Software Technology 08

Design Patterns

See https://refactoring.guru/design-patterns/catalog

Design Pattern

- Best practices
- Good examples
- Never code it directly!

    → Always write custom code tailored to your specific needs
- Common way of referring problems
- Naming convention
- Low-level stuff works high-level as well

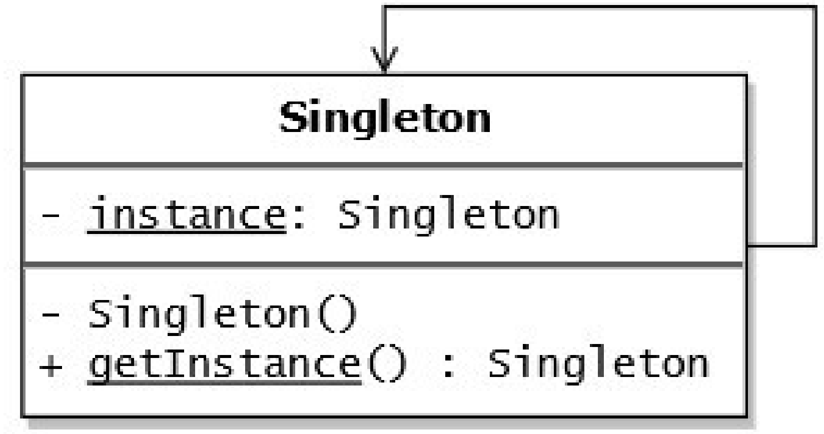
Design Pattern Types

- Creational
- Structural
- Behavioral
- Concurrency
- Architectural
- Distributed
- Algorithm Strategy
- Implementation Strategy

Creational Design Patterns

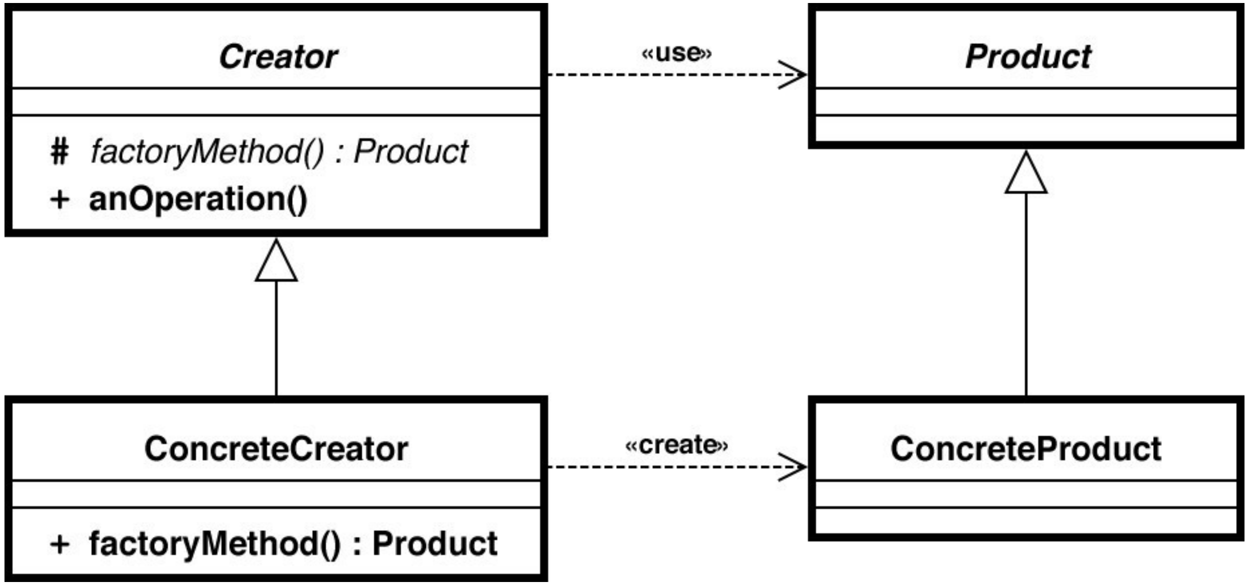
Singleton

- Can be an anti-pattern
- Testing



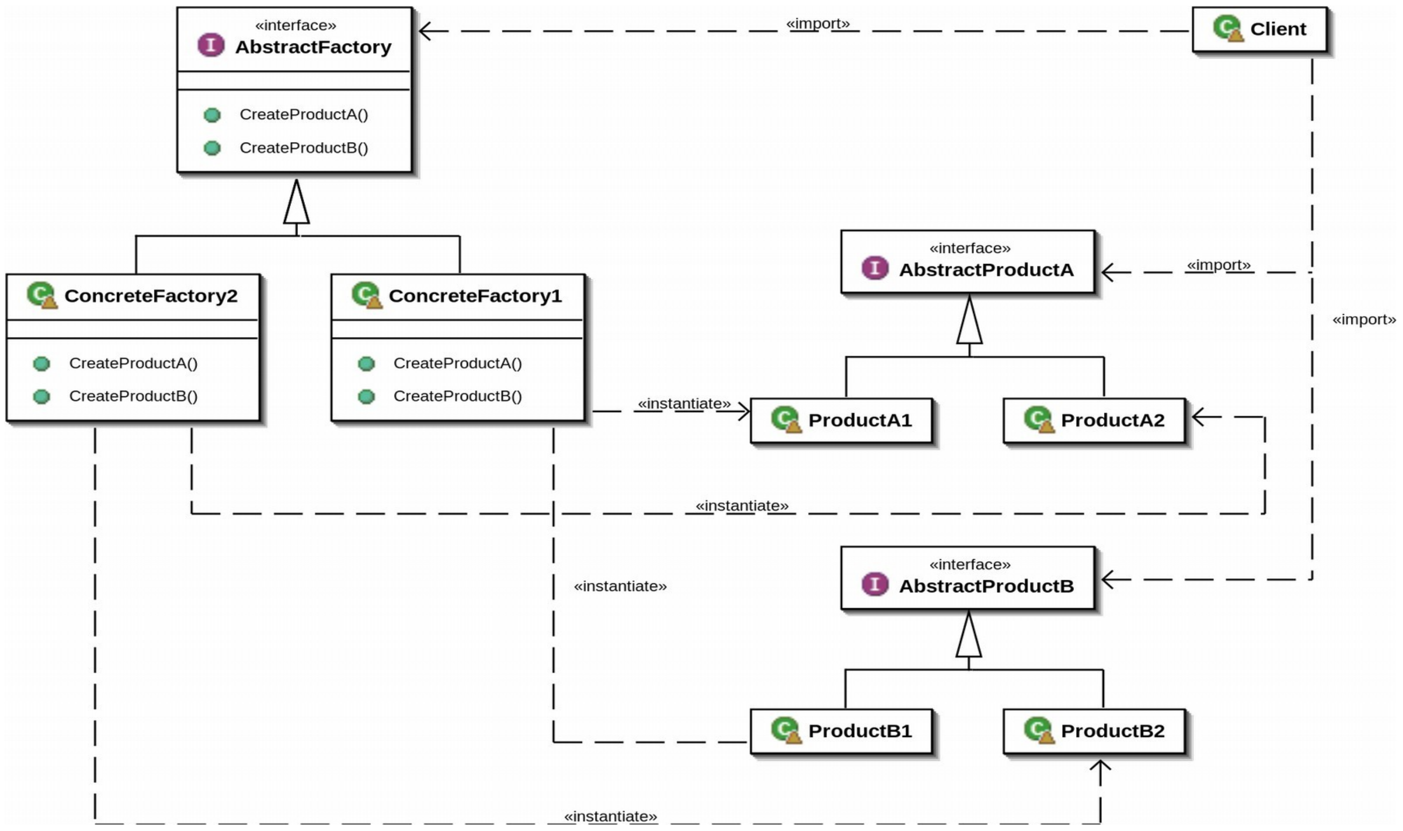
Factory Method

- Dependency Injection

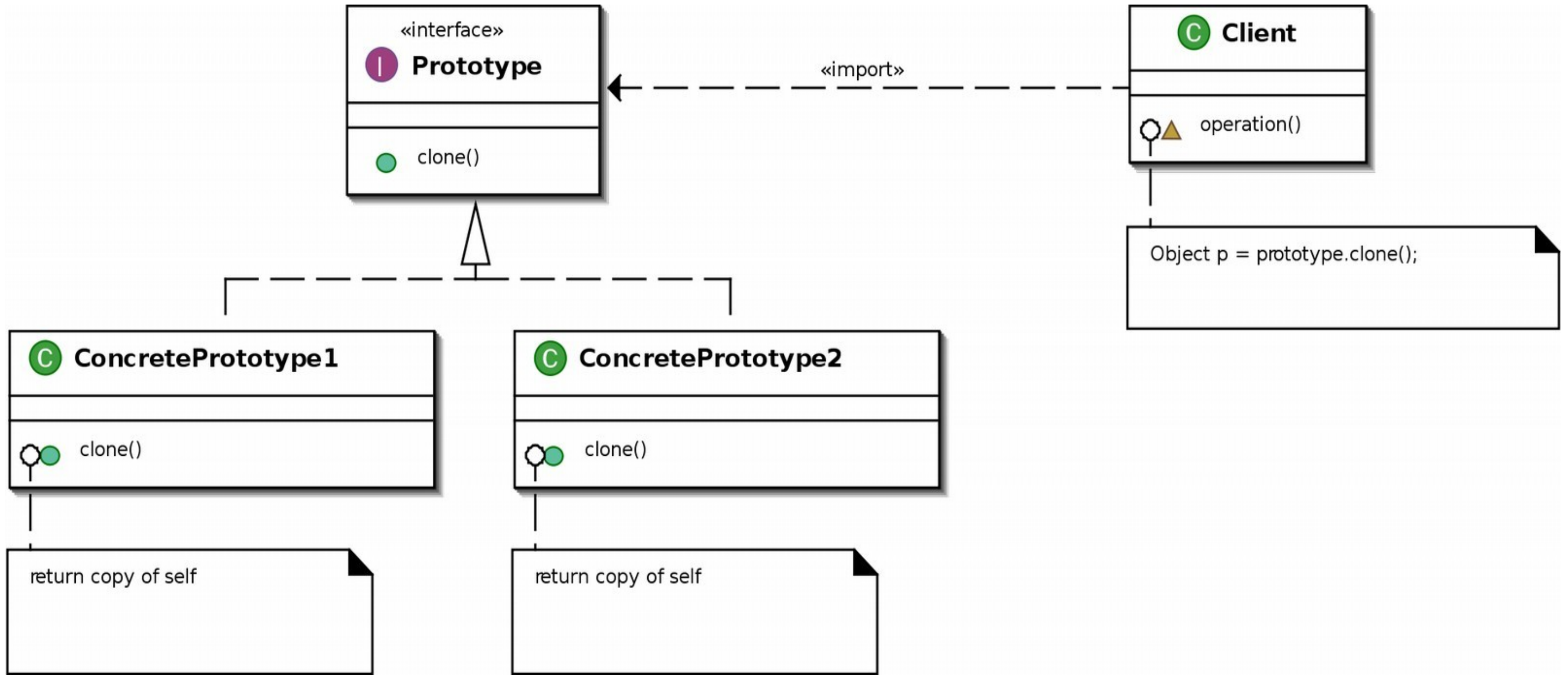


Abstract Factory

- Encapsulate multiple factories hiding implementation



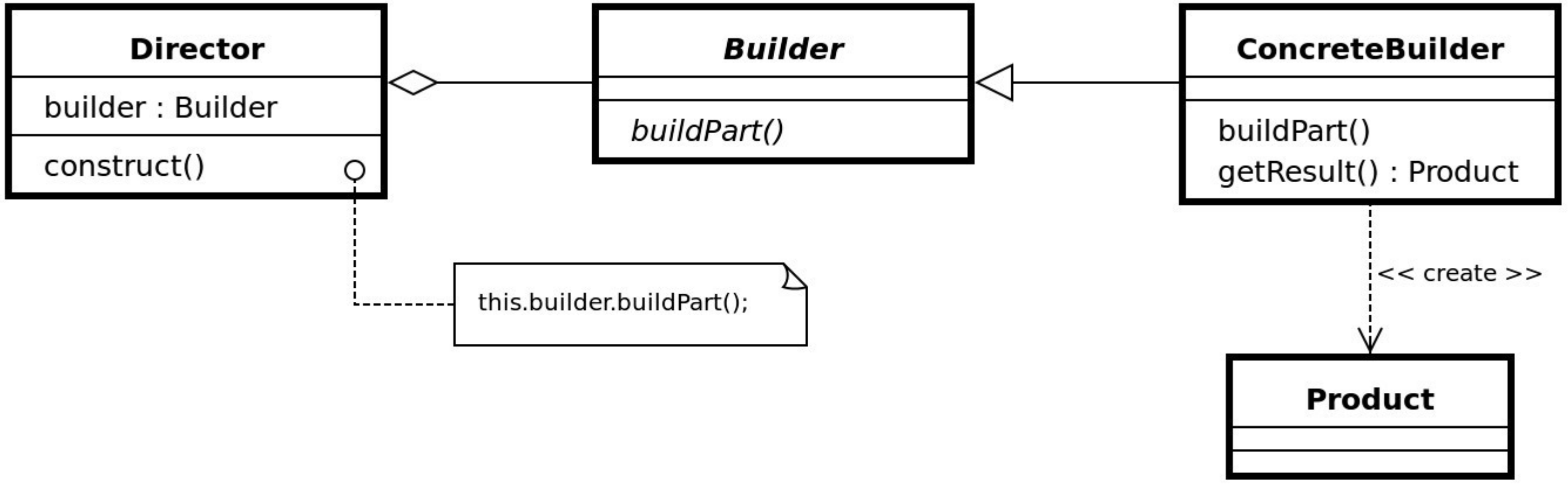
Prototype



«interface»
Prototype
clone()
«import»
Client
operation()
Object p = prototype.clone();
ConcretePrototype1
clone()
ConcretePrototype2
clone()
return copy of self
return copy of self

