Samchon Framework

Guidance for Developers

Jeongho Nam

2015-08-24

samchon.org

Index

Samchon Framework - Select an index, then move to the section

Index

Samchon Framework - Select an index, then move to the section

1 Samchon Framework

1.1 Project outline

1.1.1 Purpose

오픈소스 프로젝트, 삼촌 프레임워크는 C++의, 클라우드 서버 및 분산처리 시스템을 구현하기 위한, 프레임워크로써 다음과 같은 목적을 위하여 제작되었습니다.

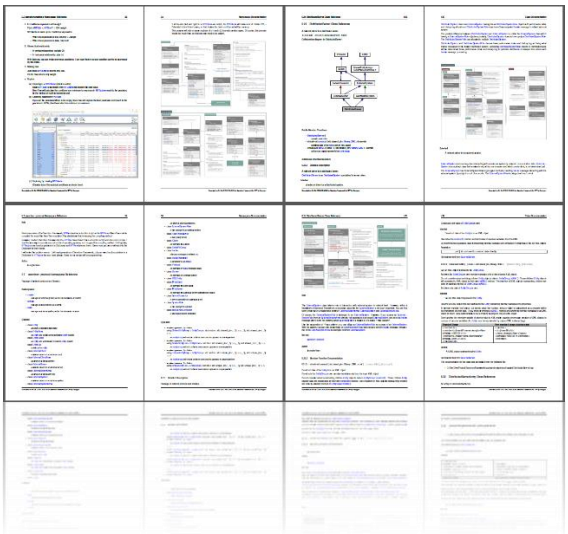
- 성능과 메모리 관리가 중요한 중대형 클라우드 시스템 구축
- 기존의 C++ 솔루션을 클라우드 서비스로 빌드
- 표준적인 메시지 프로토콜과 데이터 표기법을 사용하여 통합 시스템을 수월히 구축
- 복잡한 네트워크 시스템(ex: 트리 구조의 분산처리 시스템)을 S/W 적인 관점에서 수월히 구축
- 운영체제에 독립적인 (크로스 컴파일이 가능한) C++ 라이브러리 사용

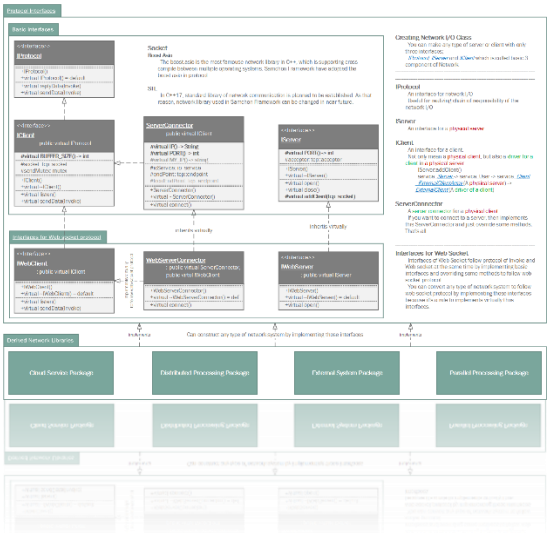
삼촌 프레임워크를 사용하고자 하는 개발자들을 위하여, 프레임워크의 주요 언어인 C++ 모듈을 필두로 하여, 삼촌 프레임워크로 구현한 클라우드 서버에 접속하기 위한 TypeScript 및 Flex(JS) 라이브러리 및 각 언어들의 API 문서가 제공됩니다.

또한, 오픈소스 프로젝트인 삼촌 프레임워크에 참여하고자 하는 분들 및 삼촌 프레임워크를 변경해 쓰고자 하는 개발자 및 설계자 분들을 위하여, 삼촌 프레임워크를 제작하는 데 쓰였던 설계도와 소스 일체를 제공합니다.

삼촌 프레임워크의 모듈, 소스, 설계도 및 API 문서 일체는 공식 홈페이지 및 깃브를 통하여 배포됩니다.

- 공식 홈페이지: http://samchon.org/framework
- 깃허브: https://github.com/samchon/framework





