Software Engineering

Part 2: UML & Gestion de version

Guillaume Swaenepoel & Thomas Aubin

Design principle

How to make a clean code

Design Principle

- Separation of concerns
- Factoring
- Don’t Repeat Yourself
- Keep it simple, stupid
- You Ain’t Gonna Need It
- Maximal encapsulation

Separation of concerns

Maybe the most important design principle.

It’s a design principle for separating a computer program into distinct sections, each sections must a precise concern.

Building your software in a modular way will help you and others to understand your project.

Example:

A part who is in charge of screen display must not contain code used to access data in model.

A class which aims to have to reasons to change must be split in two different class.

Decomposition (Factoring)

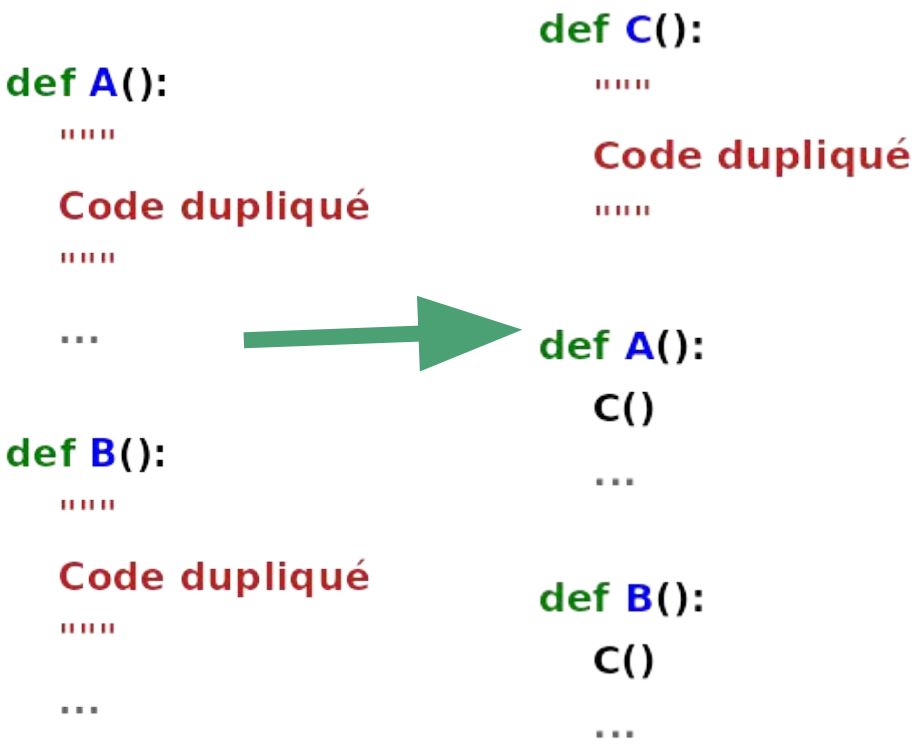
- Do not rewrite something already existing.
- The developers lost a long time by rewriting their program entirely.
- The OpenSource world offer a lot of software component tested and approved by a lot of developer. Most language offer a code package system to import easily this components in your software. (Maven in Java, NPM in NodeJS, Pypi in Python, ...)

Don’t Repeat Yourself (DRY)

Avoid copy and past of your code in your code.

Try to factorize your code as much as possible.

Your code will be easier to update and to test. A lot of bug appear because of repetition of code not updated.

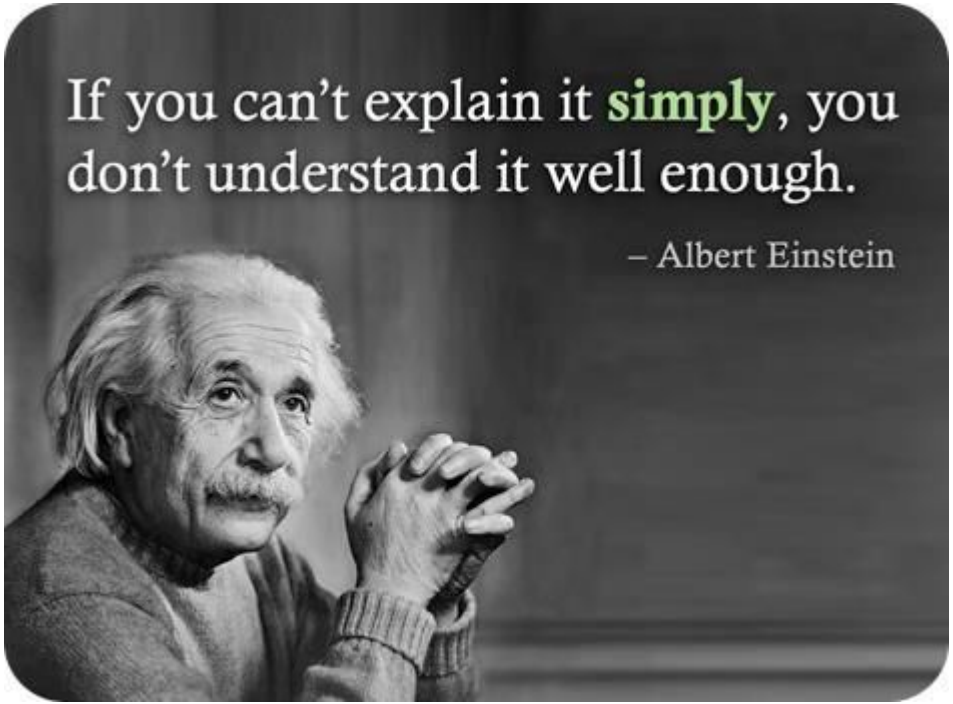


Keep It Simple, Stupid

Your code must the the most simple possible.

Complex code are often difficult to explain and maintain. Team work can be slowed down.

You must think of your code organization before starting development.

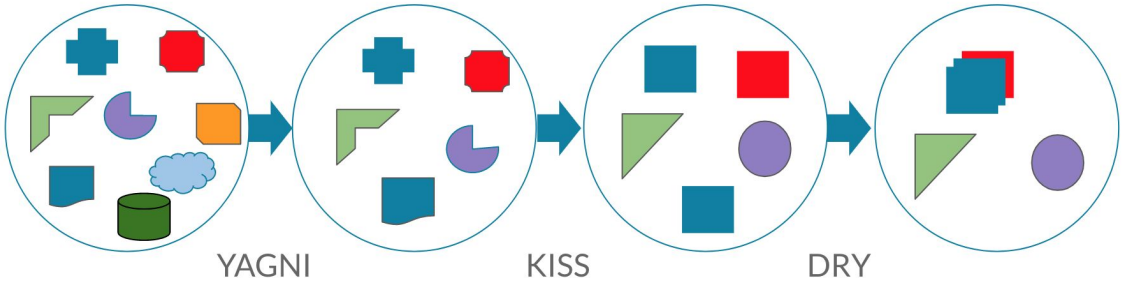


You Ain’t Gonna Need It (YAGNI)

Useless functions or class must not be keep for “evolution” in your code. This will make code reading more complex.

Conclusion

Follow these principles, will allow you to write better code ! A clean code is easier to maintain, easier to understand and for sure it will save your time when you need to change or implement something. Avoid use duplicated code, try to keep your code as simple as possible, and just implement features when it’s really necessary.



UML

Modelisation language

UML

The UML is a visual language build to provide a standard way to visualize the design of a system.

UML diagrams

UML 2.0 defines thirteen types of diagrams, divided into three categories:

Structure Diagrams include the Class Diagram, Object Diagram, Component Diagram, Composite Structure Diagram, Package Diagram, and Deployment Diagram.

Behavior Diagrams include the Use Case Diagram (used by some methodologies during requirements gathering); Activity Diagram, and State Machine Diagram.

Interaction Diagrams, all derived from the more general Behavior Diagram, include the Sequence Diagram, Communication Diagram, Timing Diagram, and Interaction Overview Diagram.

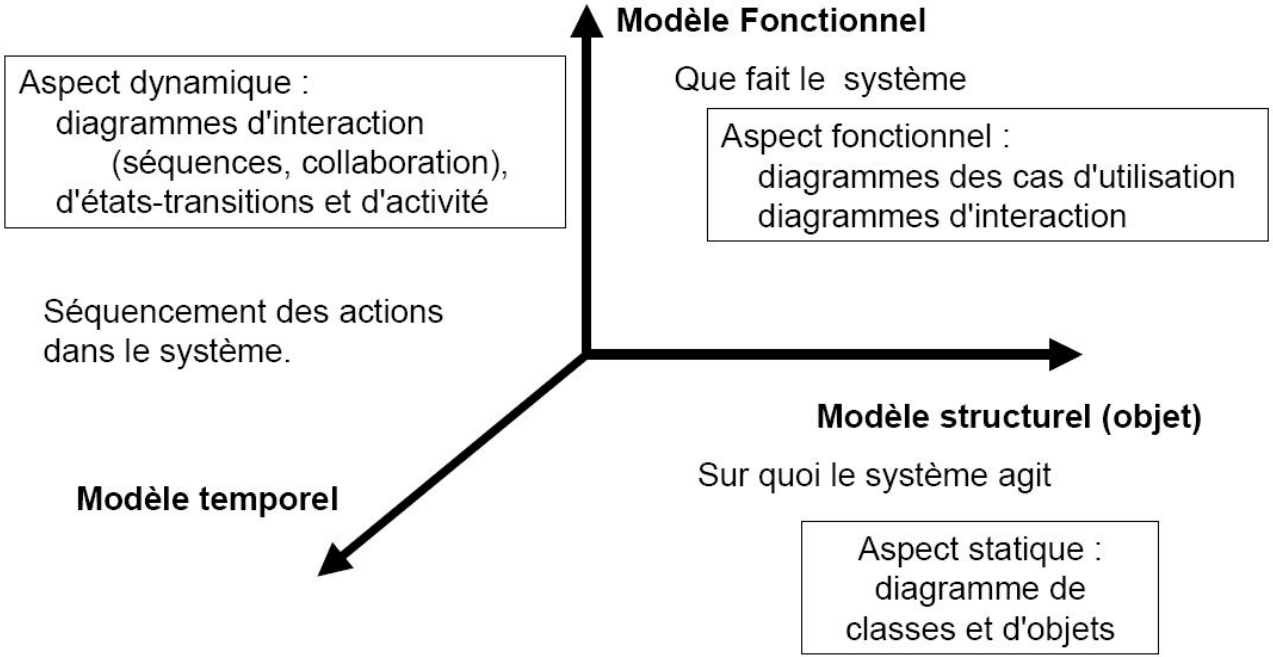
Why model in UML ?

A model is a simplification of the reality to help us to better understand the way to develop.

It allows:

- Visualize the system as it should be.
- Validation of the model with the client.
- Visualize data structures and system behavior.
- Provide a guide for building the software.
- System documentation and decisions justifications.

Model components

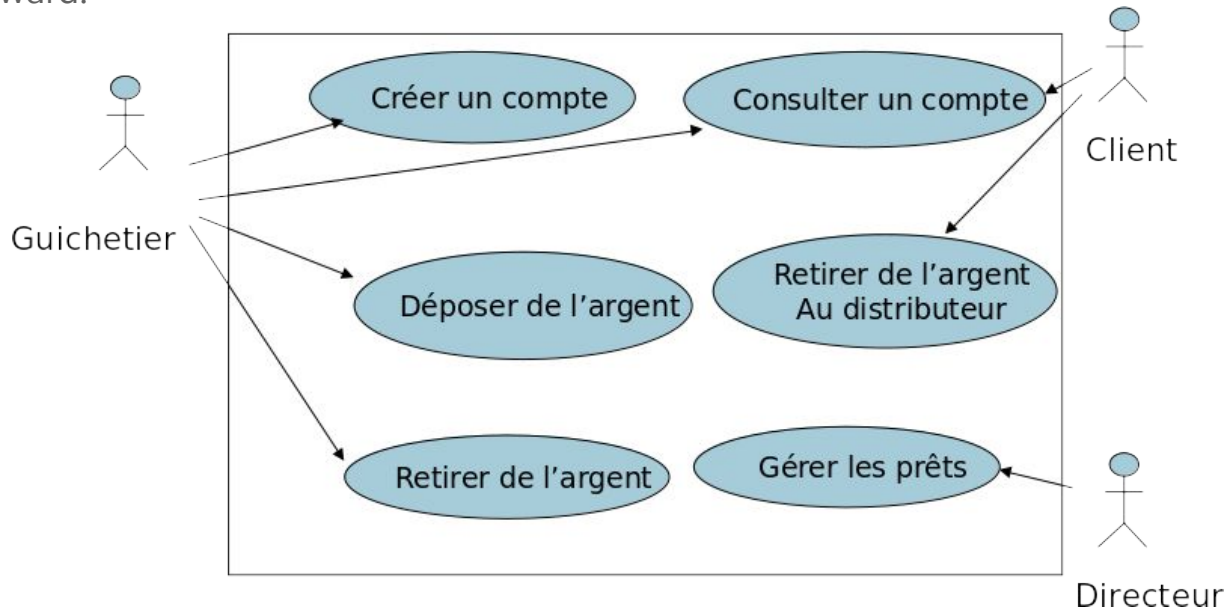


Use case Diagram

The goal is to understand customer needs to write specifications.