Builder

- Instead of constructors of long / different parameter lists
  → Use setters and a factory method

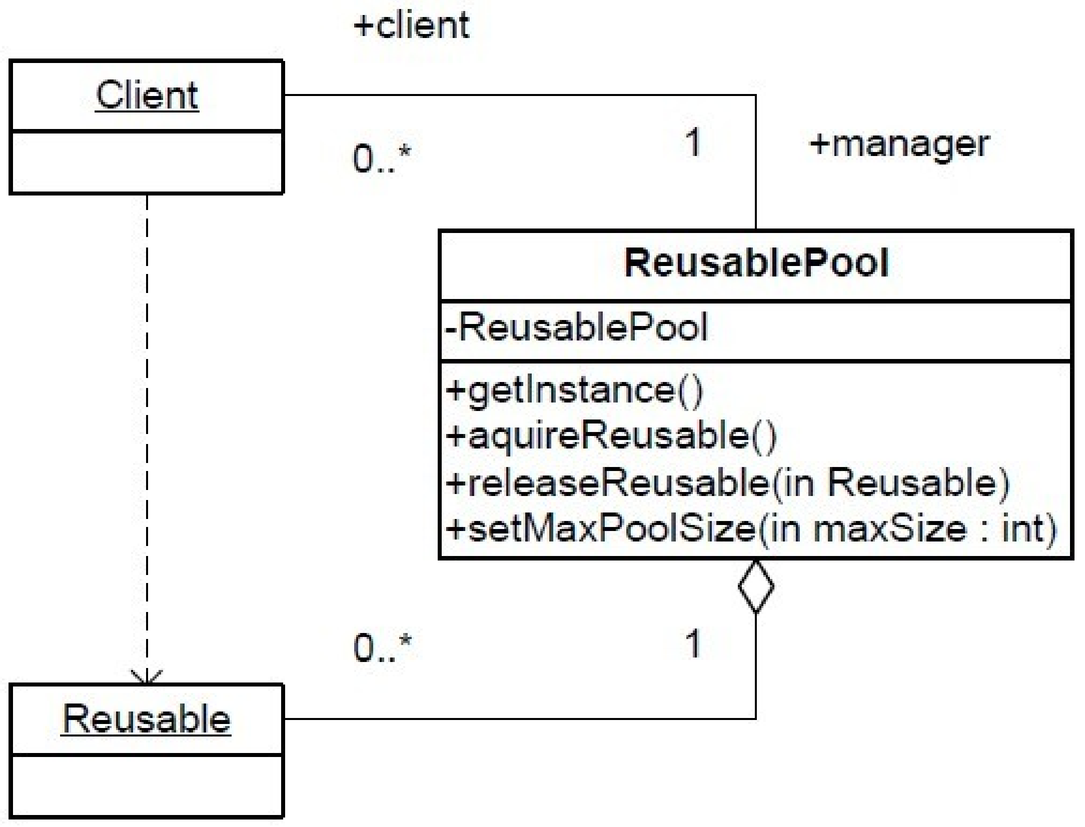


Lazy Initialization

- Only initialize when needed!
- Implementation: Something is `null` until it is not required
- See Singleton!!!!

Object Pool

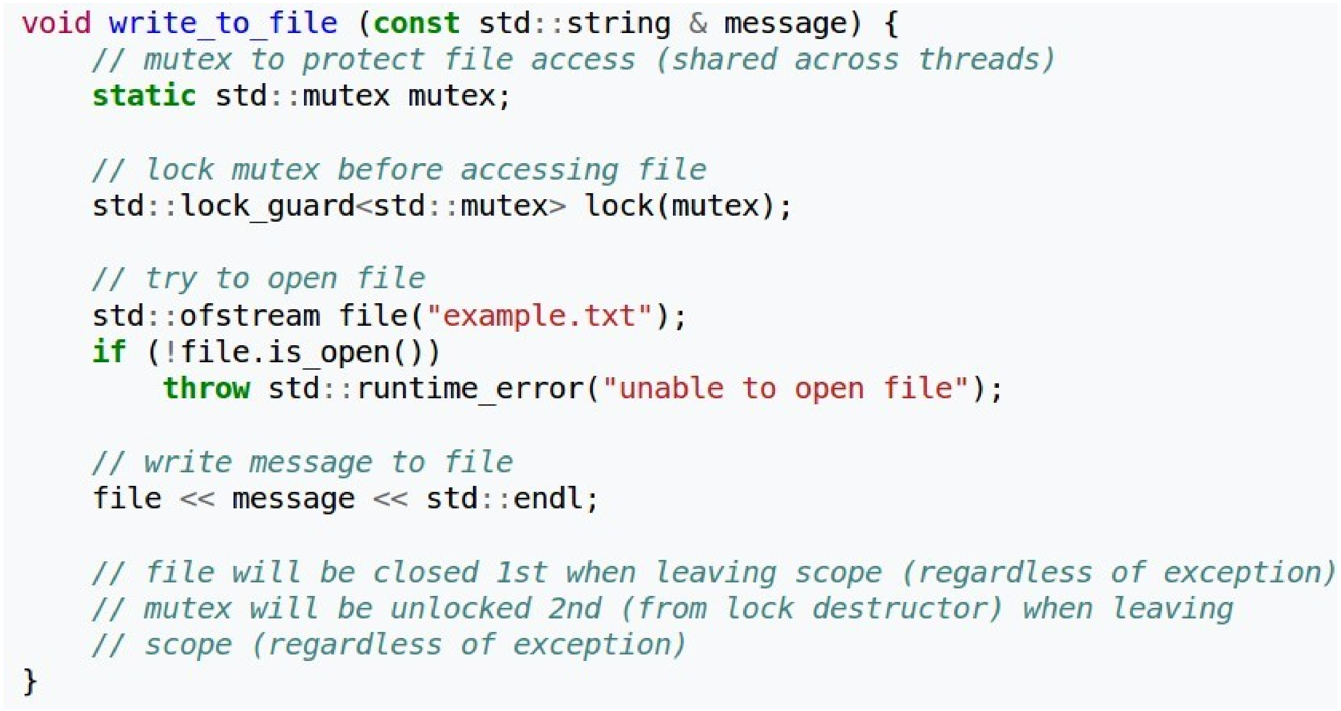
- The Pool itself can be a Singleton



RAII

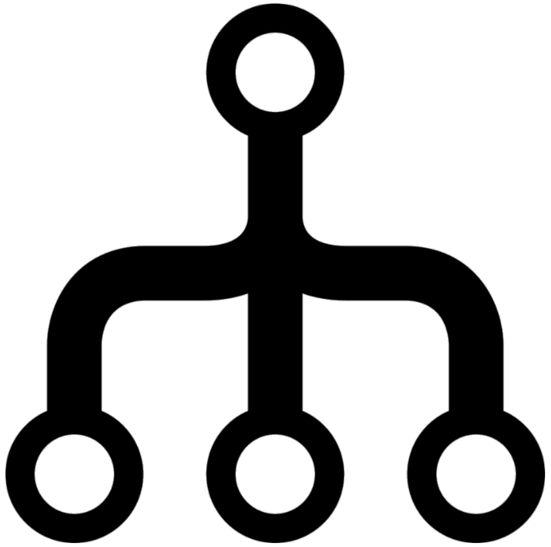
– Resource Acquisition Is Initialization

- Constructor Acquires, Destructor Releases
- Scope-Based Resource Management



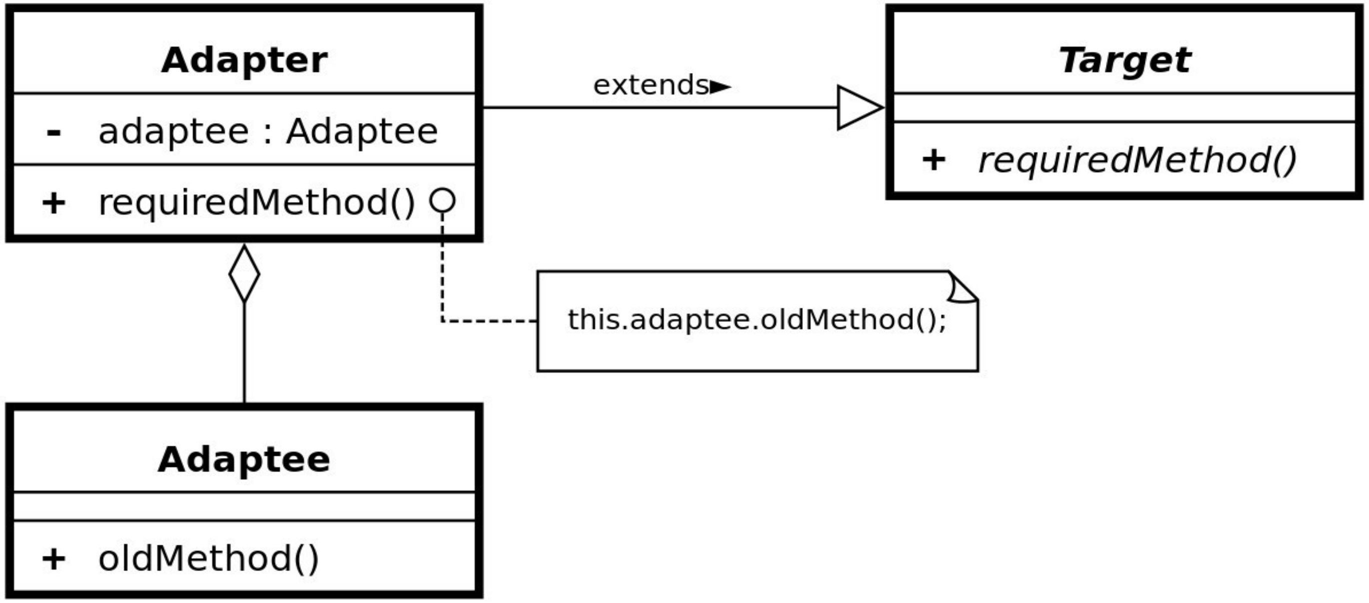
```cpp
void write_to_file (const std::string & message) {
    // mutex to protect file access (shared across threads)
    static std::mutex mutex;

    // lock mutex before accessing file
    std::lock_guard<std::mutex> lock(mutex);

    // try to open file
    std::ofstream file("example.txt");
    if (!file.is_open())
        throw std::runtime_error("unable to open file");

    // write message to file
    file << message << std::endl;

    // file will be closed 1st when leaving scope (regardless of exception)
    // mutex will be unlocked 2nd (from lock destructor) when leaving
    // scope (regardless of exception)
}
```