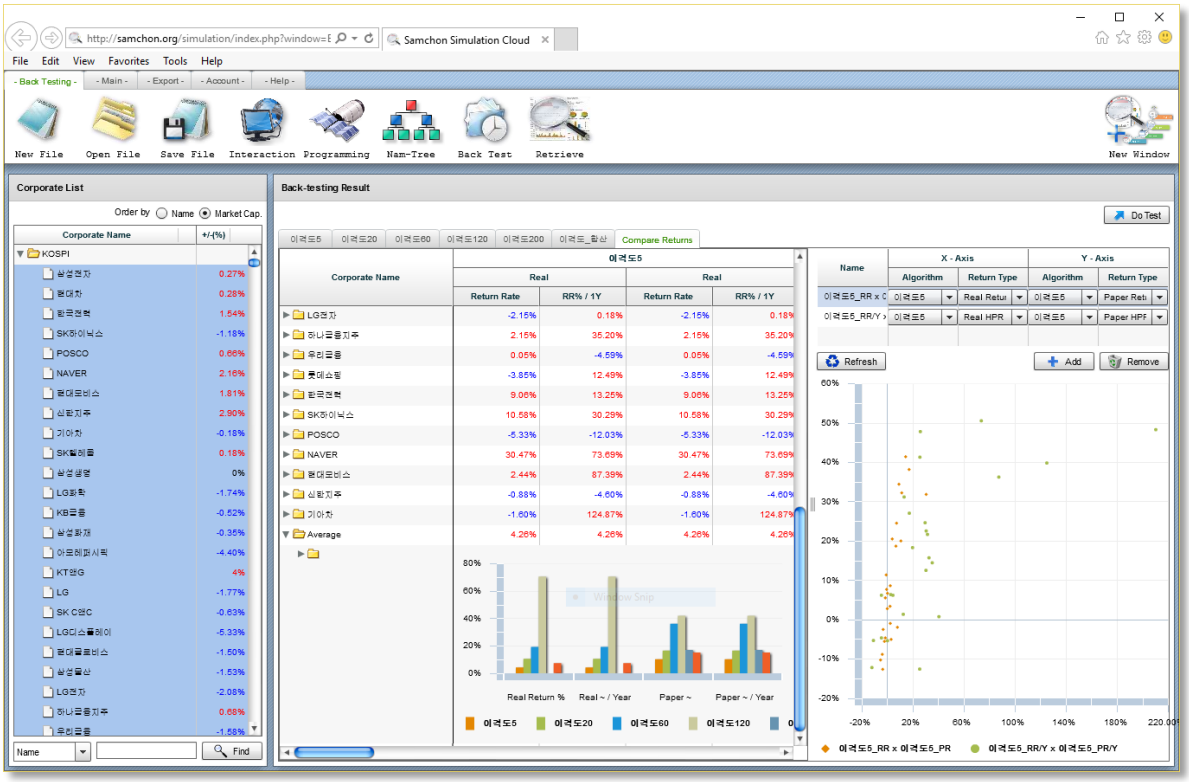
1.1.2 Examples

삼촌 프레임워크는 소스, 설계도 및 API 문서뿐만이 아니라, 삼촌 프레임워크를 활용하는 데 도움을 드리고자, 다양한 예제 코드 및 설계도를 제공해 드립니다.

간단하게는 일개 클래스나 부분 모듈에 대한 예제서부터 시작하여, 최종적으로는 솔루션 급에 이르기까지, 다양한 예제와 아키텍처 디자인을 통하여, 삼촌 프레임워크에 대하여 심도 있게 배워나가실 수 있습니다.

2015 년 10 월 3 일 현재 제공되는 솔루션급 예제

- **Samchon Simluation**
  - 주식 시뮬레이션 프로그램
  - 구 버전 Samchon Framework 에 대한 전반적인 활용 사례
  - http://samchon.github.io/simulation
- **Hansung timetable**
  - 모의 수강신청 프로그램
  - 구 버전 Nam-Tree 활용 사례
  - http://samchon.github.io/timetable



1.1.3 Specifications

- **프로젝트 일정**
  - 1 차: Hangang Simulation, 2011.09 ~ 2012.11
  - 2 차: Samchon Simulation, 2012.12 ~ 2014.05
  - 3 차: Samchon Library, 2014.06 ~ 2014.12
  - 4 차: Samchon Framework: 2015.01 ~
- **프로젝트 사용기술 및 규모**
  - 설계
    - 종류
      a. Class Diagrams
      b. Sequence Diagrams
      c. Entity Relationship Diagrams
    - 주요 기법: Prototyping 후 Waterfall 모델 적용
    - 주요 패턴
      a. Composite
      b. Factory
      c. Chain of responsibility
      d. Proxy
  - 사용 개발툴
    - 설계 도구
      a. StartUML
      b. Microsoft Visio
    - 언어 도구
      a. Visual Studio Community 2015
      b. Eclipse
      c. Flash Builder
    - 문서화 도구
      a. Doxygen
      b. TypeDoc
      c. AsDoc
  - 사용 언어
    - C++
    - Flex

        - TypeScript
        - T-SQL
    - 사용 라이브러리
        - Boost.Asio[1]
        - ODBC[2]

1.1.4 License

삼촌 프레임워크의 라이선스는 BSD 라이선스[3] 입니다. 누구던 이를 수정하여 재배포할 수 있으며 (다만, 재배포 시에는 꼭 원저자를 명시해 주셔야 합니다), 삼촌 프레임워크를 상업적인 용도로 사용함에 있어서도, 아무런 제약이 없습니다.

더불어 삼촌 프레임워크를 사용하는 프로젝트나 혹은 삼촌 프레임워크를 개작한 소스에 대하여도 별도의 공개 의무를 가지지 않습니다.

1.2 Components

1.2.1 API & Documents

- **API**
    - C++: http://samchon.github.io/framework/api/cpp/
    - TypeScript: http://samchon.github.io/framework/api/js/
    - Flex: http://samchon.github.io/framework/api/flex/
    - DB-SQL: 데이터베이스 API 는 이 개발자 가이드를 참조하세요.
- **Documents**
    - Development guide: http://samchon.github.io/framework/doc/development_guide.pdf
    - 개발가이드 한글: http://samchon.github.io/framework/docs/development_guide_kr.pdf
- **Architecture Designs**
    - C++ Class: http://samchon.github.io/framework/design/cpp_class_diagram.pdf
    - JS (Flex & TS) Class: http://samchon.github.io/framework/design/js_class_diagram.pdf
    - Sequence Diagram: http://samchon.github.io/framework/design/sequence_diagram.pdf
    - ERD: http://samchon.github.io/framework/design/entity_relationship_diagram.pdf

[1] Boost.Asio: http://www.boost.org/doc/libs/1_59_0/doc/html/boost_asio.html
[2] ODBC: https://msdn.microsoft.com/en-us/library/ms710252(v=vs.85).aspx
https://en.wikipedia.org/wiki/Open_Database_Connectivity
[3] BSD 허가서: https://ko.wikipedia.org/wiki/BSD_%ED%97%88%EA%B0%80%EC%84%9C

1.2.2 Source and Github

- 깃허브: https://github.com/samchon/
- 프레임워크 공식 홈페이지: http://samchon.org/framework

1.2.3 List of folders and files

- api
  - cpp.doxygen & cpp.bat
  - ts.bat
  - flex.bat
- cpp
  - samchon
    - library
    - protocol
      - service
      - master
      - slave
    - namtree
- design
  - cpp_class_diagram.vsdx
  - js_class_diagram.vsdx
  - sequence_diagram.vsdx
  - entity_relationship_diagram.vsdx
- doc
  - development_guide.docx
  - simulation_manual.pdf
  - nam_tree_manual.pdf
- flex
  - bin/SamchonFramework.swc
  - src
- js
  - SamchonFramework.ts
  - SamchonFramework.js
- old_version
  - v0.1