The Use case diagram must put forward:

- Main uses of the system.
- System environment.
- System limits



Case study 1 : Use case diagram

Le cas d'étude concerne un système simplifié de distributeur d'argent.

Il offre les services suivants:

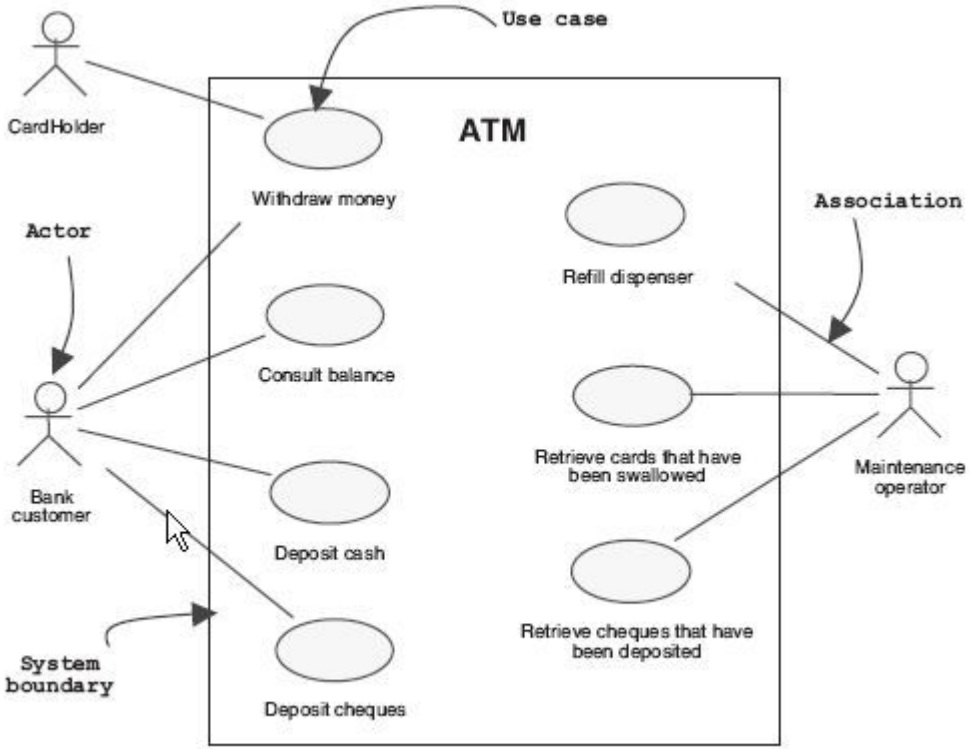
1. Distribution d'argent à tous les détenteurs d'une carte de crédit via un lecteur de carte d'une agence bancaire
2. Consultation du solde du compte , possibilité de déposer de l'argent liquide et d'encaisser des chèques.

Penser aux éléments suivants:

1. Toutes les transactions doivent être sécurisées.
2. Il est nécessaire de recharger l'automate bancaire en argent liquide.

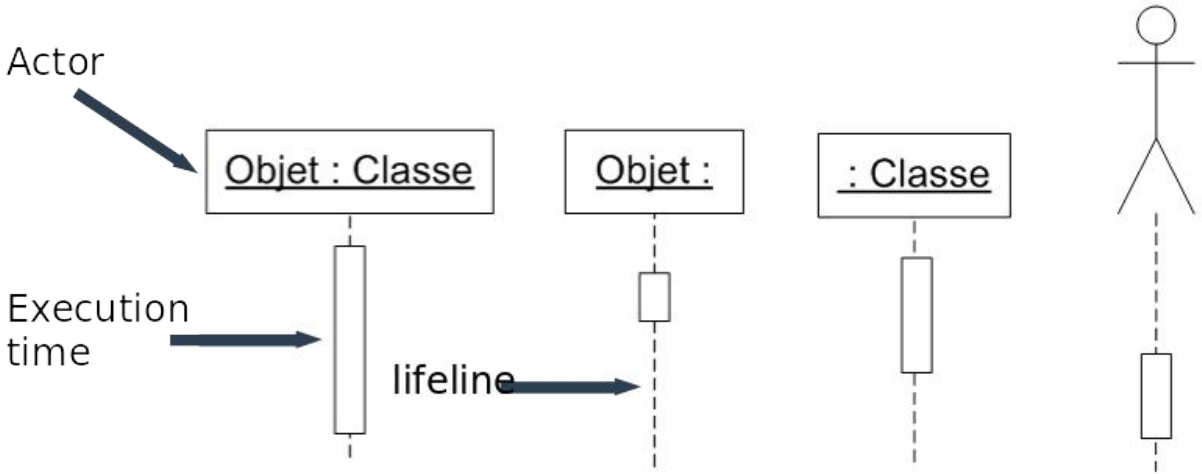
Écrire le diagramme d'utilisation associé à ce cas.

Case study 1 : Use case diagram

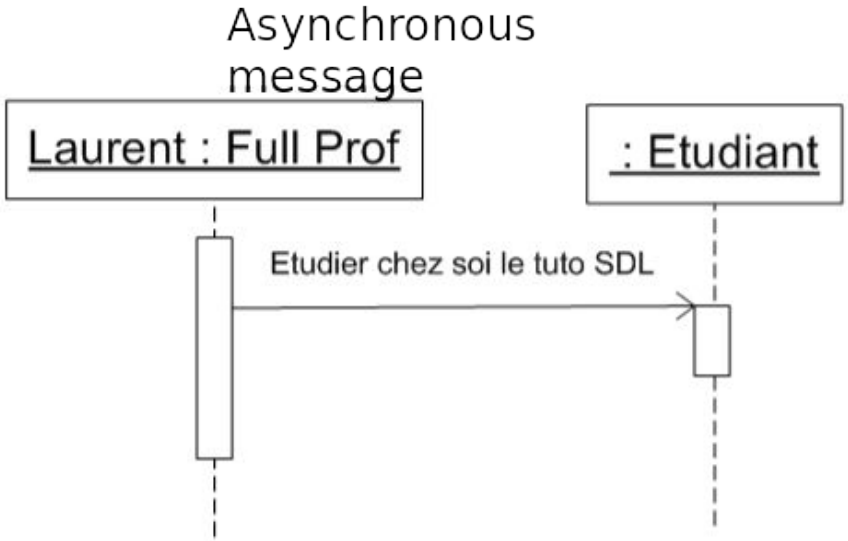


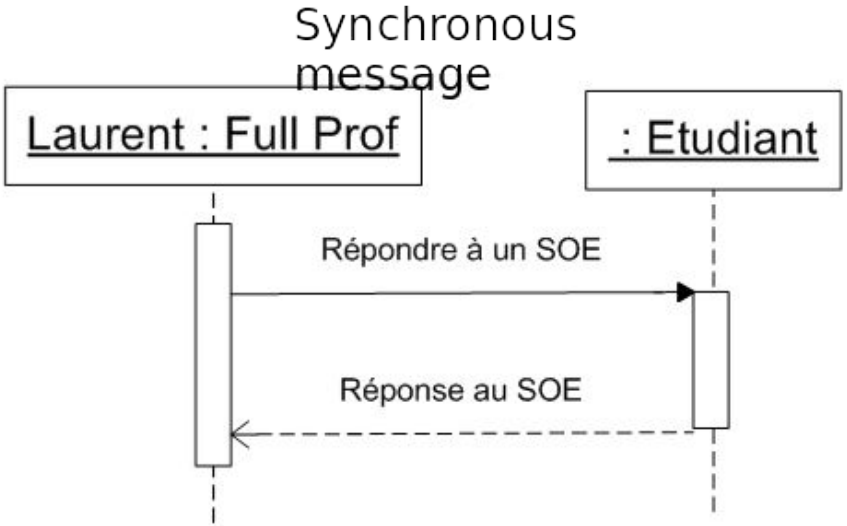
Sequence diagram

Sequence diagram are “more oriented” with independent software components (objects) that communicate (through method calls). This is a temporal representation of exchange inside the system. Exchange are represented chronologically from top to bottom.



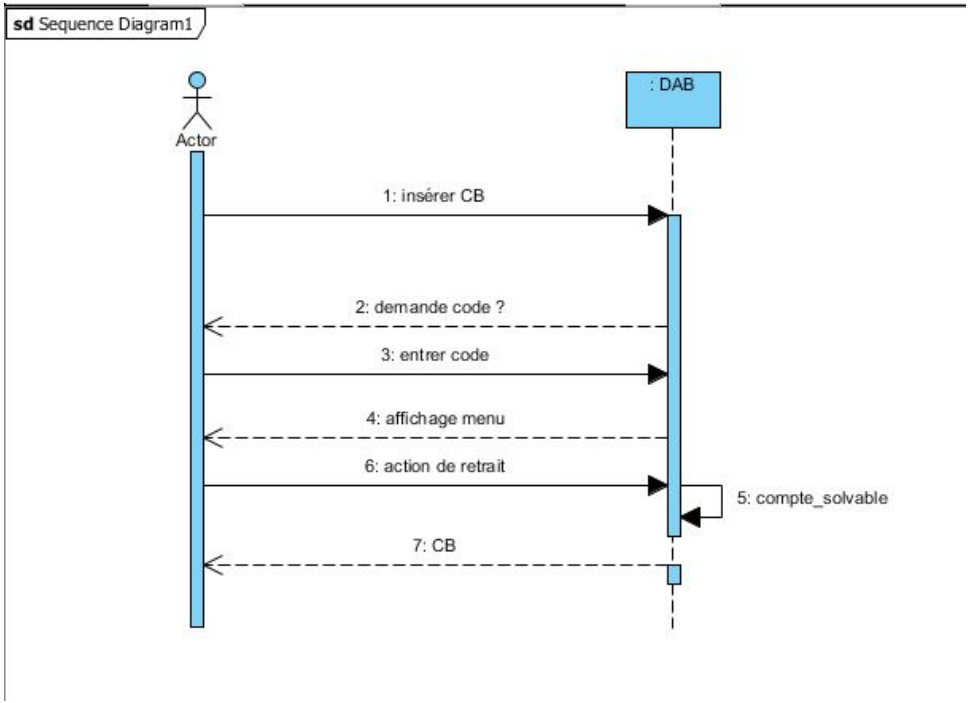
Sequence diagram





Study case 2: Sequence diagram

From previous study case, we want to write a sequence diagram of an actor taking money in cash distributor.



Class diagram

This is maybe the most important diagram but also the most complicated part, because this is where you get down to the tiny detail.

Unlike use case diagram, diagram class show the structure of services inside the software. This is an abstract representation of system objects which interact together to realize a use case.

Main elements are classes and there relationship:

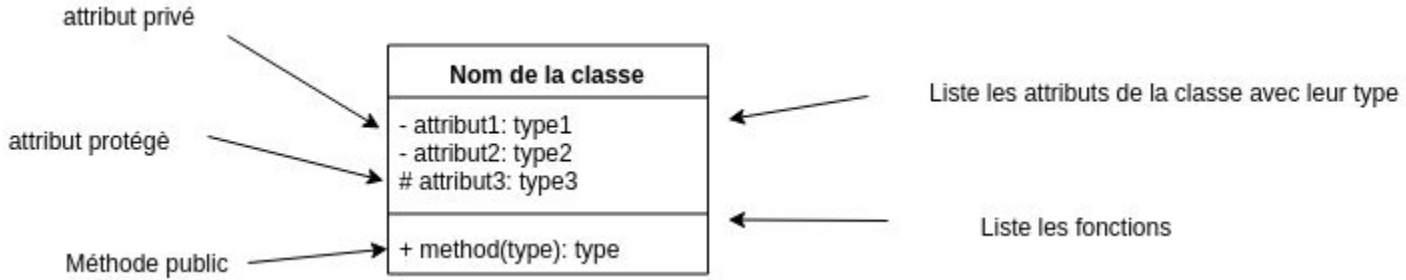
- Inheritance: generalization/specialization relationship
- association: relation between objects of a class.
- Composition: capacity and belonging relationship
- Dependencies: relation between two or more classes in which a change in one may force change in the other.

Class diagram: class

The classes in a class diagram represent both the main elements, interactions in the application, and the classes to be programmed.

In the diagram, classes are represented with boxes that contain three compartments:

- The top compartment contains the name of the class. It is printed in bold and centered, and the first letter is capitalized.
- The middle compartment contains the attributes of the class. They are left-aligned and the first letter is lowercase.
- The bottom compartment contains the operations the class can execute. They are also left-aligned and the first letter is lowercase.