Structural Design Patterns



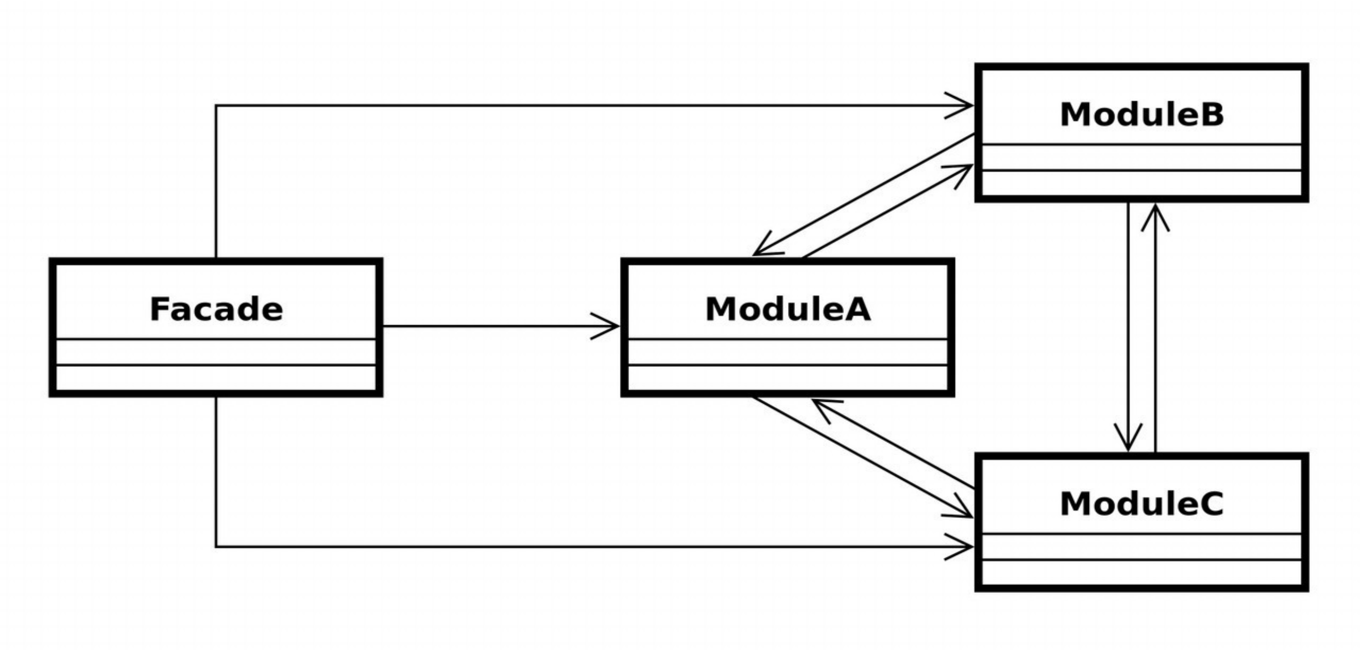
Adapter

alias Wrapper or Translator



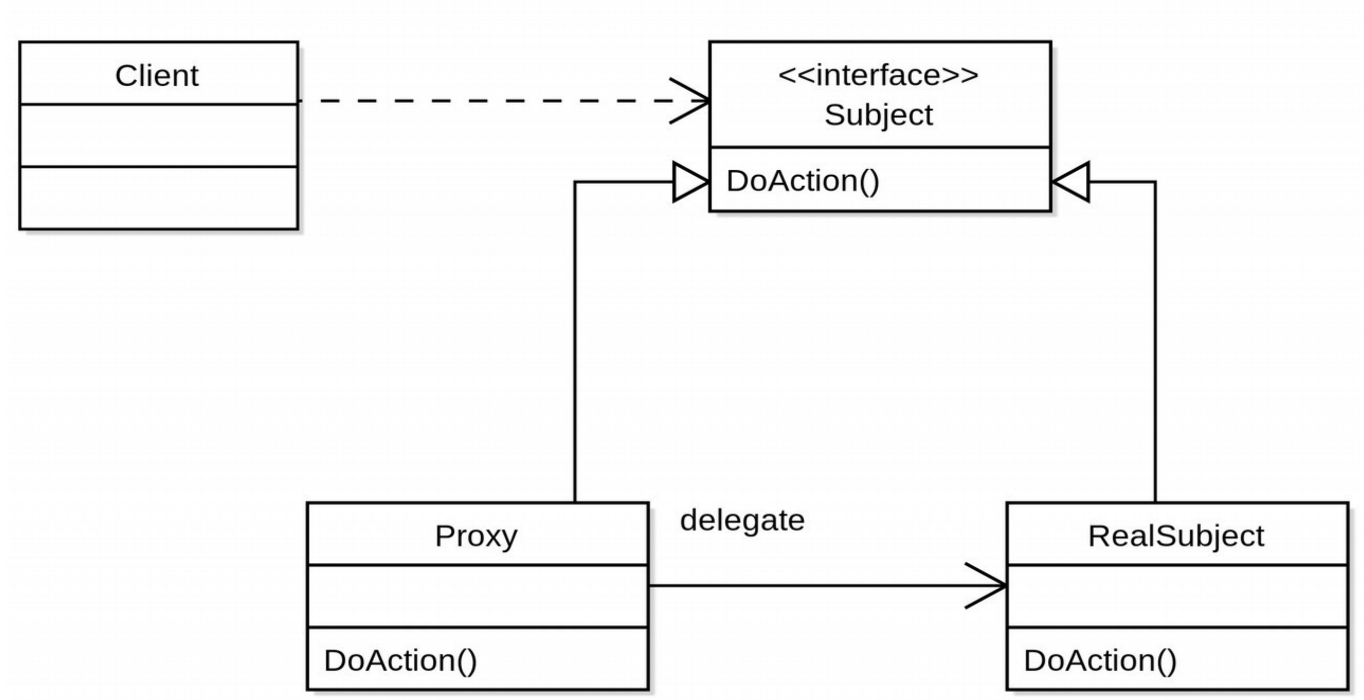
Facade

- Simplified interface to something complex



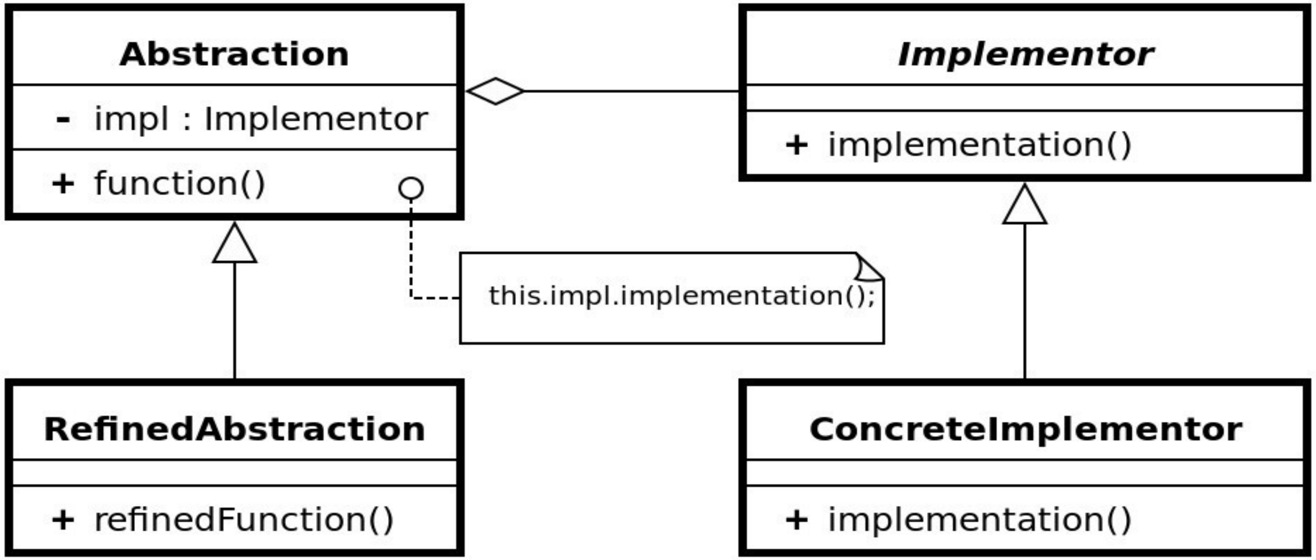
Proxy

- RMI (Remote Method Invocation)
- Protection
- Caching

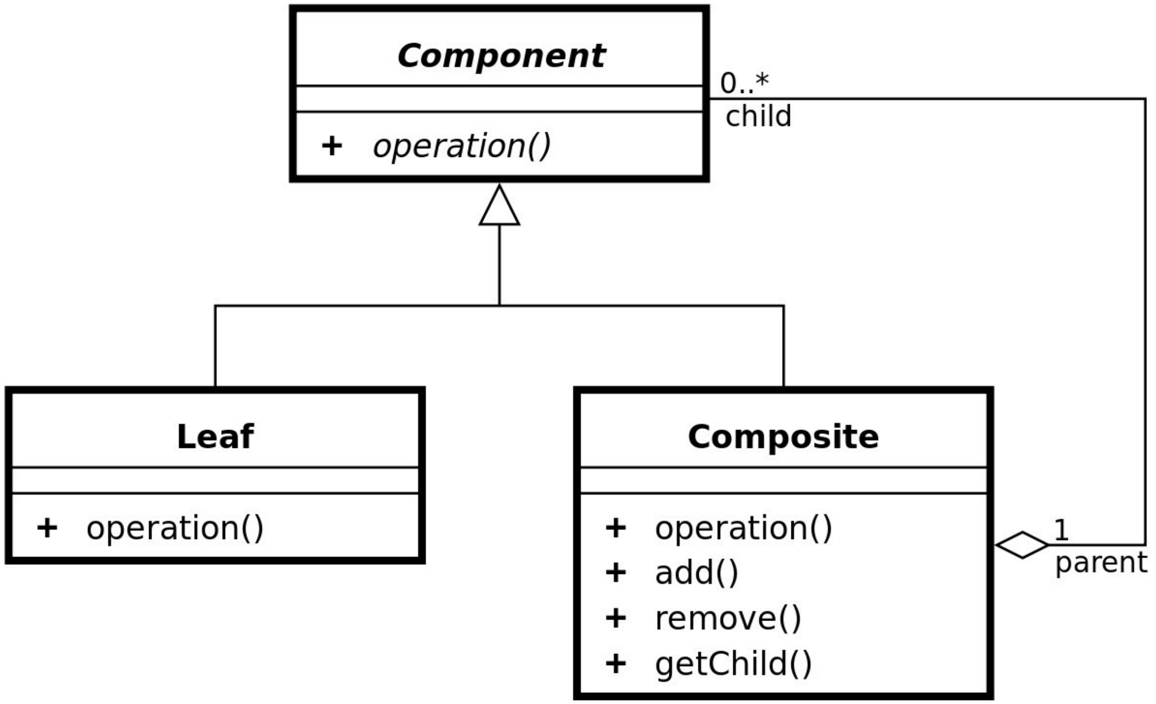


Bridge

- Decouple abstraction and implementation to make them independent
- `pimpl` idiom