2 C++ Guidance

2.1 Outline

library

protocol

service

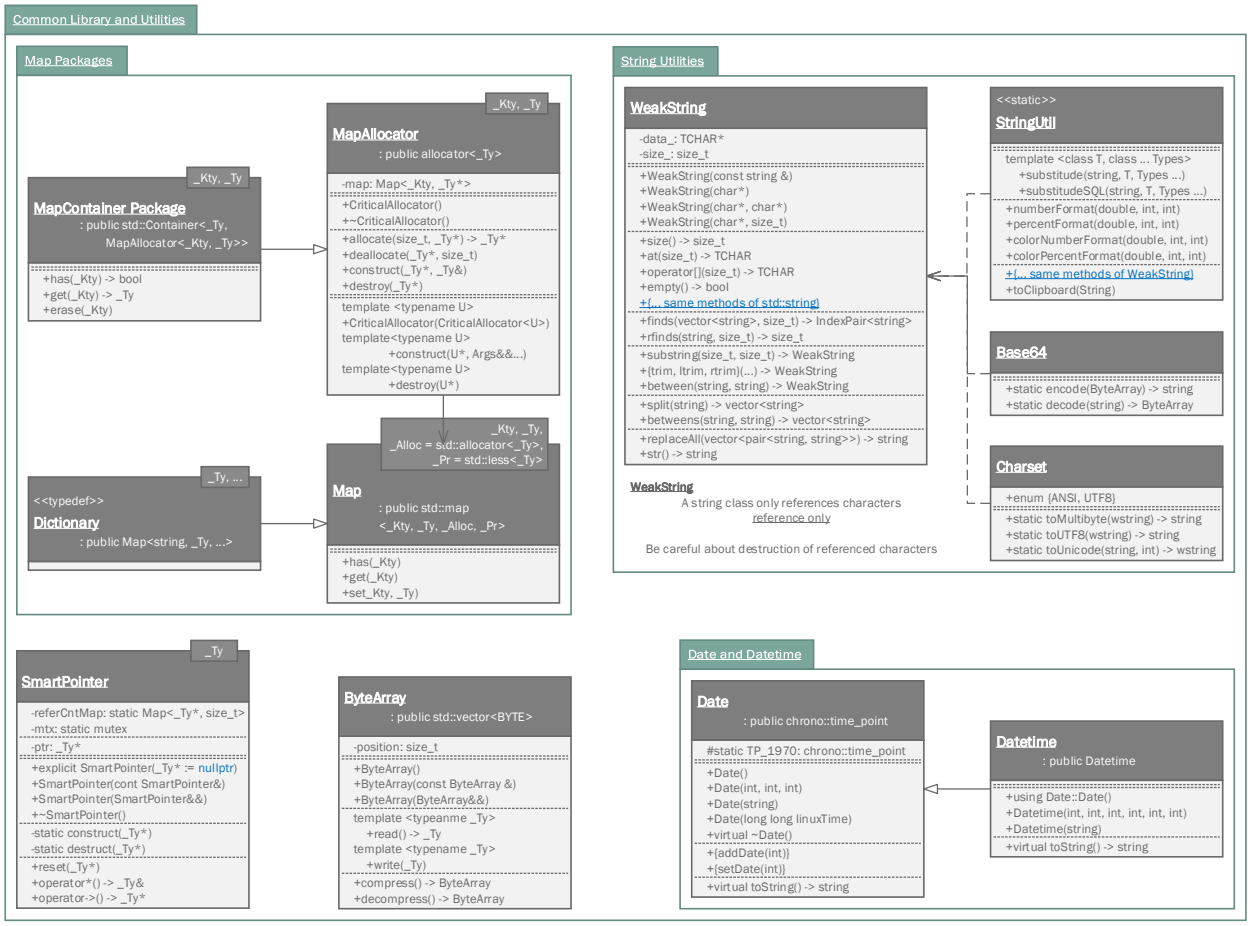
master

slave

namtree

2.2 Library

2.2.1 Utilities



- **WeakString**

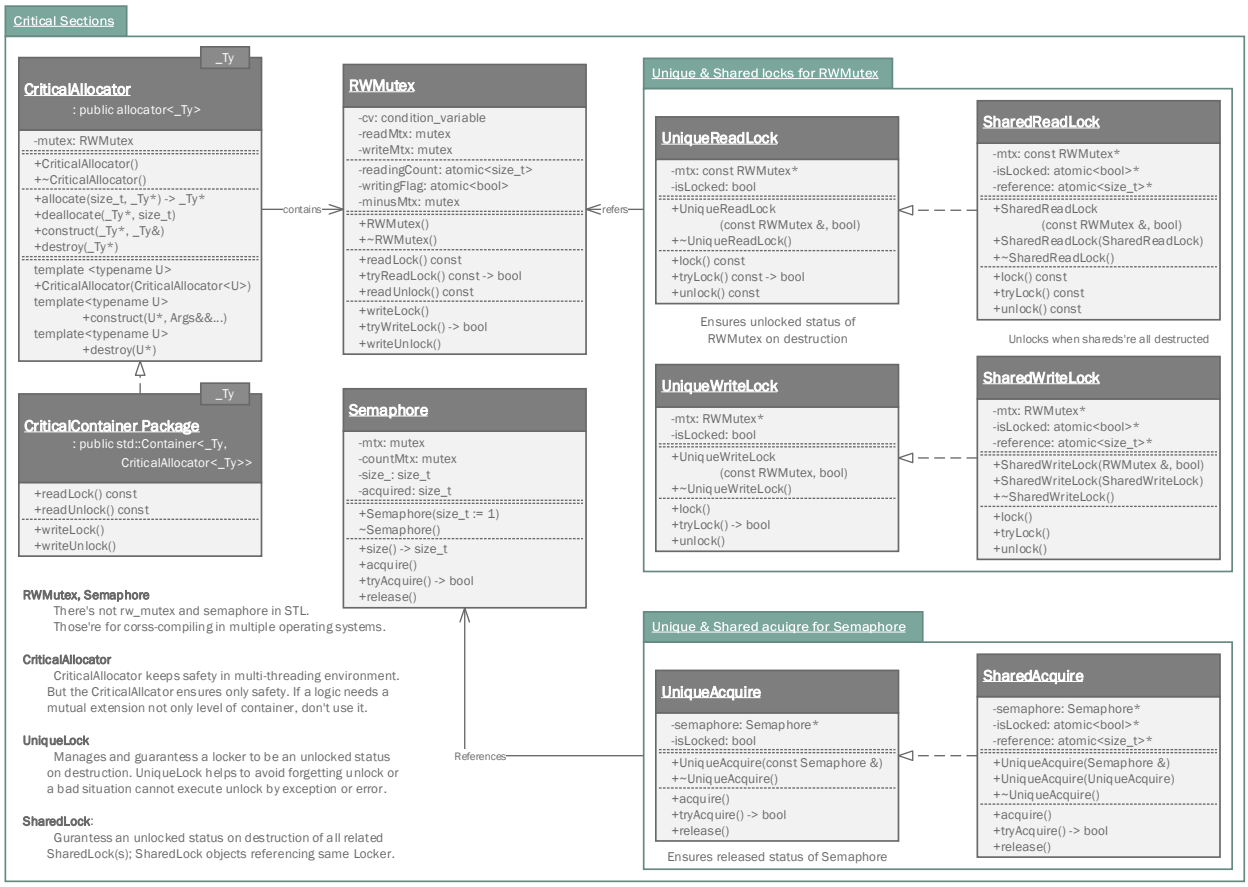
  문자열(char*)에 관한 유틸리티 클래스로써, std::string 과는 다르게 char*에 관하여 오로지 참조만 할 뿐, 생성과 소멸에는 일절 관여치 않습니다.