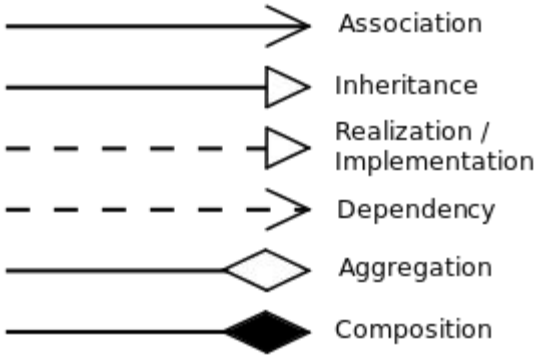
Class diagram: relationship

If a class is defined, it is not necessary to represent its properties.

A relation is represented as a line. An association is named, and the ends of an association can be adorned with role names, ownership indicators, multiplicity, visibility, and other properties.

Multiplicity:

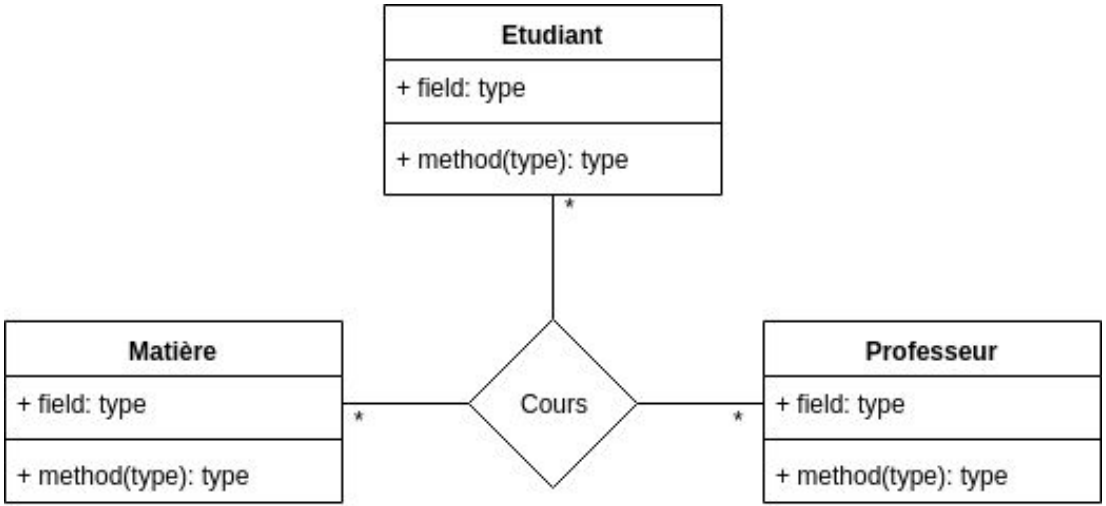
- exactly one: 1 or 1..1 ;
- many : * or 0..* ;
- At lease one: 1..* ;
- Between one and six : 1..6.



Class diagram: binary association

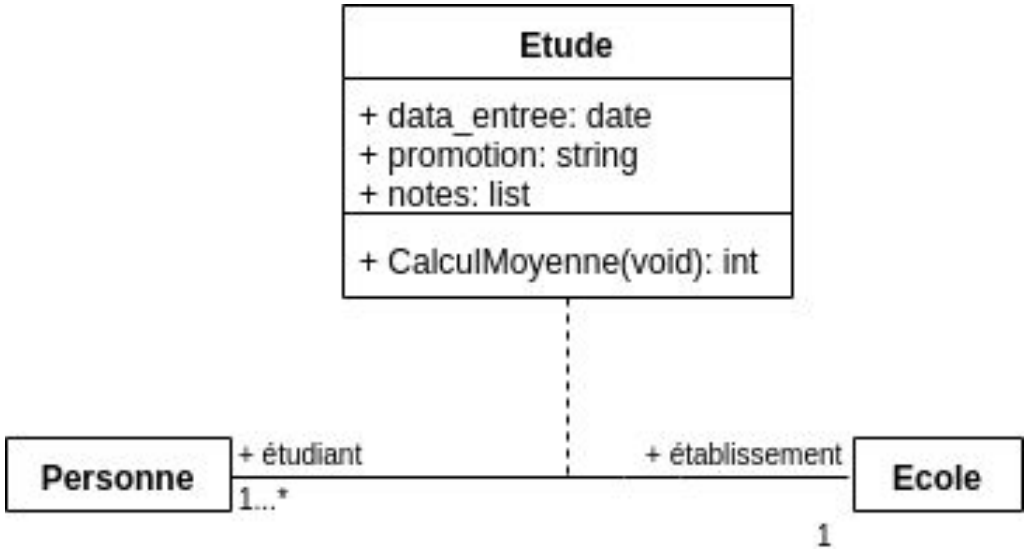
At least one student is studying in the school.

Class diagram: ternary association



Class diagram: association class

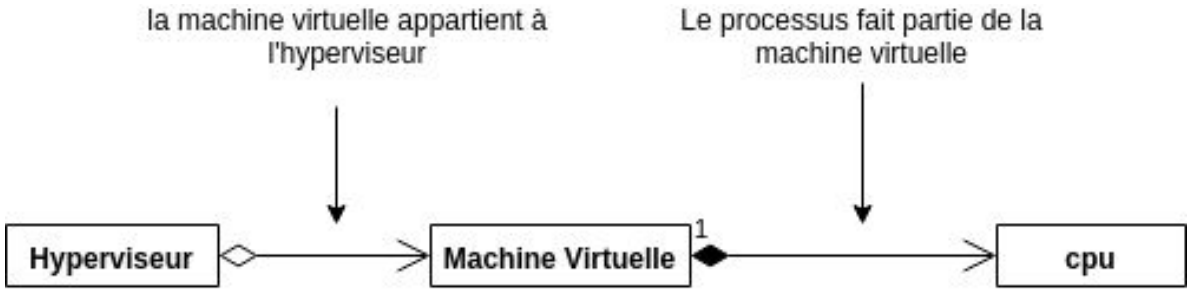
An association class is a class that is part of an association relationship between two other classes



Class diagram: aggregation and composition

An aggregation has no name (contains is implicit). In this type of relationship the aggregate contains from 0 to an unspecified number of items, but an item can also appear in several aggregates.

A composition is a particular case of aggregate with stronger constraints. An item can only belong to one composite. Additionally, the item has no life of its own outside the composite. If the composite is deleted, all the items it contains disappear with it.

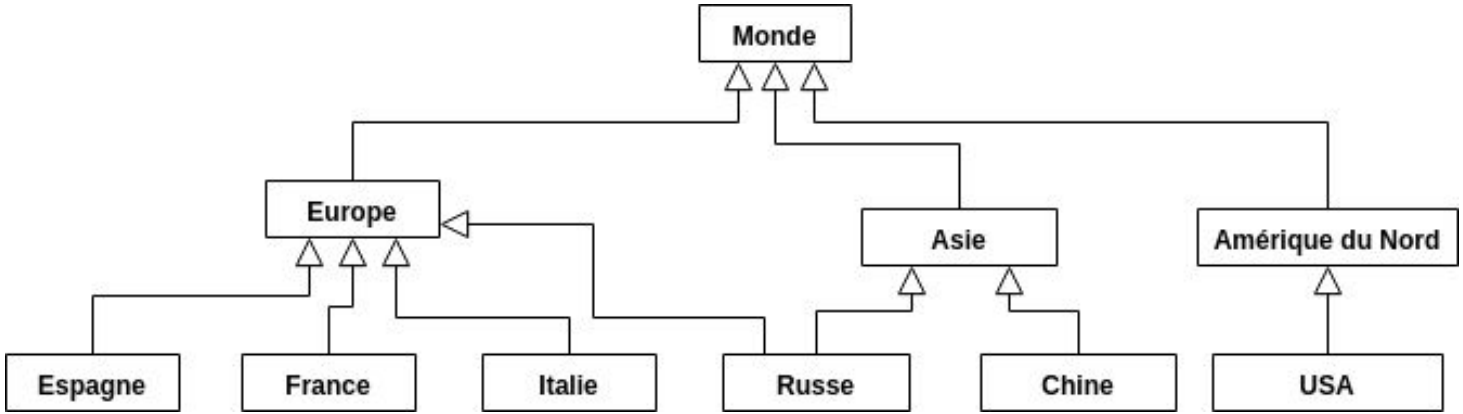


Class diagram: inheritance

It indicates that one of the two related classes (the subclass) is considered to be a specialized form of the other (the super type) and the superclass is considered a Generalization of the subclass.

The superclass (base class) in the generalization relationship is also known as the "parent", superclass, base class, or base type.

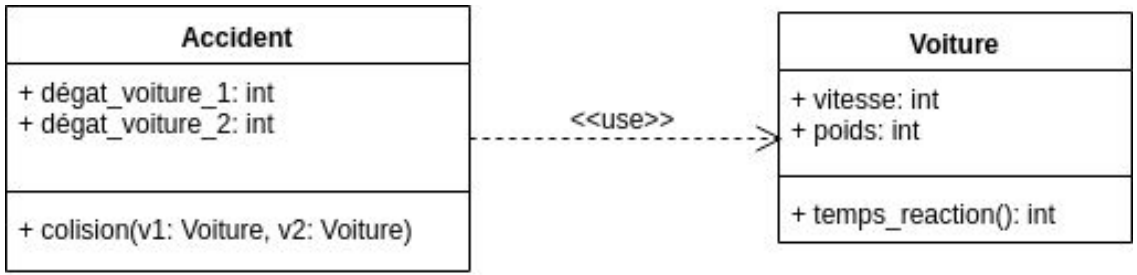
The subtype in the specialization relationship is also known as the "child", subclass, derived class, derived type, inheriting class, or inheriting type.



Class diagram: dependency

Dependency is a weaker form of bond that indicates that one class depends on another because it uses it at some point in time.

Dependencies are often used to put forward the of a class by another.



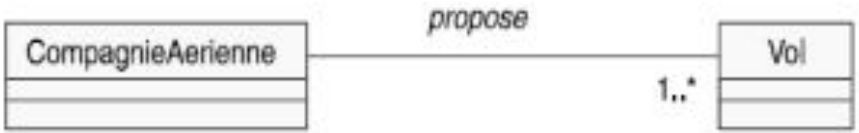
Case study 3: Class diagram

Modéliser dans un diagramme de classe le logiciel de réservation de vol répondant aux contraintes suivants:

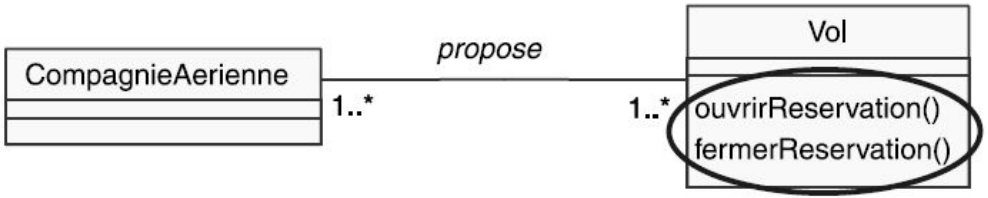
1. Des compagnies aériennes proposent différents vols.
2. Un vol est ouvert à la réservation et refermé sur ordre de la compagnie.
3. Un client peut réserver un ou plusieurs vols, pour des passagers différents.
4. Une réservation concerne un seul vol et un seul passager.
5. Une réservation peut être annulée ou confirmée.
6. Un vol a un aéroport de départ et un aéroport d'arrivé.
7. Un vol a un jour et une heure de départ, et un jour et une heure d'arrivée.
8. Un vol peut comporter des escales dans des aéroports.
9. Une escale a une heure d'arrivée et une heure de départ.
10. Un vol dessert une ou plusieurs villes.

Case study 3: Class diagram

1) Des compagnies aériennes proposent différents vols.

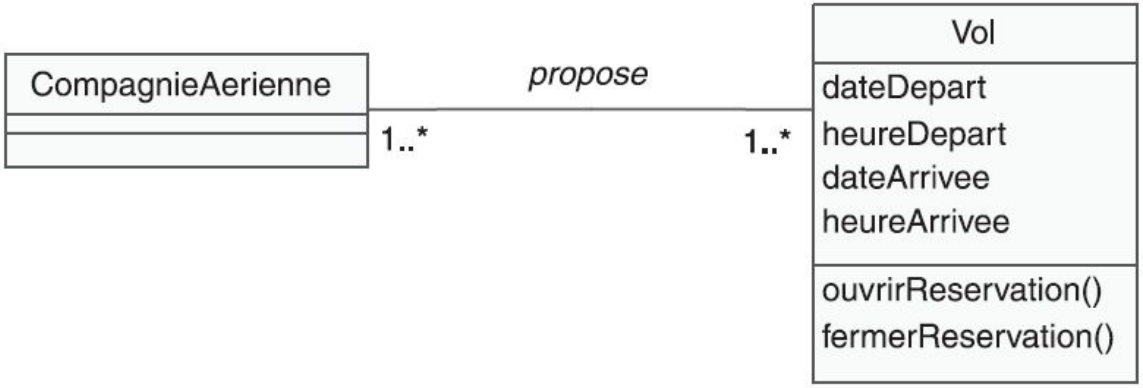


2) Un vol est ouvert à la réservation et refermé sur ordre de la compagnie.