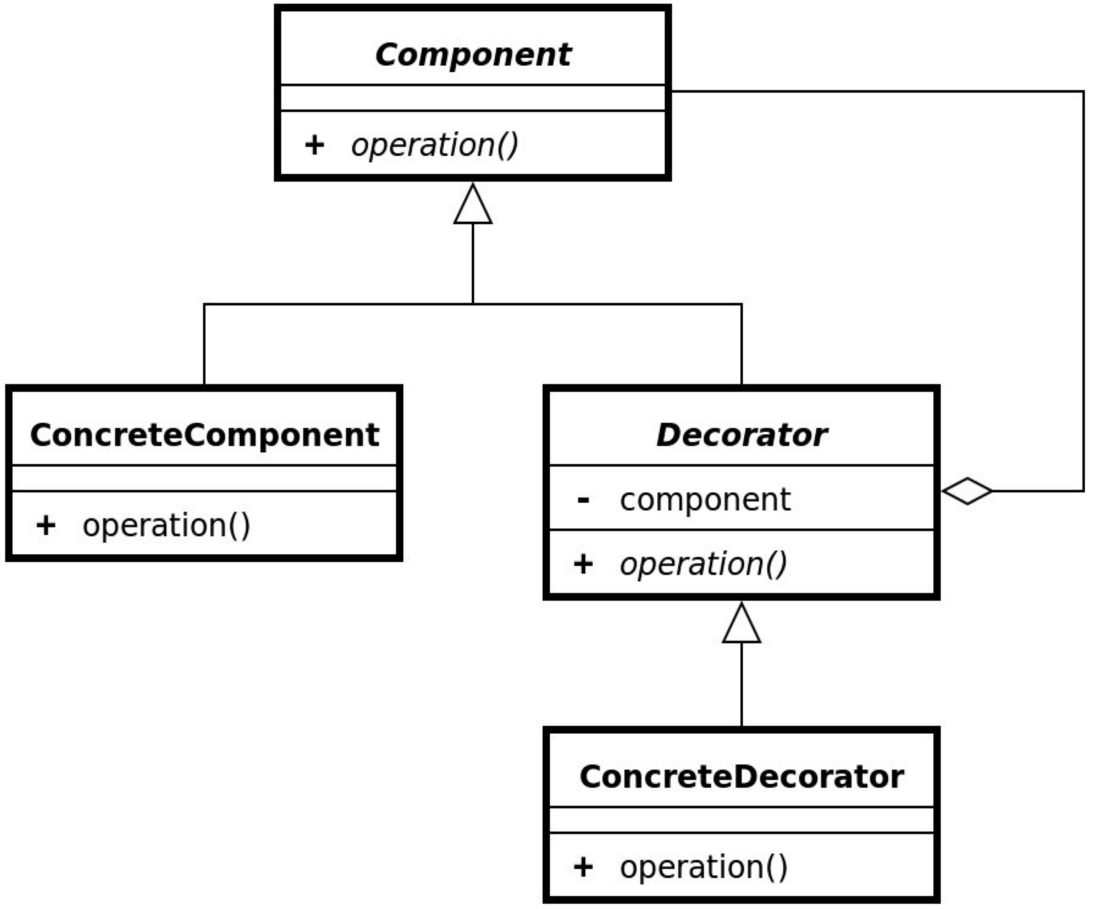


Composite


Component
+ operation()
0..*
child
Leaf
+ operation()
Composite
+ operation()
+ add()
+ remove()
+ getChild()
1
parent

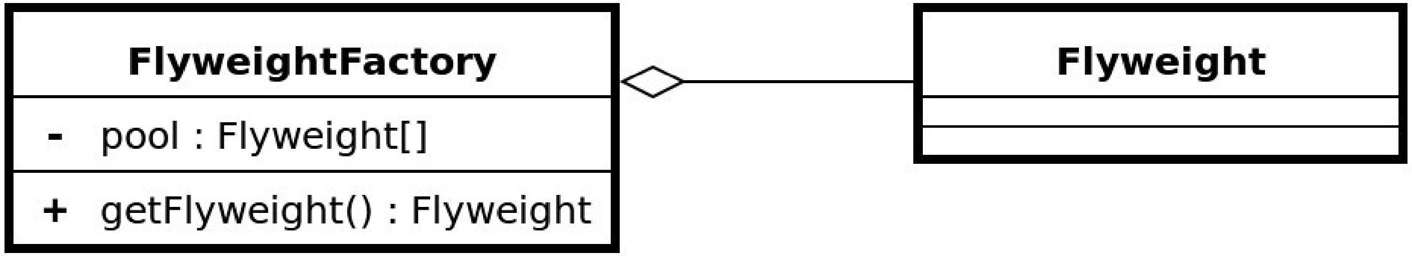
Decorator

- for Single Responsibility Principle
- Subclassing at run-time



Flyweight

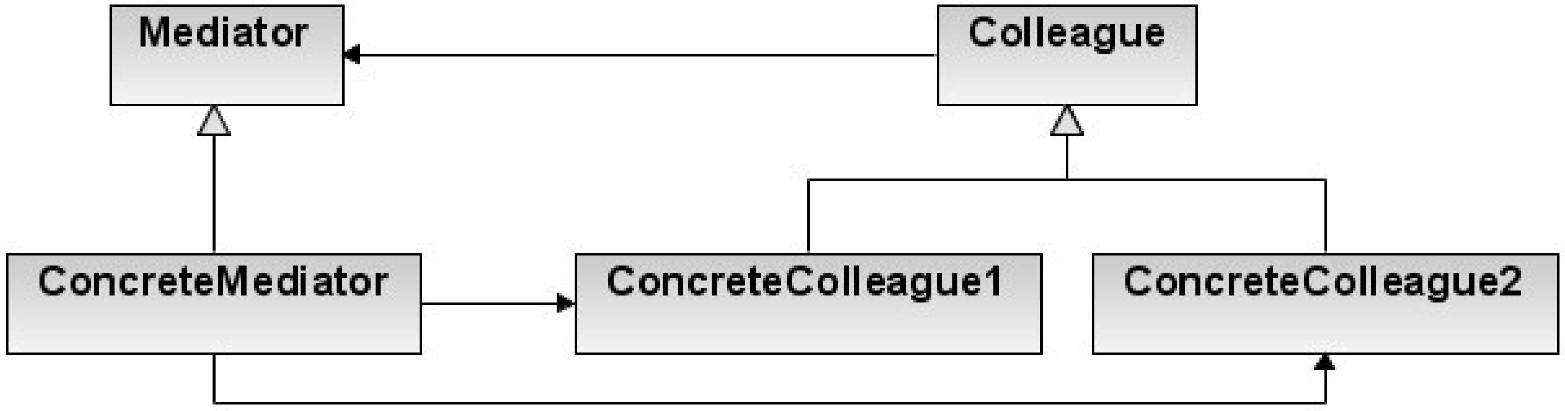
- Optimize storage of a lot of objects in special structure
- String interning
- Copy-on-write



Behavioral Design Patterns

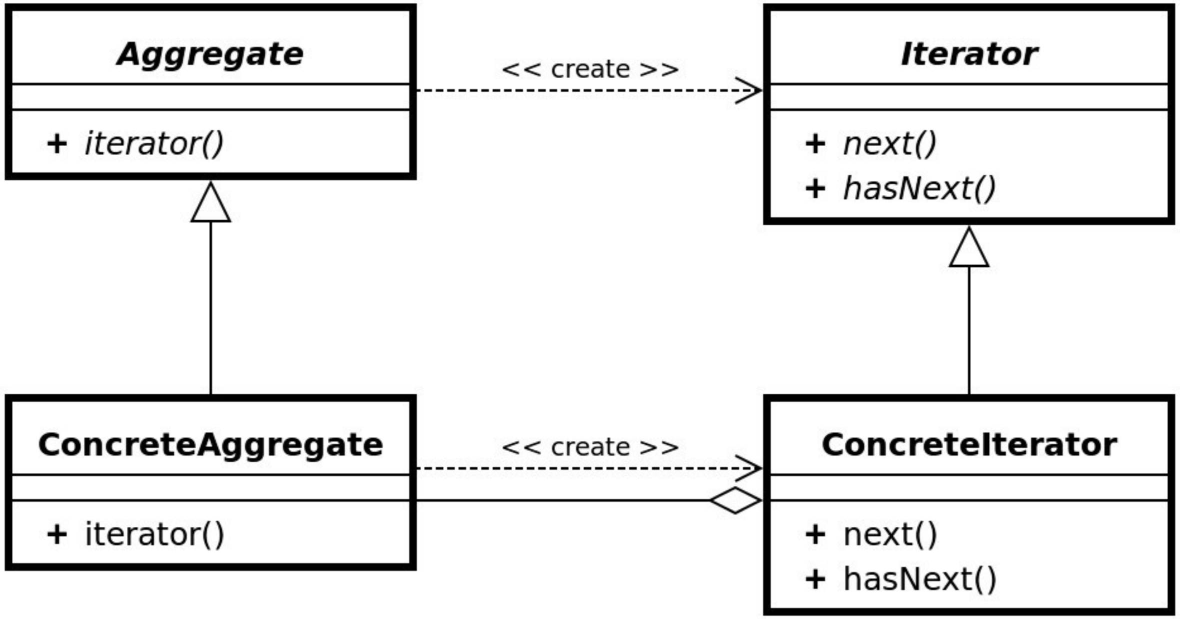
Mediator

- for Refactoring
- Keep objects from directly referencing each other (loose coupling)



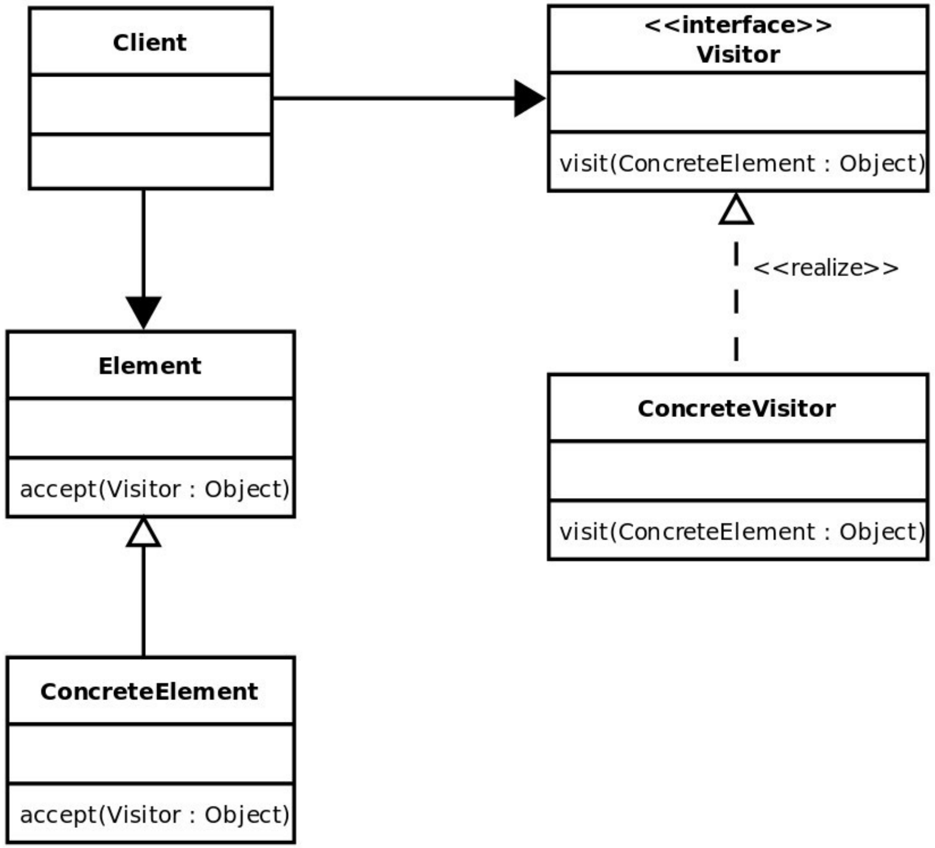
Iterator

- On abstract aggregate object
- Hide implementation



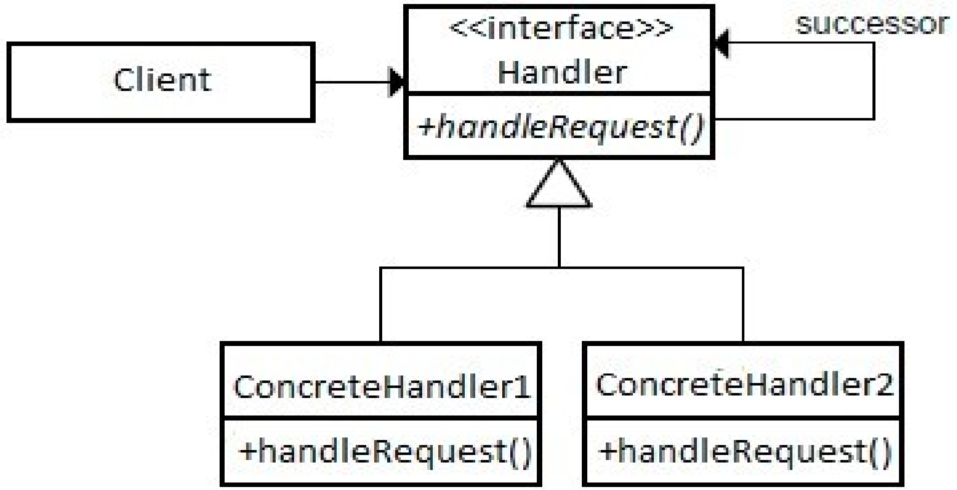
Visitor

- New function to all aggregate members without modifying them (API plug-in)
- Double dispatch (Element and Visitor chosen dynamically)



