

# C++ Class Diagram

1. library
2. protocol
  - 2-1. service
  - 2-2. master
  - 2-3. slave
3. nam-tree

# Library

string & container

critical section

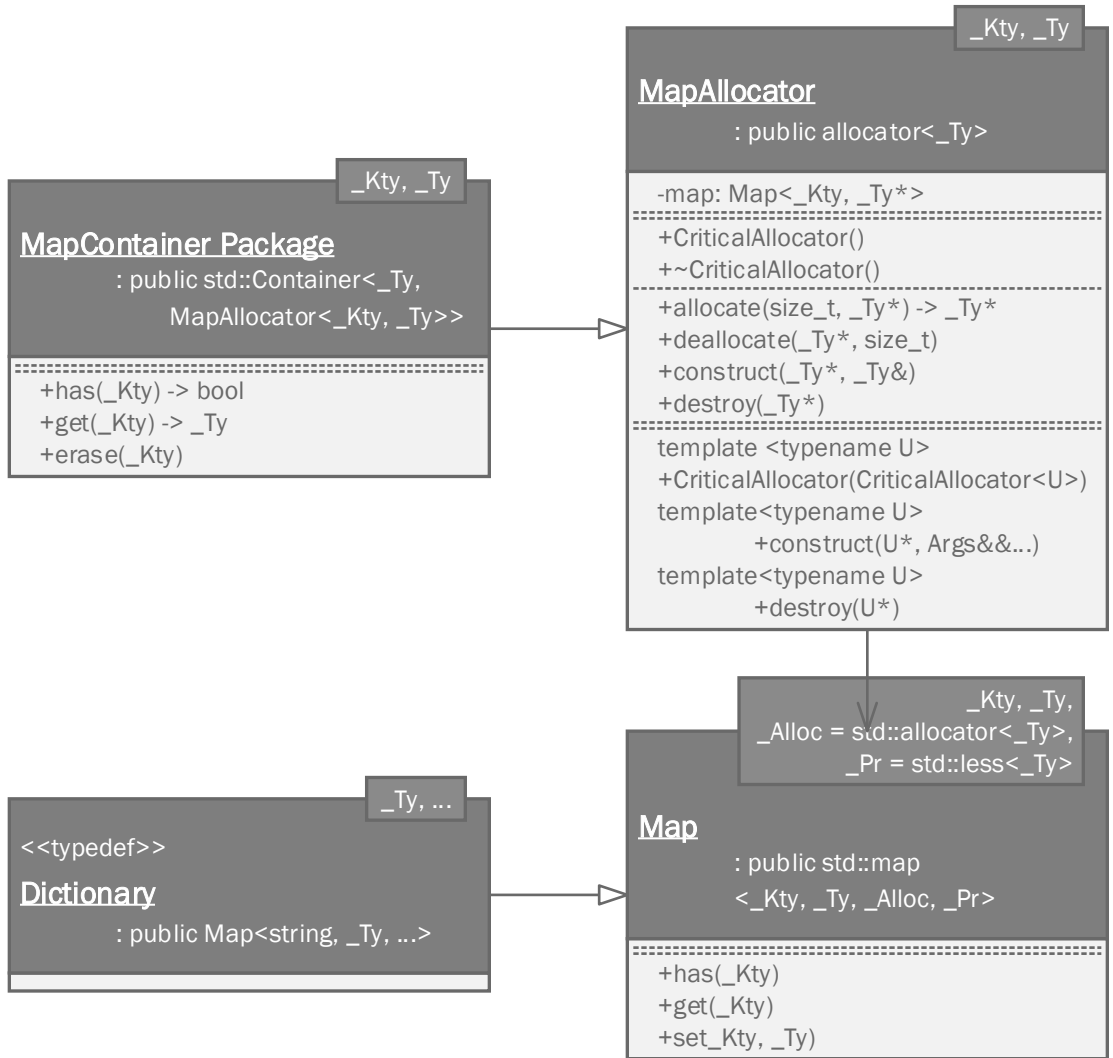
math

xml & sql driver

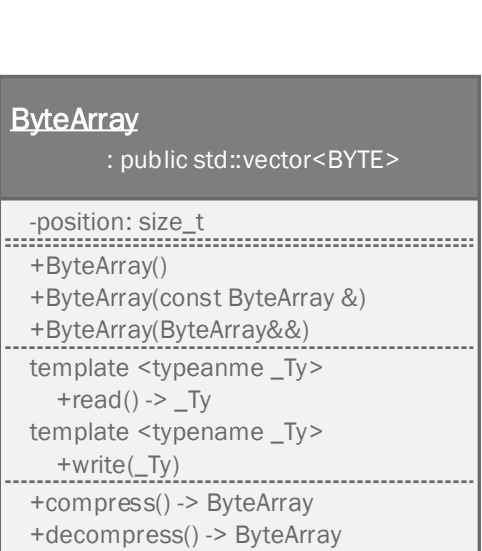
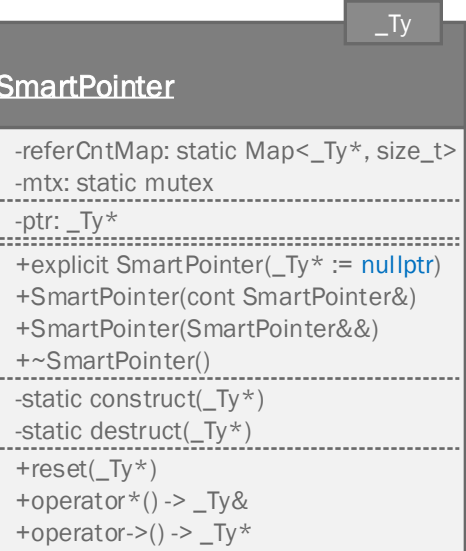
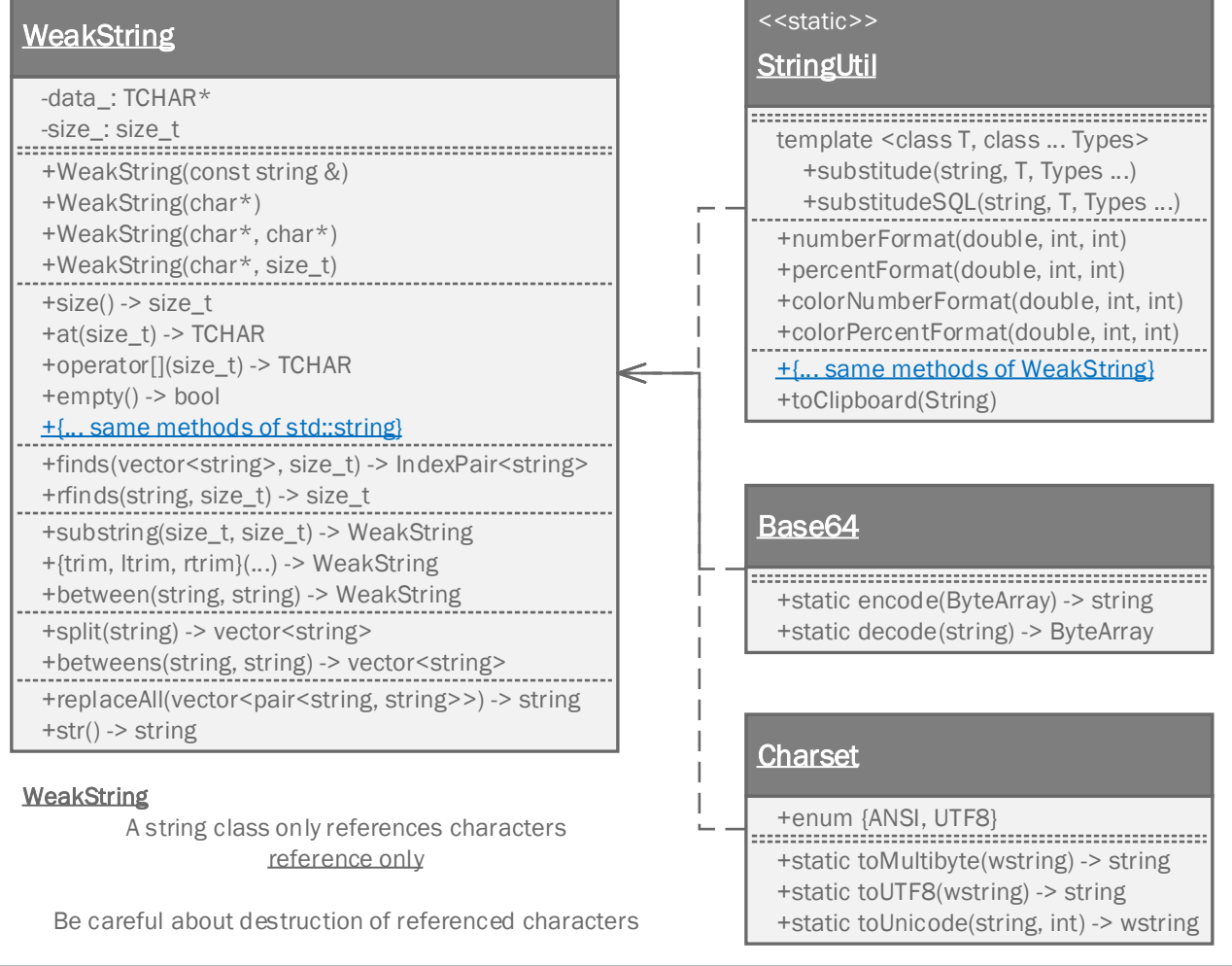
event

File tree

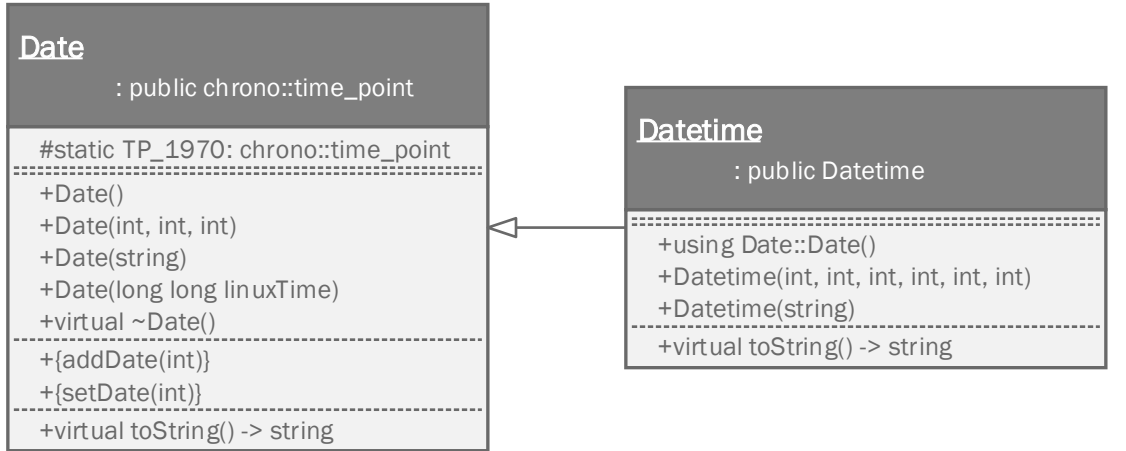
Map Packages



String Utilities



Date and Datetime



## Math Package

```
<<typedef>>
```

Matrix

: public vector<vector<\_Ty>>

Genes, \_Pr := std::less<Genes>

GAPopulation

```
-children: vector<Genes>
-.....
-GAPopulation(size_t)
+GAPopulation(Genes, size_t)
-.....
+fitTest() -> Genes
```

Genes, \_Pr := std::less<Genes>

GeneticAlgorithm

: public EventDispatcher

```
+typedef GAPopulation Pop.
-unique: bool
-mutationRate: double
-tournament: size_t
-.....
+GeneticAlgorithm(bool, double, size_t)
+inline evolveGeneArray(
    Genes,
    population, generation: size_t
) -> GeneArray
+evolvePopulation(Pop.) -> Pop.
-.....
-selection(Pop.) -> Genes
-crossover(Genes, Genes) -> Genes
-mutate(Genes)
```

## CaseGenerator Package

CaseGenerator

```
#dividerArray: vector<size_t>
#size_: size_t
#n_: size_t
#r_: size_t
-.....
+CaseTree(size_t, size_t)
+virtual ~CaseGenerator() = default
+size() -> size_t
+operator[](size_t) -> vector<size_t>
+virtual at(size_t) -> vector<size_t>
-.....
+toMatrix() -> Matrix<size_t>
```