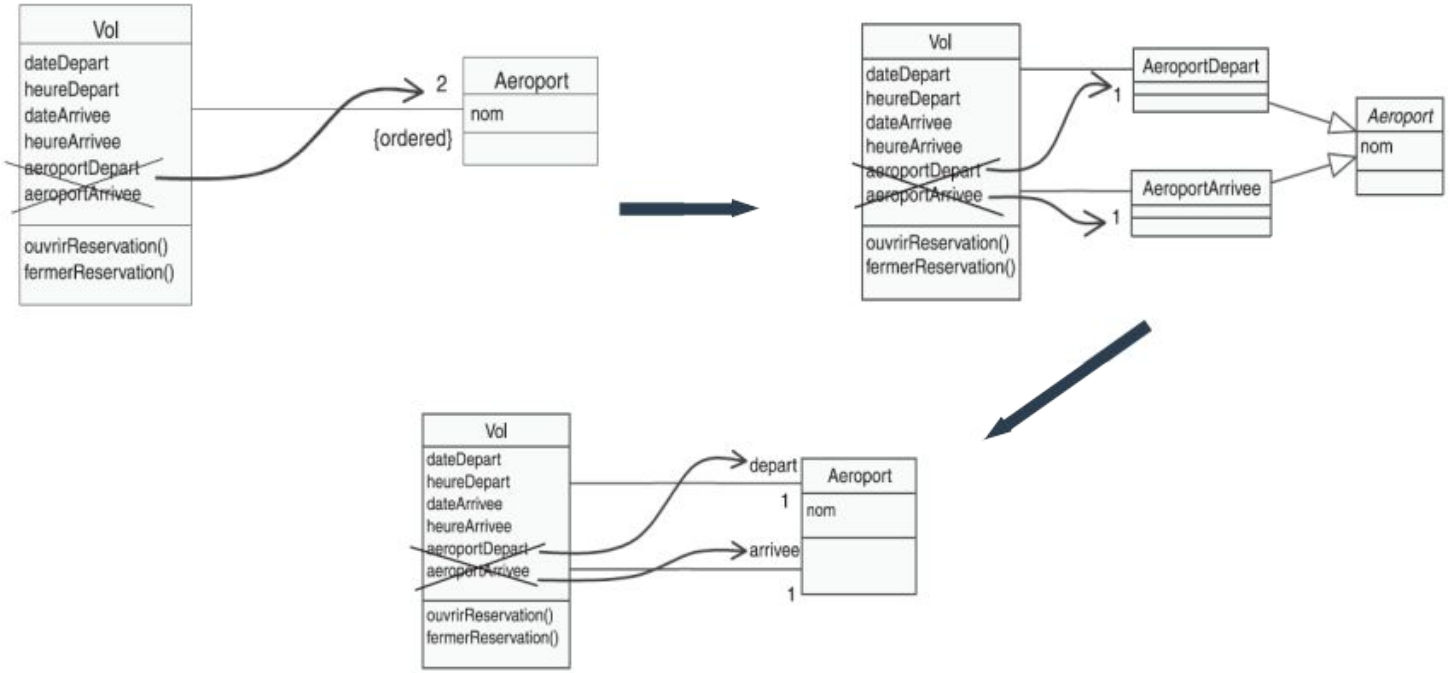
Case study 3: Class diagram

7) Un vol a un jour et une heure de départ, et un jour et une heure d'arrivée.



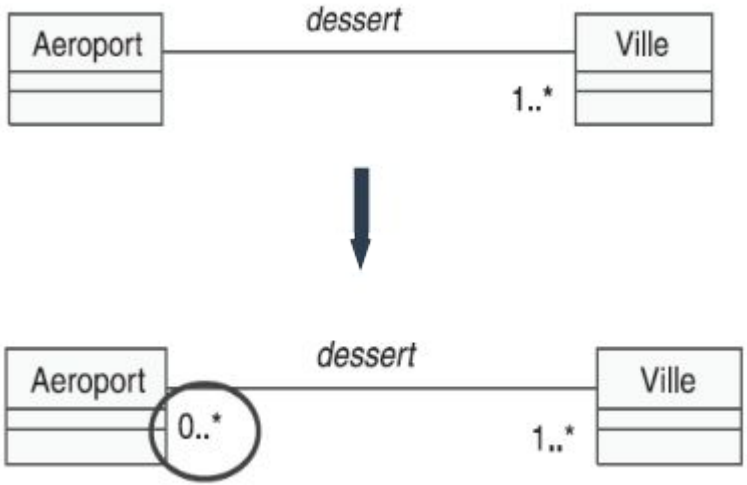
Case study 3: Class diagram

6) Un vol a un aéroport de départ et un aéroport d'arrivé.



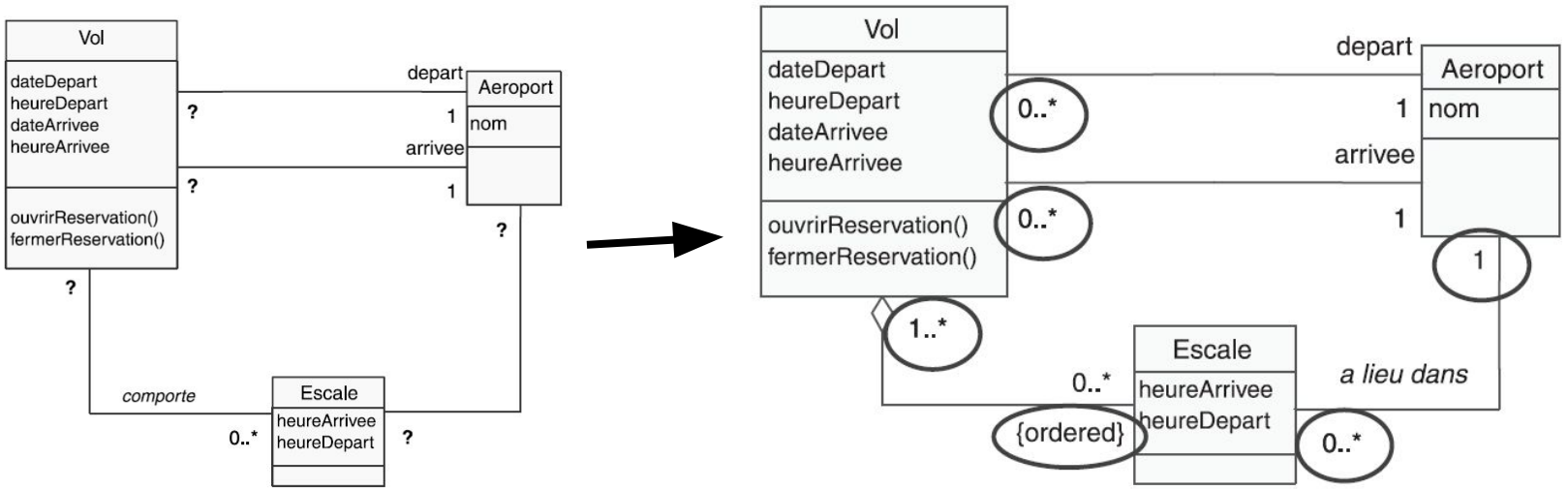
Case study 3: Class diagram

10) Un vol dessert une ou plusieurs villes.

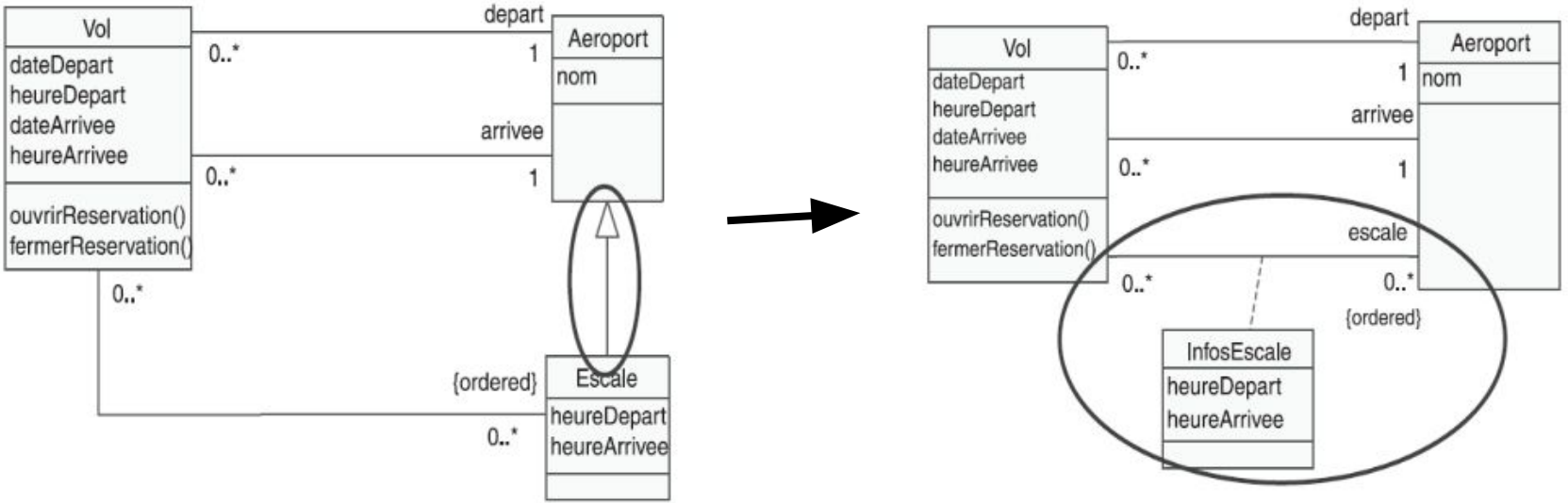


Case study 3: Class diagram

8) Un vol peut comporter des escales dans des aéroports.
9) Une escale a une heure d'arrivée et une heure de départ.