  std::string 특유의 과도한 복제(strcpy)를 막고자 하면, char* 포인터에 직접 액세스하여 C 스타일로 불편하게 써야 하고, std::string 특유의 유틸리티 함수들을 써 편하게 문자열을 다루자면, 과도한 복제를 막을 길이 없기에 구현하였습니다.

  std::string 에서 지원하는 모든 유틸리티 함수와 몇 가지 추가적인 유틸리티 함수를 제공합니다.

- **SmartPointer**: 전역 단위의 공유 포인터입니다.
- **ByteArray**: 바이너리 데이터를 다루는 유틸리티 클래스입니다.
- **StringUtil**: std::string 에서 지원하지 않는 여러가지 유틸리티 함수를 제공합니다.

2.2.2 Critical section



삼촌 프레임워크는 임계영역을 다루는 모듈을 제공합니다. 물론, MFC 에서 지원하는 세마포어와 리눅스 C 에서 지원하는 rw_mutex 등이 있지만, 이들은 운영체제 종속이 있기에, 크로스 컴파일을 할 수 없습니다.

삼촌 프레임워크는 이와 같은 이유로 STL 에서 지원하지 않는 rw_mutex, semaphore 및 여타 라이브러리들을 STL 스타일에 맞춰 제공합니다.

- **RWMutex**

  읽기, 쓰기락을 모두 지원하는 뮤텍스입니다.

  (Unique-Shared)(Read-Write)Lock 를 통하여 간접적으로 사용하시는 것이 좋습니다.

- **Semaphore**

  세마포어입니다.

  (Unique_Shared)Acquire 를 통하여 간접적으로 사용하시는 것이 좋습니다.

❖ **Unique Lockers**

- UniqueReadLock
- UniqueWriteLock
- UniqueAcquire

RWMutex 및 Semaphore(이하 Locker)의 해제 상태를 보장하는 라이브러리입니다.

Unique Locker 를 통하여 생성시 혹은 중도에 lock(또는 acquire)을 걸게 되면, Unique Locker 가 소멸되는 시점에 이를 해제해 줍니다. Locker 를 Unique Locker 를 통하여 락을 걸게 되면, 스택이 return 구문이나 예외상황을 만나 예기치 못하게 종료된다 하더라도 반드시 해제를 보장하기에 거뜬합니다.

❖ **Shared Lockers**

- SharedReadLock
- SharedWriteLock
- SharedAcquire

Locker 의 해제 상태를 보장하는 것은 Unique Lockers 와 마찬가지이되, 단일 객체가 아닌 연관된 모든 Shared Locker 객체가 소멸될 때 연관 Locker 를 해제시키게 됩니다.

Shared Locker 를 사용하면, 복잡한 연관 관계를 띄어야 하는 임계 영역을 관리함에 있어서도 매우 수월하게 관리할 수 있습니다. 단지 연관 관계의 영역마다 Shared Locker 를 복제하면 됩니다. 연관된 SharedLocker 가 모두 소멸될 때, Locker 가 해제됩니다.

