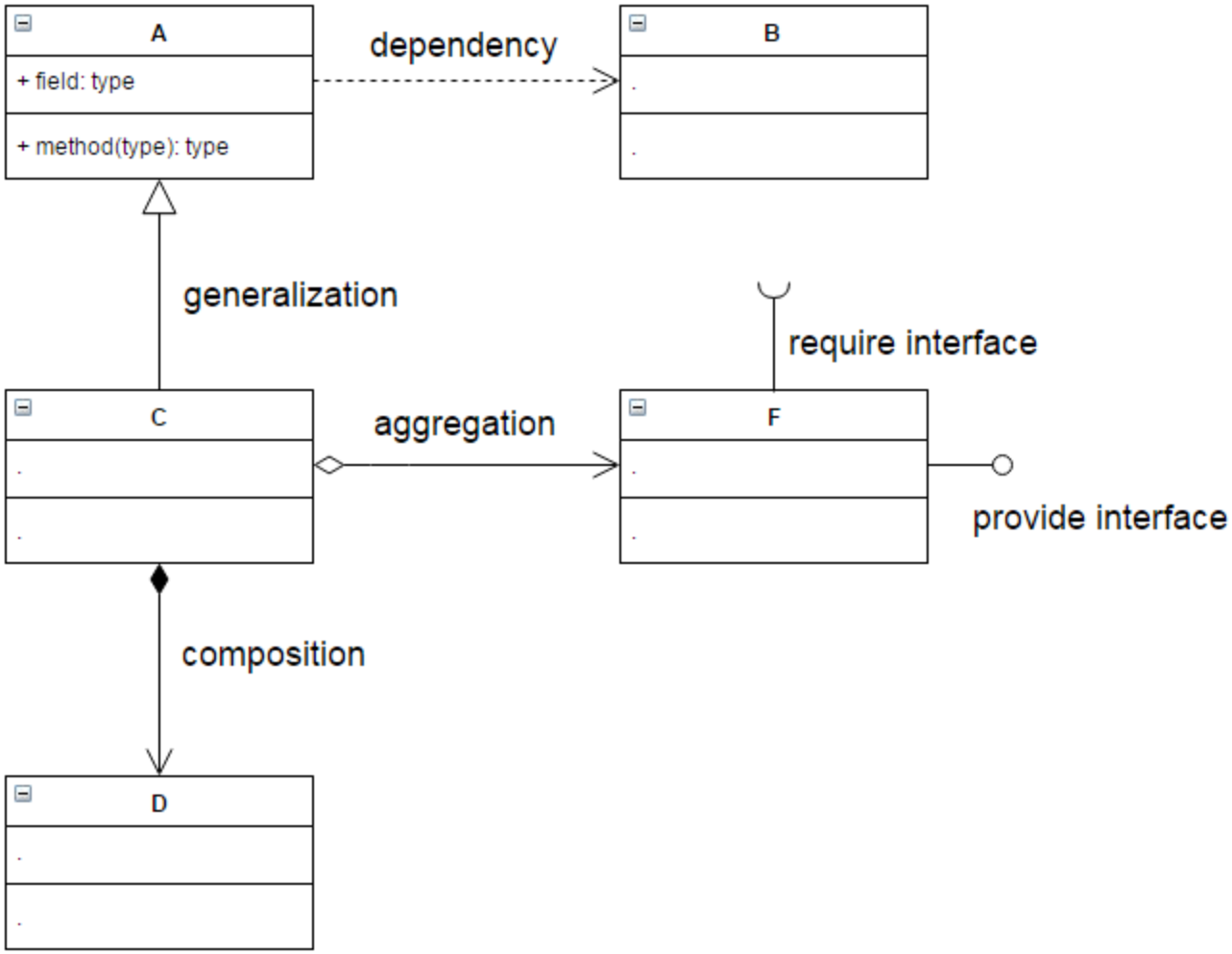
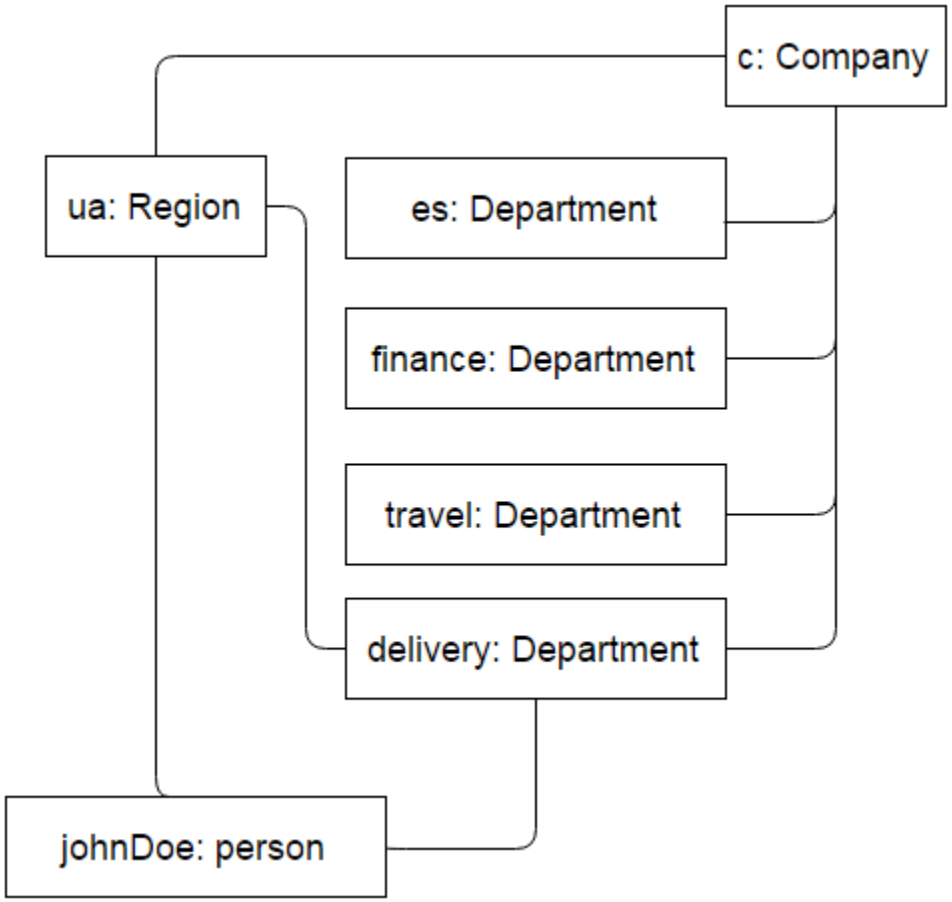
UML & Design patterns, Best practices