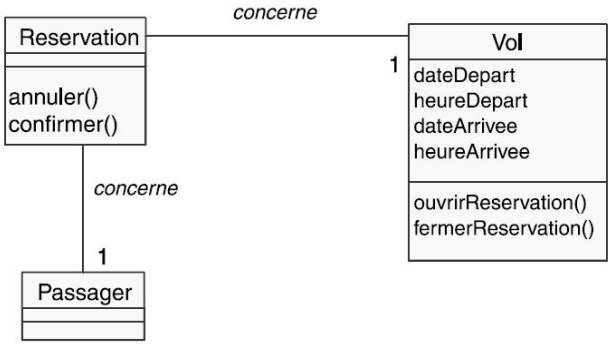


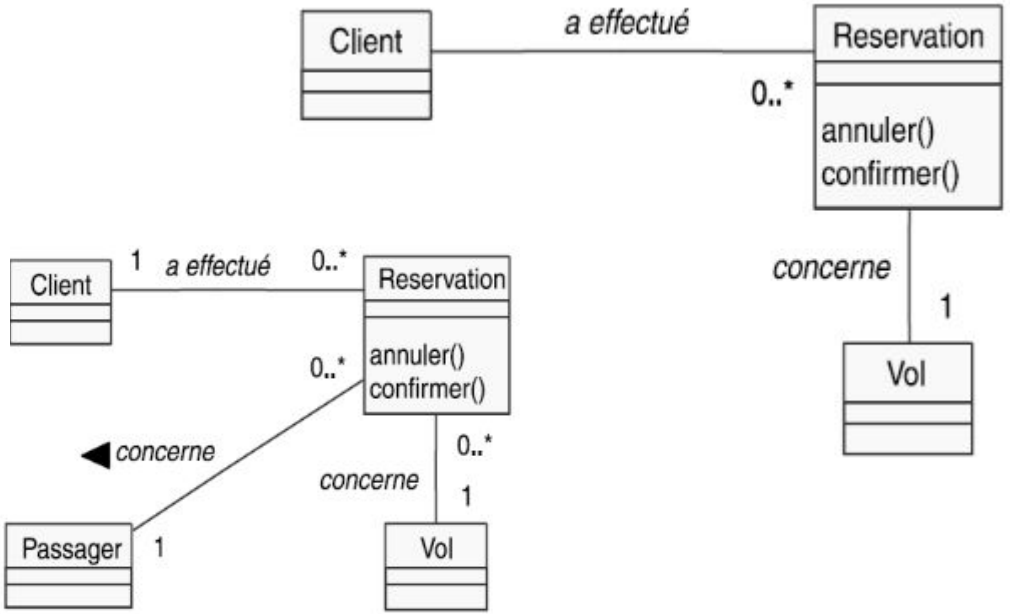
Case study 3: Class diagram



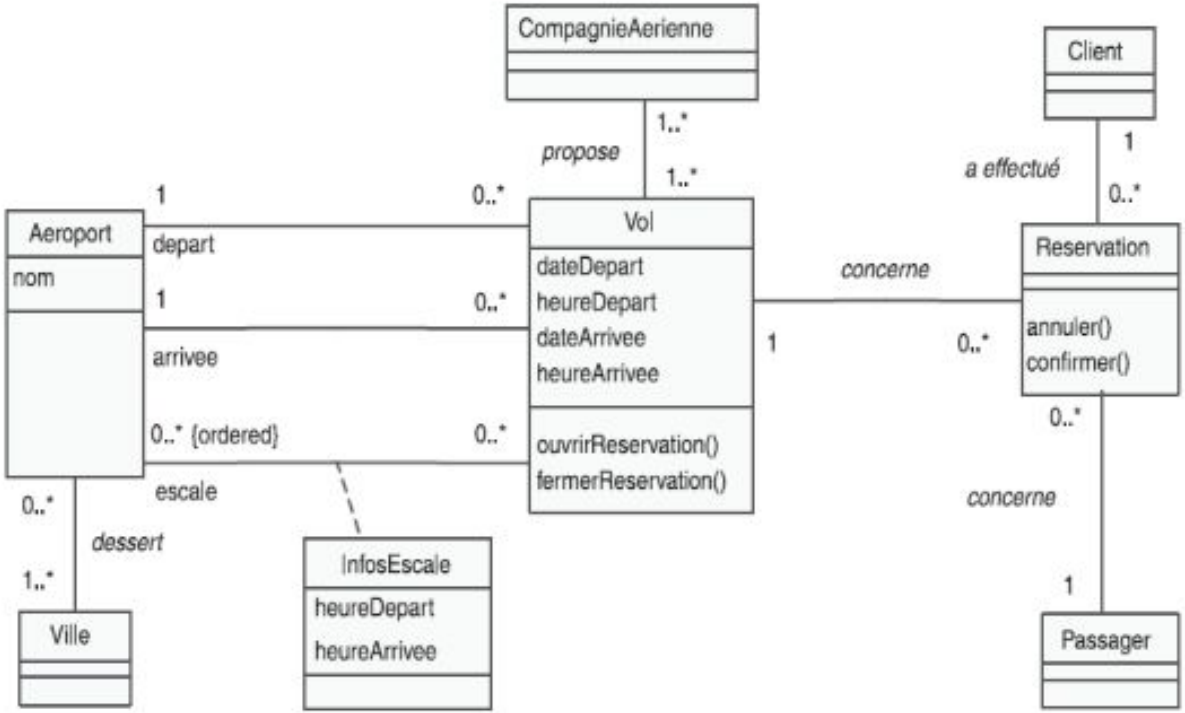
Case study 3: Class diagram

3) Un client peut réserver un ou plusieurs vols, pour des passagers différents.
4) Une réservation concerne un seul vol et un seul passager.
5) Une réservation peut être annulée ou confirmée.



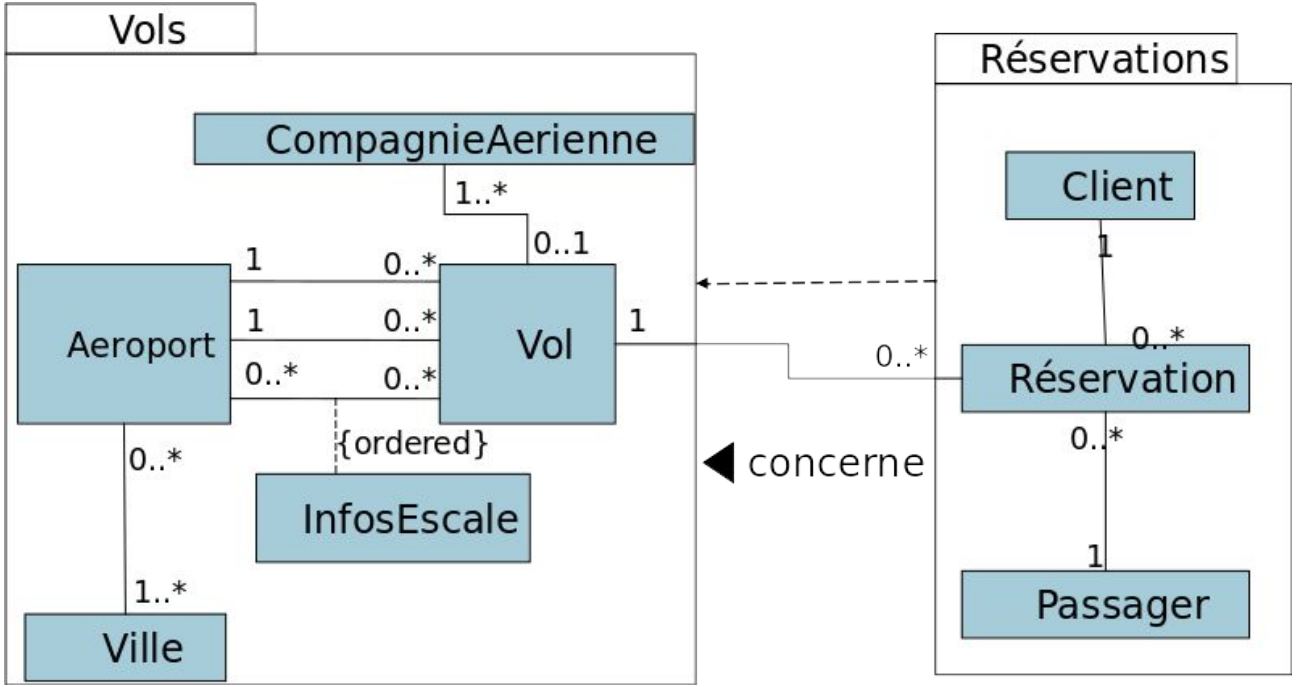


Case study 3: Class diagram



Package

There are other notions in UML such as the "package" that is a logical grouping of classes (... as Java or Python packages). Of course relationships can cross packages.



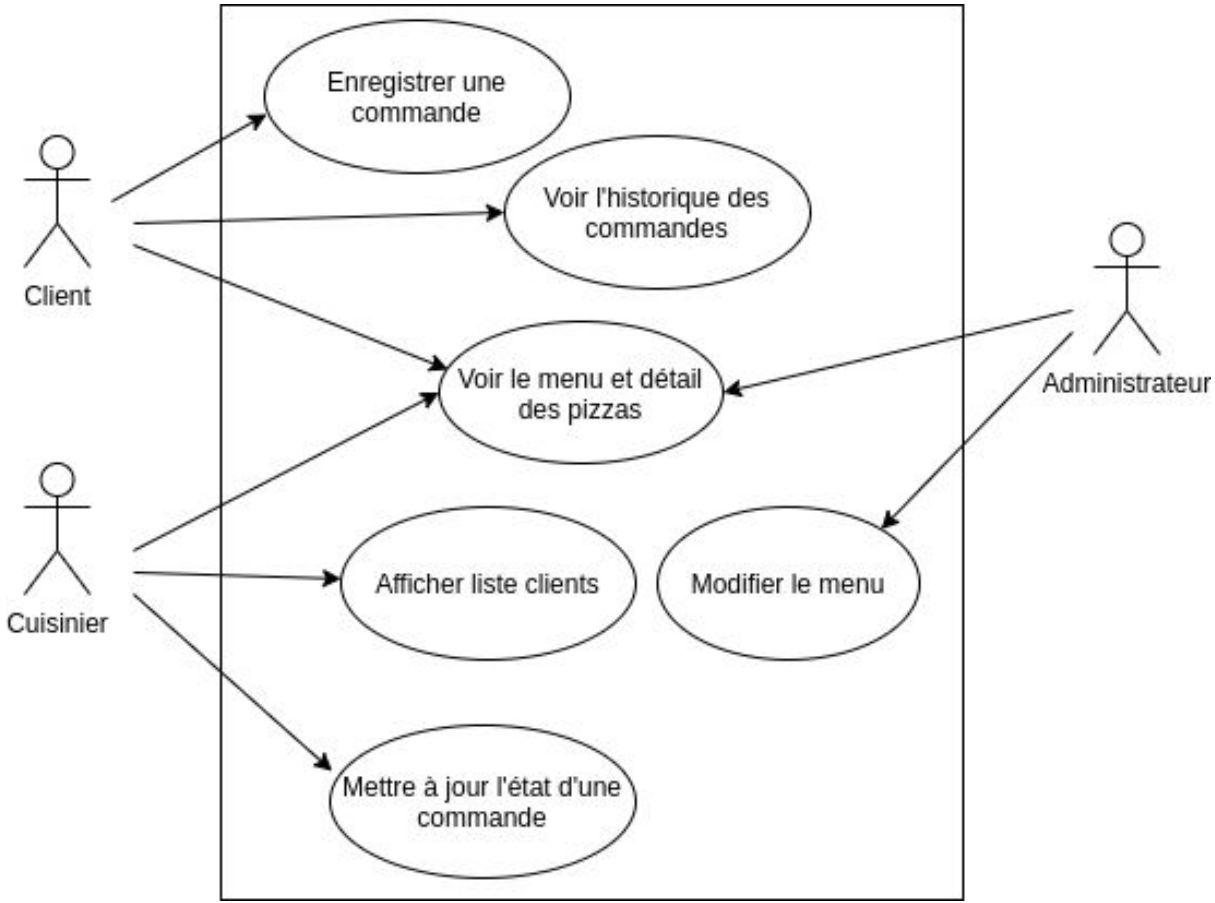
Exercice

Vous devez réaliser un logiciel pour un restaurant de pizza à emporter.

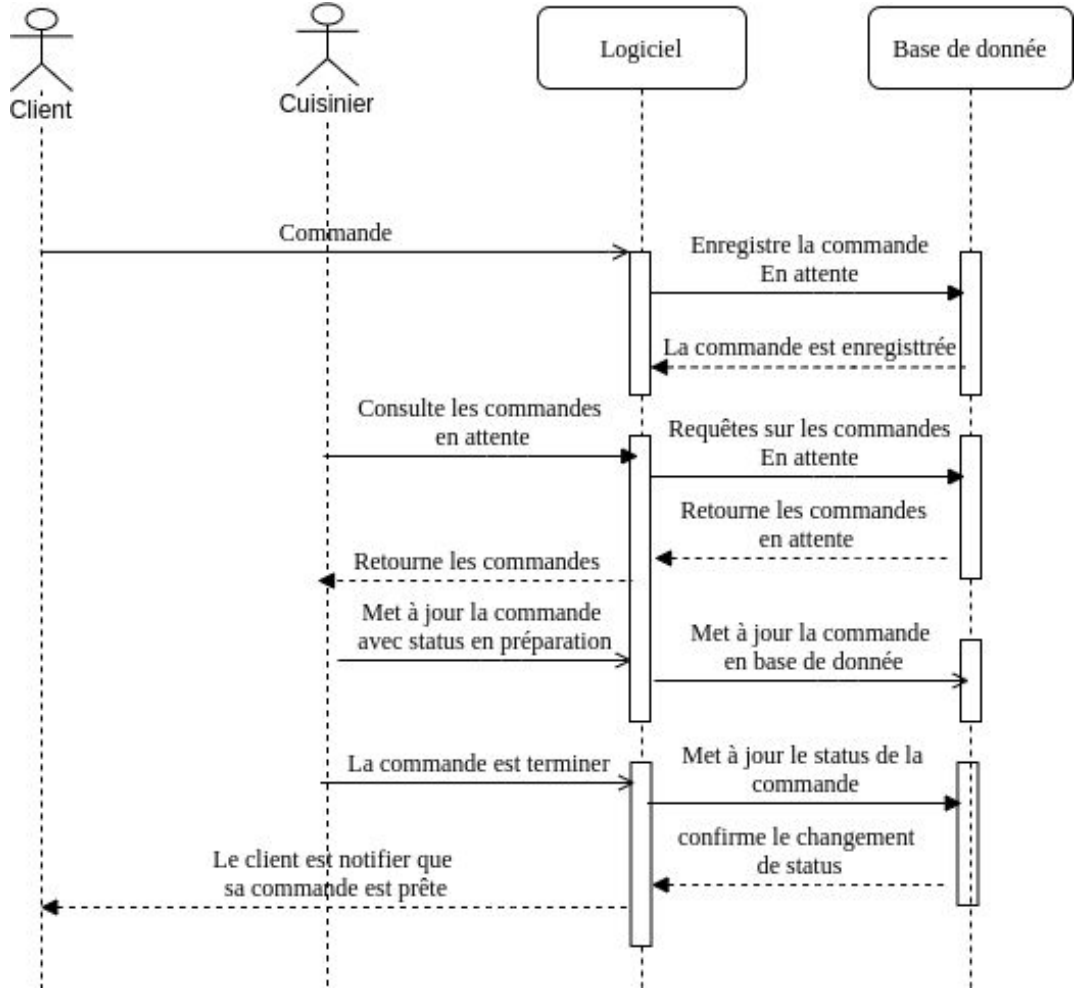
1. Les commandes sont enregistrées par les clients.
2. Une commande comprend une heure de commande, un état d'avancement et une liste de pizza.
3. Chaque pizza est composé d'une liste d'ingrédients et d'un prix.
4. Le client doit pouvoir calculer le prix de sa commande.
5. Les comptes utilisateurs clients ne peuvent voir que les commandes qui leur appartiennent et peuvent afficher les détails de chacune des commandes.
6. Les clients n'ont pas accès aux commandes des autres clients.
7. Les administrateurs peuvent afficher les liste des clients et voir les profils de chacun d'entre eux
8. Les administrateurs peuvent modifier le menu (liste des pizzas proposées aux clients).
9. Les cuisiniers peuvent voir la liste complète des commandes et voir le contenu de chacune des commandes.
10. Les cuisiniers peuvent mettre à jour l’état d'une commande (En attente, en cours de préparation, terminer)

Créez un diagramme des cas d'utilisation, un diagramme séquence sur une commande puis créez un diagramme de classes.