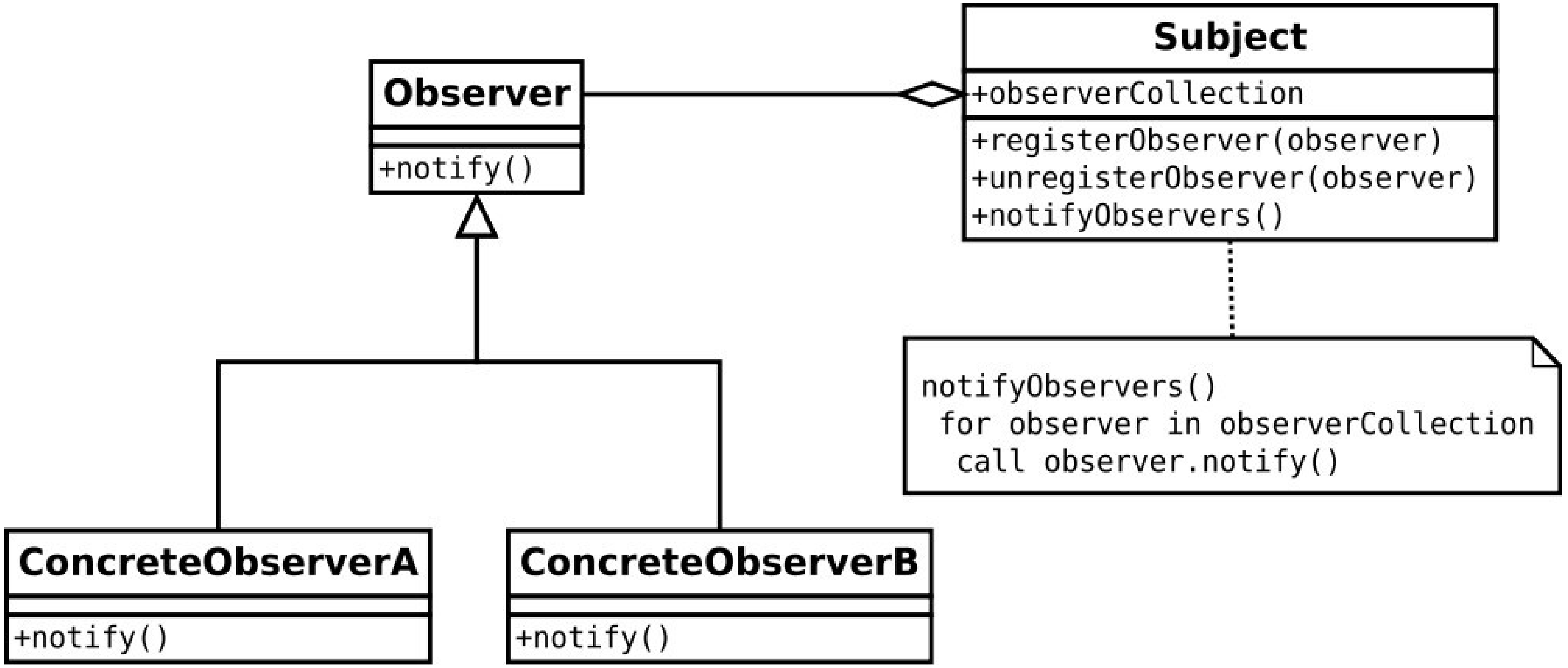
Chain of Responsibility

- Recursive dispatch of not handled events



Observer

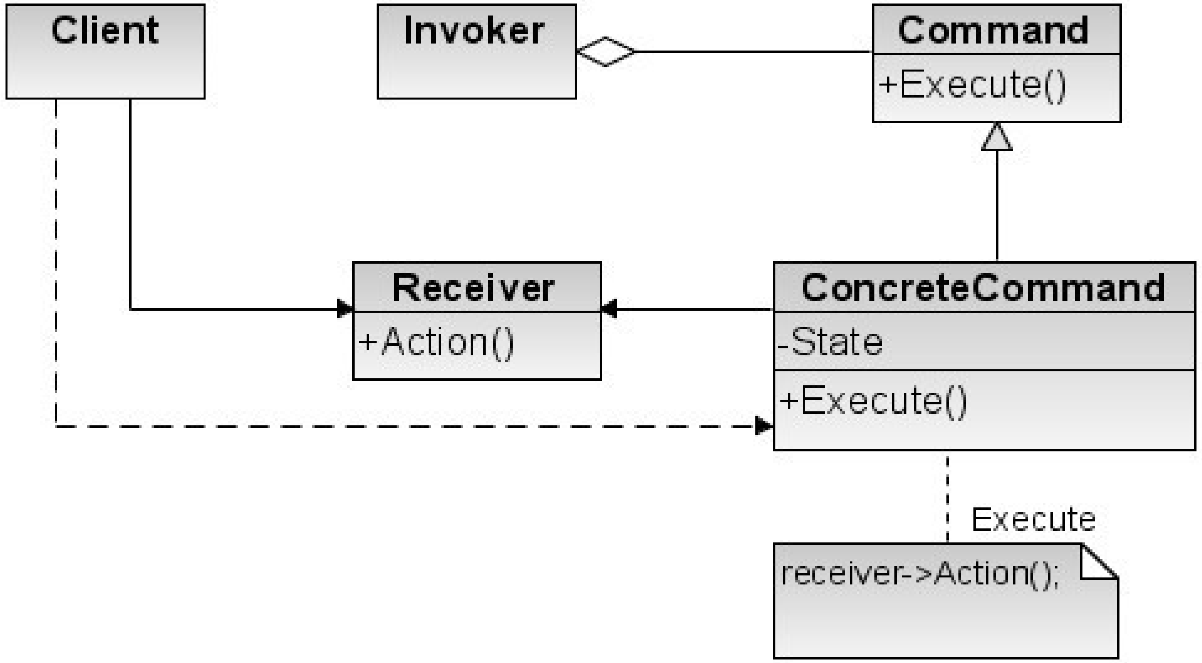
- Event handling



```
notifyObservers()
 for observer in observerCollection
  call observer.notify()
```

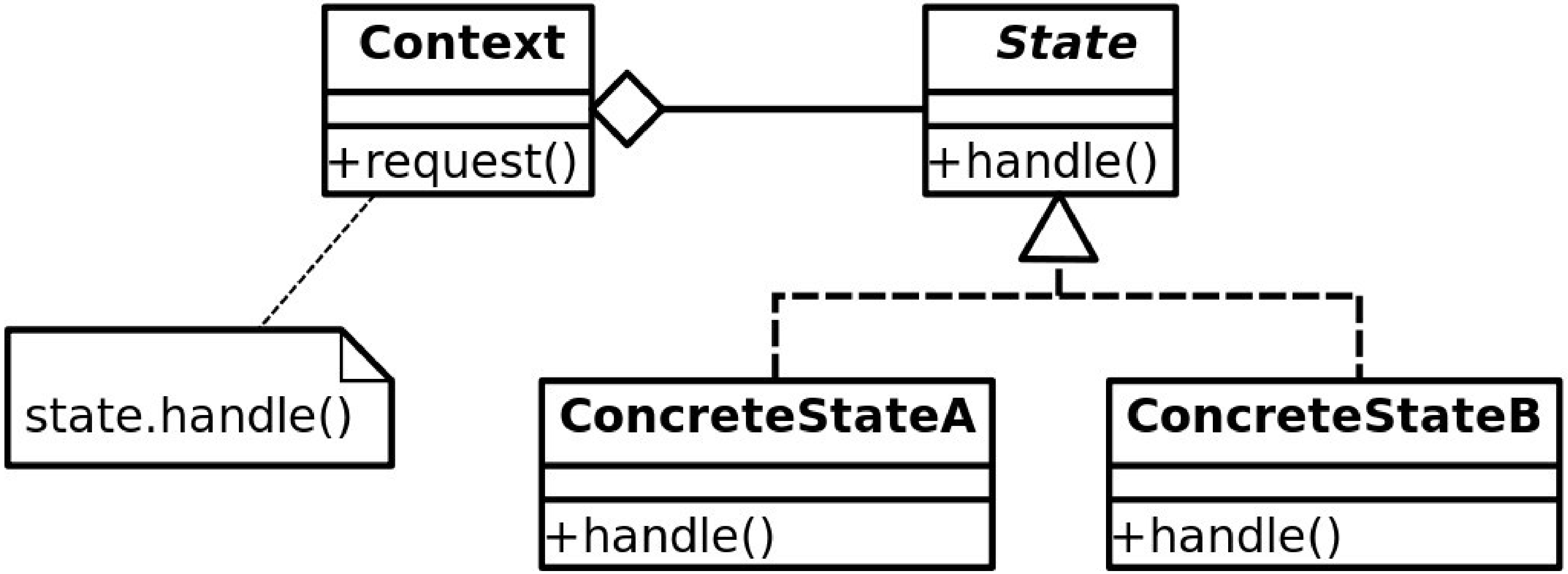
Command

- Encapsulate everything about an action
- Undo



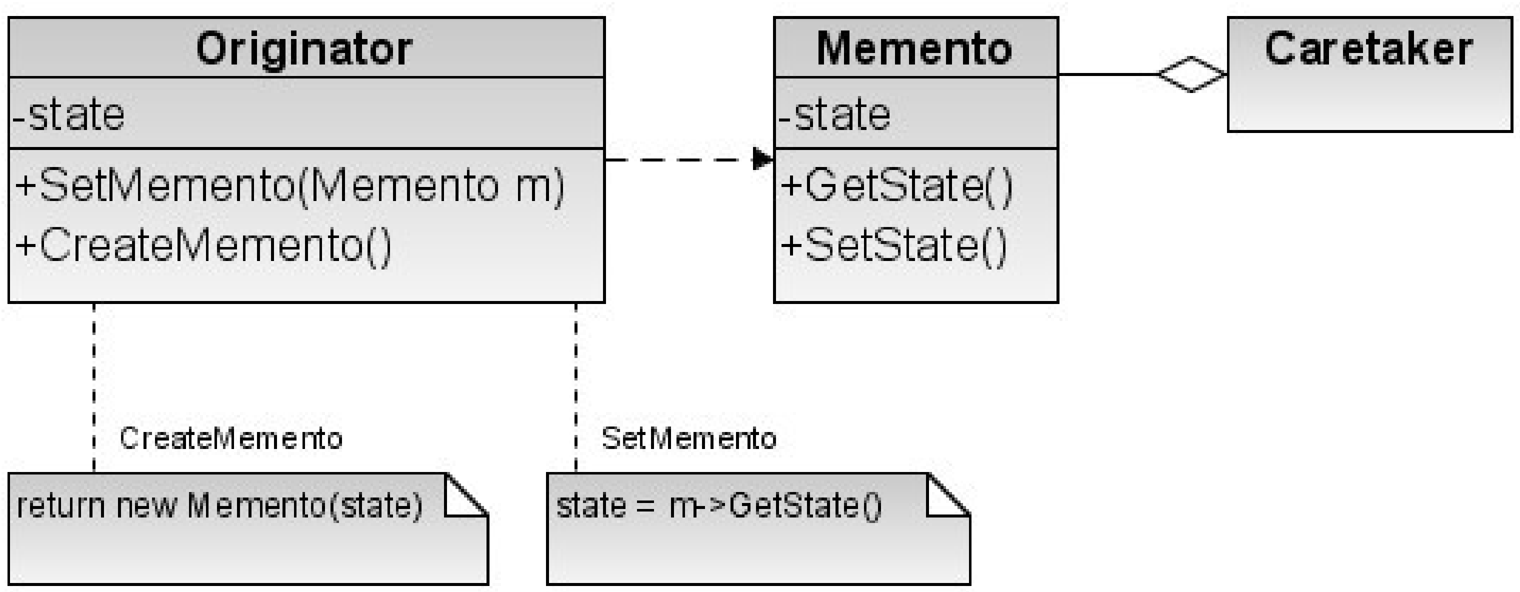
State

- State machine with definable transition actions
- Same call, state dependent behavior