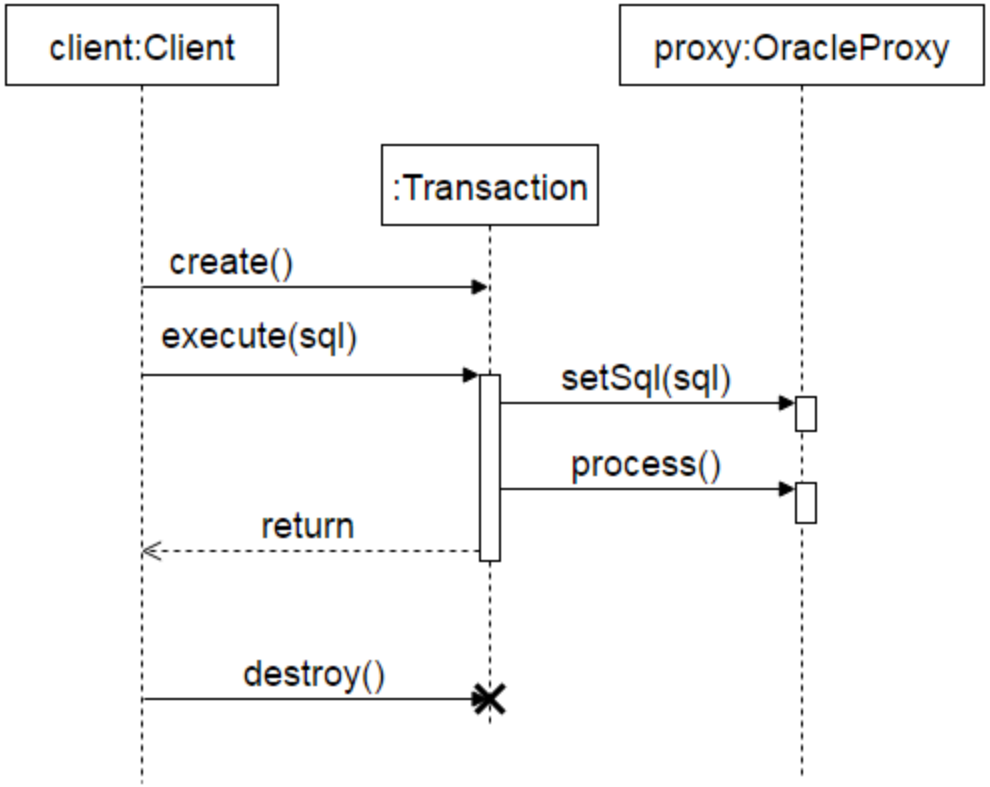
UML - Classes diagrams



UML - Objects diagrams



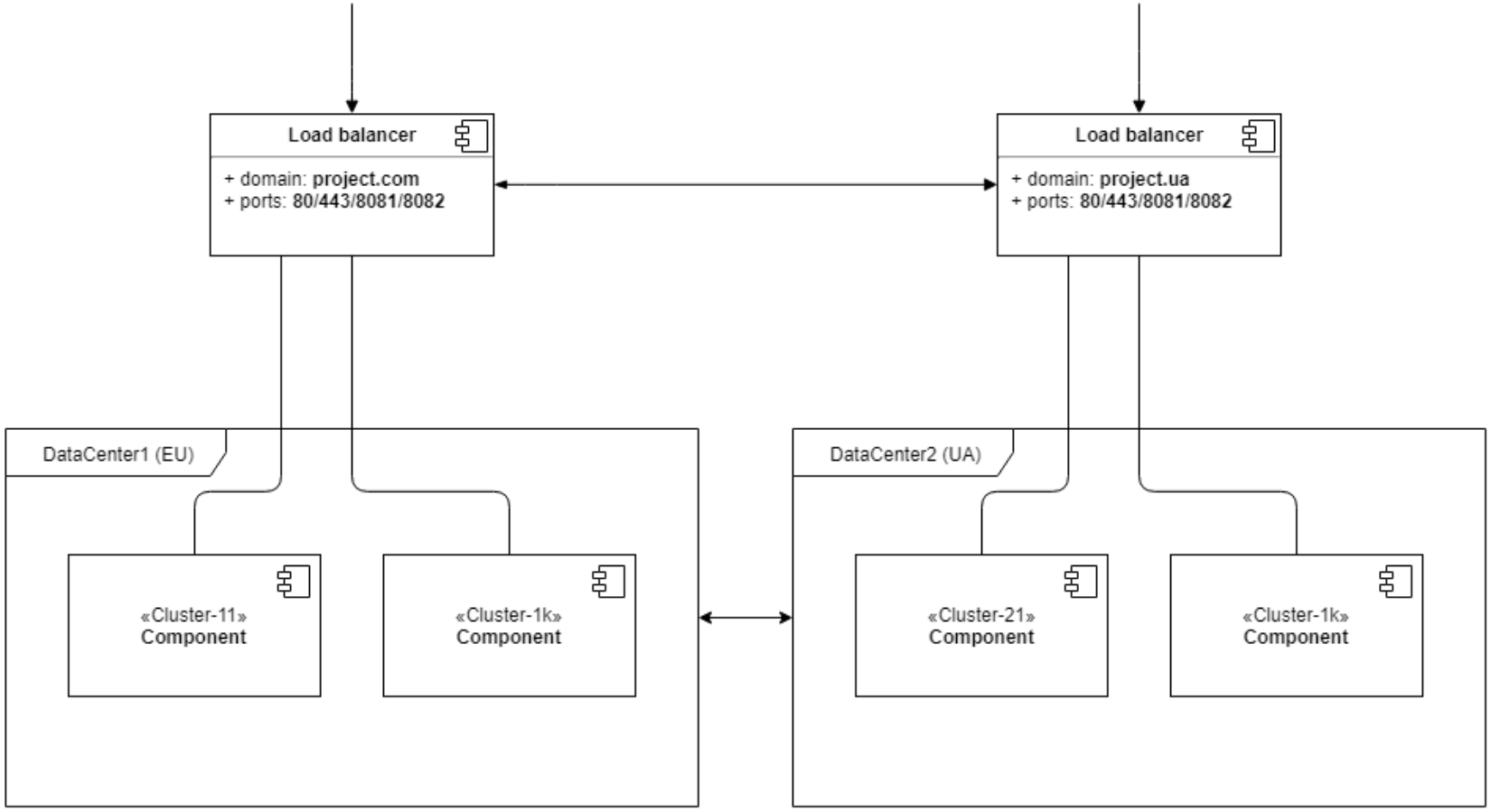
UML - Sequence Diagrams



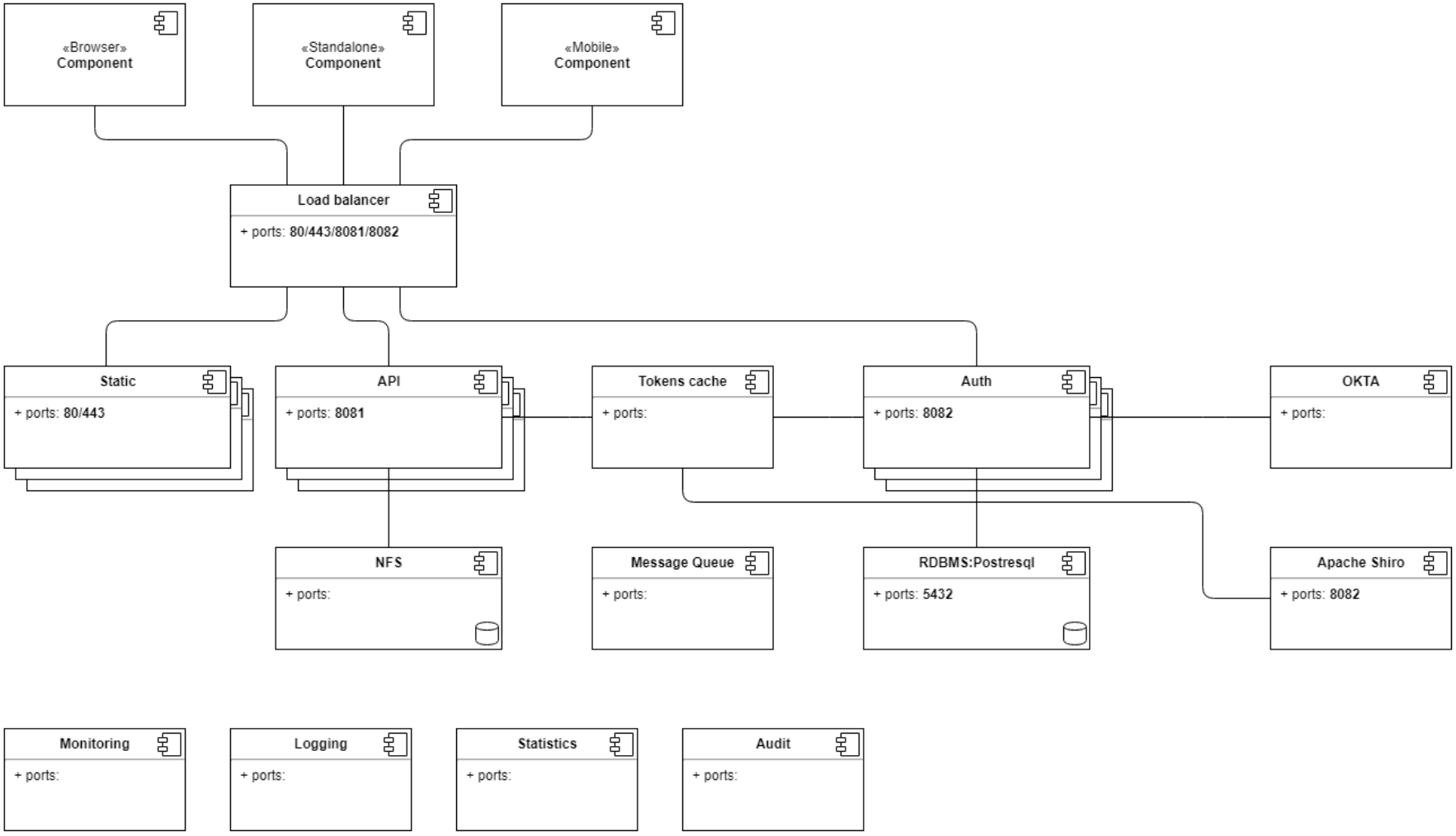
Dropbox-like service: high-level requirements

- Automatic file sub-system synchronization between single backend server and multiple consumer applications
- Version based back API
- Web client
- Standalone clients Win/Mac/Linux
- Mobile client
- Full Audit subsystem
- Versioning subsystem for stored files
- Integration with external services (OAuth)