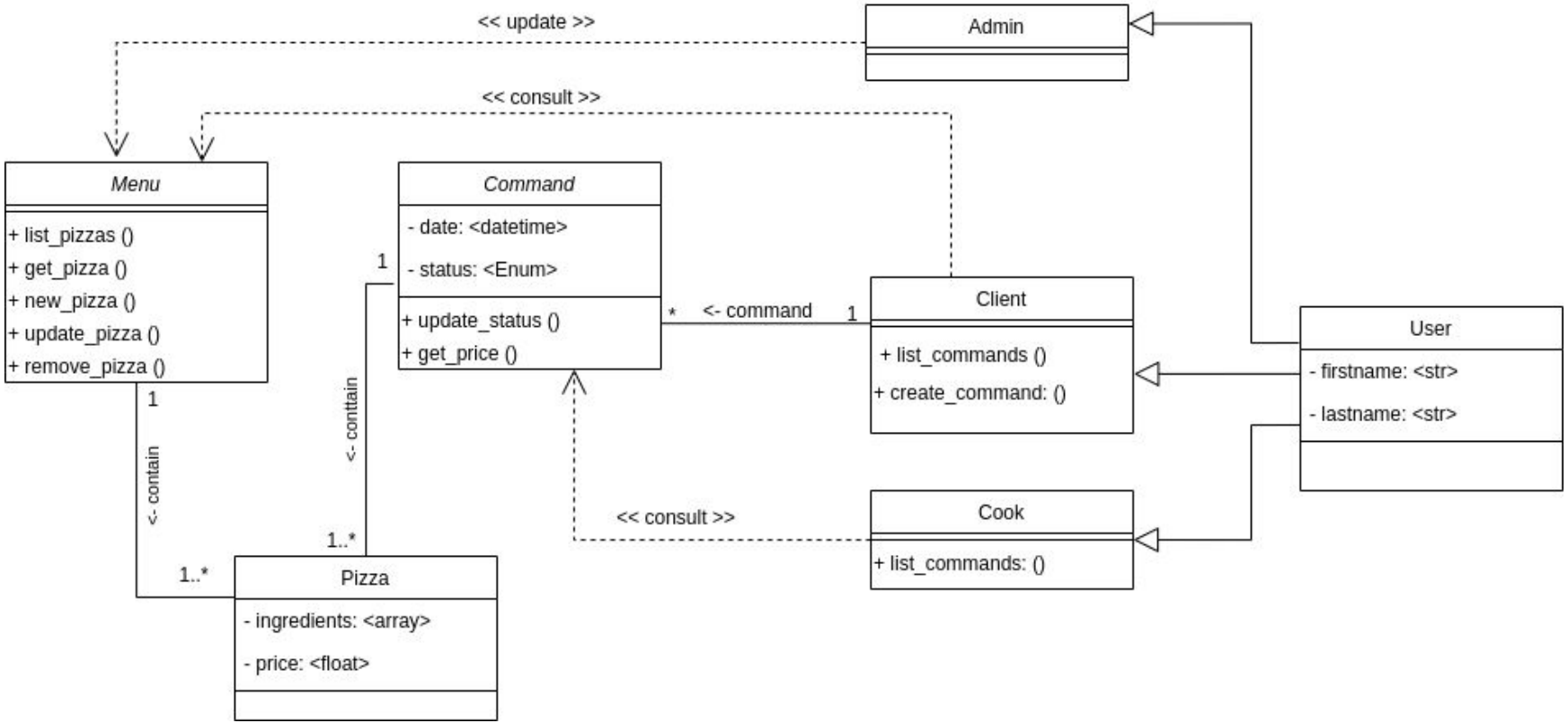
Exercice


Enregistrer une commande
Voir l'historique des commandes
Voir le menu et détail des pizzas
Afficher liste clients
Modifier le menu
Mettre à jour l'état d'une commande
Client
Cuisinier
Administrateur

Exercice


Client
Cuisinier
Logiciel
Base de donnée
Commande
Enregistre la commande
En attente
La commande est enregisttrée
Consulte les commandes
en attente
Requêtes sur les commandes
En attente
Retourne les commandes
en attente
Retourne les commandes
Met à jour la commande
avec status en préparation
Met à jour la commande
en base de donnée
La commande est terminer
Met à jour le status de la
commande
confirme le changement
de status
Le client est notifier que
sa commande est prête

Exercice


Admin
<< update >>
<< consult >>
Menu
+ list_pizzas ()
+ get_pizza ()
+ new_pizza ()
+ update_pizza ()
+ remove_pizza ()
Command
- date: <datetime>
- status: <Enum>
+ update_status ()
+ get_price ()
1
* <- command 1
Client
+ list_commands ()
+ create_command: ()
User
- firstname: <str>
- lastname: <str>
1
<- contain
1..*
<- conttain
1..*
Pizza
- ingredients: <array>
- price: <float>
<< consult >>
Cook
+ list_commands: ()

Design patterns

Typical solutions to common problems

Design pattern

A design pattern is a standard solution to a problem of conception. The set of design pattern constitute a catalog of good practices. These practices are build from experience of the community. They form a common vocabulary to identify immediately a solution.

Why using a design pattern

- To speed up the development process by providing proven development paradigms.
- To anticipate issues that may not become visible until later in the implementation.
- To improve readability of the code by providing standardization.

Design patterns were popularized by the book “Design Patterns - Elements of Reusable Object-Oriented Software” released in 1995 and co-written by four authors (Gang Of Four, or GoF). This book describe 23 patterns to which others have been added.

Design Pattern

The "Gang of Four" book organizes patterns by purpose.

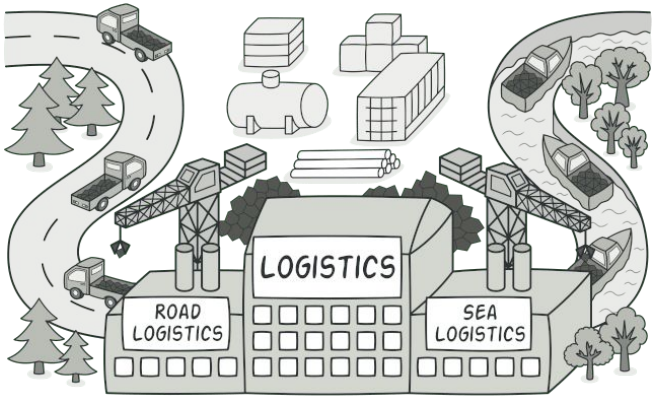
- **Creational** (abstracting the object-instantiation process)
  - Factory, Abstract, Builder, Prototype, Singleton

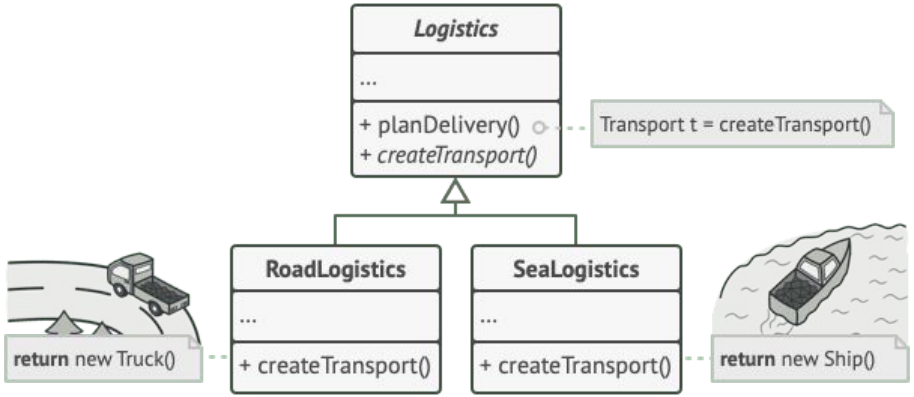
- **Structural** (how objects/classes can be combined to form larger structures)
  - Adapter, Bridge, Composite, Decorator, Facade, Flyweight, Proxy

- **Behavioral** (communication between objects)
  - Chain of responsibility, Command, Iterator, Mediator, Memento, Observer, State, Strategy, Template method, Visitor

Creation pattern : Factory

This pattern is one of the most used. It create an interface to build objects in mother class but leave sub-objects his types.
Example : A delivery system which can send merchandises by boat or by truck



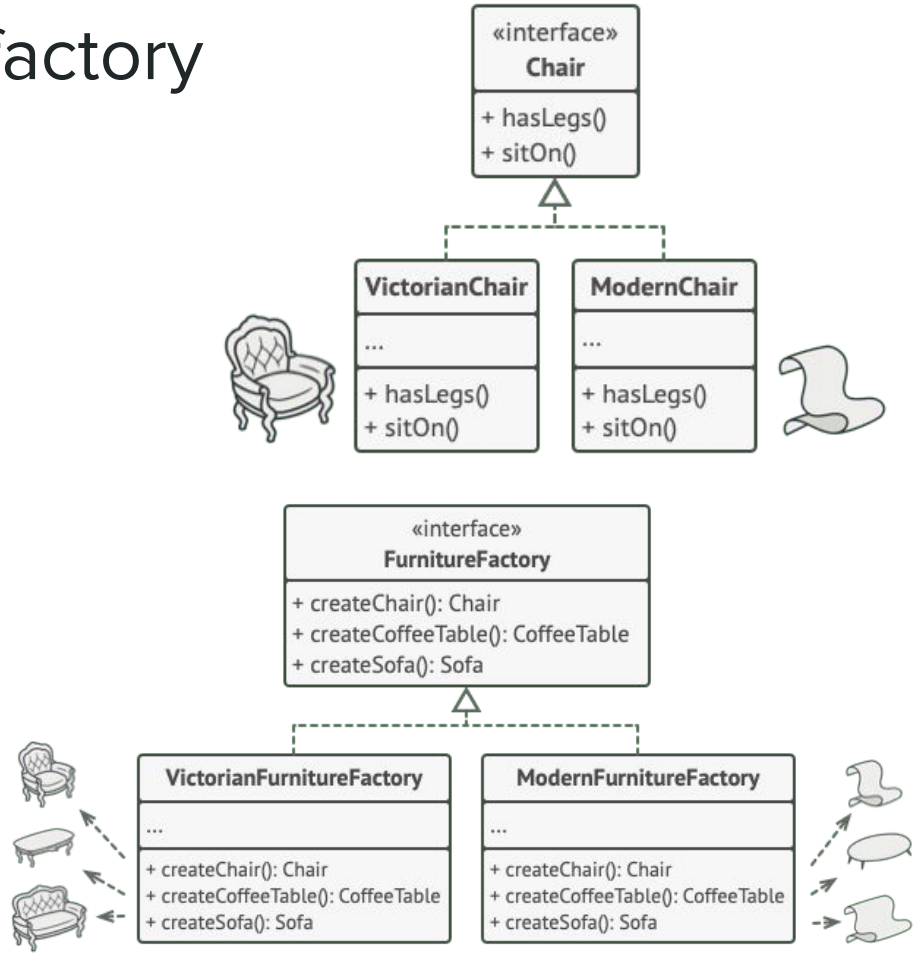


!!! This method supposed to use same functions name and this overloading of them !!!