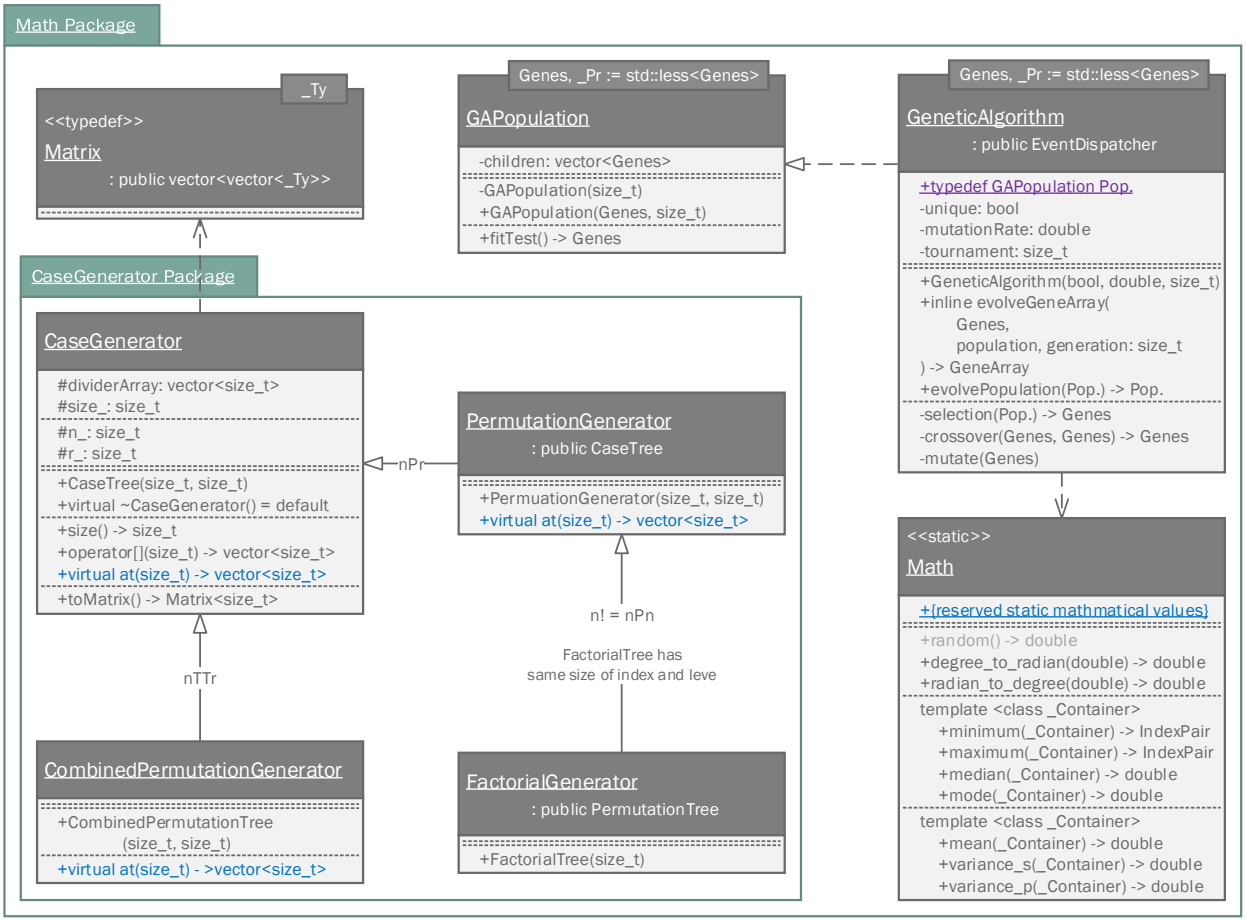
❖ **CriticalAllocator**

멀티 스레딩 환경에서 컨테이너의 병행 제어를 보장하는 std::allocator 입니다.

CriticalAllocator 는 안에 RWMutex 를 보유하고, 컨테이너 원소의 변경 때마다 쓰기락을 수행합니다. 하지만, 이처럼 보장하는 것은 오로지 병행 제어일 뿐입니다. 한 원소에 동시에 변경을 수행하는 것을 막아 오류로 인한 프로그램 종류만을 막을 뿐입니다.

RWMutex 가 로직 단계에서의 상호 배제를 지원하는 것은 아닙니다. 컨테이너 단위의 임계 영역을 제어함에 있어, 로직 단계의 상호 배제가 필요하다면, CriticalAllocator 를 쓰시면 안 됩니다.

2.2.3 Math



❖ **Math**

수학에 관련된 전역 상수 및 전역 메소드를 제공합니다.

자주 쓰이는 지수 및 로그 값을 전역 상수로써 제공하며, 각종 통계 및 몬테카를로 시뮬레이션 등에 관련된 함수들을 전역 메소드의 형태로써 제공합니다.

❖ **CaseGenerator**

경우의 수를 생성해주는 라이브러리입니다.

각 경우의 수에 대하여 index 에 관한 배열로써 리턴해 줍니다. 전체 경우의 수에 대하여, 각 경우의 수를 담은 배열, 즉 행렬처럼 사용하시면 됩니다.

- CombinedPermutationGenerator: $n\prod r$
- PermuationGenerator: nPr
- FactorialGenerator: $n!$

```
PermutationGenerator pg(7, 3);
for (size_t i = 0; i < pg.size(); i++) //0 to 7*6*5
{
        vector<size_t> &row = pg[i];

        //DO SOMETHING WITH ROW REPRESENTS A CASE
}
```

❖ **GeneticAlgorithm**

```
template <typename GeneArray, typename Compare = std::less<GeneArray>>
class GeneticAlgorithm
```

유전자 알고리즘[4]을 구현하기 위해 만들어진, 템플릿 클래스입니다.

템플릿 파라미터 GeneArray 에는 염기서열(sequence listing)에 해당하는 클래스를, Compare 에는 적합도 검증에 쓰일 구조체 함수를 넣어주십시오. 염기서열에 해당하는 GeneArray 는 반드시 std::array 나 std::vector 그 자체, 혹은 이를 상속한 클래스여야만 합니다.

적당한 수준의 돌연변이율, 세대 수 및 각 세대 당 인구수, 그리고 토너먼트 횟수를 통해 유전자 알고리즘 특유의 진화(학습효과)를 시행하십시오.