Dropbox-like service: high-level requirements



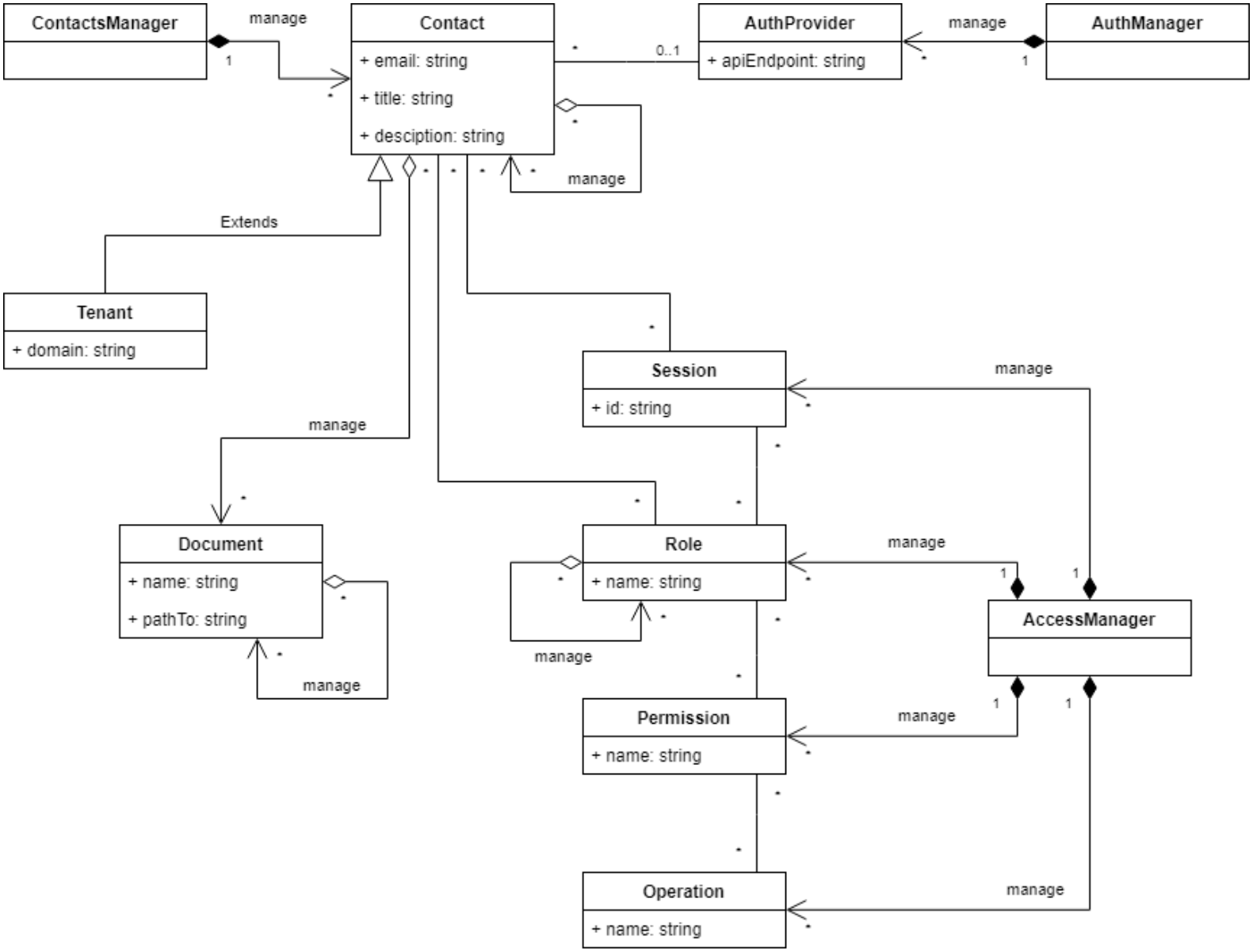
Dropbox-like service: cluster



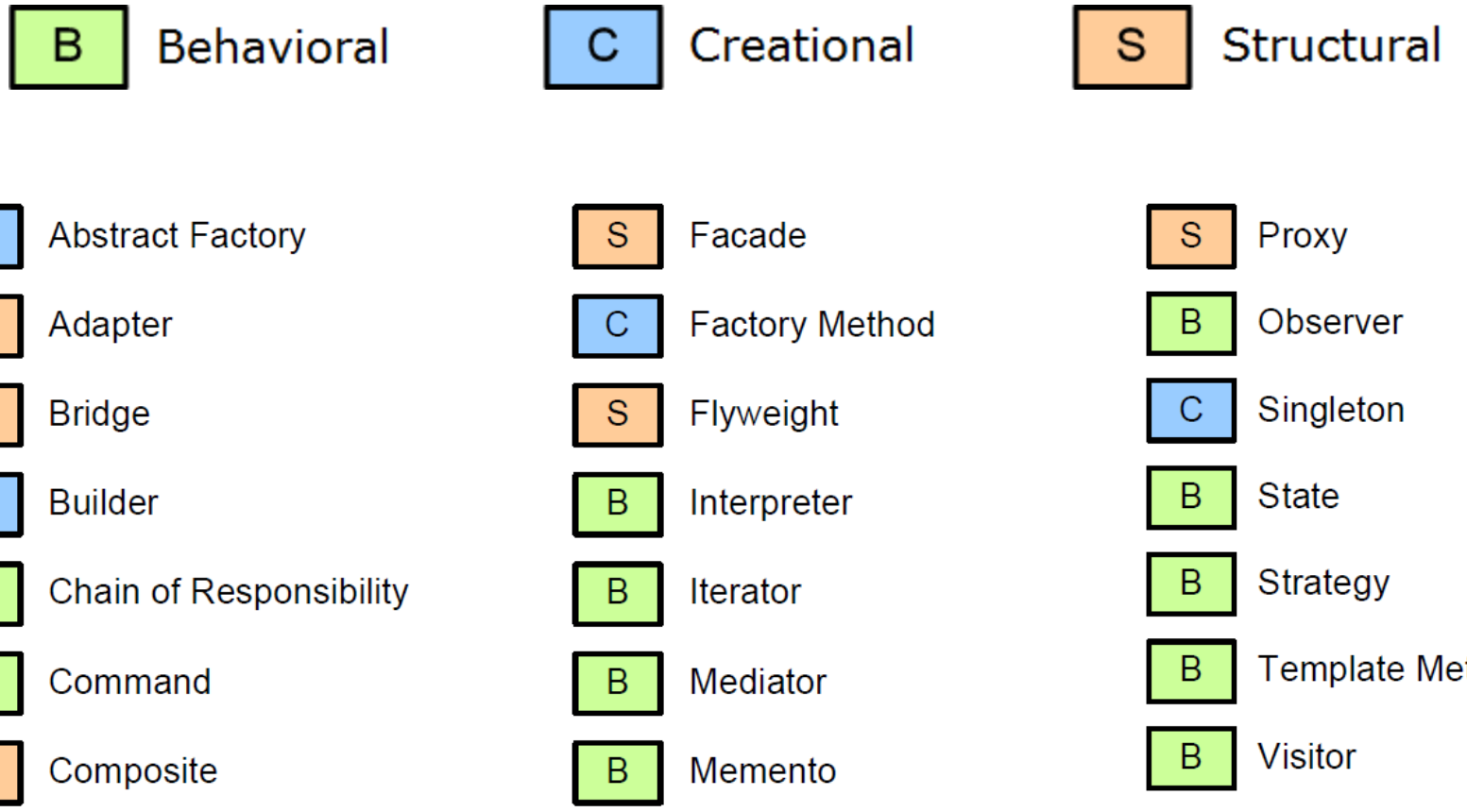
Dropbox-like service: user management

- Register user via email
- Bind accounts from other services (facebook, linkedin, google, etc.)
- Ability to create/manage/login into tenant
- Switch between sub-accounts (facebook, linkedin, “master” account)
- Make API call using third-parties tokens
- CRUD for group of account, manage permissions

Dropbox-like service: user management (RBAC/ACL)



Design patterns

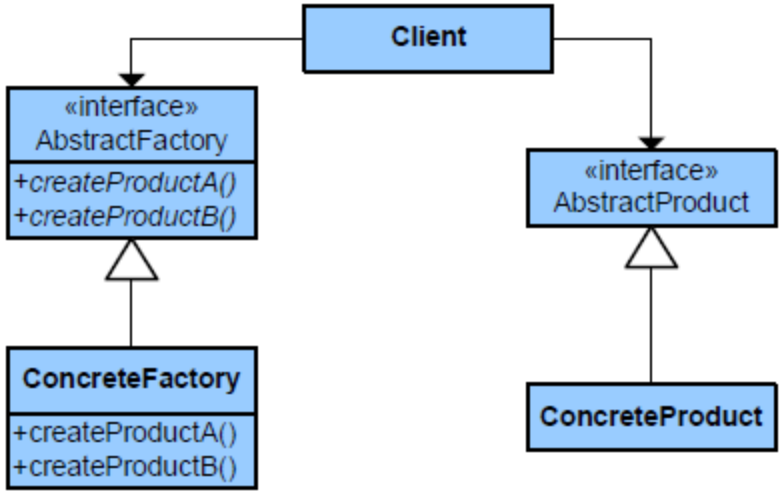


Design patterns

Abstract Factory

Type: Creational