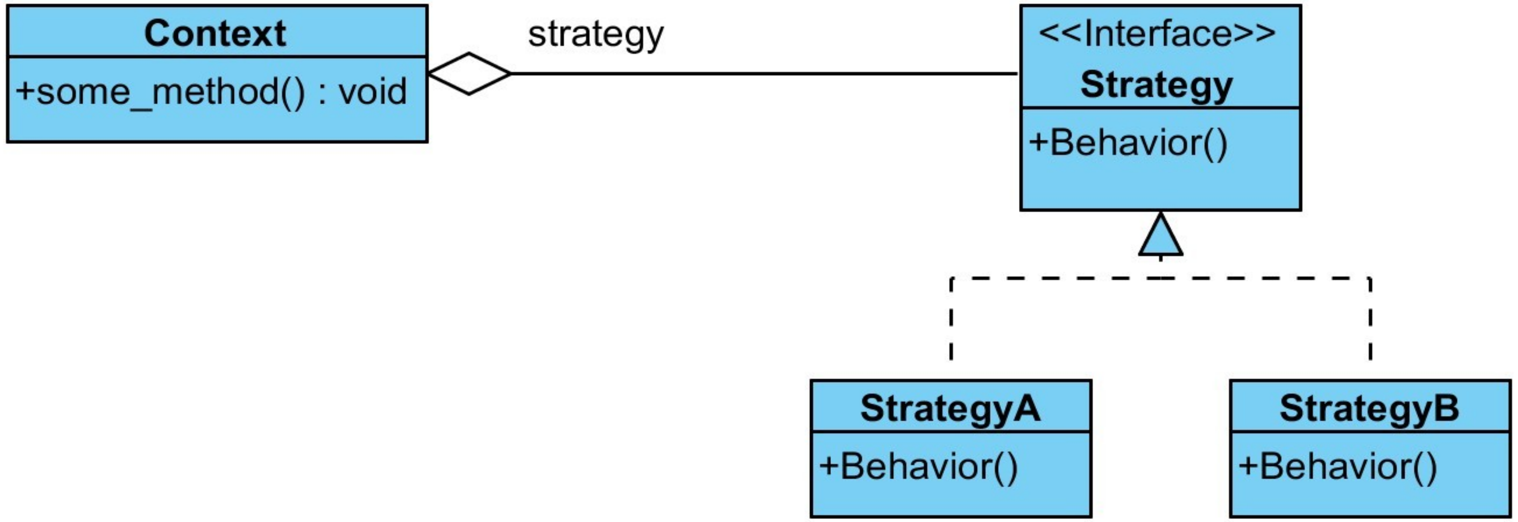
Memento

- Save, Serialize, Restore...



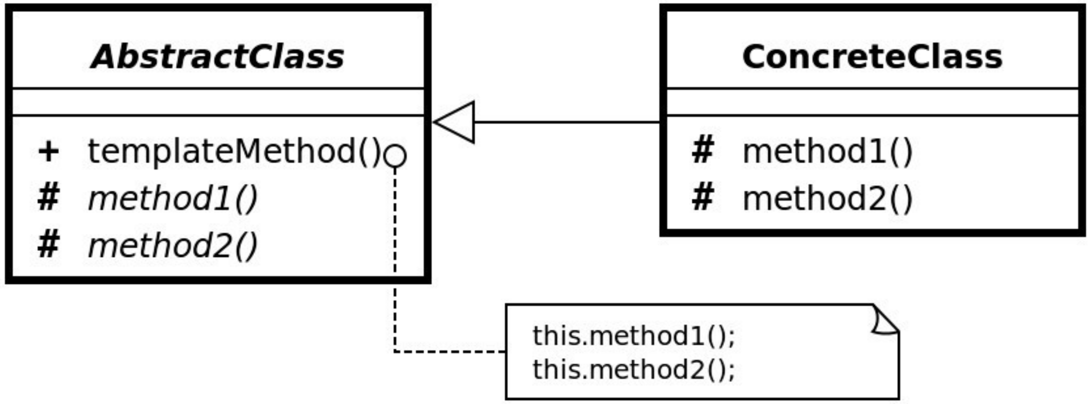
Strategy

- Encapsulate different algorithms coupled to data depending on its type



Template Method

- The algorithm / class defers some work to subclasses



```
this.method1();
this.method2();
```

Concurrency Design Patterns

Monitor Object

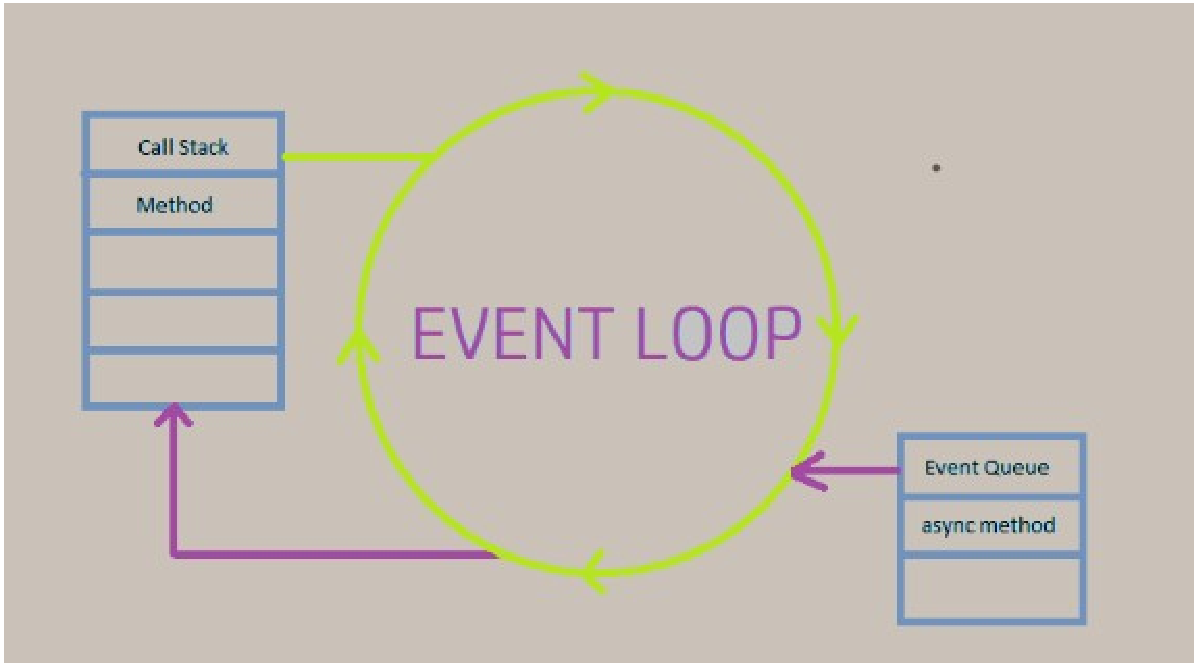
- Mutual Exclusion
- Memory access, cache
- Spin-Lock
- Lock, wait
- Condition variable

Active Object

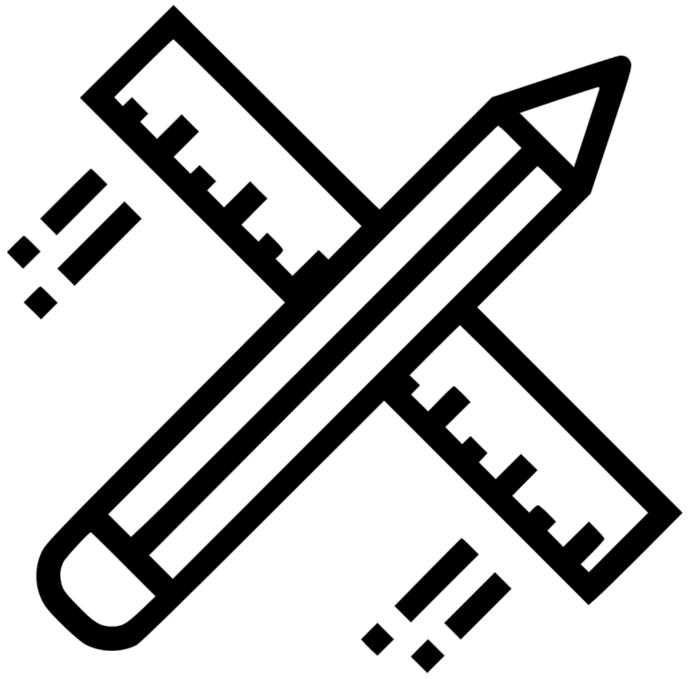
- Decouple method execution from invocation (on separate threads)
- Futures and Promises
- Asynchronous method invocation
  - Proxy
  - Scheduler
  - Implementation
  - Variable with result or Callback

Reactor

- Event loop
- Demultiplexes events synchronously
- Inverted control flow (dispatch depends on arrived event)

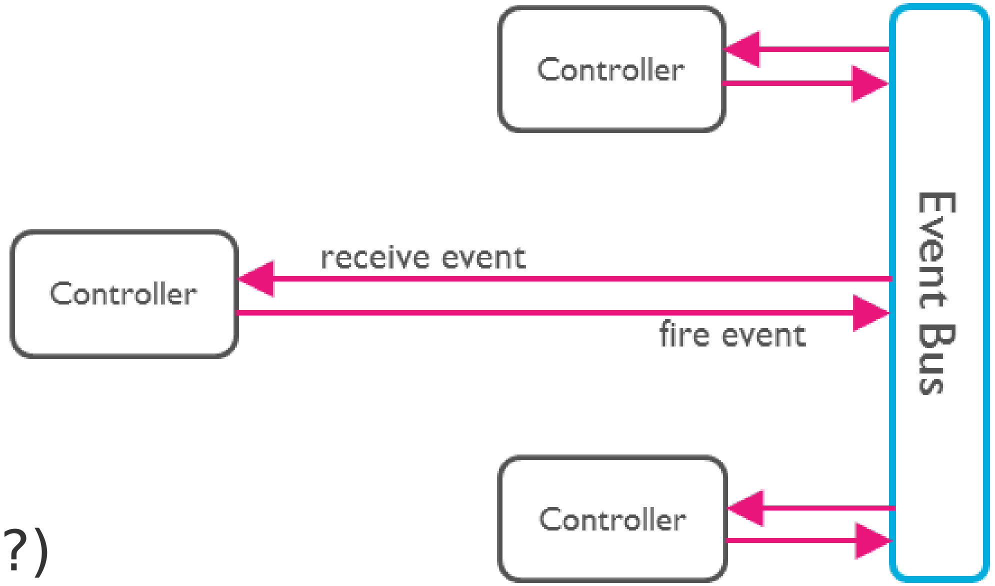


Architectural Design Patterns



Publish-Subscribe

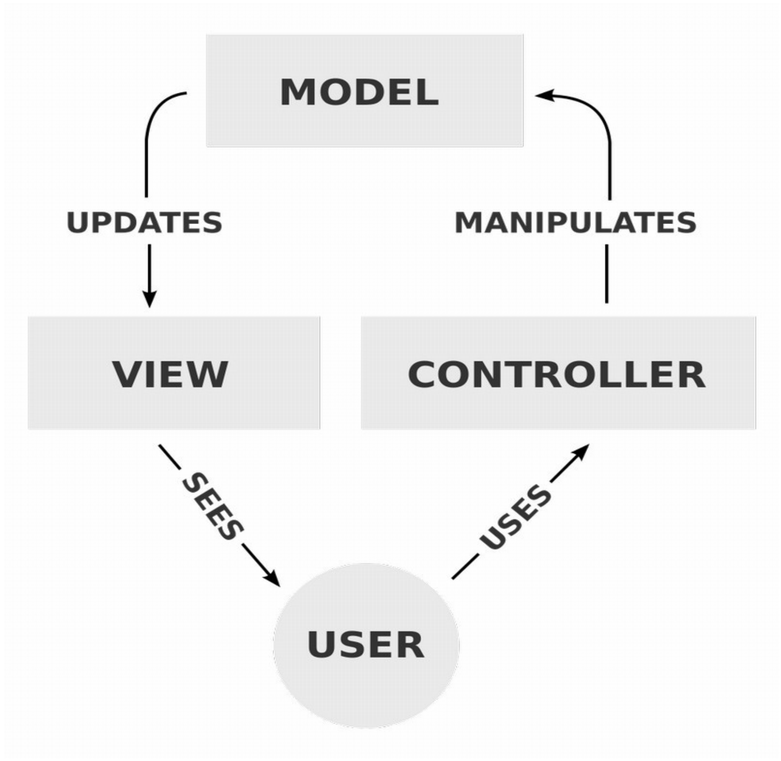
- Event Bus
- Message queues
- Registering to messages via
  - Message types or Topics
  - Subscriber filtering
- the big bad Event System
- Against Coupling (good or bad?)