PermutationGenerator

: public CaseTree

```
+PermutationGenerator(size_t, size_t)
+virtual at(size_t) -> vector<size_t>
```

```
<<static>>
```

Math

```
+{reserved static mathematical values}
-.....
+random() -> double
+degree_to_radian(double) -> double
+radian_to_degree(double) -> double
-.....
template <class _Container>
+minimum(_Container) -> IndexPair
+maximum(_Container) -> IndexPair
+median(_Container) -> double
+mode(_Container) -> double
-.....
template <class _Container>
+mean(_Container) -> double
+variance_s(_Container) -> double
+variance_p(_Container) -> double
```

CombinedPermutationGenerator

```
+CombinedPermutationTree
(size_t, size_t)
+virtual at(size_t) -> vector<size_t>
```

FactorialGenerator

: public PermutationTree

```
+FactorialTree(size_t)
```

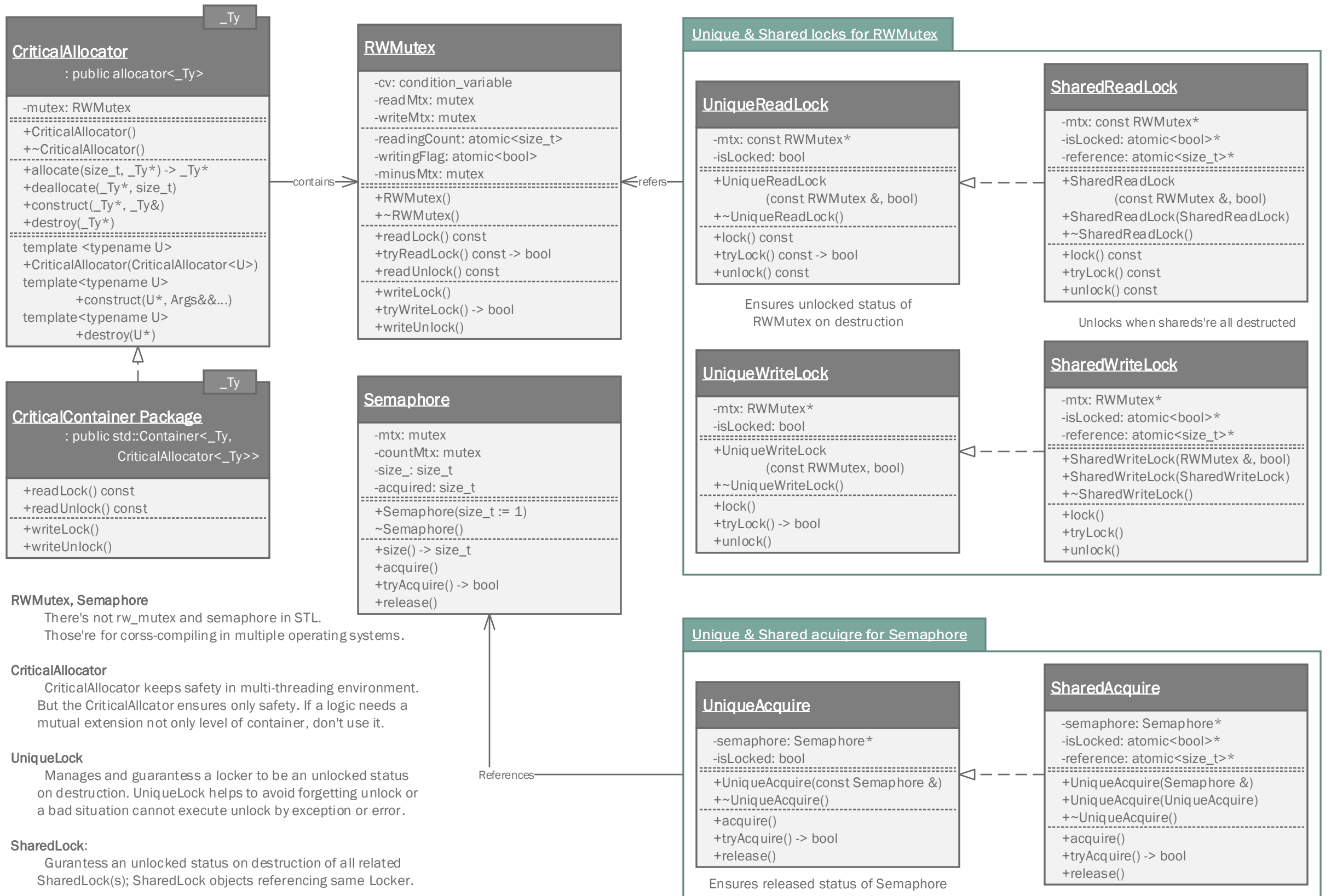
nTTr

nPr

n! = nPn

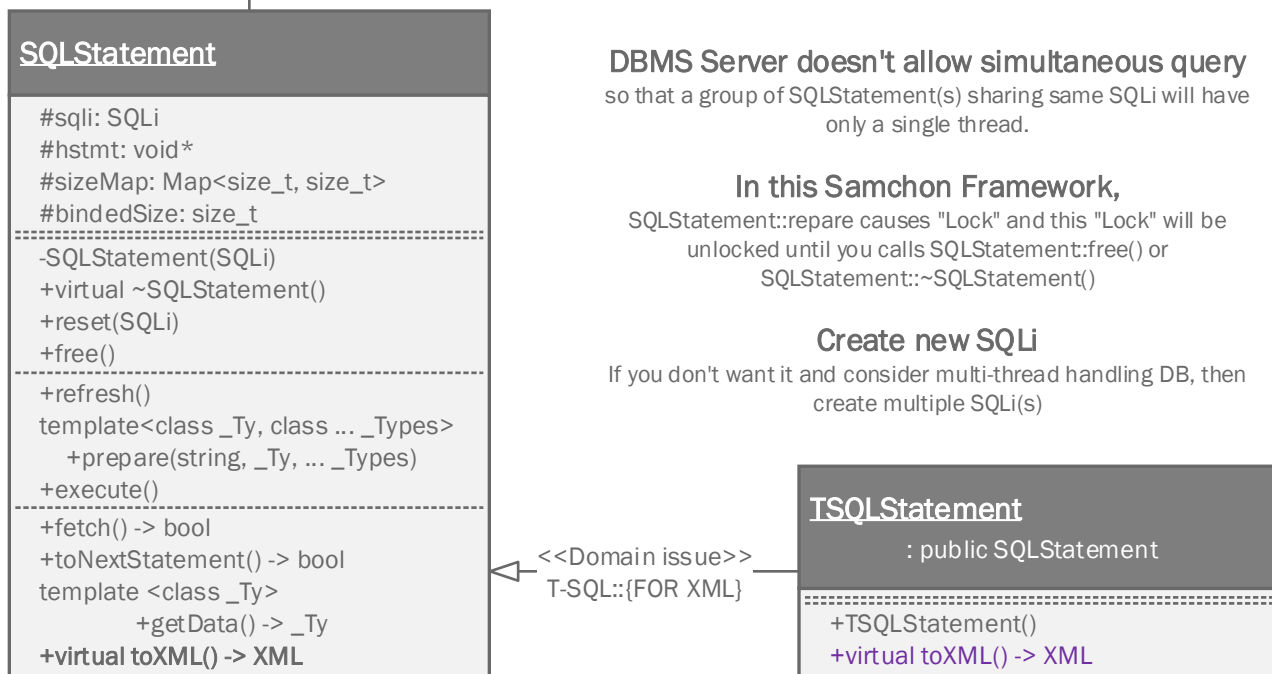
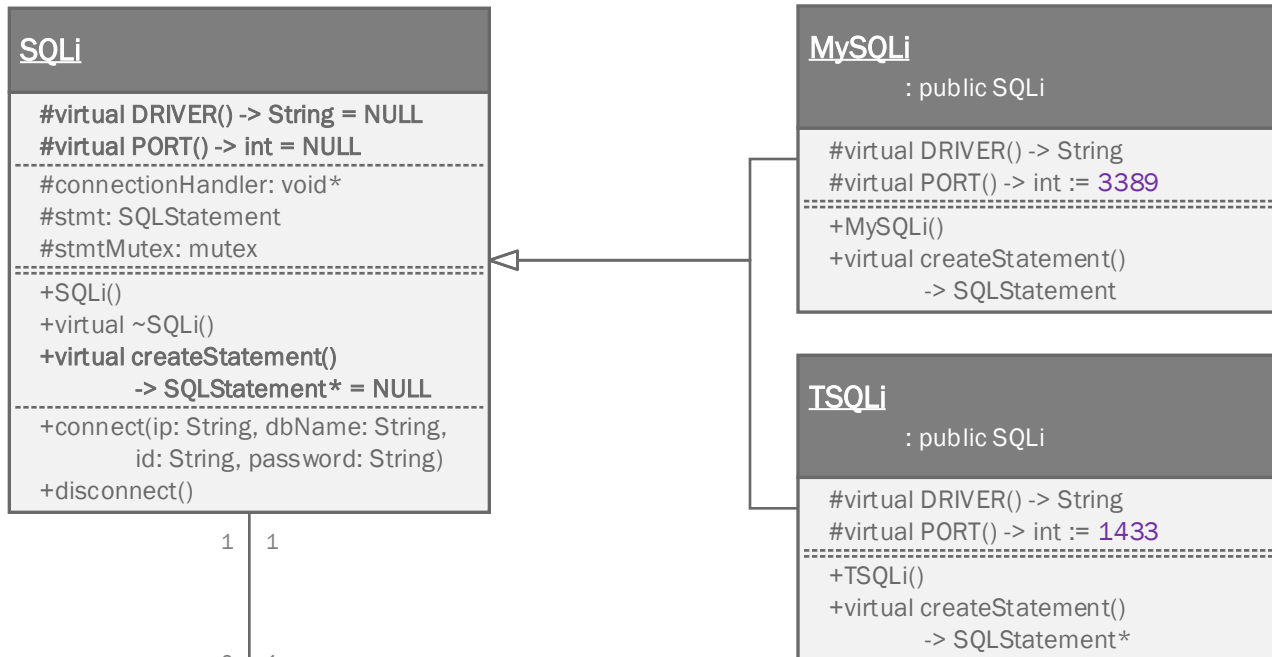
FactorialTree has  
same size of index and leve