What it is:
Provides an interface for creating families of related or dependent objects without specifying their concrete class.

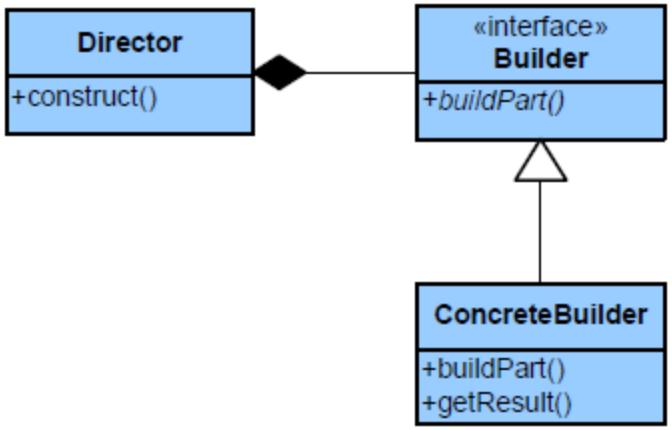


Xwindows/QT

Builder

Type: Creational

What it is:
Separate the construction of a complex object from its representing so that the same construction process can create different representations.



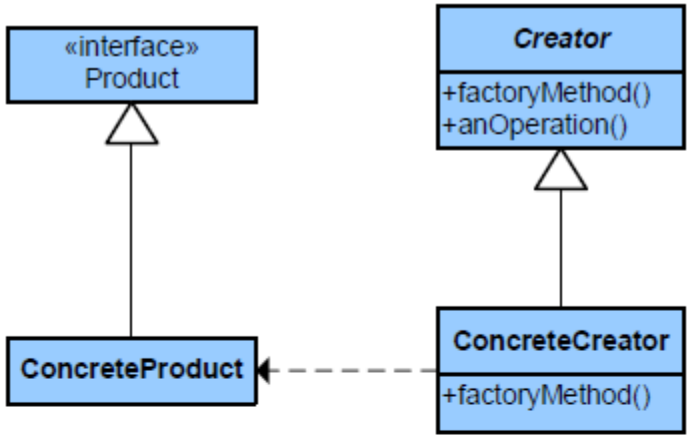
Load document from different formats

Design patterns

Factory Method

Type: Creational

What it is:
Define an interface for creating an object, but let subclasses decide which class to instantiate. Lets a class defer instantiation to subclasses.