MVC

– *Model-View-Controller*

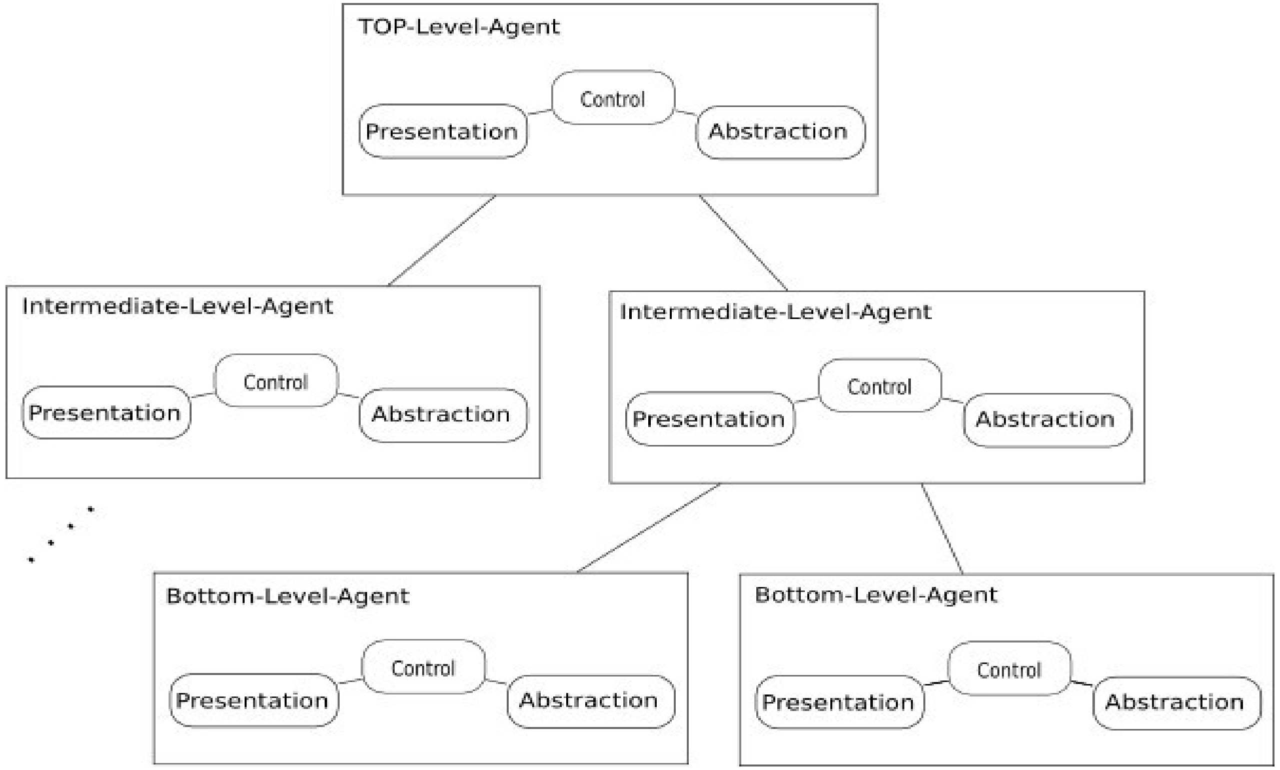
- GUI, Web, Game...



PAC

– *Presentation-Abstraction-Control*

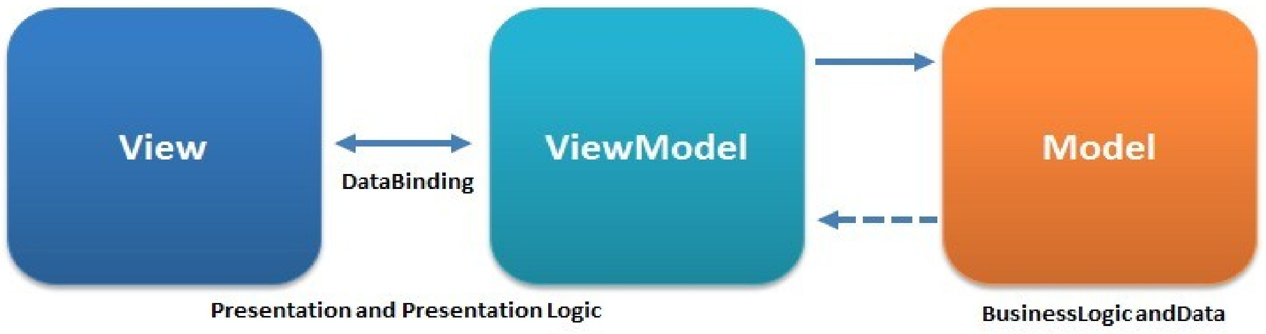
- Hierarchical MVC
- Blocks communicate only through Control objects



MVVM

– *Model-View-ViewModel*

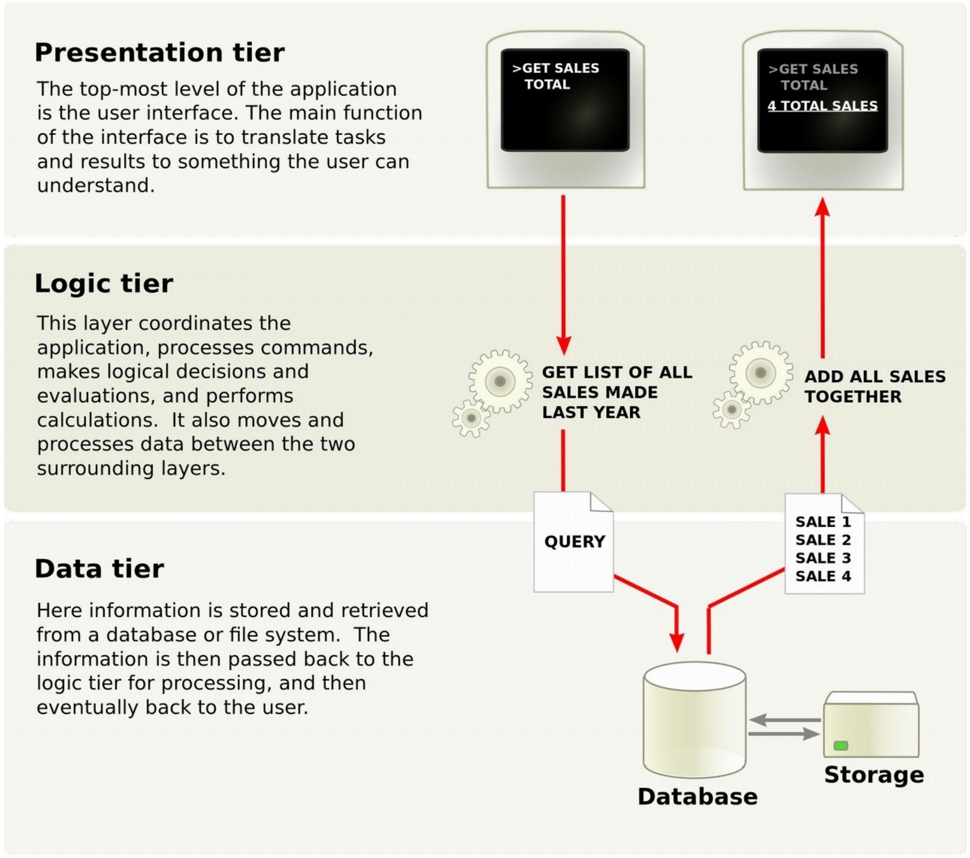
- VM: Data binder or Mediator
- Testing



Multitier

or N-tier

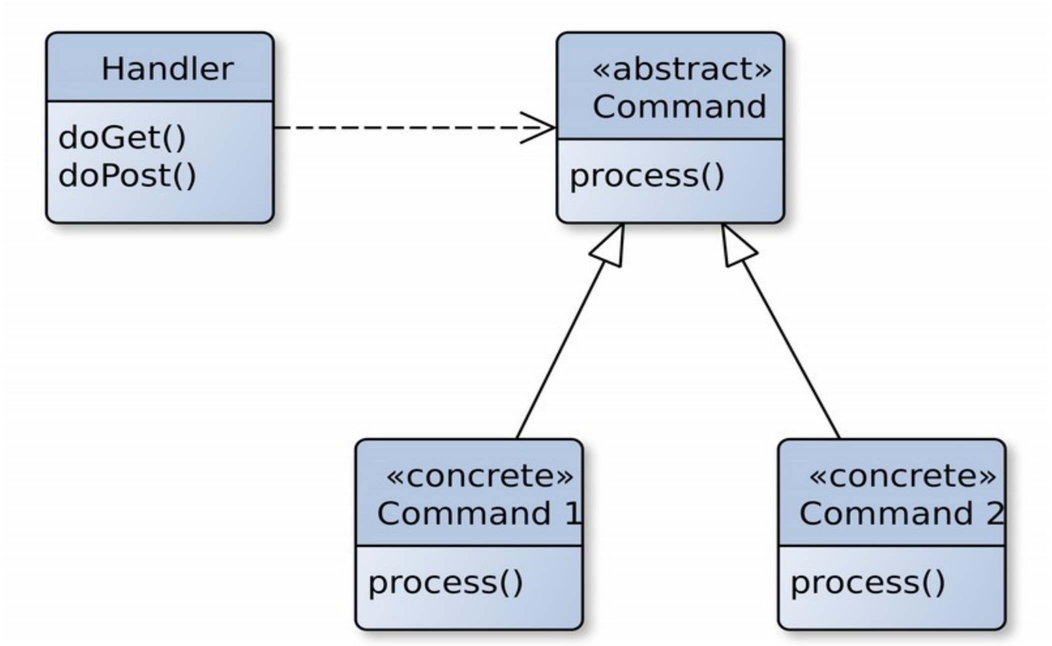
- Reduce complexity by separating system into tiers (layers)
- MVC is 3-tier
- But Web development (DB, app server, web server)



Presentation tier
The top-most level of the application is the user interface. The main function of the interface is to translate tasks and results to something the user can understand.
>GET SALES TOTAL
>GET SALES TOTAL
4 TOTAL SALES
Logic tier
This layer coordinates the application, processes commands, makes logical decisions and evaluations, and performs calculations. It also moves and processes data between the two surrounding layers.
GET LIST OF ALL SALES MADE LAST YEAR
ADD ALL SALES TOGETHER
QUERY
SALE 1
SALE 2
SALE 3
SALE 4
Data tier
Here information is stored and retrieved from a database or file system. The information is then passed back to the logic tier for processing, and then eventually back to the user.
Database
Storage

Front Controller

- Handles all requests to the system
- Serves as only entry-point for requests
- *index.php*



SOA

– *Service Oriented Architecture*

- Autonomous services (application components)
- Communicate through (network) protocol
- Find each other via registry (repository or broker)
- Can be hierarchical (service in service)
- Implementation agnostic