## Critical Sections

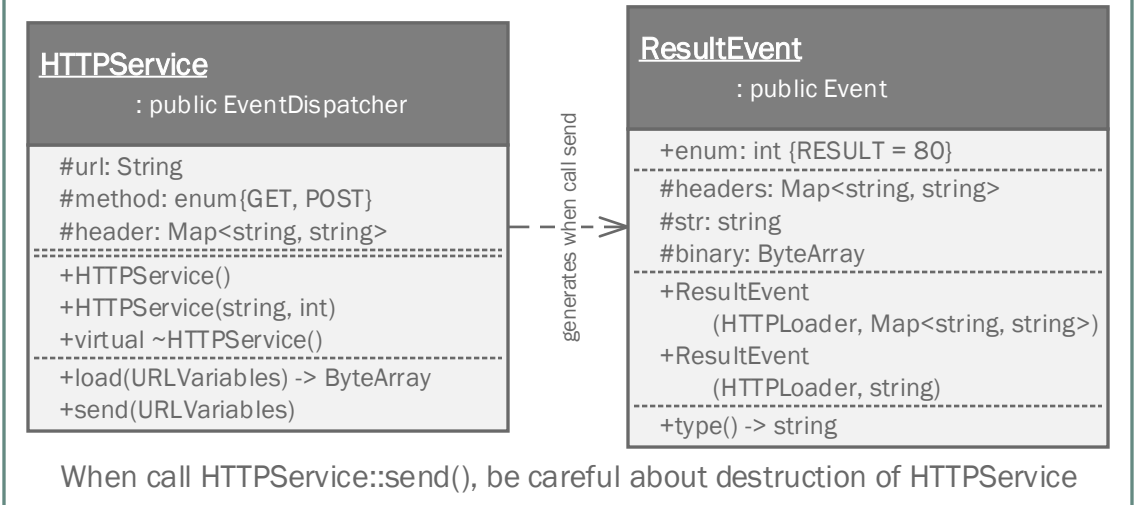


## Protocol Package

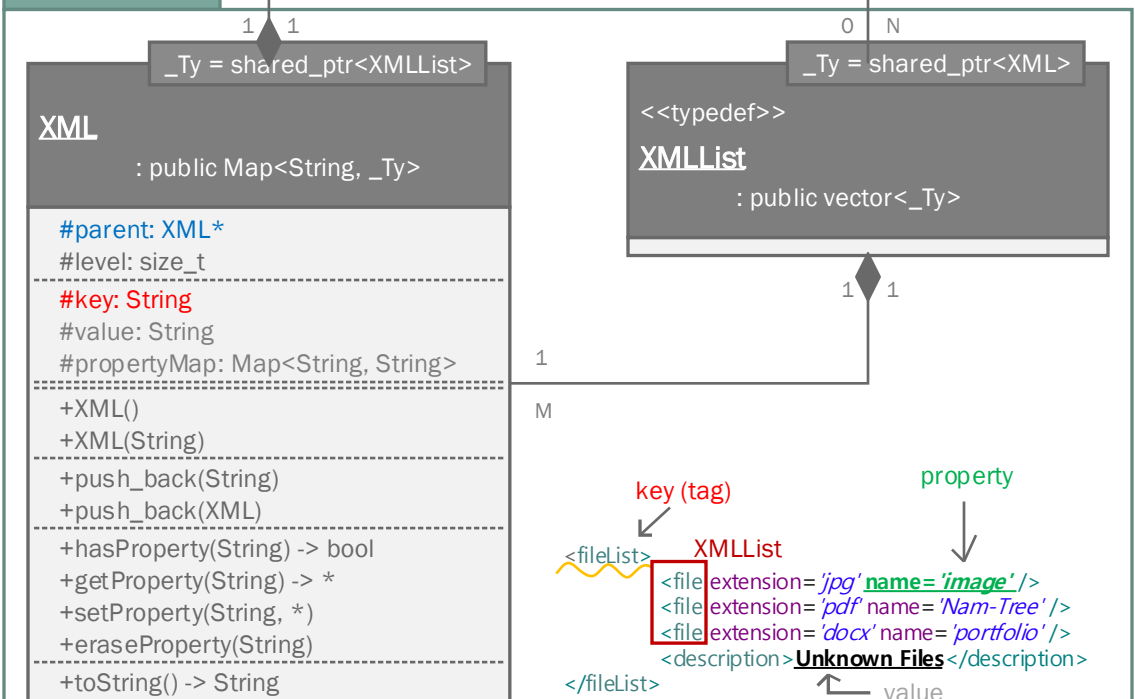
## SQL INTERFACE

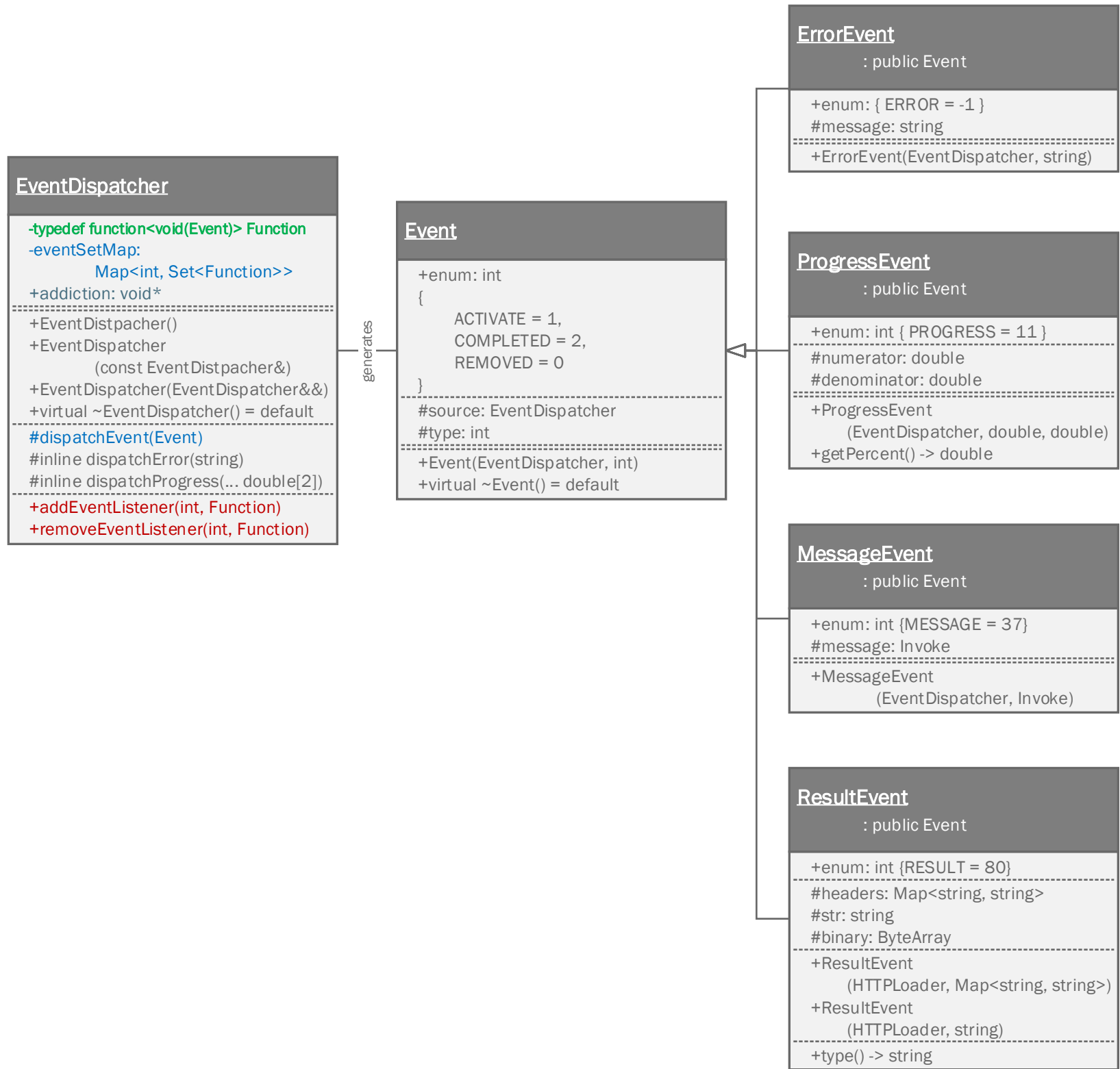


## HTTP Protocol



## XML Package





## EventDispatcher

All the events are sent asynchronously.

To protect from creating enourmous threads by asynchronous event sending, all event sending process will lock the semahore. The default size of the semaphore is 2

Event listener function has to be global or static

I couldn't specify the class to listen, so I programmed all event listener (function pointer) to be static. To send Events to a class's member method, I'm considering to make an interface to listen, "**EventListener**"

### Warning!

Since C++11, calling member method of a class by new thread passing by static method and void pointer is recommended to avoid.

By guidance of the STL, using `std::thread` and `std::bind` will be better. As that reason, Event and EventDispatcher can be depreciated in next generation of Samchon Framework

## Event

A basic class for expressing an event.

Determined Events are "ACTIVE" & "COMPLETE"  
You can add any new event type, if you want.

## ErrorEvent

Cannot throw exception as you called some process asynchronous, you can use this ErrorEvent, insteadly

## ProgressEvent

An event representing a progress.  
It's good for expressing a progress of a process influences to whole system.

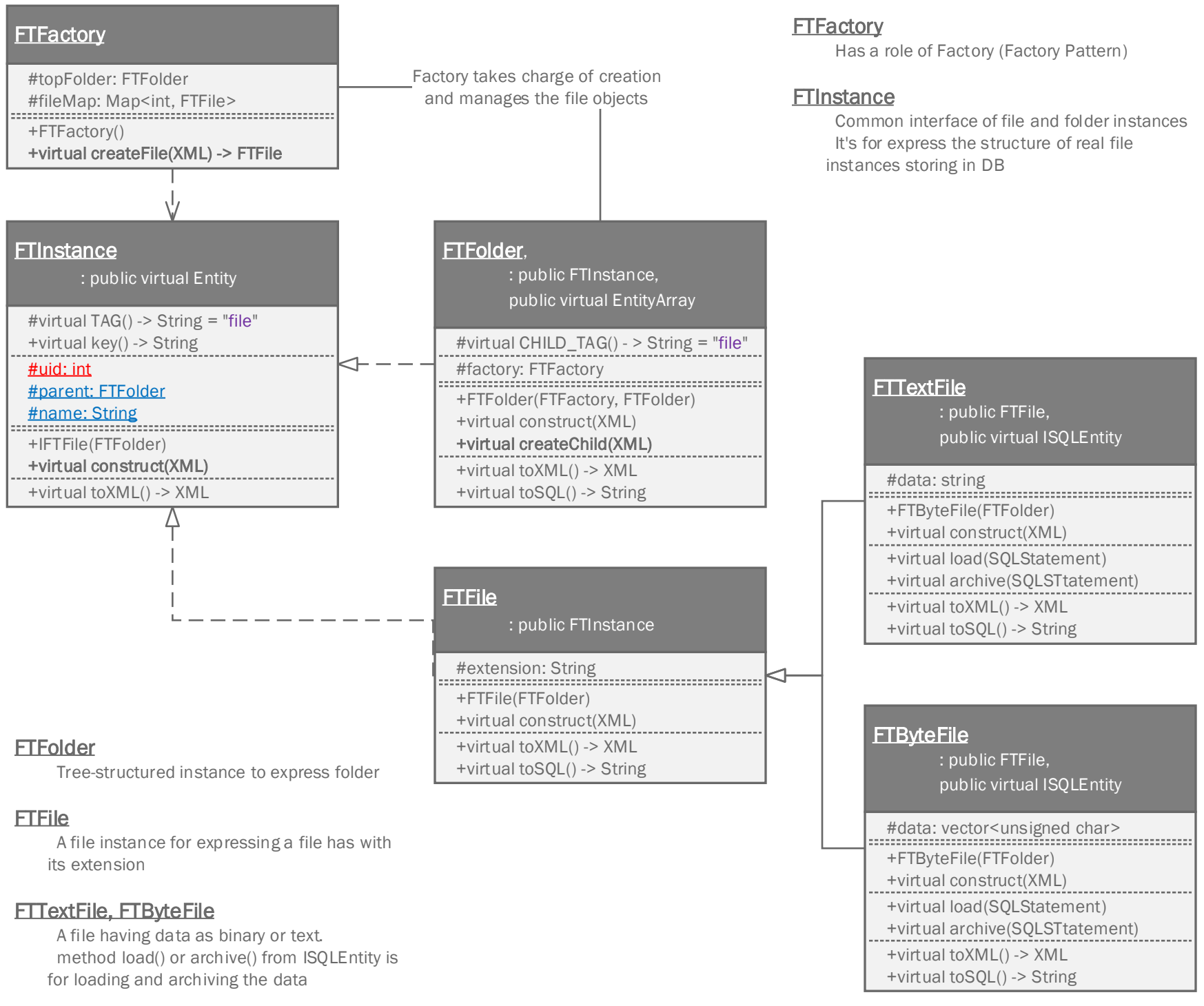
## MessageEvent

—An event containing an Invoke message  
**Depreciated since v1.0.**  
Use **chain of responsibility** pattern with **IProtocol**.