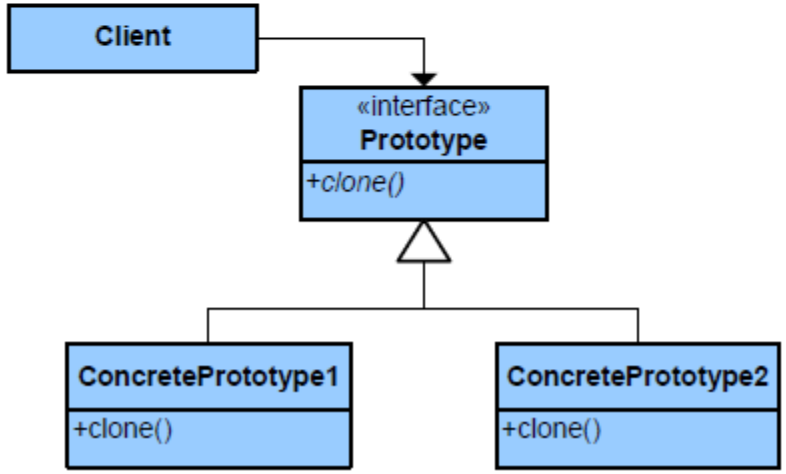


Virtual constructor.

Prototype

Type: Creational

What it is:
Specify the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype.



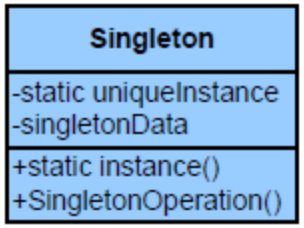
“heavy” object, when copy is much quickly than creation from scratch, reuse some parts

Design patterns

Singleton

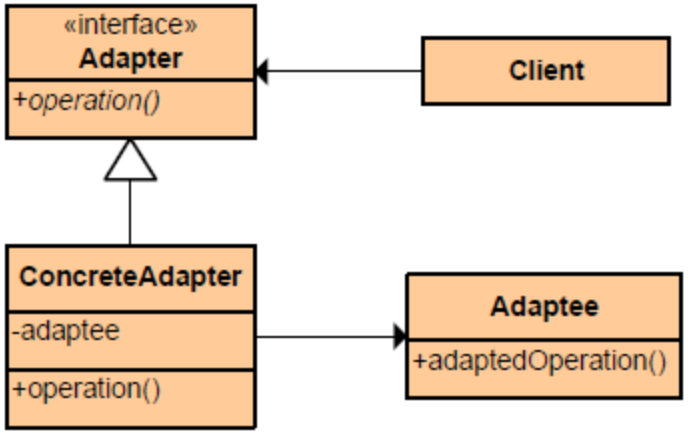
Type: Creational

What it is:
Ensure a class only has one instance and provide a global point of access to it.



Logger

Design patterns

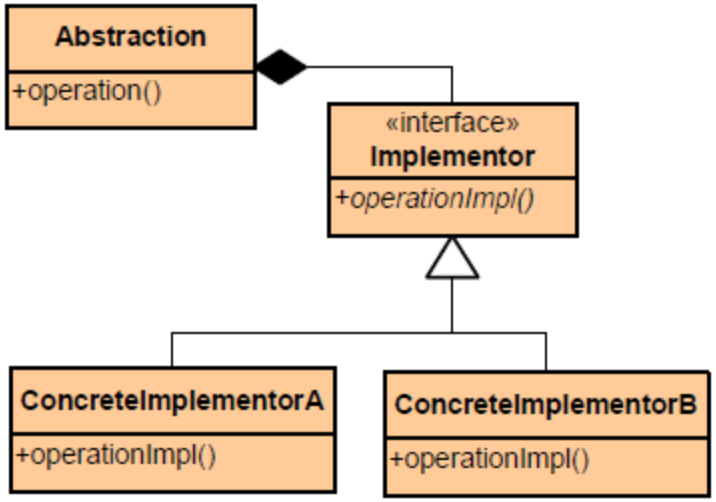


Adapter

Type: Structural

What it is:
Convert the interface of a class into another interface clients expect. Lets classes work together that couldn't otherwise because of incompatible interfaces.

Use XPath to navigate JSON



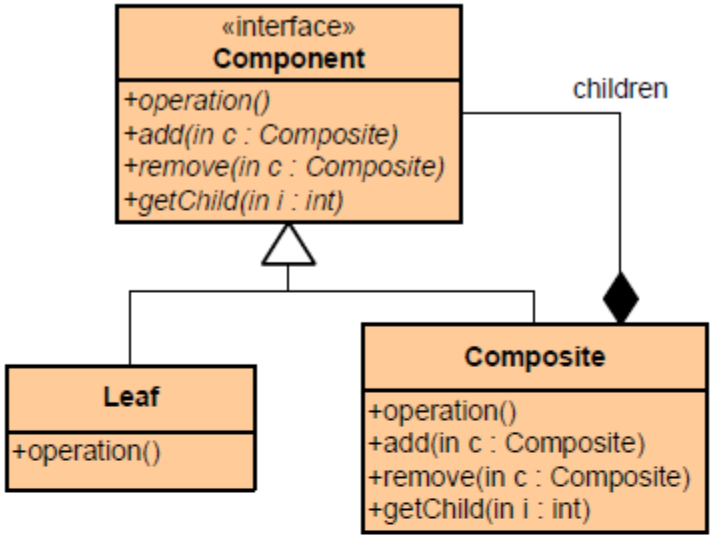
Bridge

Type: Structural

What it is:
Decouple an abstraction from its implementation so that the two can vary independently.

PIMPL – Point to Implementation

Design patterns

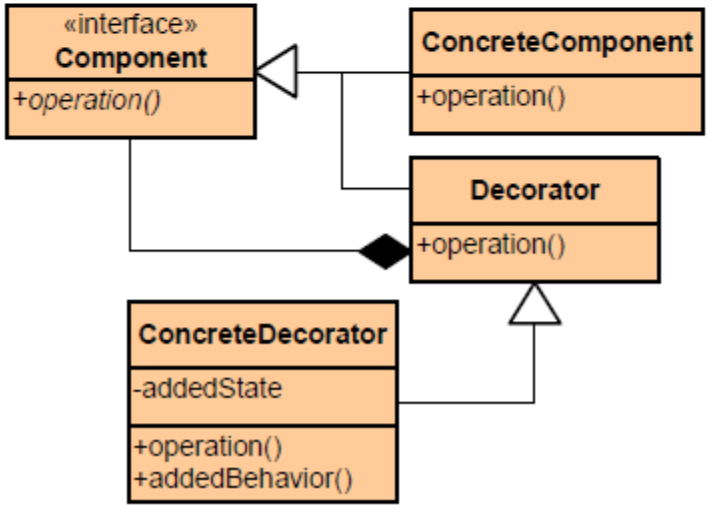


Composite

Type: Structural

What it is:
Compose objects into tree structures to represent part-whole hierarchies. Lets clients treat individual objects and compositions of objects uniformly.

GUI windows, Multi-selection



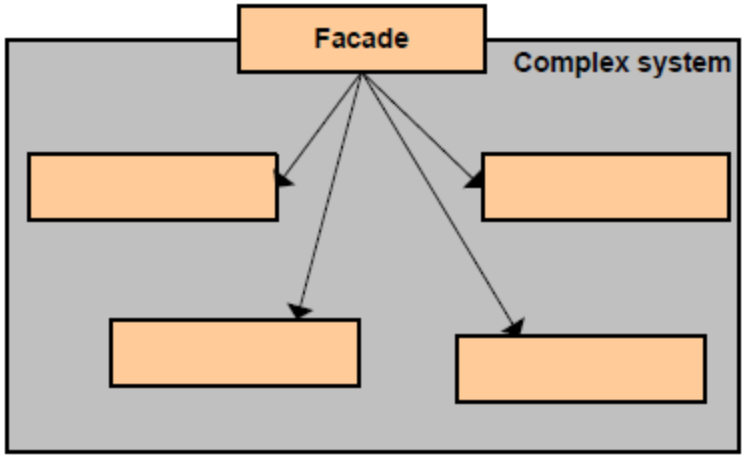
Decorator

Type: Structural

What it is:
Attach additional responsibilities to an object dynamically. Provide a flexible alternative to sub-classing for extending functionality.

