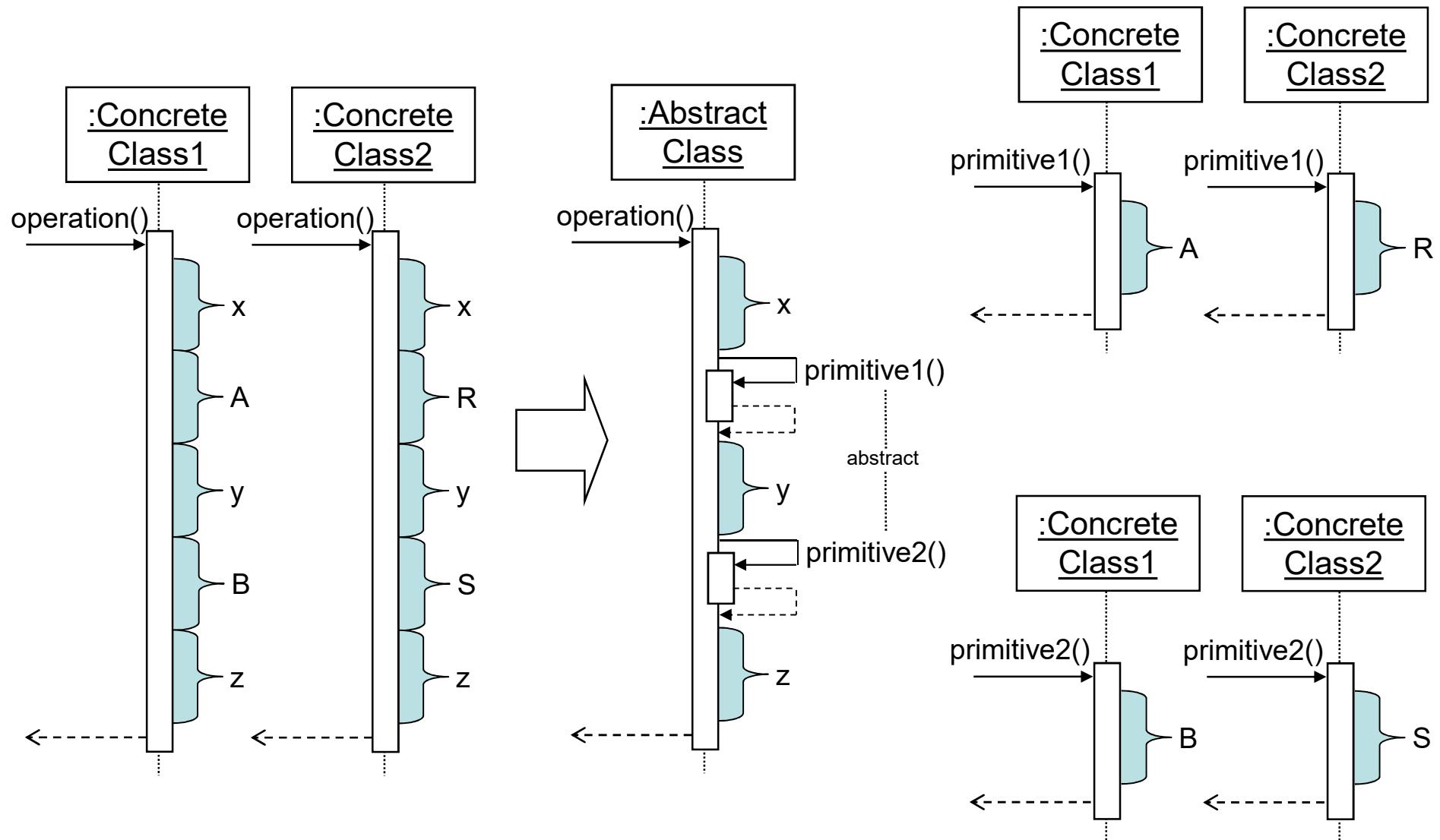
- Context: Starting situation



Static View

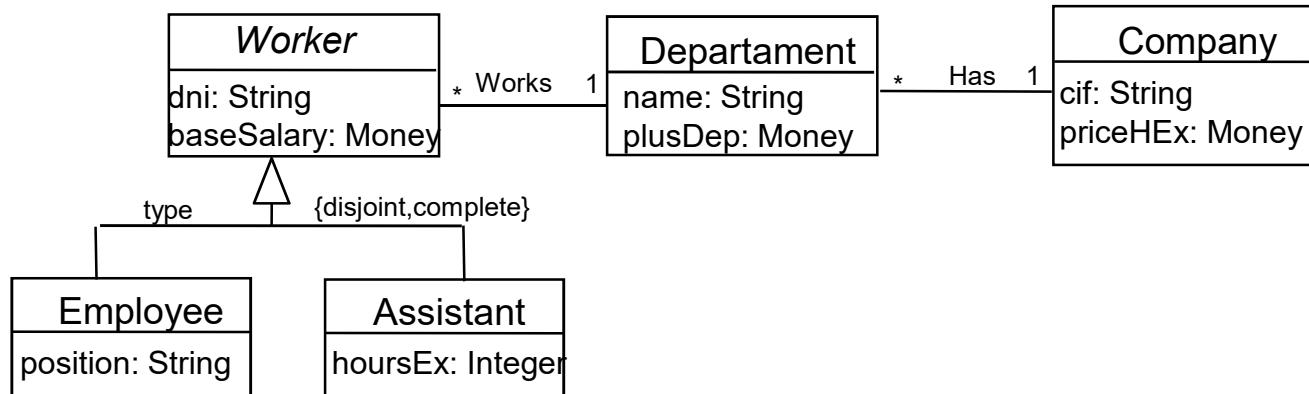