## ResultEvent

An event containing result data from a web-page  
The result type will be one of *string* and *ByteArray*

**ResultEvent::header**: Replied headers from a web-page





# Protocol

invoke

entity

interfaces

external system

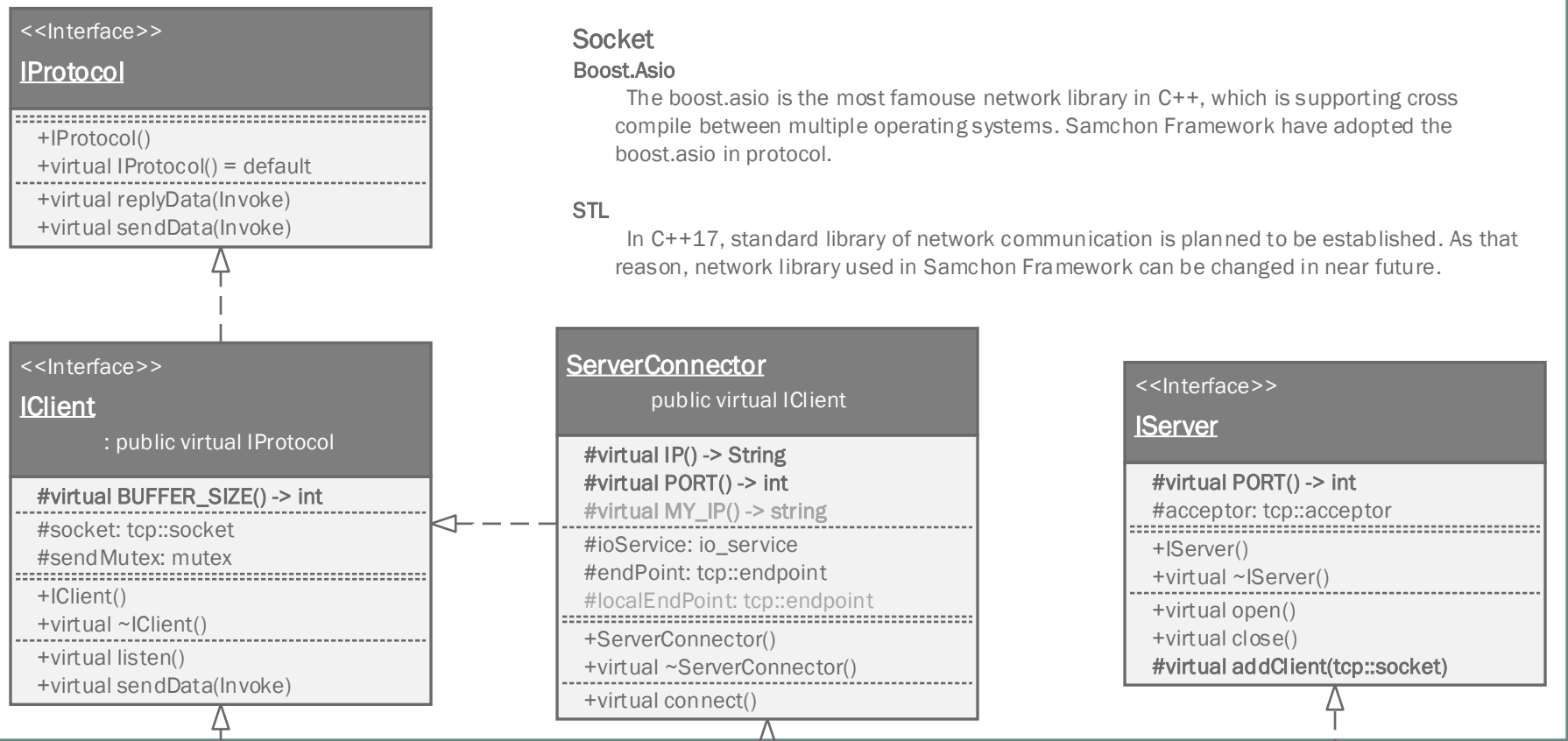
distributed system

parallel system

cloud service

Protocol Interfaces

Basic Interfaces



Creating Network I/O Class

You can construct any type of network system, even how the system is enormously complicated, by just implementing and combining:  
*IProtocol*, *IServer* and *IClient* those are called basic 3 components of Network.

IProtocol

An interface for network I/O  
To realizing chain of responsibility of the network I/O

IServer

An interface for a **physical server**

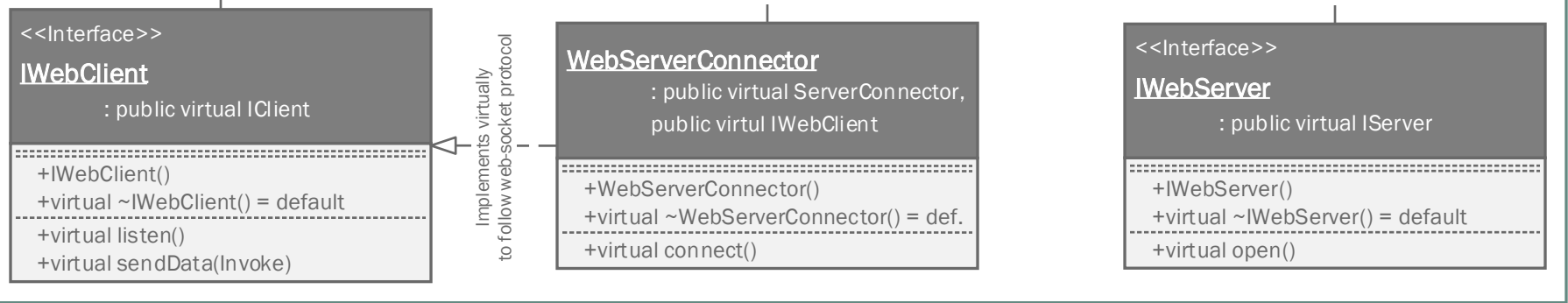
IClient

An interface for a client.  
Not only mean a **physical client**, but also a **driver for a client** in a **physical server**  
IServer::addClient()  
service::Server -> service::User -> service::Client  
ExternalClientArray (A physical server) ->  
ExternalClient (A driver of a client)

ServerConnector

A **server connector** for a **physical client**.  
If you want to connect to a server, then implements this ServerConnector and just override some methods.  
That's all.

Interfaces for Web socket protocol



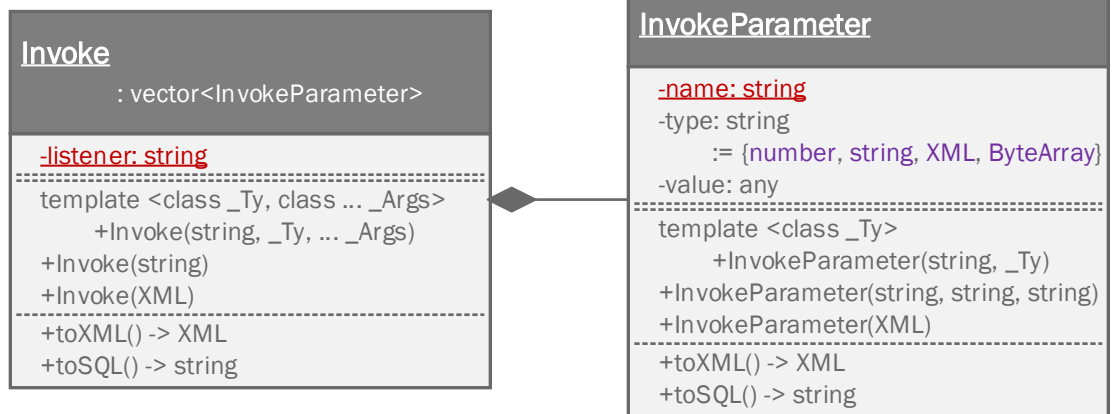
Interfaces for Web-Socket

Interfaces of Web-Socket follow protocol of Invoke and Web socket at the same time by implementing basic interfaces and overriding some methods to follow web-socket protocol.  
You can convert any type of network system to follow web-socket protocol by implementing those interfaces because it's a rule to implement virtually those interfaces.

Derived Network Libraries



Invoke Package



**Invoke:** Express a message (function)  
`Invoke::listener` := almost same with name of a function.

**InvokeParameter:** Parameter in a function.  
When a parameter is not atomic data like a Data-set(structure, list), use XML.

Invoke is

Designed to standardize message structure to be used in network communication. By the [standardization of message protocol](#), user does not need to consider about the network handling. Only concentrate on system's own domain functions are required.

At next page, "Protocol - Interface", you can find "[Basic 3 + 1 Components](#)" required on building some network system; **IProtocol**, **IServer** and **IClient**.

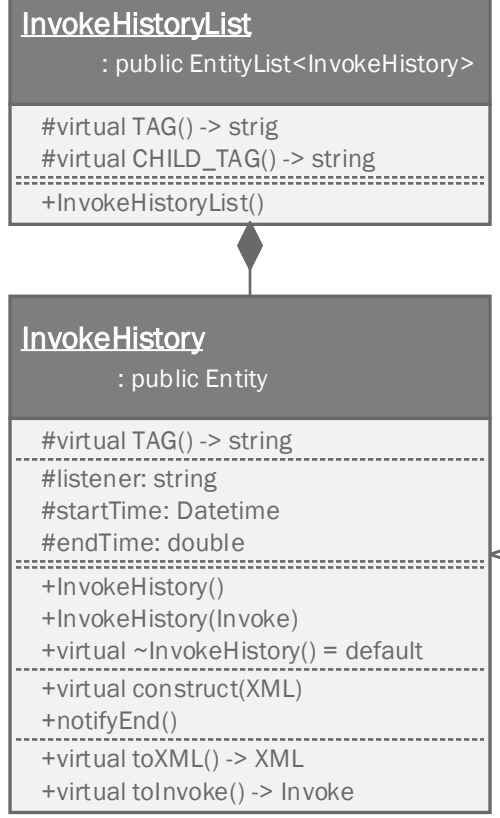
You can construct any type of network system, even how the system is enormously complicated, by just implementing and combining those "[Basic + 1 Components](#)"

Secret of we can build any network system by only those basic components lies in the [standardization of message protocol](#), **Invoke**

Message structure of Invoke

```
<?xml version="1.0" encoding="utf-8" ?>
<invoke listener="login">
  <parameter type="string">jhnam88</parameter>
  <parameter type="string">1234</parameter>
  <parameter type="number">4</parameter>
  <parameter type="XML">
    <memberList>
      <group>3</group>
      <member id="guest" authority="1" />
      <member id="john" authority="3" />
      <member id="samchon" authority="5" />
    </memberList>
  </parameter>
</invoke>
```

Invoke History Package



InvokeHistory is

Designed to report a history log of an Invoke message with elapsed time consumed for handling the Invoke message. The report is directed by a mster from its slaves.  
The reported elapsed time is used to estimating performance of a slave system.

DSInvokeHistory

A reported InvokeHistory in framework of a master of distributed processing system. The master of a distributed processing system estimates performance index of a slave system by those reports.  
Master distributes roles to slave systems optimally from the estimated performance index which is calculated from those reports.

PRInvokeHistory

A reported InvokeHistory in framework of a master of parallel processing system. The master of a parallel processing system estimates performance index of a slave system by those reports.  
Master distributes quantity of handing process of slave systems from the estimated performance index which is calculated from those reports.

History for Parallel P. system



Histories for Distributed P. system



Entity is

To standardize expression method of data structure.  
Provides I/O interfaces to/from XML and Invoke.  
When you need some additional function for the Entity,  
use the chain responsibility pattern with IEntityChain.

When data-set has a "Hierarchical Relationship"

Compose the data class(entity) having children by  
inheriting EntityGroup and terminate the leaf node by  
inheriting Entity.  
Just define the XML I/O only for each variables, then  
about the data I/O, all will be done

Utility interfaces

```
<<Interface>>
ISQLException
-----
+virtual load(SQLStatement)
+virtual archive(SQLStatement)
+virtual toSQL() -> String
```

```
<<Interface>>
IHTMLEntity
-----
#CSS: static string
#HEADER: static string
-----
template <class _Ty, class ... _Args>
    #toTR(_Ty, ... _Args) -> string
template <class _Ty>
    #toTH(_Ty) -> string
template <class _Ty>
    #toTD(_Ty) -> string
+virtual toHTML() -> string
```

```
<<Interface>>
IEntityGroup
-----
#virtual CHILD_TAG() -> string
```

```

        _Container,
        _Ty := _Container::value_type

EntityGroup
: public _Container
public virtual Entity,
public virtual IEntityGroup
-----
+EntityGroup()
+virtual ~EntityGroup() = default
+virtual construct(XML)
#virtual createChild(XML) -> Entity
+has(String) -> bool
+get(String) -> _Ty
+virtual toXML() -> XML
```

```

        _Ty := extends Entity

EntityArray
: public vector<_Ty>,
public virtual IEntityGroup,
public virtual Entity
-----
+EntityArray()
+virtual ~EntityArray() = default
+virtual construct(XML)
+has(string) -> bool
+get(string) -> _Ty
+virtual toXML() -> XML
```

```
Entity
-----
#virtual TAG() -> string
+Entity()
+virtual ~Entity() = default
+virtual construct(XML)
+virtual key() -> string
+virtual toXML() -> XML
```

```
<<Interface>>
IEntityChain
-----
#entity: Entity
-----
+IEntityChain(Entity)
+virtual ~IEntityChain() = default
```

Pre-compiled EntityGroup

```
EntityArray -> Static entity array
EntityList -> EntityGroup<std::list<Entity>>>

SharedEntityArray ->
    EntityGroup<std::vector<std::shared_ptr<Entity>>>>
SharedEntityList ->
    EntityGroup<std::list<std::shared_ptr<Entity>>>>
```

Inherits to share same interface

composite pattern  
(enable to realize 1:N recursive relationship)