Serialization algorithms

Design patterns

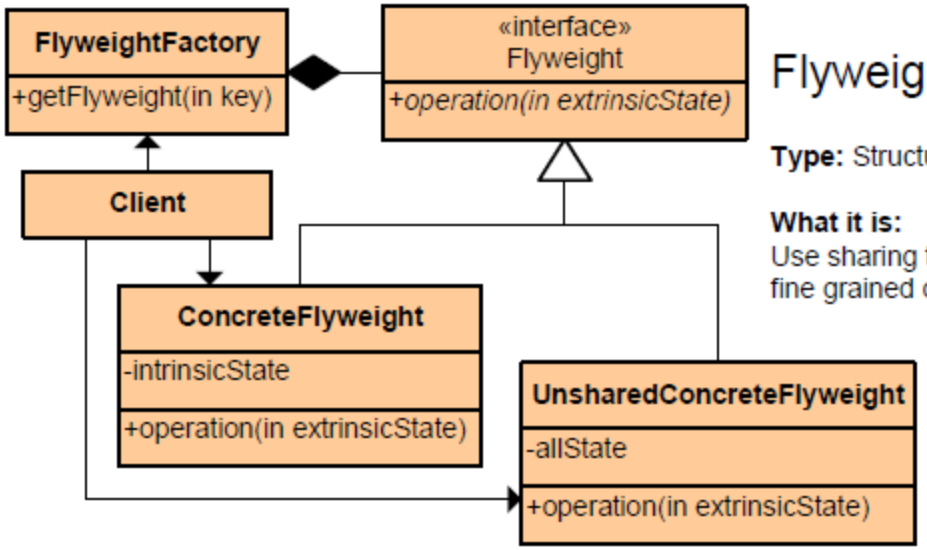


Facade

Type: Structural

What it is:
Provide a unified interface to a set of interfaces in a subsystem. Defines a high-level interface that makes the subsystem easier to use.

One-click-buy-button



Flyweight

Type: Structural

What it is:
Use sharing to support large numbers of fine grained objects efficiently.

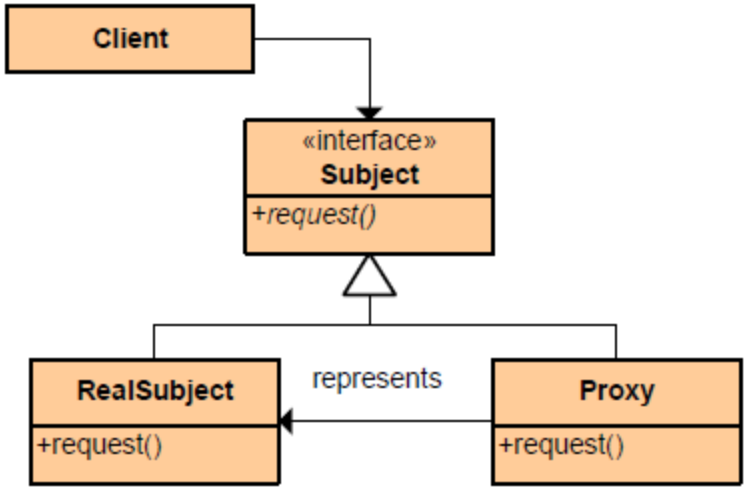
?

Design patterns

Proxy

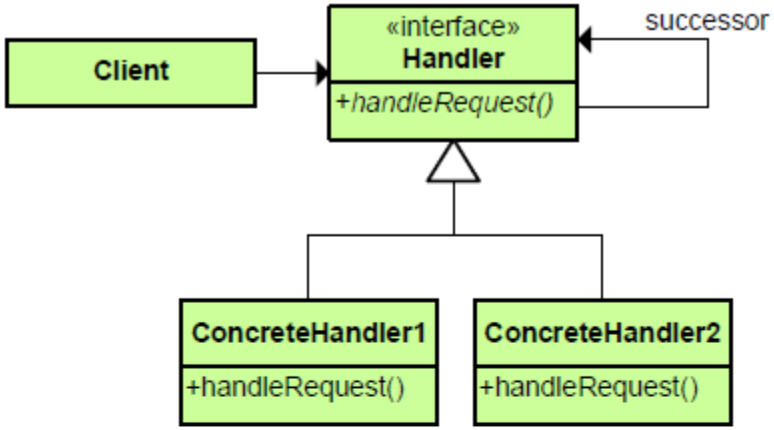
Type: Structural

What it is:
Provide a surrogate or placeholder for another object to control access to it.



Mocks, fakes

Design patterns

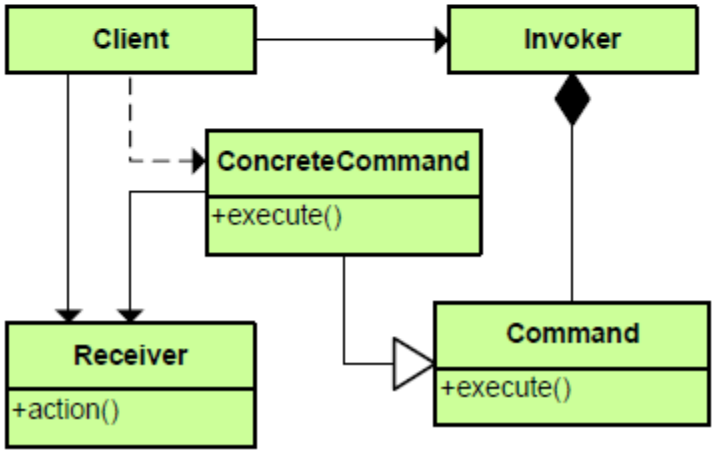


Chain of Responsibility

Type: Behavioral

What it is:
Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request. Chain the receiving objects and pass the request along the chain until an object handles it.

Event handler in browser



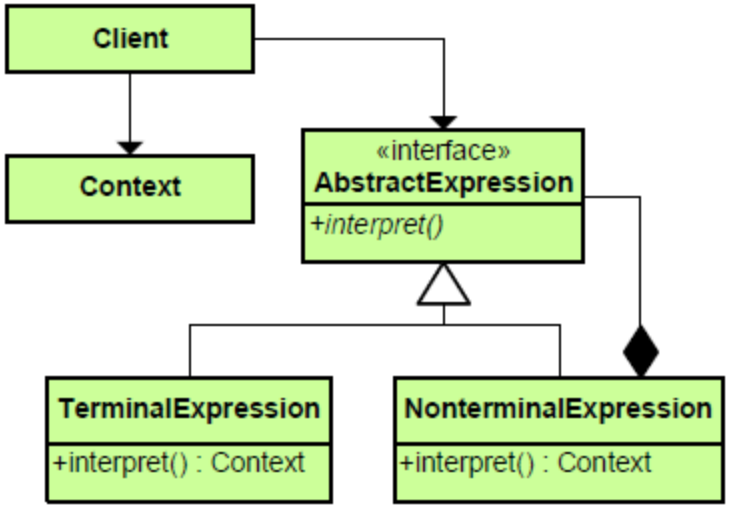
Command

Type: Behavioral

What it is:
Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations.