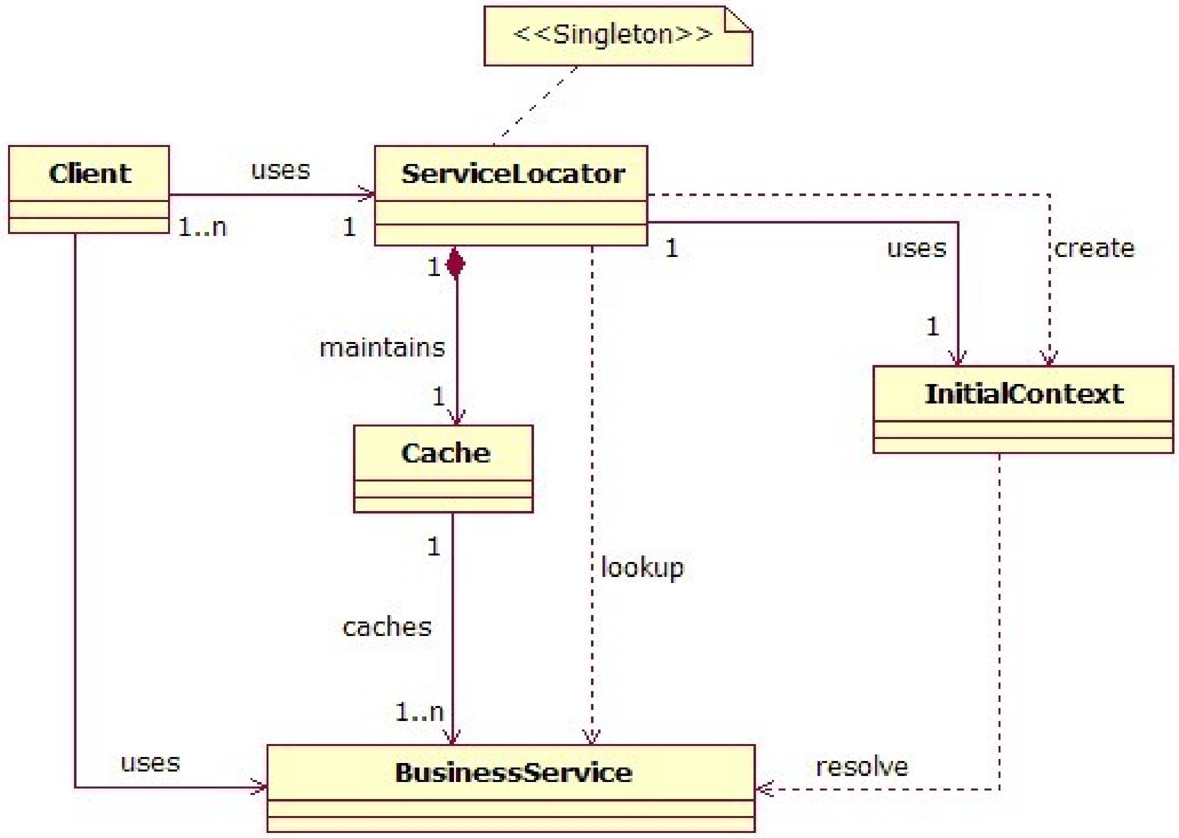
- Flexible, scalable
- Complex to maintain and test

REST

– *Representational State Transfer*

- Stateless operation
- Not a standard, just a pattern
- Access and manipulate text through web
- URIs
- Uniform interface
  - input HTTP GET, POST, PUT, DELETE
  - output JSON, XML, HTML...

Service Locator



Dependency Injection

- Creator of object supplies (injects) all dependencies to the object
- Automate the construction of dependent services
- Separated usage and construction
- Through
  - Setter
  - Constructor
  - Interface

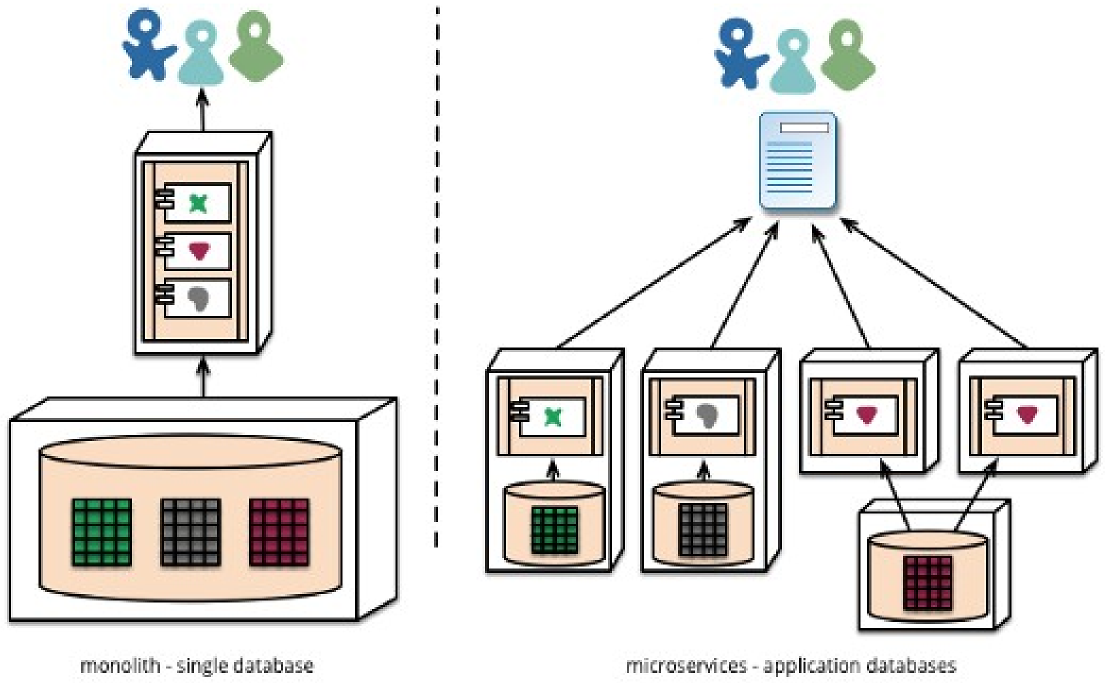
IoC

– *Inversion of Control*

- Normally custom code governs the program, calls general methods, libraries
- But a generic framework can also call specialized functions
- Implementors
  - Service Locator
  - Dependency Injection
  - Strategy
  - Template method
  - Any modern SDK?

Microservices

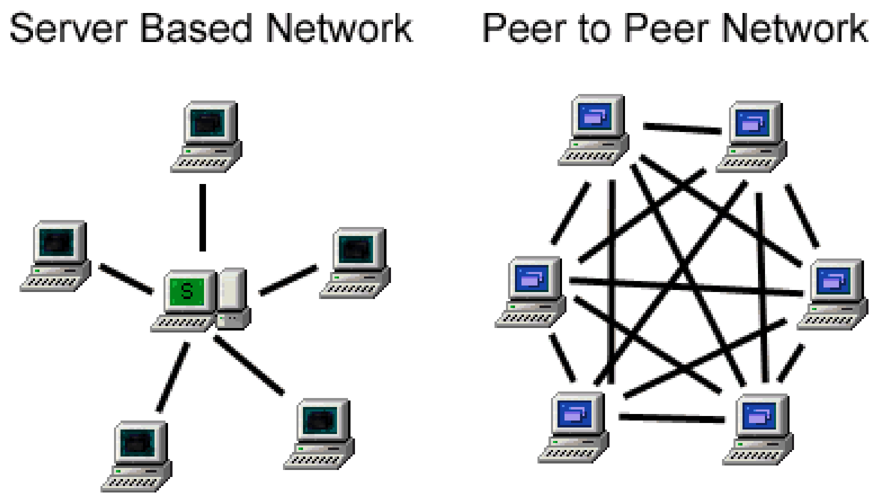
- = SOA + DevOps
- Small services
- Full automation of tests and deployment
- + automate
  - Fault tolerance
  - Scalability


monolith - single database
microservices - application databases

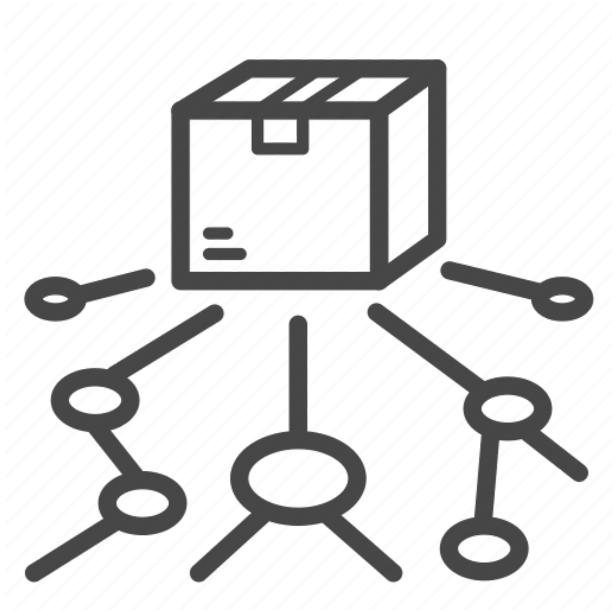
P2P

– *Peer-to-peer*

- No client-server
- Routing and peer discovery
- Distributed
  - Storage
  - Search
  - Processing

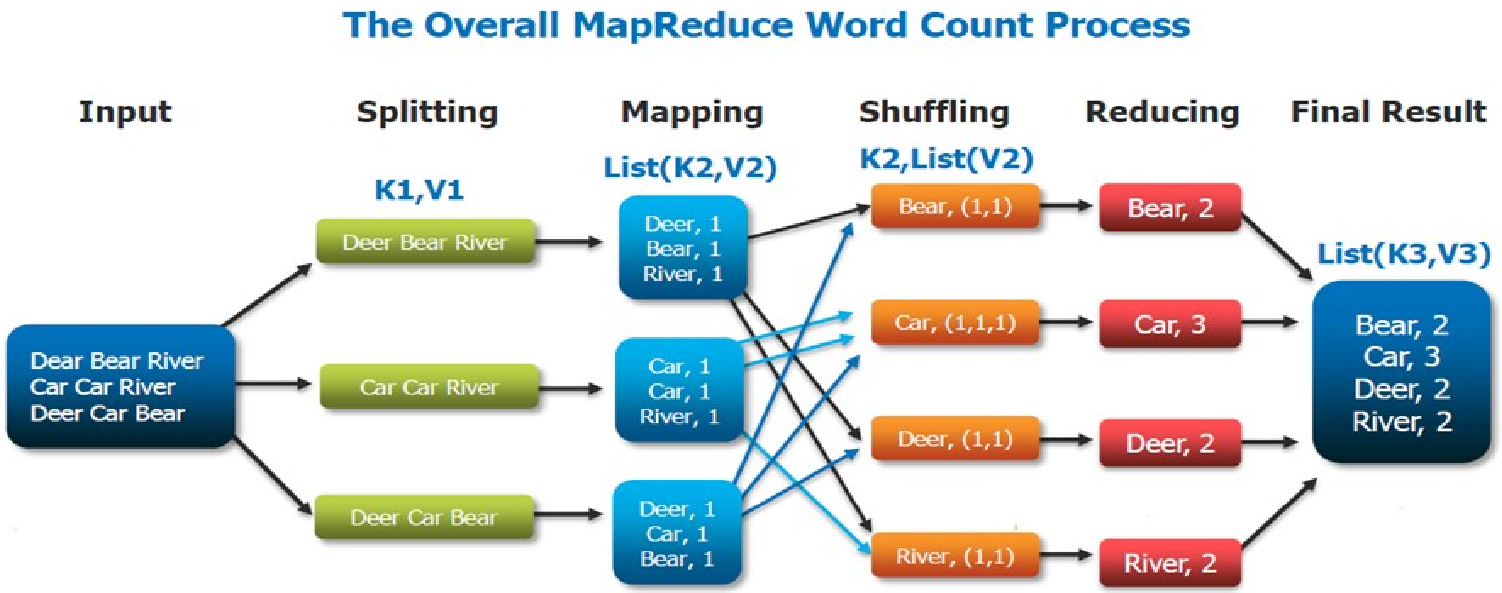


Distributed Design Patterns



MapReduce

- From functional programming
- Steps
  - Map
  - Shuffle
  - Reduce


The Overall MapReduce Word Count Process
Input
Splitting
Mapping
Shuffling
Reducing
Final Result
K1,V1
List(K2,V2)
K2,List(V2)
List(K3,V3)
Dear Bear River
Car Car River
Deer Car Bear
Deer Bear River
Car Car River
Deer Car Bear
Deer, 1
Bear, 1
River, 1
Car, 1
Car, 1
River, 1
Deer, 1
Car, 1
Bear, 1
Bear, (1,1)
Car, (1,1,1)
Deer, (1,1)
River, (1,1)
Bear, 2
Car, 3
Deer, 2
River, 2
Bear, 2
Car, 3
Deer, 2
River, 2