In my framework, Entity is the main character,  
so that concentrates on to the Entity 1st

implements

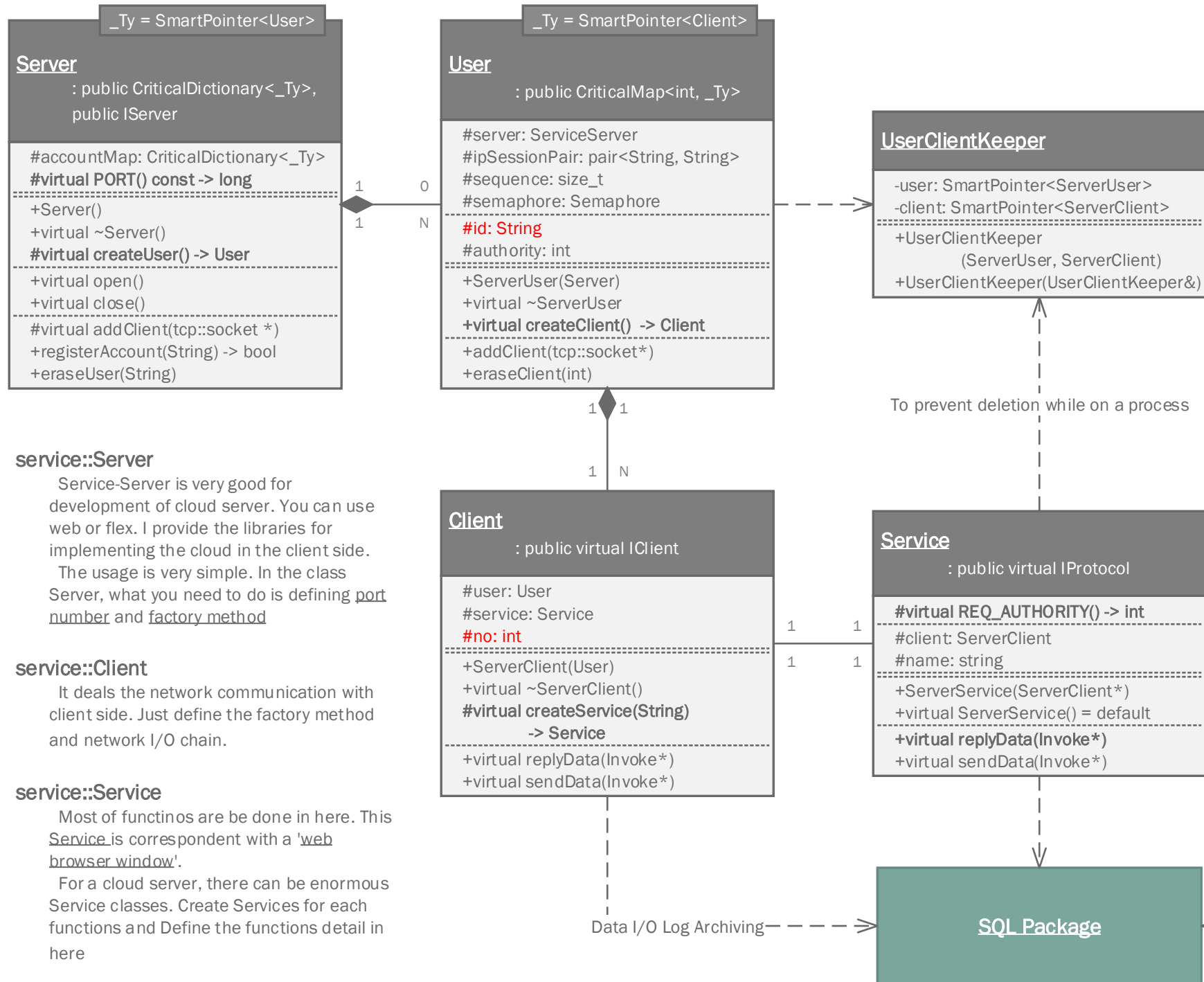
0 1  
composite

inherits

Role of the "Chain Responsibility"

0 1

## SERVICE SERVER PACKAGE



### service::User

ServerUser does not have any network I/O and its own special work something to do. It's a container for grouping clients by their ip and session id.

Thus, the service::User corresponds with a User (Computer) and service::Client corresponds with a Client(A browser window)

### service::UserClientKeeper

You can prevent the object to be deleted until the method is in a process. ServerUser, ServerClient and ServerService provides a macro instruction for it.

ServerUser: KEEP\_USER\_ALIVE  
ServerClient: KEEP\_CLIENT\_ALIVE  
ServerService: KEEP\_SERVICE\_ALIVE

### service::Server

Service-Server is very good for development of cloud server. You can use web or flex. I provide the libraries for implementing the cloud in the client side.

The usage is very simple. In the class Server, what you need to do is defining port number and factory method

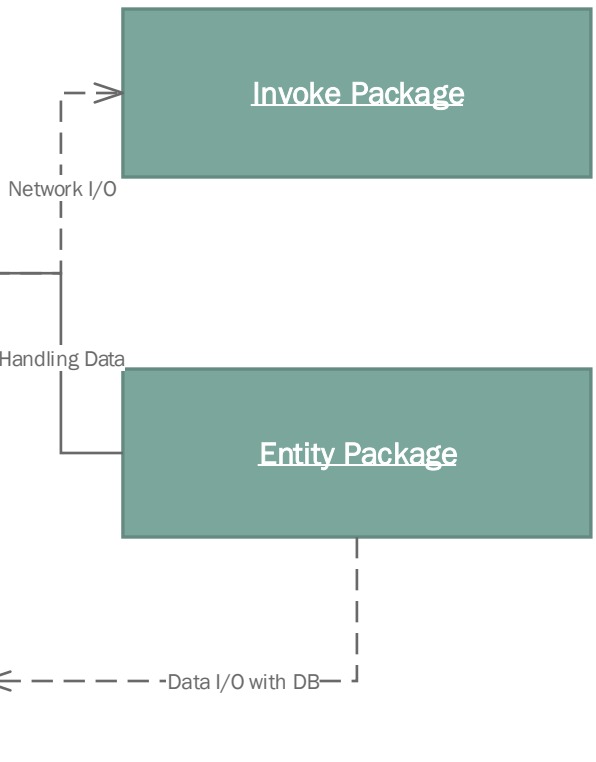
### service::Client

It deals the network communication with client side. Just define the factory method and network I/O chain.

### service::Service

Most of funtinos are be done in here. This Service is correspondent with a 'web browser window'.

For a cloud server, there can be enormous Service classes. Create Services for each functions and Define the functions detail in here





## Deriveds

### Distributed Processing System

This Package's System and Role have those own index for performance.  
Master distributes Role(s) to Slavess) optimally

Also, Master and Slave can be delegated by composing tree-structure

### Parallel Processing System

Parallel System does not have Role.  
ParallelSystemMaster distributes works uniformly

## External System Network Chain

### ExternalSystemClientSocket

: public ExternalSystemSocket,  
public virtual IClient

+ExternalSystemClientSocket  
(ExternalSystem, tcp::socket)  
+virtual start()

### ExternalSystemServerSocket

: public ExternalSystemSocket,  
public virtual IServerConnector

+ExtSysServerSocket(ExternalSystem)  
+virtual start()

### ExternalSystemSocket

: public virtual IClient

#system: ExternalSystem  
+ExtSysSocket(ExternalSystem)  
+virtual ~ExtSysSocket = default  
+virtual start()  
+virtual replyData(Invoke)

## Basic System

### ExternalSystemArray

: public virtual EntityArray,  
public virtual IProtocol,  
private IServer

#virtual TAG() -> String := "systemArray"  
#virtual CHILD\_TAG() -> String  
+virtual DIRECTION()  
-> enum {SERVER, CLIENT}  
+virtual BUFFER\_SIZE() -> int  
-----  
#parent: IProtocol  
#roleMap: Map<String, ExtSysRole>  
-myIP: String  
-port: int  
-----  
+ExternalSystemArray(IProtocol)  
+virtual construct(XML)  
#virtual createChild(XML) -> Entity  
+virtual start()  
-----  
+hasRole(String) -> bool  
+getRole(String) -> ExtSysRole  
-----  
+virtual sendData(Invoke)  
+virtual replyData(Invoke)

### ExternalSystemRole

ExternalSystemMaster and ExternalSystem expresses the physical relationship between your system(master) and the external system.

But ExternalSystemRole enables to have a new, logical relationship between your system and external servers.

You just only need to concentrate on the role what external systems have to do.

Just register and manage the Role of each external system and you just access and orders to the external system by their role

### ExternalSystemArray

This class set will be very useful for constructing parallel distributed processing system.

Register distributed servers on ExternalSystemArray and manage their roles, and then communicate based on role.

### Access by Role

```
ExternalSystemMaster *master;  
ExternalSystem *system =  
    master->getRole(String)->getSystem();  
server.sendData(Invoke)
```

### ExternalSystem

: public virtual EntityArray,  
public virtual IProtocol

#virtual TAG() -> String := "system"  
#virtual CHILD\_TAG() -> String  
#master: ExternalSystemArray  
-socket: ExternalSystemSocket  
-ip: String  
-myIP: String  
-port: int  
-----  
+ExternalSystem(ExternalSystemArray)  
+virtual construct(XML)  
#virtual createChild(XML) -> Entity  
+virtual start()  
-----  
+virtual sendData(Invoke)  
+virtual replyData(Invoke)

<<Interface>>

### ExternalSystemRole

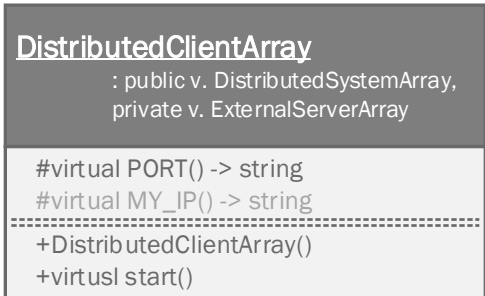
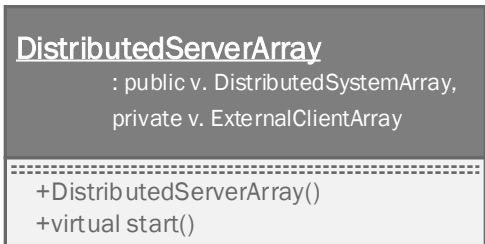
: public virtual Entity

#virtual TAG() -> String := "role"  
#virtual key() -> String  
#system: ExternalSystem  
-----  
+ExternalSystemRole(Slave)  
+virtual ~ExternalServerRole()

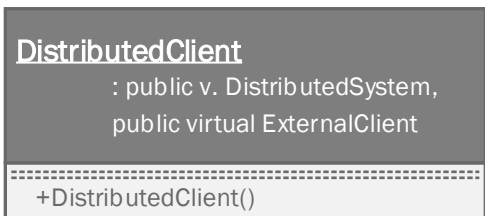
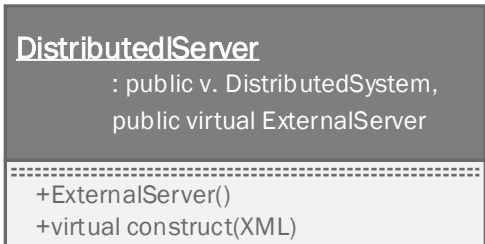


Derived classes

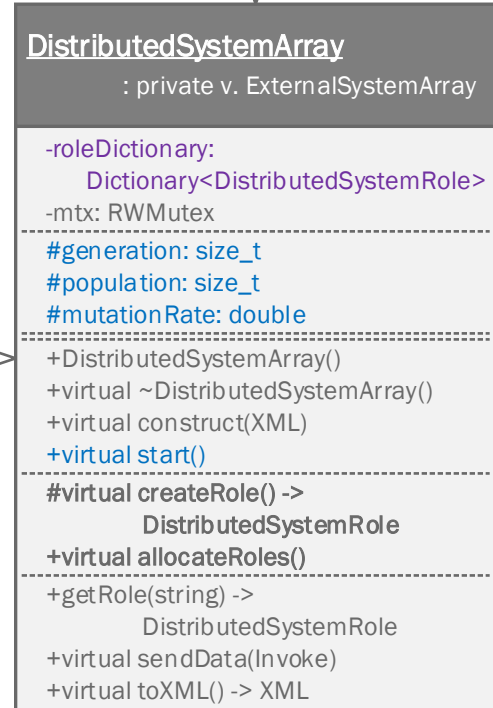
Masters



Slaves



Base classes



Different aspect with ExternalSystem

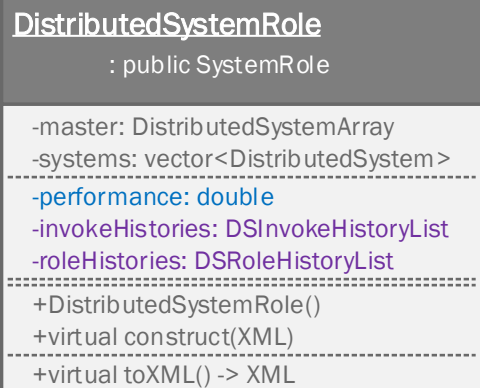
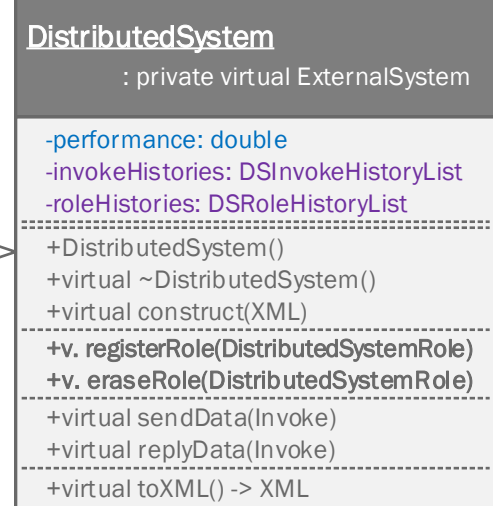
Relationship between Role is different.  
The Role is not belonged only to a System and  
the Role is not even created by SystemArray.