Undo/Redo

Design patterns

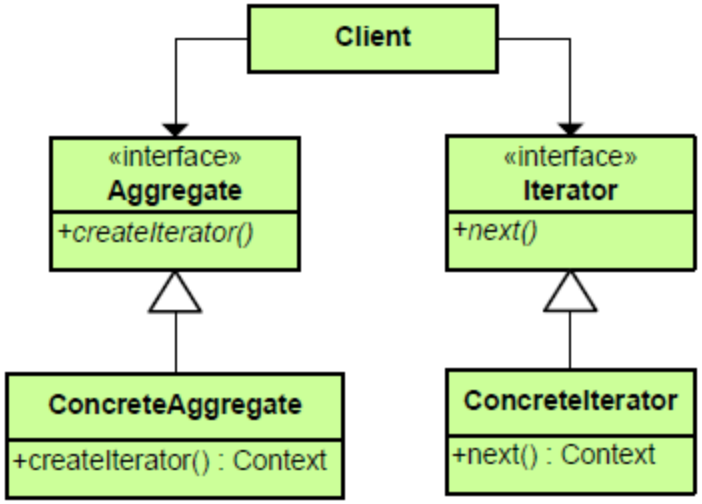


Interpreter

Type: Behavioral

What it is:
Given a language, define a representation for its grammar along with an interpreter that uses the representation to interpret sentences in the language.

DSL



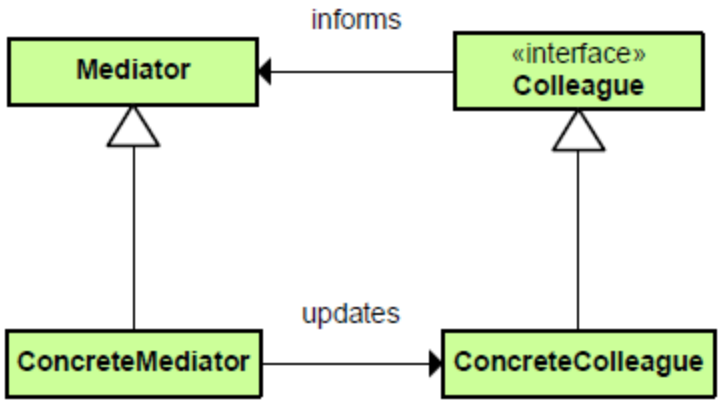
Iterator

Type: Behavioral

What it is:
Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.

foreach, pairs, intervals

Design patterns



Mediator

Type: Behavioral

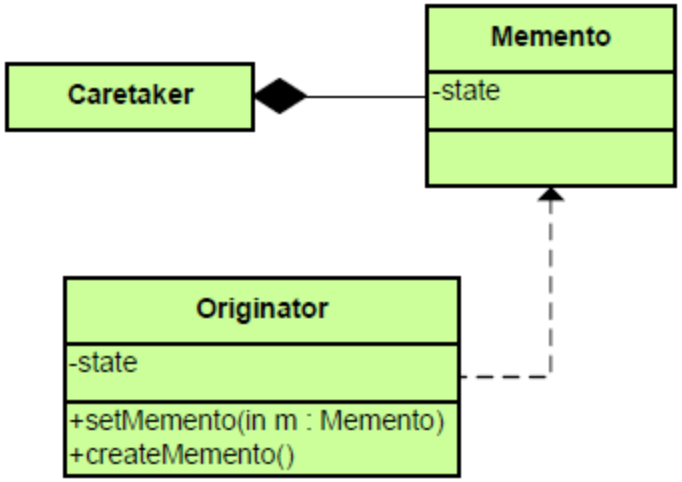
What it is:
Define an object that encapsulates how a set of objects interact. Promotes loose coupling by keeping objects from referring to each other explicitly and it lets you vary their interactions independently.

MVC

Memento

Type: Behavioral

What it is:
Without violating encapsulation, capture and externalize an object's internal state so that the object can be restored to this state later.



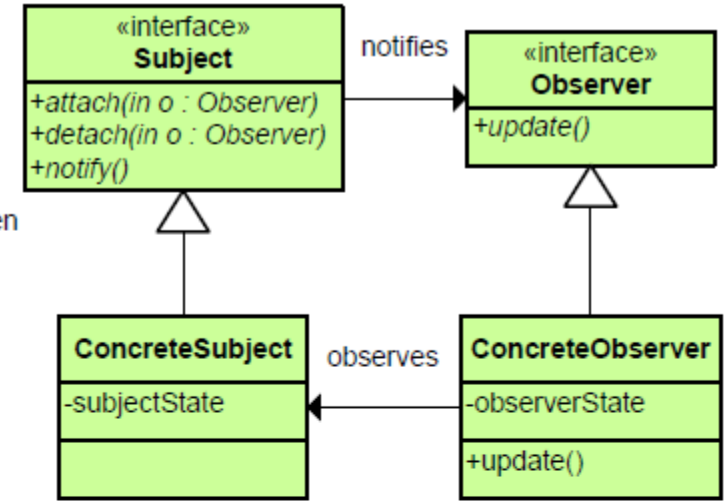
Save/Load
Serialization

Design patterns

Observer

Type: Behavioral

What it is:
Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.

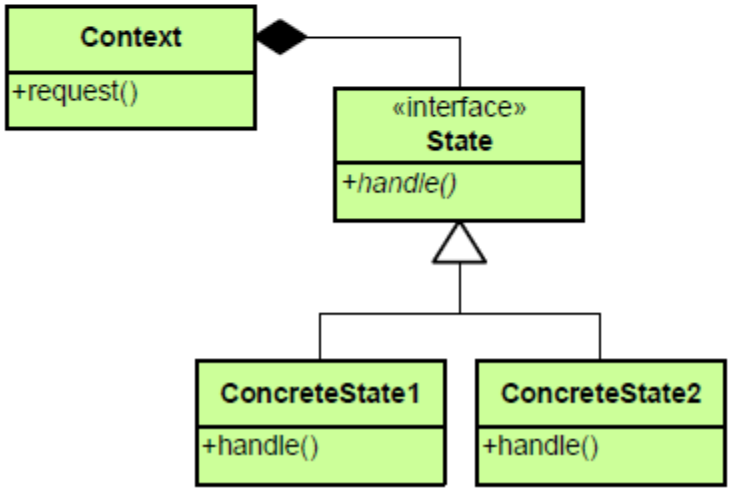


Broadcast mechanism

State

Type: Behavioral

What it is:
Allow an object to alter its behavior when its internal state changes. The object will appear to change its class.