Dynamic View



Example

- Specification data conceptual model:



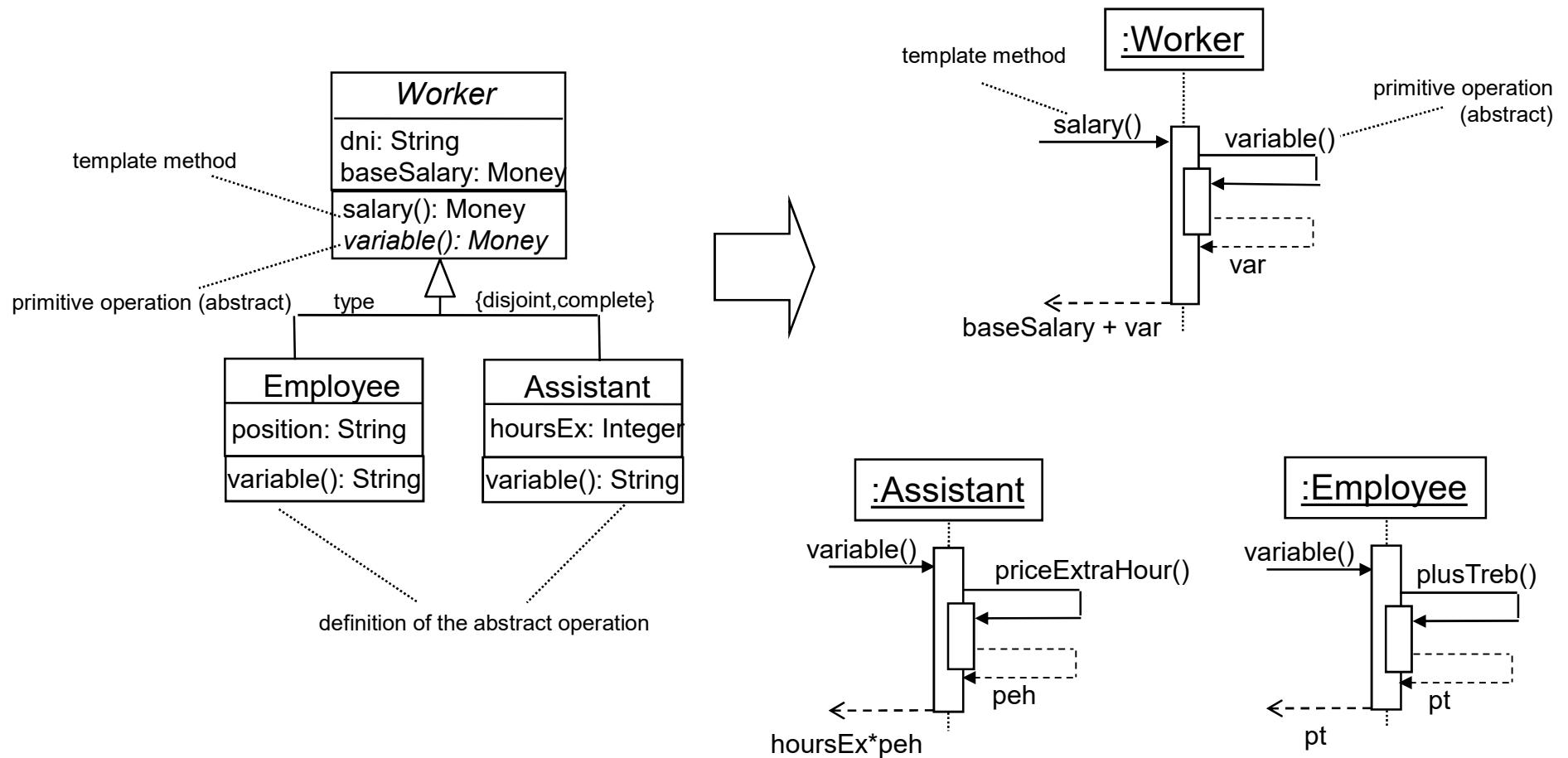
Textual Integrity Constraints:

Identifiers: (`Worker`, `dni`); (`Company`, `cif`)

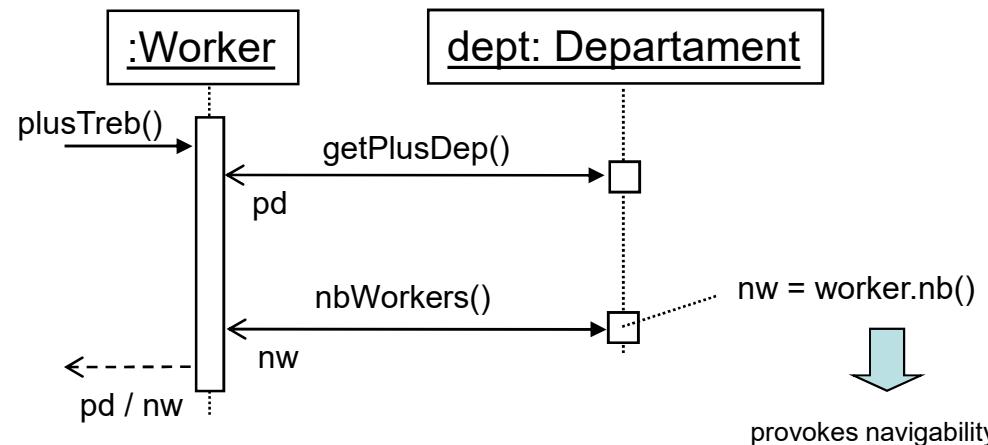
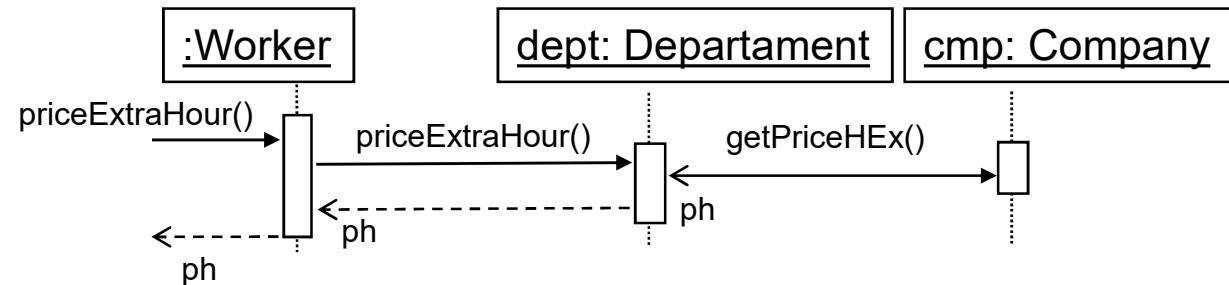
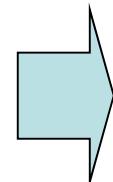
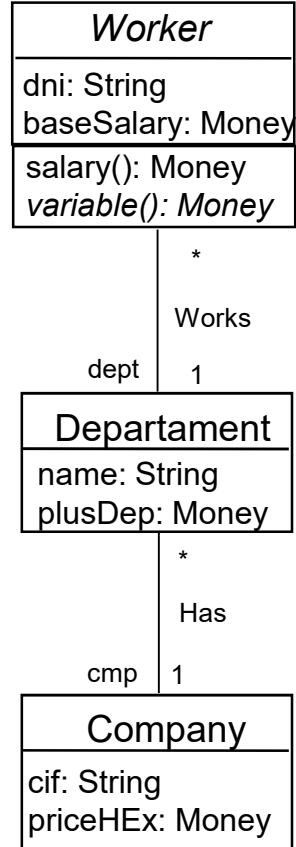
There cannot exist two departments with the same name in the same company

- 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* + $plusDep / \text{number of } Workers$

Example



Example



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
- <https://www.youtube.com/watch?v=7ocpwK9uesw>
- <https://refactoring.guru/es/design-patterns/template-method>