Allocating Roles to Systems

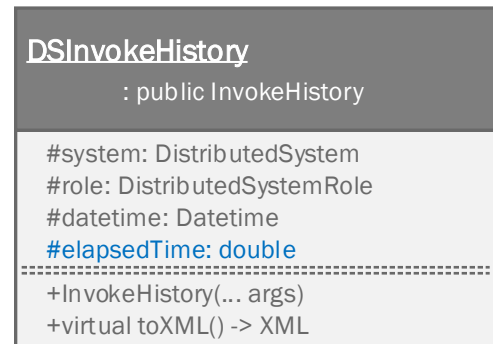
History of message transmissions and those  
elapsed times are archived. SystemArray  
calculates performance index of Systems and  
Roles.  
Roles will be allocated or re-allocated System  
from those performance indices by genetic  
algorithm. If number of Systems and Roles are not  
too much, then combined permutation case  
generator will be used instead of the genetic  
algorithm.

Mediator

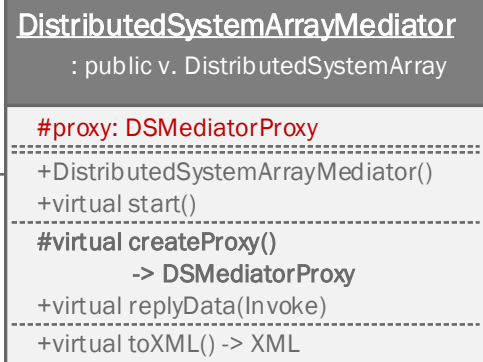
You even can compose tree-structured  
distributed processing system with  
DistributedSystemArrayMediator



Histories

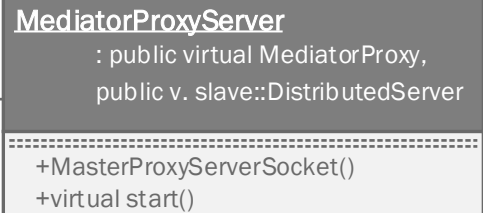
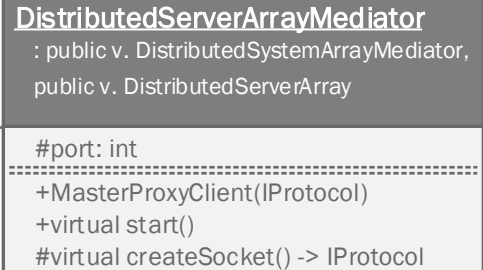
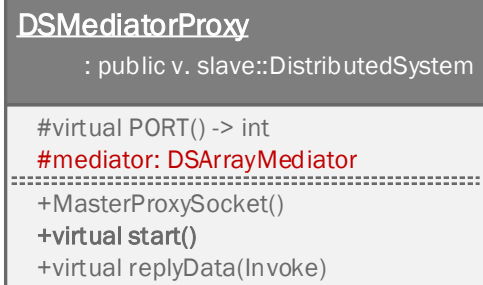


Mediator



<<Mediator to real master>>

DistributedSystemRole::replyData()  
-->>  
DistributedSystemArrayMediator::replyData()  
-->>  
DSMediatorProxy::sendData()



DistributedSystemArrayMediator is a DistributedSystem  
in view of the real-DistributedSystemArray



Parallel Server Package

**ParallelServerArray**  
: public v. ParallelSystemArray

---

+ParallelServerArray()  
+virtual ~ParallelServerArray() = def.  
+virtual start()

**ParallelServer**  
: public virtual ParallelSystem,  
public virtual ServerConnector

---

#virtual IP() -> string  
#virtual PORT() -> int  
#virtual MY\_IP() -> string

---

+ParallelServer()  
+virtual ~ParallelServer() = default

Parallel Client Package

**ParallelClientArray**  
: public v. ParallelSystemArray,  
public virtual IServer

---

#virtual PORT() -> int  
#virtual MY\_IP() -> string

---

+ParallelClientArray()  
+virtual ~ParallelClientArray() = def.  
+virtual start()

**ParallelClient**  
: public virtual ParallelSystem,  
public virtual IClient

---

+ParallelClient()  
+virtual ~ParallelClient() = def.

Parallel System Package

**ParallelSystemArray**  
: public SharedEntityArray,  
public virtual IProtocol

---

#virtual TAG() -> string := "systemArray"  
#virtual CHILD\_TAG -> string := "system"  
#myIP: string

---

+ParallelSystemArray()  
+virtual ~ParallelSystemArray() = def.  
+virtual start()  
+virtual replyData(Invoke)  
+virtual sendData(Invoke)  
+sendData(Invoke, size\_t, size\_t)  
+virtual toXML() -> XML

**ParallelSystem**  
: public Entity,  
public virtual IProtocol

---

#virtual TAG() -> string = "system"  
#parent: ParallelSystemArray  
#ip: string  
#port: int  
#performance: double

---

+ParallelSystem()  
+virtual ~ParallelSystem() = default  
+virtual construct(XML)  
+virtual replyData(Invoke)  
+sendData(Invoke, size\_t, size\_t)  
+virtual toXML() -> XML

Mediator

**ParallelSystemArrayMediator**  
: public v. ParallelSystemArray

---

#proxy: PRMediatorProxy

---

+DistributedSystemArrayMediator()  
+virtual start()  
#virtual createProxy()  
-> PRMediatorProxy  
+virtual replyData(Invoke)  
+virtual toXML() -> XML

<<Mediator to real master>>

ParallelSystem::replyData()  
-->>  
ParallelSystemArrayMediator::replyData()  
-->>  
PRMediatorProxy::sendData()

**PRMediatorProxy**  
: public virtual IProtocol

---

#virtual PORT() -> int  
#mediator:  
DistributedSystemArrayMediator

---

+MasterProxySocket(MasterProxy)  
+virtual start()  
+virtual replyData(Invoke)

**DistributedServerArrayMediator**  
: public v. DistributedSystemArrayMediator,  
public v. DistributedServerArray

---

#port: int

---

+MasterProxyClient(IProtocol)  
+virtual start()  
#virtual createSocket() -> IProtocol

**DistributedClientArrayMediator**  
: public v. MasterProxy,  
public v. MasterServer

---

#masterIP: String

---

+MasterProxyServer(IProtocol)  
+virtual construct(XML)  
+virtual start()  
#virtual createSocket() -> IProtocol  
+virtual toXML() -> XML

**DSMediatorProxyServer**  
: public v. MasterProxySocket,  
public v. OneToOneServer

---

+MasterProxyServerSocket  
(MasterProxy)  
+virtual start()

**DSMediatorProxyClient**  
: public v. MasterProxySocket  
public v. ServerConnector

---

#virtual IP() -> String

---

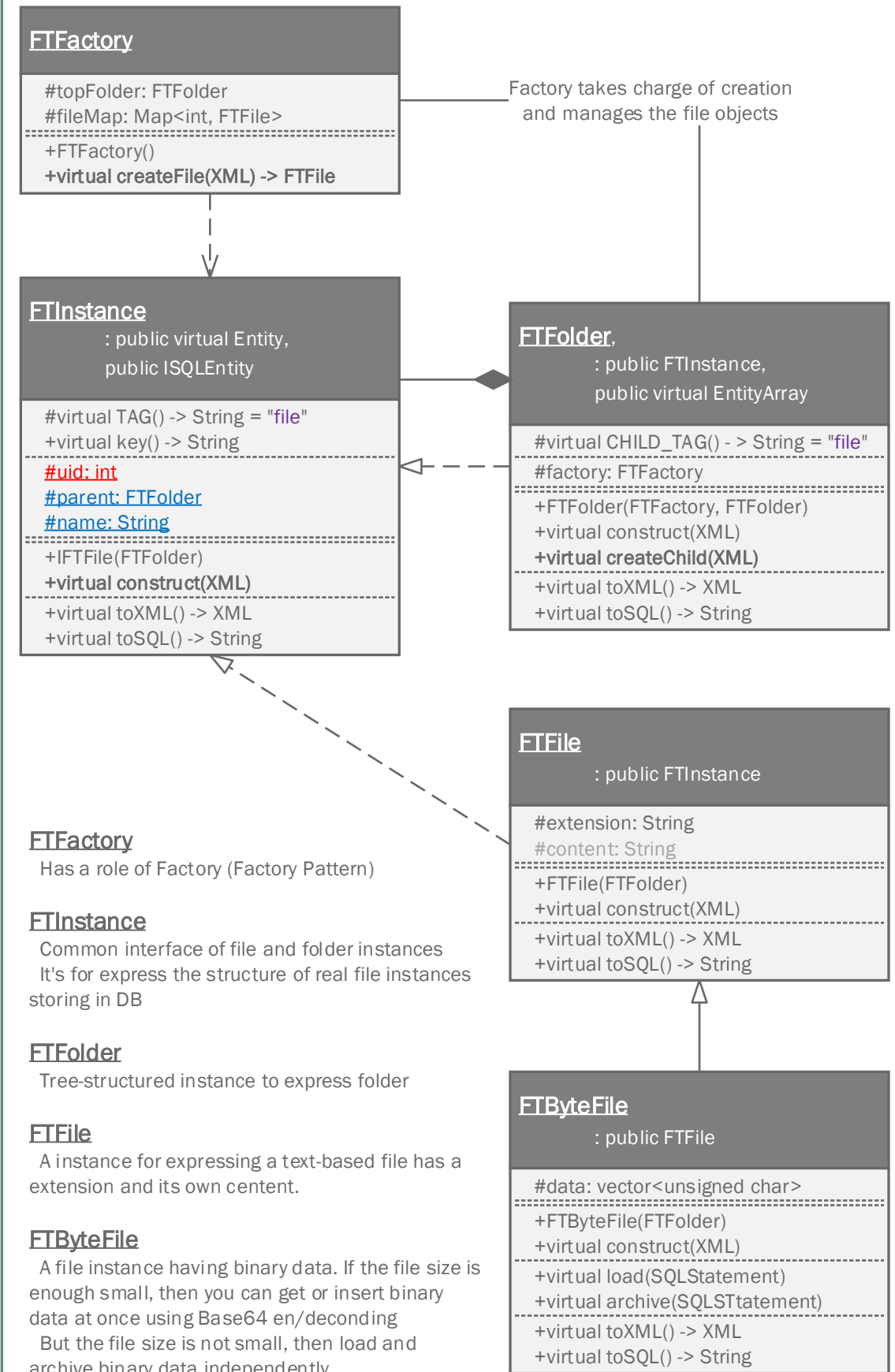
+ExternalSystemProxyClient  
(ExternalSystemProxy)