- `typedef GAPopulation<GeneArray, Compare> MyPopulation`

❖ **GeneticPopulation**

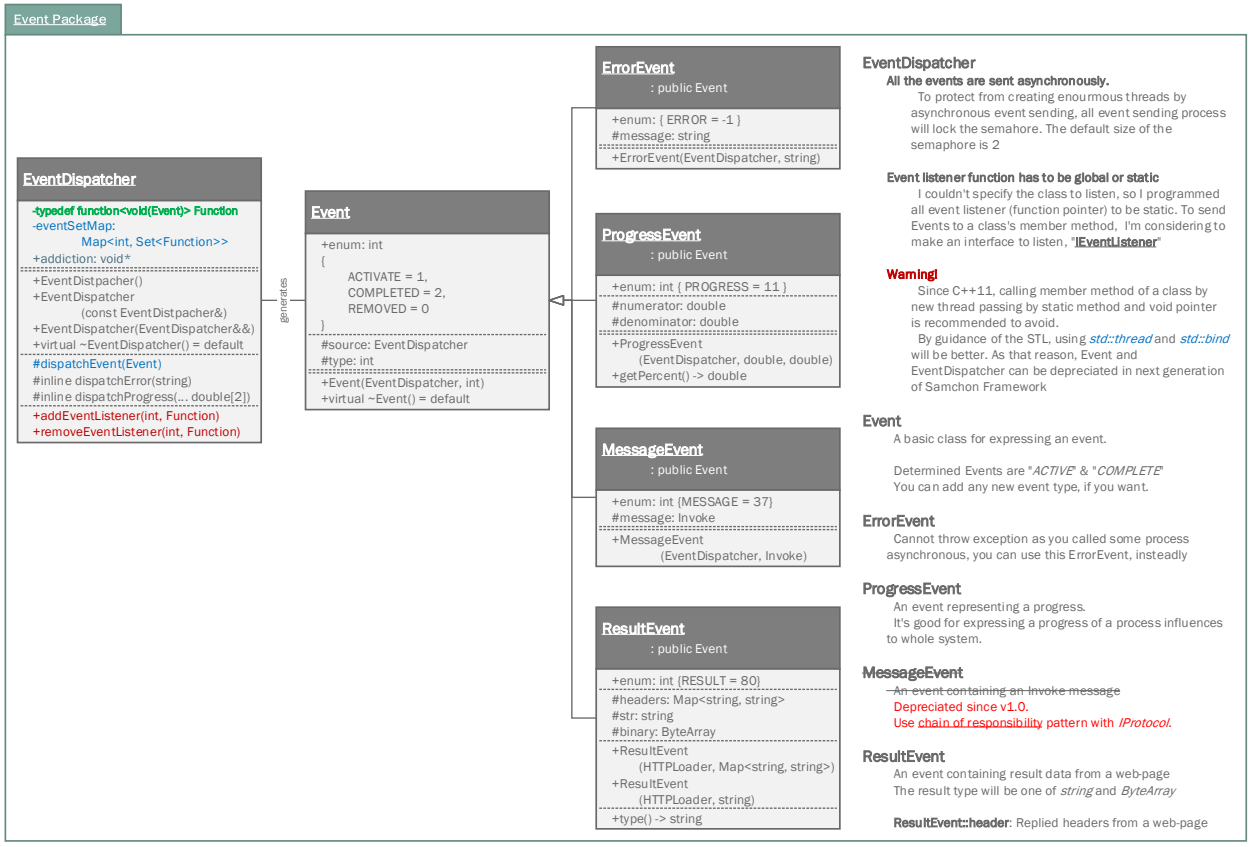
```
template <typename GeneArray, typename Compare = std::less<GeneArray>>
class GAPopulation
```

유전자 알고리즘에 사용되는 개념 중 한 세대(A population)를 표현하는 템플릿 클래스로써 염기서열에 대한 배열, 즉 염기에 대한 행렬이기도 합니다.

더불어 GeneticPopulation 은 GeneticAlgorithm 클래스에서도 정의했던 바 있던 템플릿 파라미터, 적합도 검증 함수를 의미하는 Compare 가 실제로 실시되는 클래스이기도 합니다.

[4] 유전자 알고리즘: https://en.wikipedia.org/wiki/Genetic_algorithm

2.2.4 Event



이벤트 모듈은 백그라운드에서 동작하는 비동기 이벤트를 구현하기 위하여 설계되었습니다.

❖ EventDistpacher[5]

해당 객체에 관한 이벤트를 발생시키며, 이벤트의 리스너 목록을 총괄하는 추상객체입니다.

dispatchEvent() 메소드를 통하여 이벤트를 발생시킬 수 있으며, 모든 이벤트의 처리(발생 및 리스닝)는 별도의 스레드 풀에서 비동기적으로 이루어집니다. 해당 객체에 할당되는 스레드 풀의 개수를 조정할 수 있습니다 (EventDispatcher::semaphore. 기본 2 개).

[5] EventDispatcher 는 추후 삭제되거나 크게 변경될 여지가 있습니다.
현재 EventDispatcher 는 리스너 함수들은 모두 전역 함수의 형태입니다.
만일, 이벤트를 전달받을 함수가 특정 클래스의 멤버 메소드라면, EventDispatcher 의 addication(void*)을 통해서 해당 클래스의 포인터 주소를 입력하고, 전역 함수에서 이를 reinterpret_cast 를 통하여 해당 클래스의 멤버 메소드로 전달해야 합니다.
하지만, 새 표준인 C++11 에선 이와 같은 void*와 전역함수의 사용을 피하도록 권고하고 있기에, Event 모듈은 추후 삭제되거나 크게 변경될 수 있습니다. 당분간은 **std::thread** 와 **std::bind** 를 사용하세요.

❖ **Event**

백그라운드에서 돌아가는 이벤트를 의미하는 추상객체입니다.

이벤트 타입이 무엇인지, 어느 객체에서 발생한 이벤트인지 등을 담고 있습니다.

발생코자 하는 이벤트에서 위 이상의 정보가 필요할 경우, 해당 정보에 관하여서는 이 Event 클래스를 상속하여 정의토록 해 주십시오. ErrorEvent 와 ProgressEvent 가 이에 대한 사례 중 하나일 것입니다.

미리 정의된 이벤트 타입 (numeration code)

- **ACTIVATE**
- **COMPLETED**

❖ **ErrorEvent**

오류 및 예외상황을 전달하기 위해 설계되었습니다.

별도의 스레드를 사용, exception 이 발생해도 이를 catch 하기 어려운 경우에 적합합니다.

- **ERROR**

❖ **ProgressEvent**

특정 작업의 진행 상황(m of n 또는 x%)을 전달하기 위해 설계된 이벤트 클래스입니다.

ProgressEvent 는 현재의 진행상황을 화면에 출력하거나 연관 네트워크 시스템에 통보하는 작업 등 또한, 적잖은 시간을 소요하여 본래의 연산에 부담이 될 때, 또는 진행상황에 대한 통보가 여러 군데에 동시다발적으로 이루어져야 할 때 특히 적합합니다.

EventDispatcher 와 ProgressEvent 를 이용하여, 진행상황 출력에 관한 작업을 백그라운드에서 동작하도록 만드십시오.