Creation pattern : Abstract factory

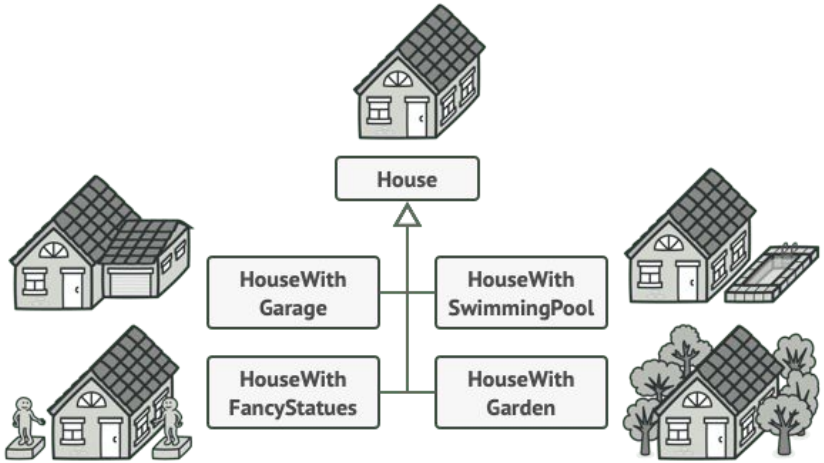
An extension of the factory design to classified your objects, without mentioning their complete class.

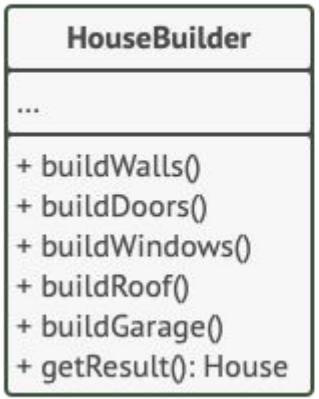
Example : A furniture shop, with different designs.

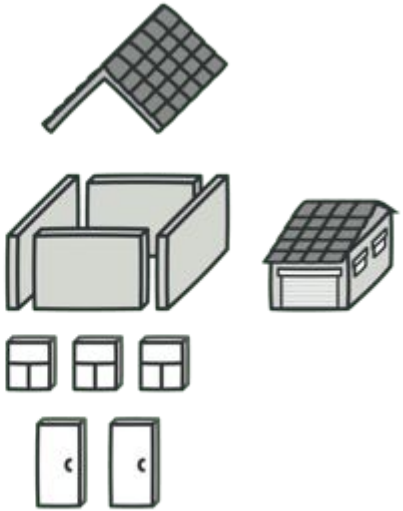


Creation pattern : Builder

The Builder is pattern made to help you build complex object, which can have different variation.

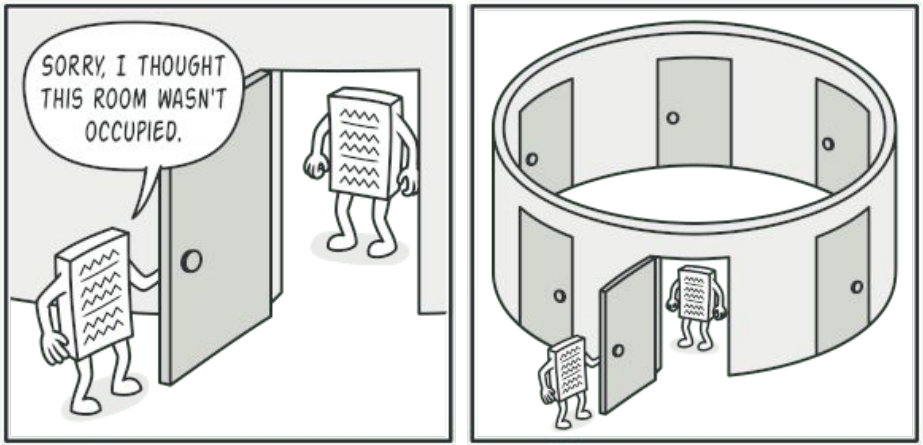


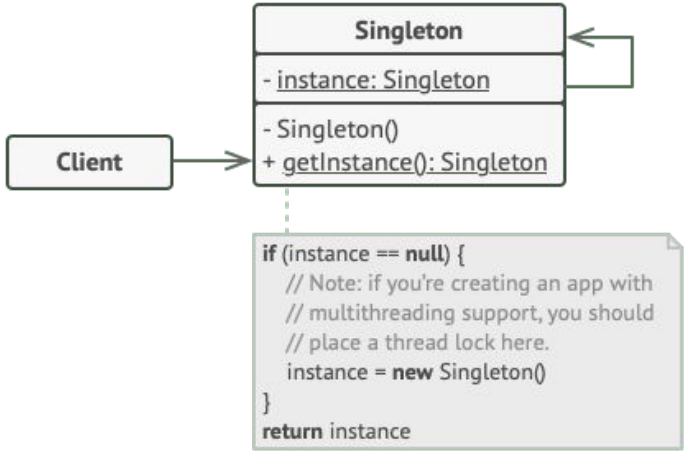




Creation pattern : Singleton

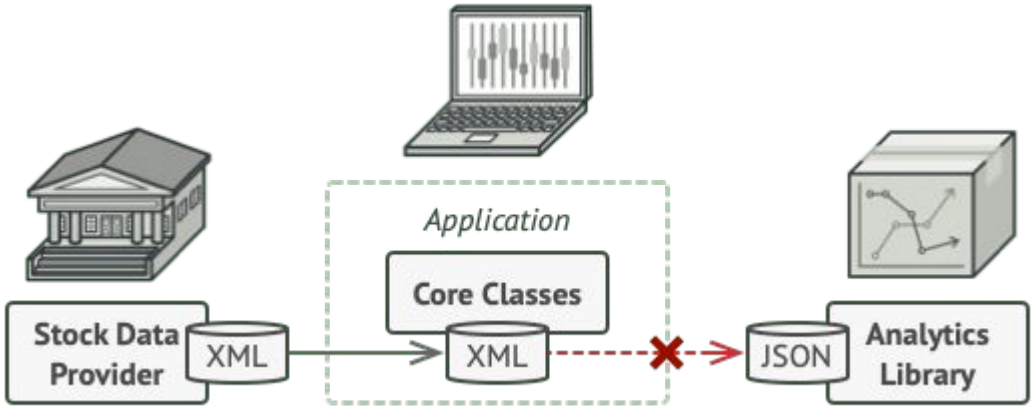
The singleton is pattern made to for your object to have only one instance, which can be access by multiple entry points.

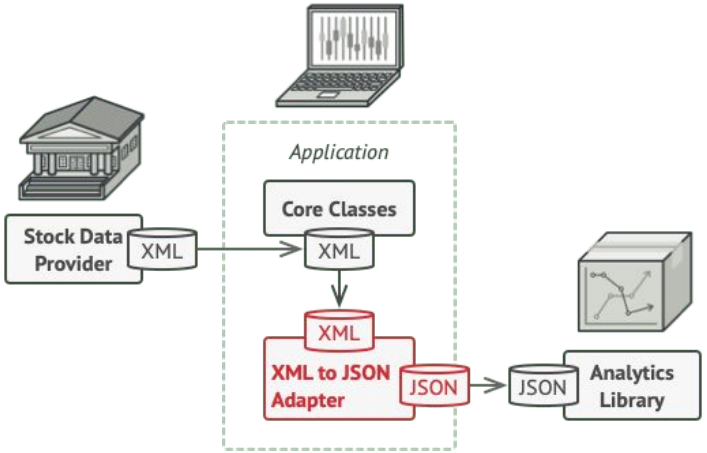




Structural pattern : Adapter

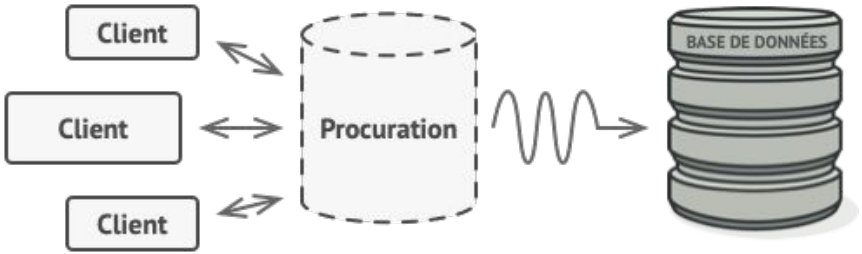
An adapter design pattern allows objects to collaborate even if they have incompatible interfaces.

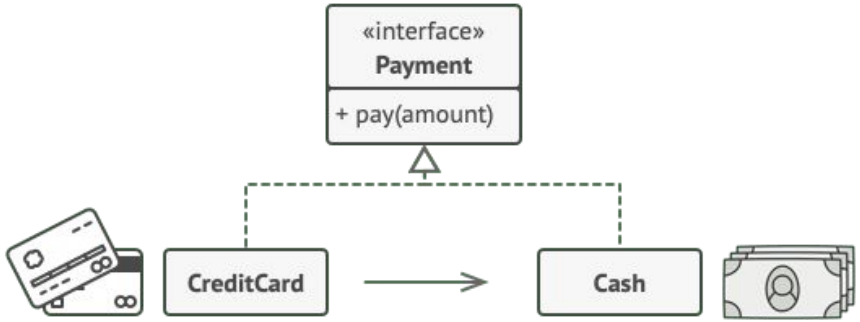




Structural pattern : Proxy

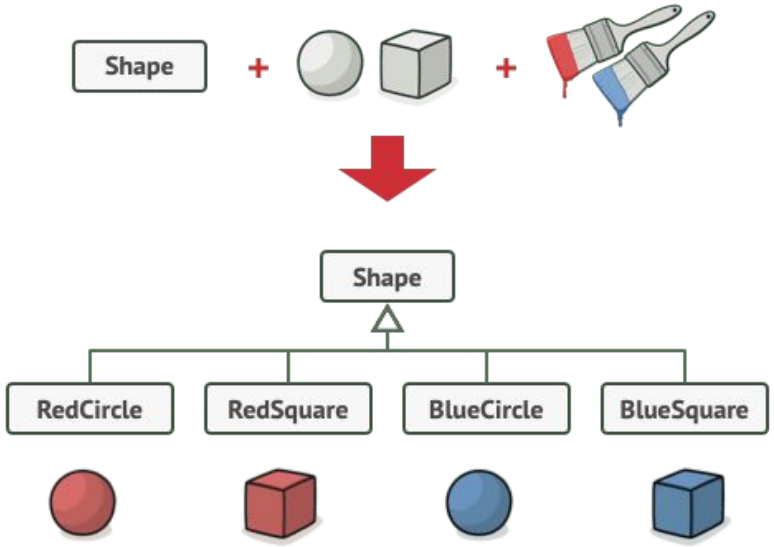
The proxy pattern is a pattern to control the access and allows you to act before and after the request.

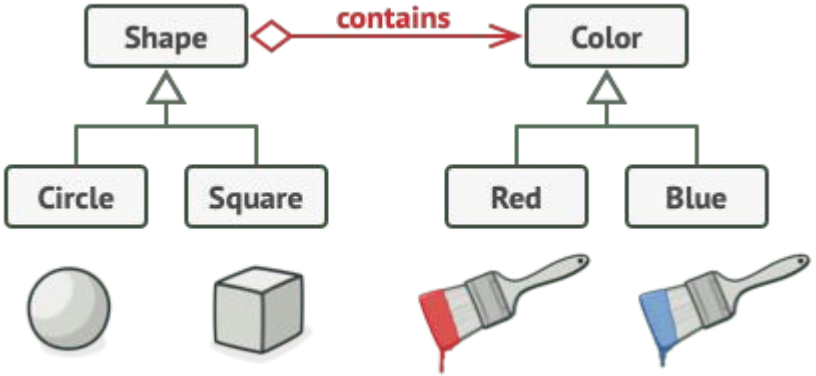




Structural pattern: Bridge

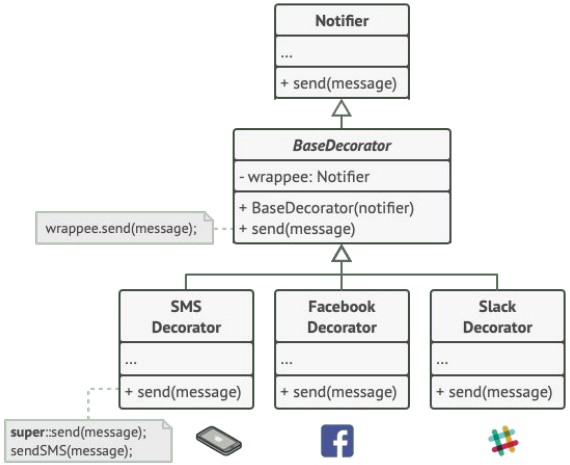
Bridge is a pattern to let you split a large class or two seperate classes/or hierarchies.