# Nam-Tree

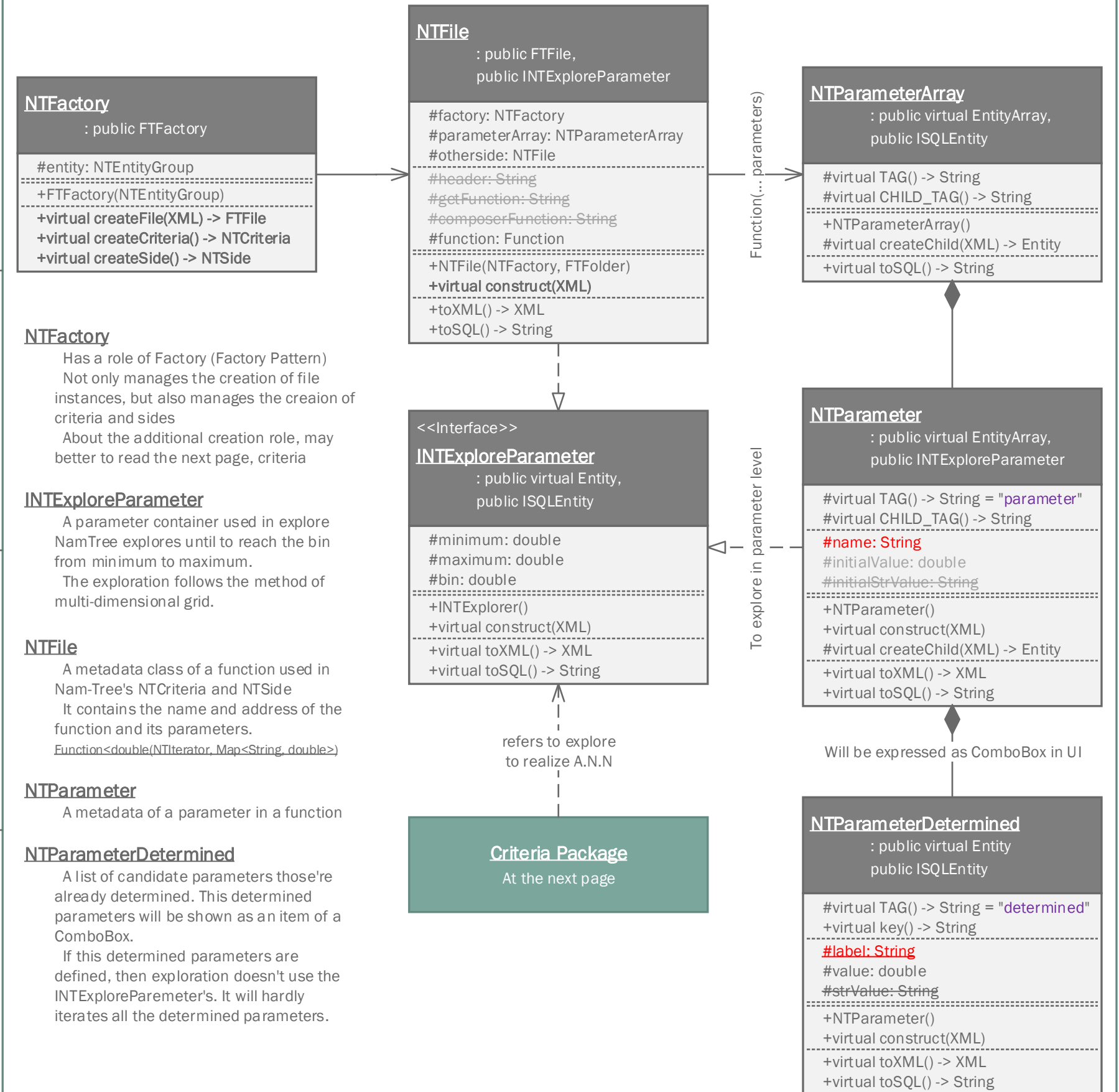
file  
criteria

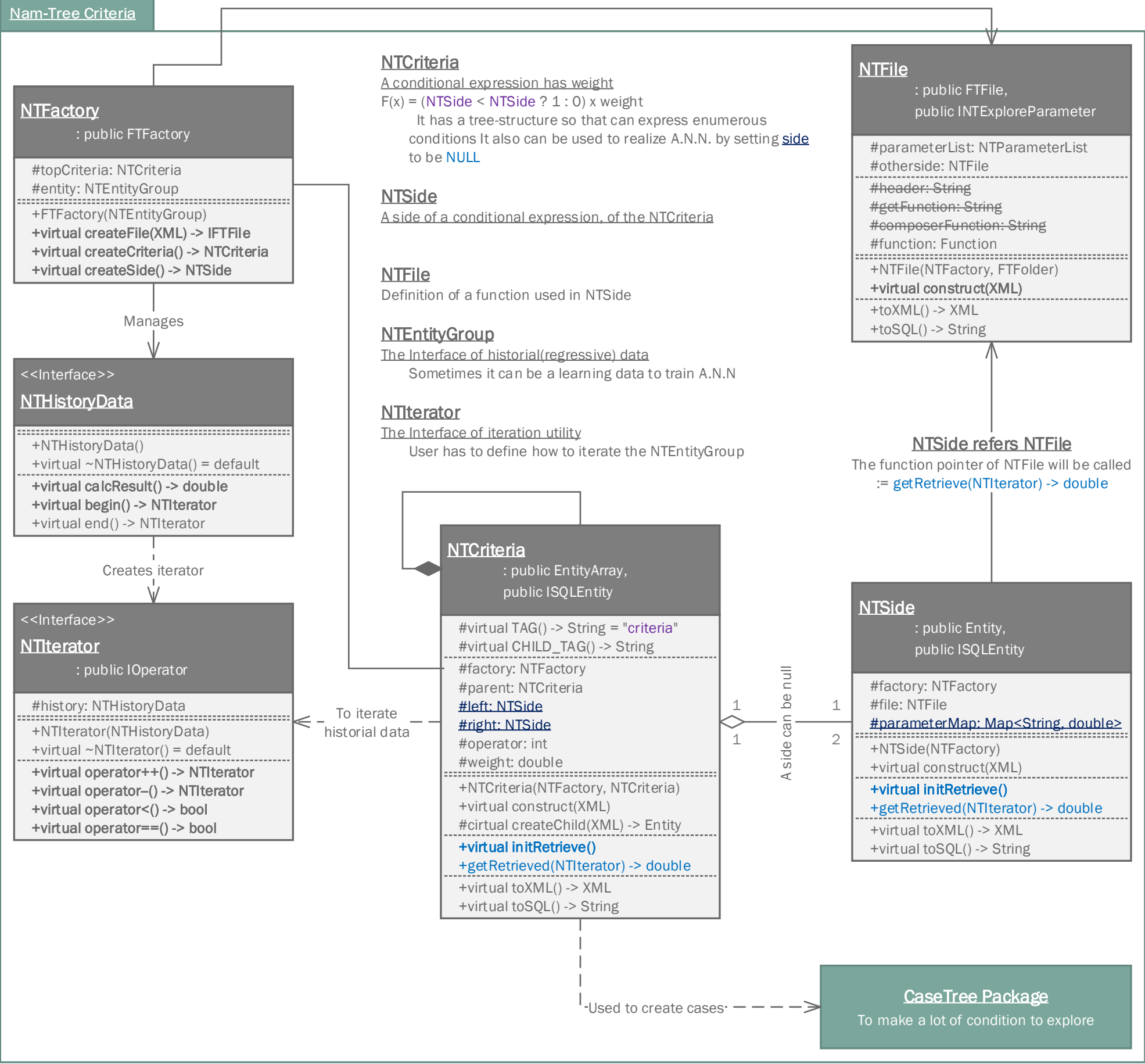
## Nam-Tree File Package

## File-Tree Package



## Nam-Tree File Package





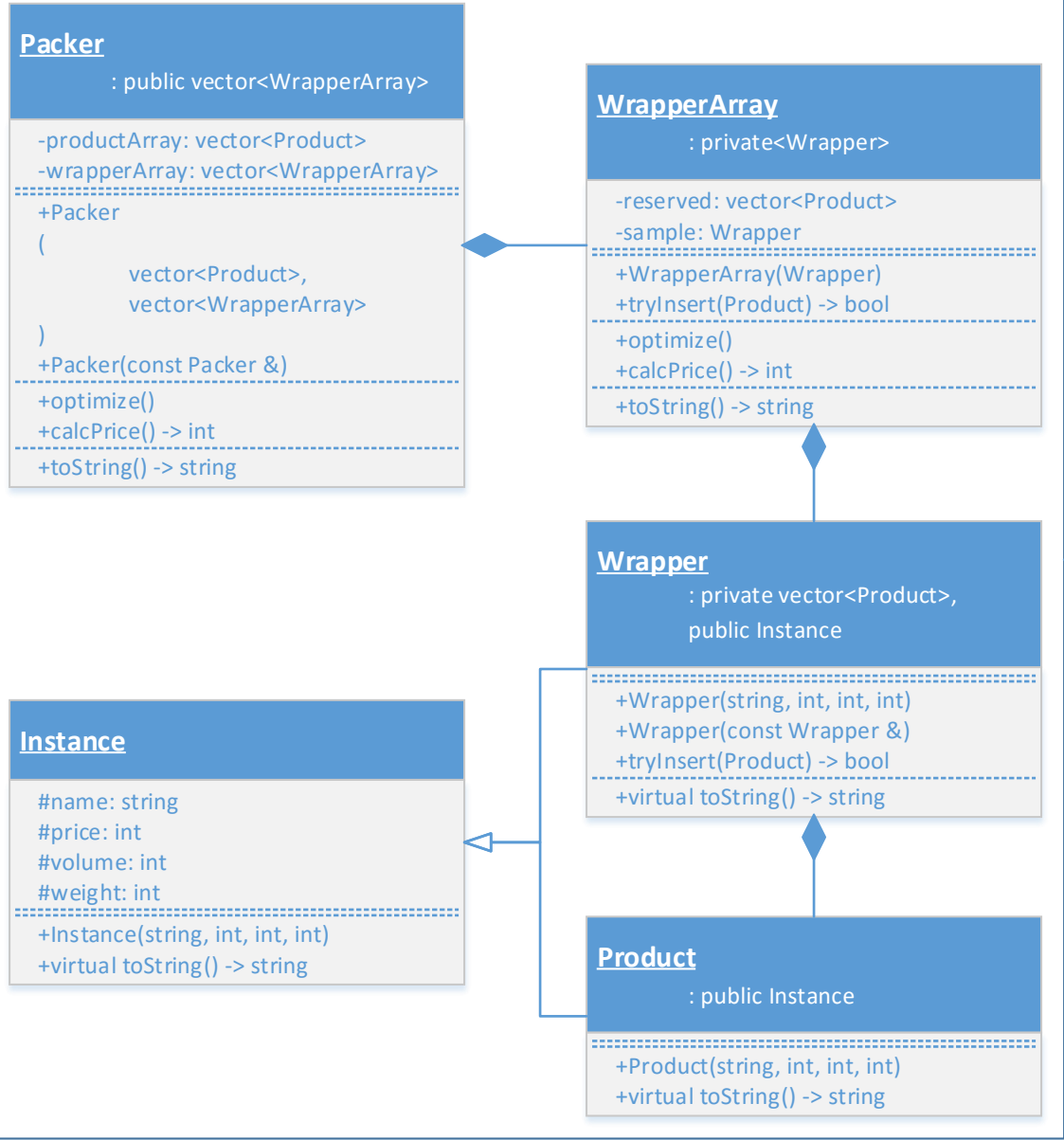
# Example

packer – case generator

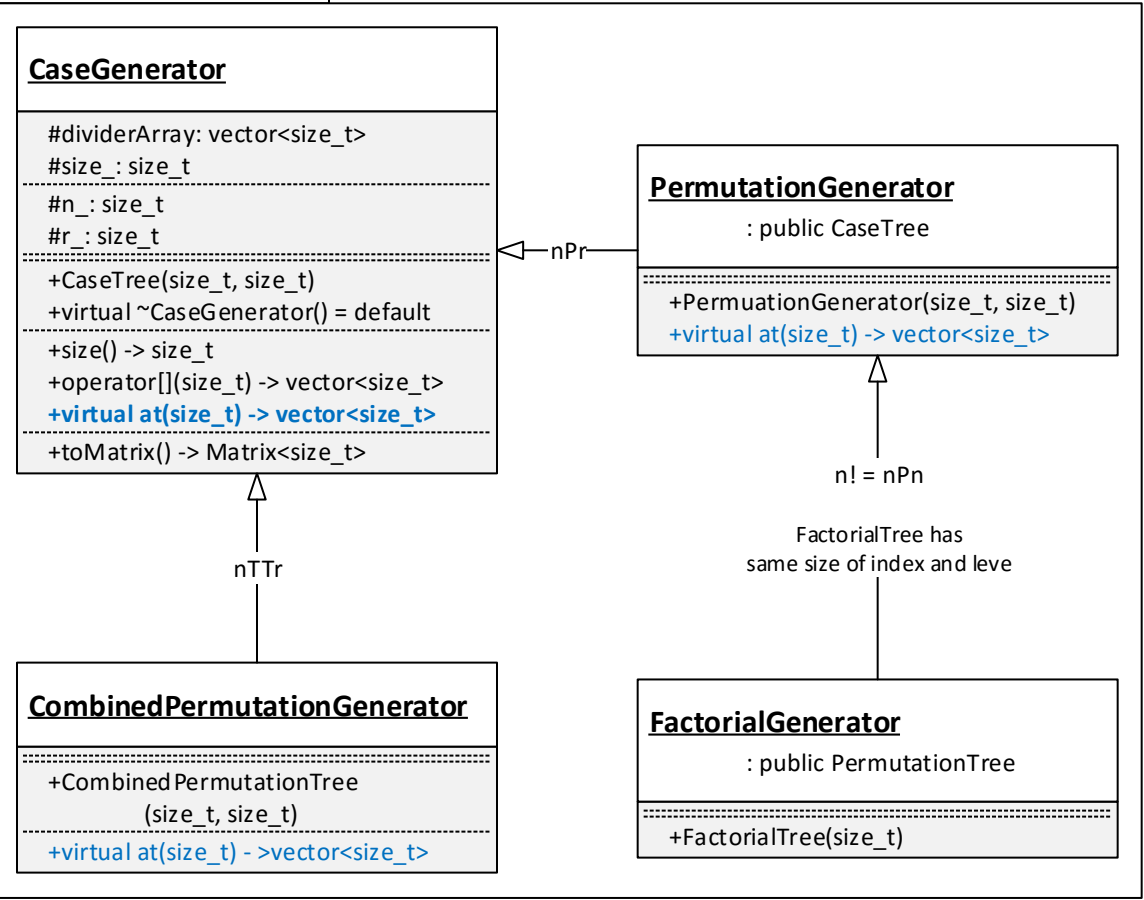
traveling salesman problem – genetic algorithm