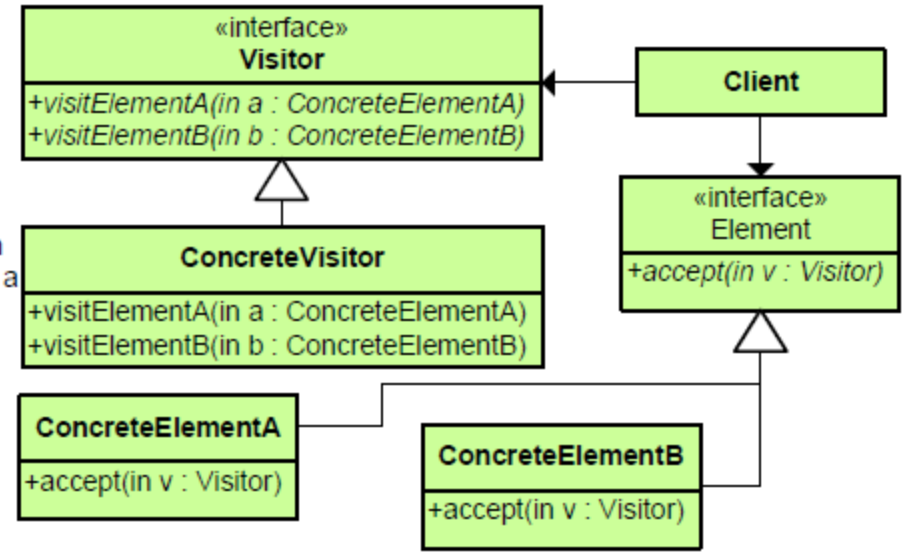
Finite State Machine

Design patterns

Visitor

Type: Behavioral

What it is:
Represent an operation to be performed on the elements of an object structure. Lets you define a new operation without changing the classes of the elements on which it operates.



Render hierarchy of object using different toolset (GPU/Sodt)

Law of Demeter

- Each unit should have only limited knowledge about other units: only units "closely" related to the current unit.
- Each unit should only talk to its friends; don't talk to strangers.
- Only talk to your immediate friends.

Best practices

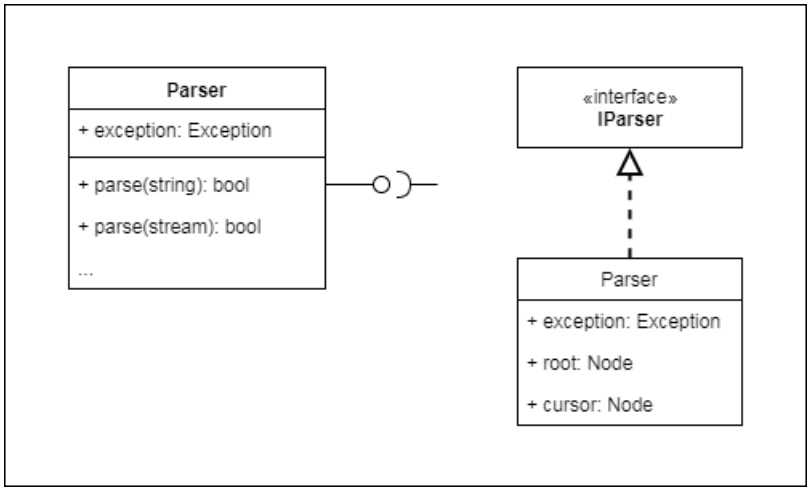
PP	Pareto Principle
KISS	Keep It Short and Simple
YAGNI	You Aren't Gonna Need It
NIH	Not Invented Here
HIN	Habit Inhibiting Novelty
DRY	Don't Repeat Yourself
OAOO	Once And Only Once
SSOT	Single Source Of Truth
SVOT	Single Version Of Truth
SoC	Separation of Concerns
YOLO	You Only Load it Once
POGE	Principle Of Good Enough
POLA	Principle Of Least Astonishment
SoD	Separate of Duties

S.O.L.I.D.

- S - Single Responsibility Principle
- O - Open-Closed Principle
- L - Liskov Substitution Principle
- I - Interface Segregation Principle
- D - Dependency Inversion Principle

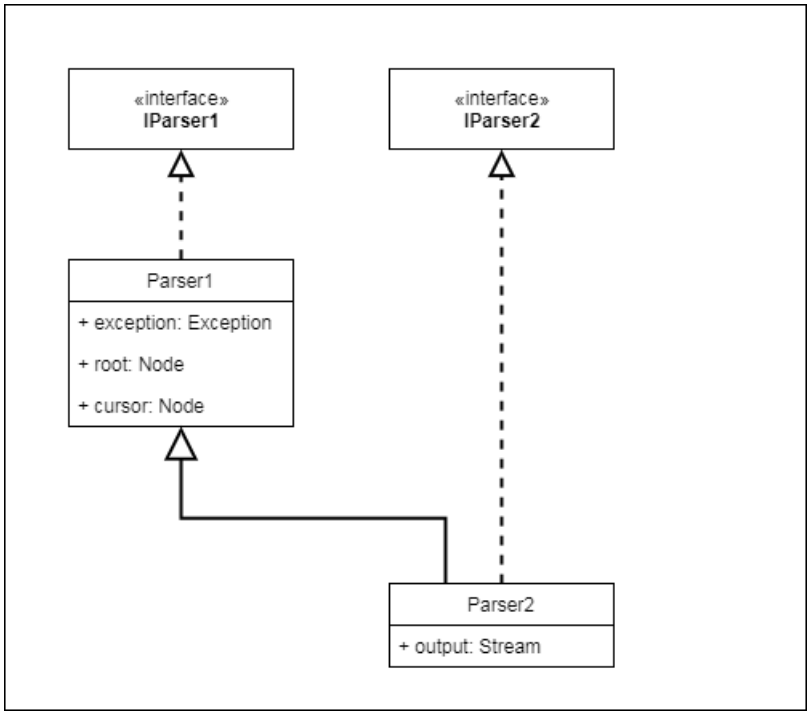
S.O.L.I.D. - Single Responsibility Principle

- **Single** interface to implement
- **Single** reason to modify



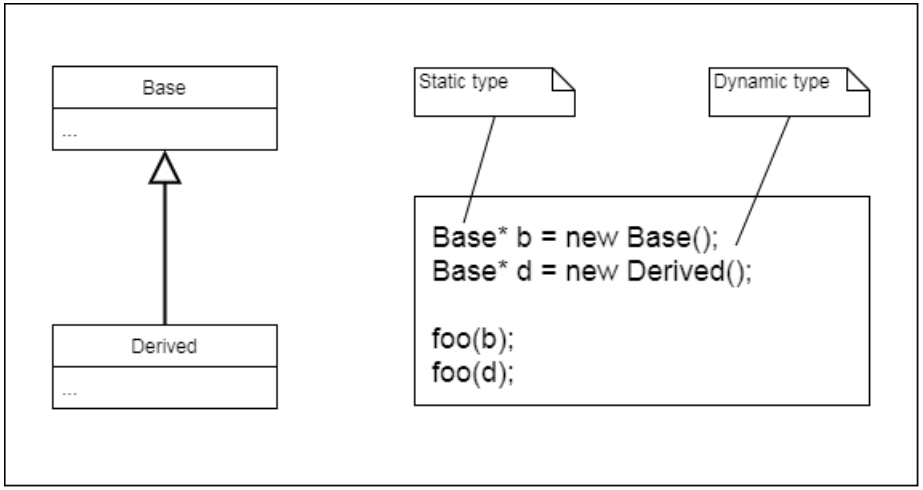
S.**O**.L.I.D. - Open-Closed Principle

- Interface should be **closed** for modification
- Interface can be **opened** for extension
- Interface is **opened** for errors fixing



S.O.**L**.I.D. - Liskov Substitution Principle

- **Derived** class instance **have** to be able to **use** anywhere **instead** of Base
- **Derived** class can make **pre conditions weaker** and/or **post conditions stronger** (requiring less and guaranteeing more)



S.O.L.I.D. - Interface Segregation Principle

- Orthogonal interface

S.O.L.I.**D**. - Dependency Inversion Principle

- **Minimal relations** between classes
- Depends on **interface** not **implementation**