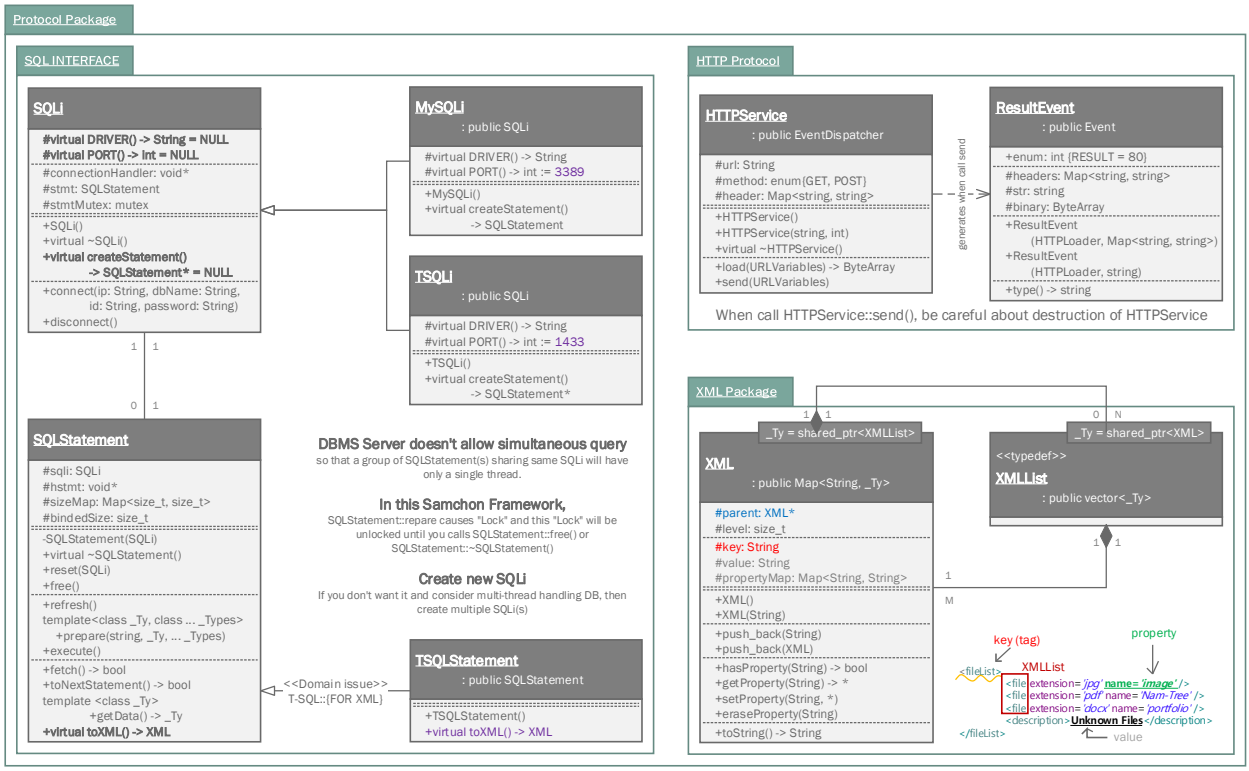
- **PROGRESS**

❖ **MessageEvent, ResultEvent**

MessageEvent 는 Invoke 메시지의 수신을 알리는 이벤트였고, ResultEvent 는 HTTP 페이지를 완전히 불러들였을 때 발생하는 이벤트를 구현하기 위한 객체였습니다. 모두 현재는 삭제되어 구 버전에서만 존재합니다.

MessageEvent 는 Invoke 메시지 전달 방식이 "Chain of responsibility Pattern" 으로 변경되어 사라졌으며, ResultEvent 는 이를 발생시키던 HTTPLoader 객체가 삭제되어 사라졌습니다.

2.2.5 Data



데이터 모듈은 삼촌 프레임워크에서 제공하는 데이터의 입출력 및 표기형식에 관한 모듈입니다.

❖ XML

XML 데이터의 구성 및 파싱을 위하여 제작된 오브젝트입니다.

트리 구조를 가지고 있으며, 각 태그와 속성에 대하여 {key, value} pair 의 형태로 액세스합니다.

```
<?xml version="1.0" encoding="utf-8" ?>
<invoke listener="setDepartment">
    <parameter type="string">samchon</parameter>
    <parameter type="XML">
        <memberList department="programmer">
            <member id="john" name="John Doe" authority="3" />
            <member id="samchon" name="Jeongho Nam" authority="5" />
        </memberList>
    </parameter>
</invoke>
```

```
shared_ptr<XML> xml(new XML("위의 XML 문자열 ....."));
```

- 첫번째 parameter 의 value, "samchon" 에 접근

```
shared_ptr<XMLList> &parameterList = xml->get("parameter");
shared_ptr<XML> &parameter = parameterList->at(0);
cout << parameter->getValue() << endl;
```

- 두 번째 member 의 속성값 id="samchon"에 접근

```
cout << xml->get("parameter")->at(1)
        ->get("memberist")->at(0)->get("member")->at(1)
            ->getProperty("id") << endl;
```

❖ **SQLi**

DBMS 서버로의 접속 및 **SQLStatement** 오브젝트의 관리를 담당하는 클래스로써, ODBC 를 개체지향적으로 쓸 수 있게 몇 가지 메소드를 제공하는 Adaptor pattern 에 해당합니다

한 가지 주의할 점은, 하나의 접속된 **SQLi** 객체는 DBMS 서버 측에서 보면 하나의 세션에 해당합니다. 그리고 DBMS 에 접속된 세션은, 오로지 한 번에 한 가지 작업밖에 수행할 수 없습니다. 때문에 **SQLi** 는 **SQLStatement** 를 관리하며, **SQLStatement** 에서 쿼리가 수행될 시에, **SQLi** 를 잠가버림[6]으로써, 다른 **SQLStatement** 에서 해당 **SQLi** 및 질의를 사용할 수 없게 됩니다.

다수의 SQL 쿼리가 동시에 필요하다면, 그만큼의 **SQLi** 를 생성, 각기 DBMS 에 접속하여 개별적으로 세션을 획득한 후에라야 가능합니다.

❖ **SQLStatement**

SQL 문의 질의를 수행하고, 다시금 해당 질의로부터의 결과값을 받아오는 클래스입니다.

SQLStatement 는 자체적으로 생성될 수 없으며, 반드시 SQLi 의 createStatement() 메소드를 통하여 생성하도록 되어있습니다. 또한, SQLStatement 의 질의는 SQLi 를 잠구는 효과를 발생시킵니다. SQLStatement 에서 작업이 끝난 후에는 반드시 SQLStatement 객체를 소멸시키거나 free() 메소드를 호출하시어 SQLi 의 잠금 상태를 풀어주십시오.