console chat – interfaces of protocol

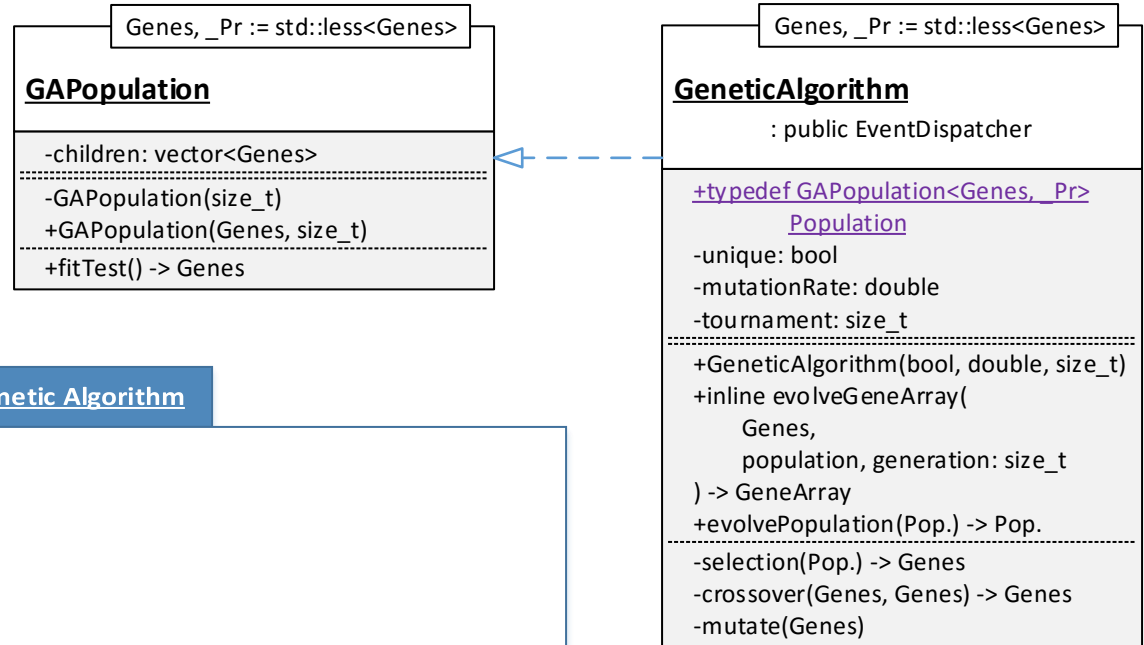
Packer, an example of CaseGenerator



CaseGenerator Package



## Traveling Salesman Problem



## Traveling Salesman Problem, an example of Genetic Algorithm

### Scheduler

-travel: Travel  
-ga\_parameters: struct GAParameters  
+Scheduler(Travel, struct GAParameters)  
+optimize() -> Travel

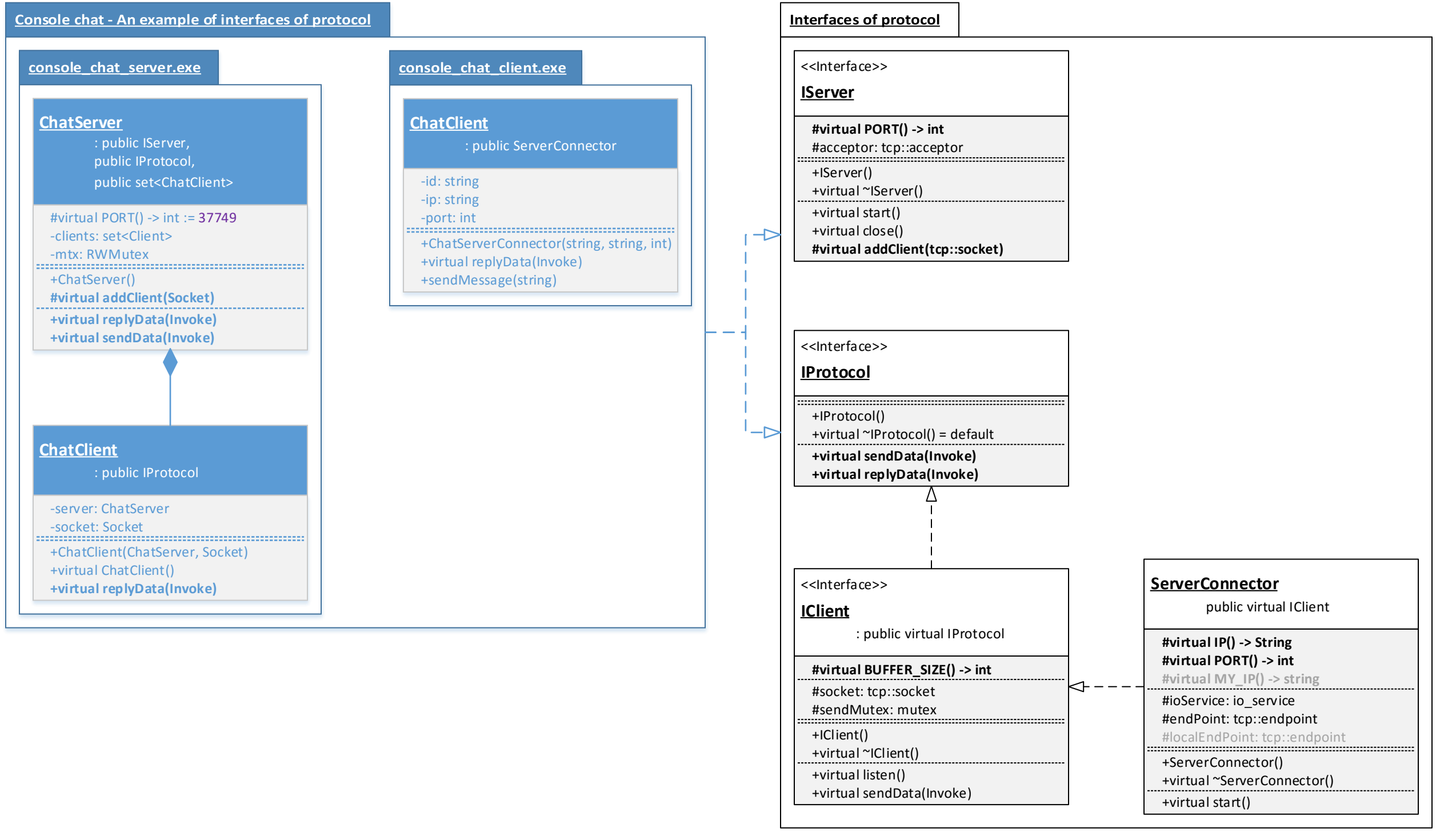
### Travel

: public vector<GeometryPoint>

+Travel()  
+Travel(const Travel &)  
+Travel(Travel&&)  
+operator<(Travel) -> double  
+calcDistance() -> double  
+toString() -> string

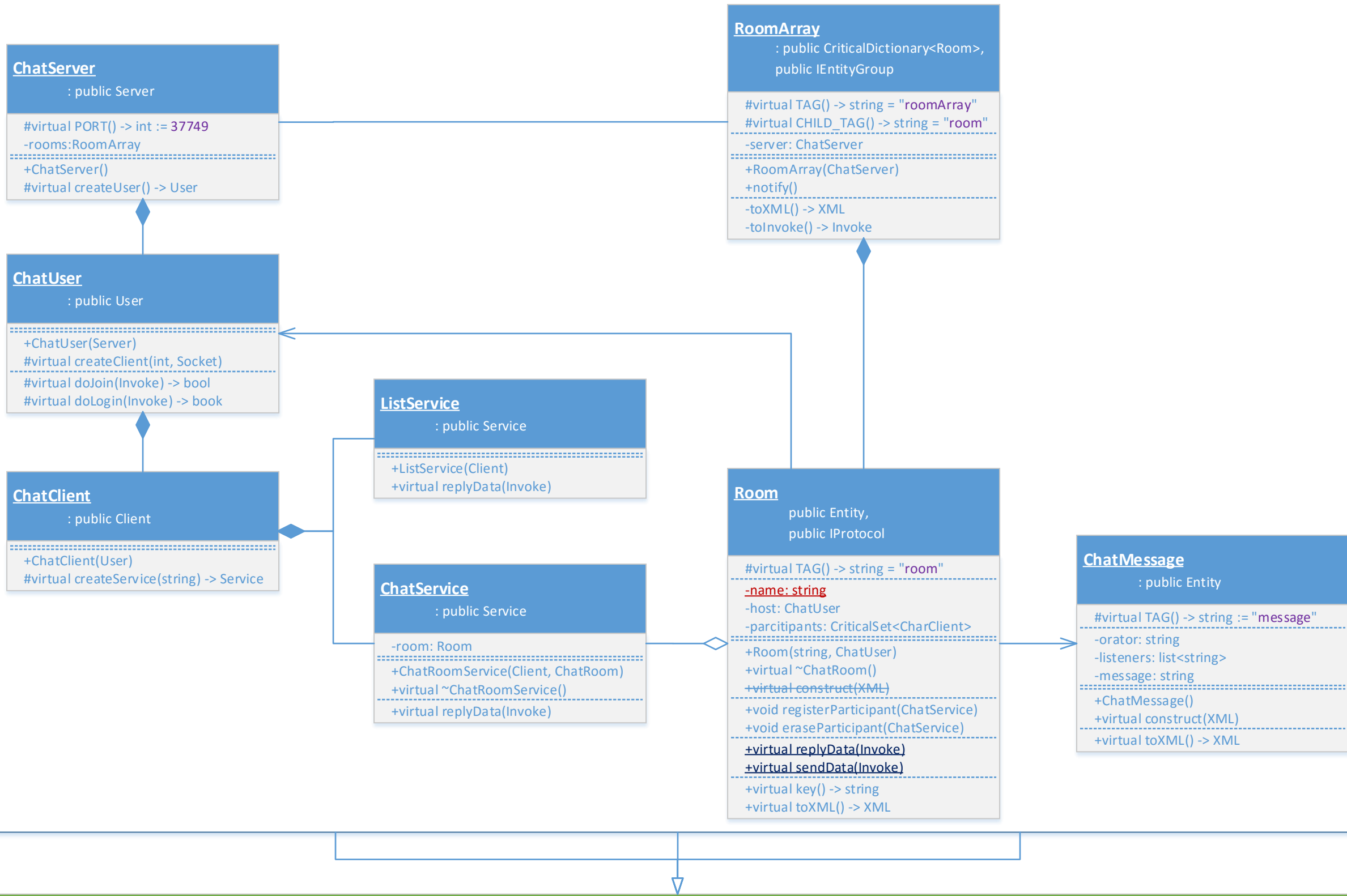
### GeometryPoint

-uid: int  
-longitude: double  
-latitude: double  
+GeometryPoint(int)  
+GeometryPoint(int, double, double)  
+calcDistance(GeometryPoint) -> double  
+toString() -> string





Chat service - an example of clouse service



Service Package in Protocol

A package for building cloud service