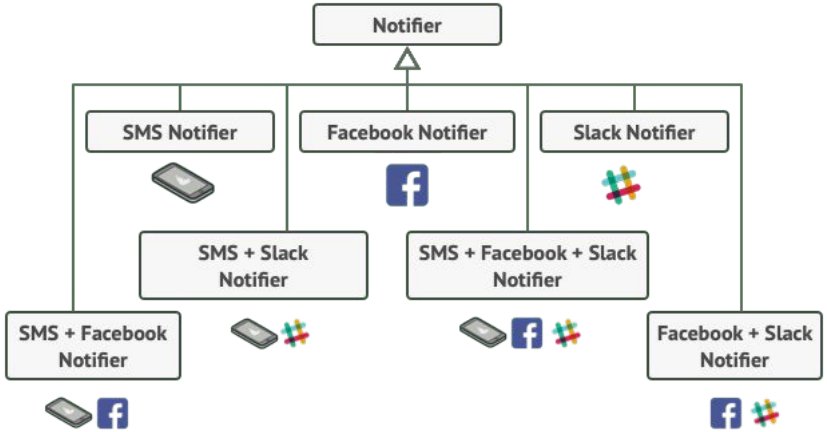


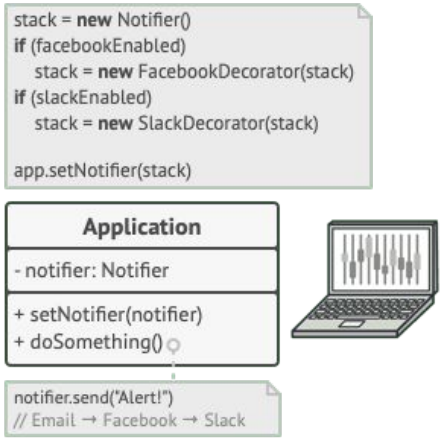


Structural pattern: Decorator

Decorator design pattern permits you to affect new behaviours to your objects by placing them into new wrapper object that contains theses behaviours.

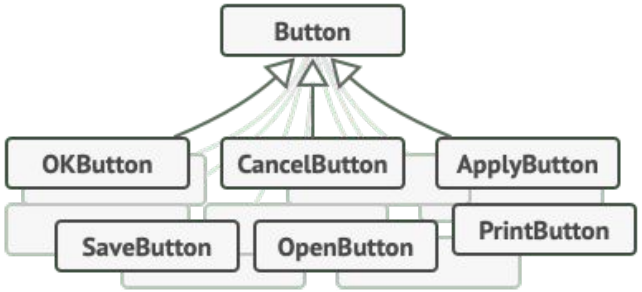


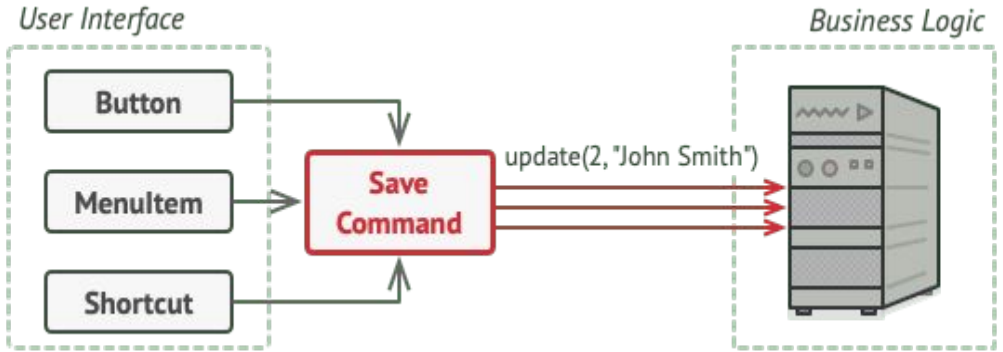




Behavior pattern : Command

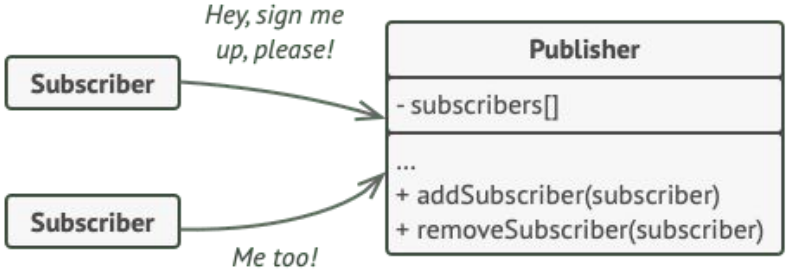
Take an action and transform it to a stand-alone object. This object contains all of the detail of the action. This transformation permit to configure methods with different action, plan their execution, put them onto queue or simply cancel them..

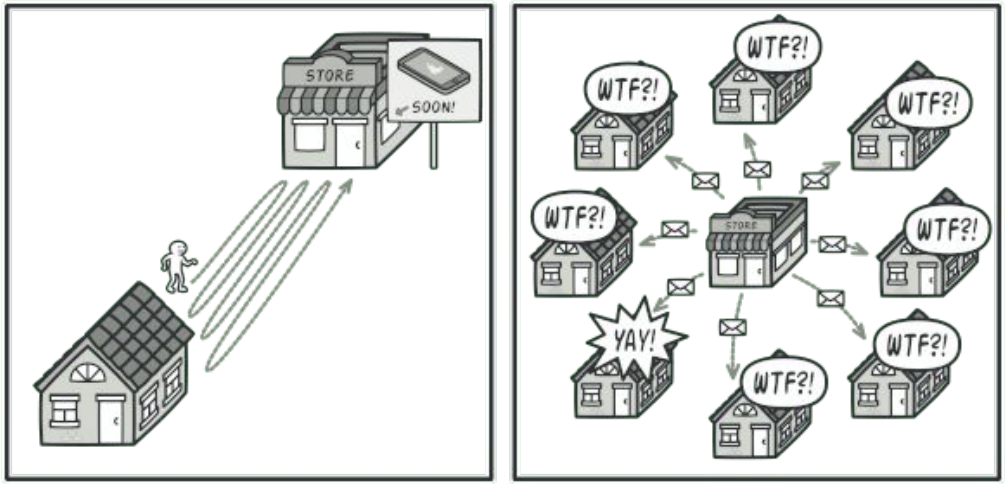


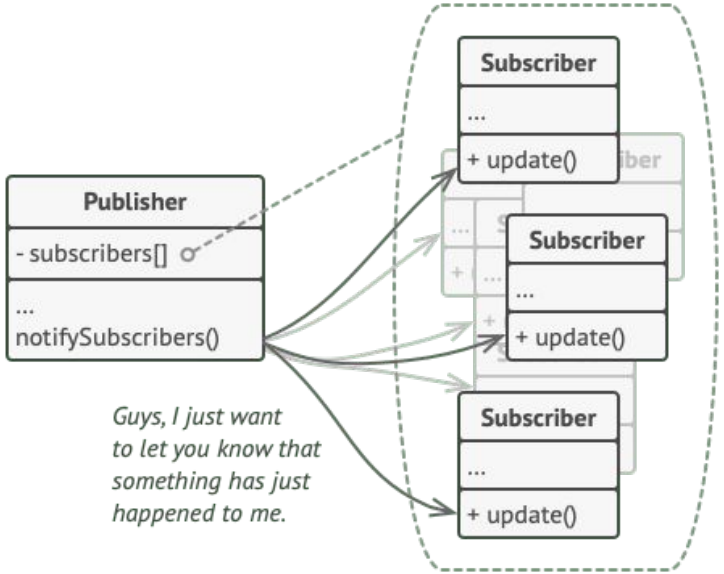


Behavior pattern : Observator

This pattern let you create a subscription system to notify multiple objects about events concerning this observed object.

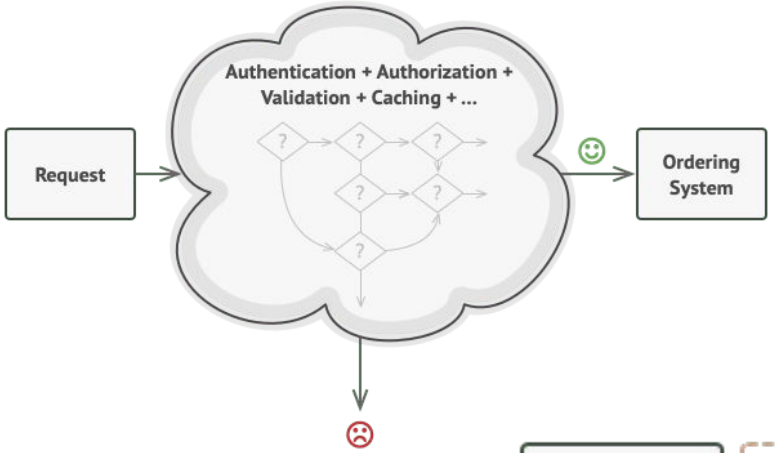


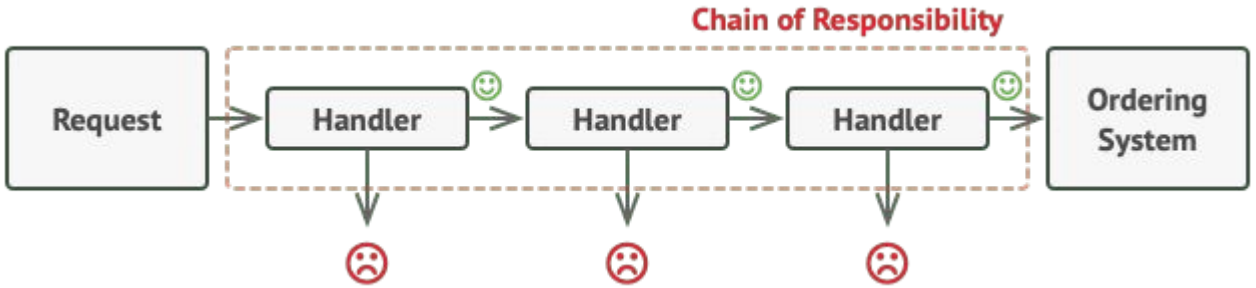




Behavior pattern : Chain of responsibility

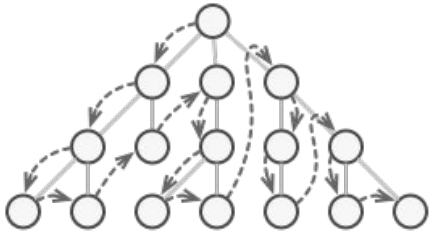
This pattern allows you to pass a request along a chain of handlers.

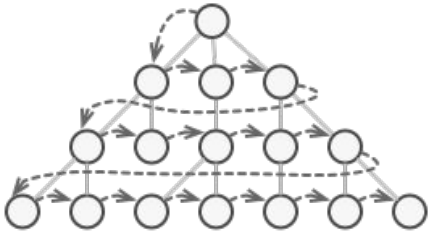




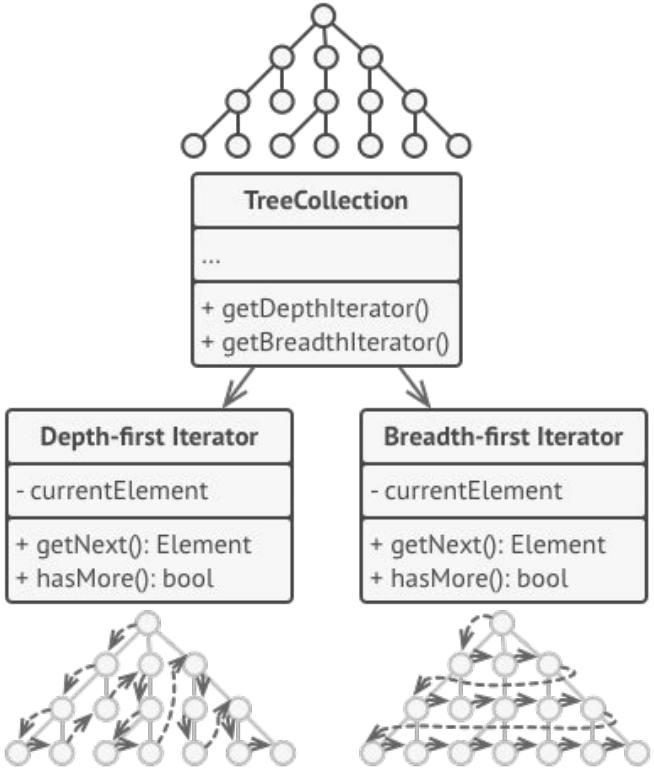
Behavior pattern : Iterator

This pattern permits you to browse through elements of a collection without showing its internal representation





How browse it efficiently ?



Design Pattern

It exist a lot more design pattern, you can visit the Refactoring.guru or the Wikipedia page about design pattern to see a complete list with description of each.

https://refactoring.guru/design-patterns/catalog
Or
https://en.wikipedia.org/wiki/Software_design_pattern

Unjustified use:

> *If all you have is a hammer, everything looks like a nail.*

This is the problem that haunts many novices who have just familiarize themselves with patterns. Having learned about patterns, they try to apply them everywhere, even in situations where simpler code would do just fine.

Questions ?