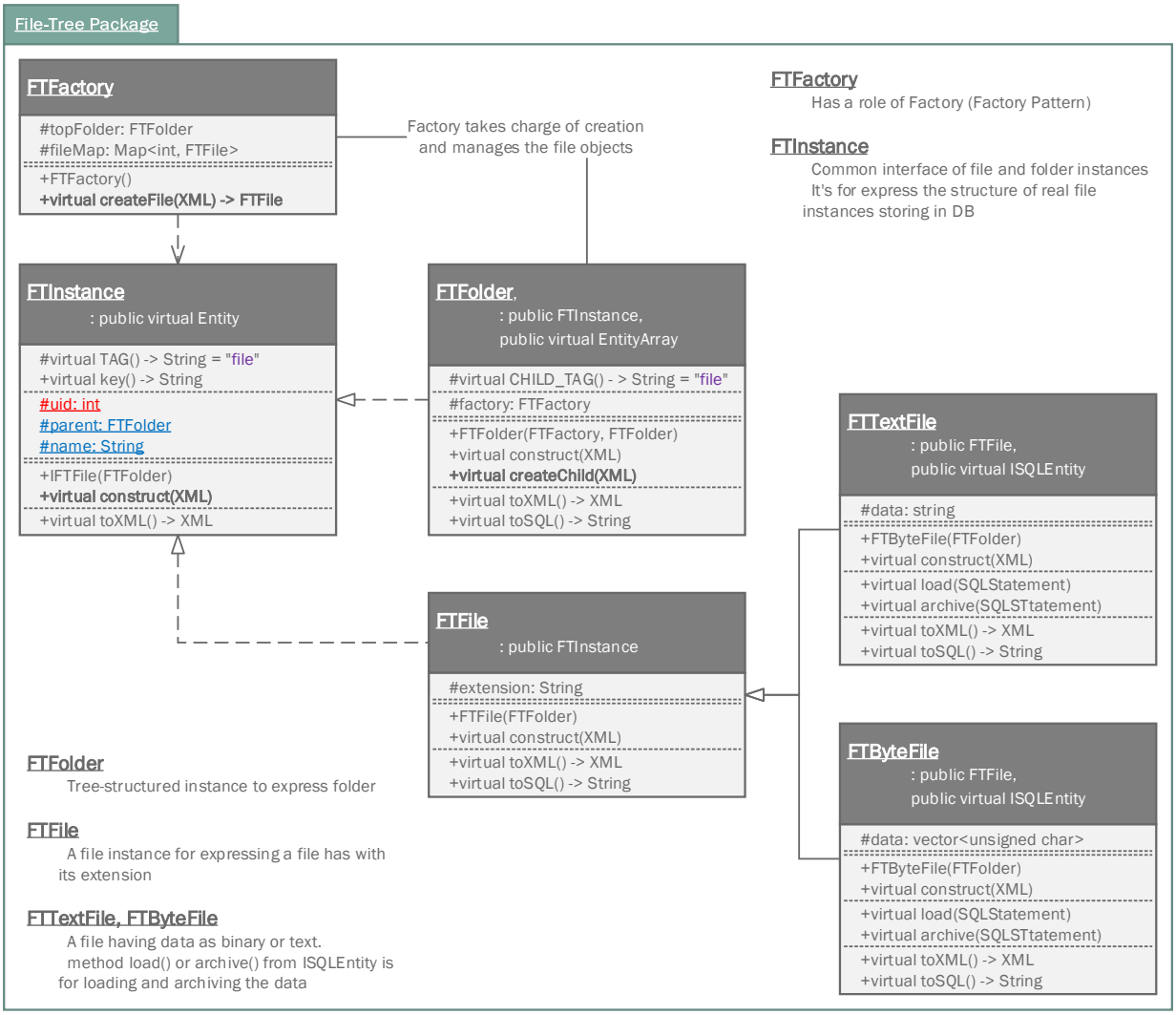
---

[6] SQLStatement 가 SQLi 를 잠근다:

ODBC 에서는 다수의 스레드가 하나의 DB 세션에 동시에 접근하는 것에 대하여 따로 대책을 마련해놓지 않았습니다. 때문에 이와 같이 다수의 스레드가 하나의 DB 세션에 접근하는 경우에는 바로 에러가 리턴되어 프로그램이 종료되고 맙니다.

이를 방지하기 위하여 SQLStatement 의 질의는 SQLi 를 잠그게끔 설계하였습니다.

2.2.6 File-tree



파일트리는 트리 구조의 폴더 및 파일을 표현하기 위한 추상 모듈입니다. 더불어 DB 와의 입출력을 지원하기에 활용하기에 따라서는 실제 폴더와 파일을 표현하는 것이 아닌, DB 에 기반을 둔. 가상의 폴더 및 파일시스템을 구현하는 것도 가능합니다.

파일트리 모듈은 **Protocol** 의 **Entity** 모듈을 상속하여 구현하였으며, 트리 구조(1:N recursive and hierarchical relationship)의 폴더 및 파일의 표현 에는 Composite 패턴이, 각 폴더 및 파일의 생성에는 Factory 패턴(by **FTFactory**)이 사용되었습니다.

이 추상 모듈을 상속하여 여러분의 환경에 맞는 파일트리 모듈을 만드시면 됩니다. 가장 가까운 예제는 **Nam-Tree** 라이브러리어 있습니다. **Nam-Tree** 라이브러리 중에 **File instances** 모듈이 이 파일트리 모듈을 상속하여 구현하였습니다.

❖ **FTFactory**

파일 객체의 생성을 담당하는 팩토리 클래스(Factory pattern)입니다.

더불어 해당 파일 패키지의 관리 및 메타 데이터와 DB 입출력을 책임지는 매니져 클래스이기도 합니다. 멤버로써 해당 파일트리 패키지가 속하는 어플리케이션과 카테고리 정보(as enumeration code), 그리고 최상위 폴더 및 모든 파일에 대한 단축 경로를 지니고 있습니다.

파일트리 모듈을 상속할 때, **FTFile** 만 상속하지 마시고, 필히 **FTFactory** 를 상속하시어 파일생성 메소드, createFile()를 재정의 해 주십시오.

❖ **FTInstance**

파일트리 모듈에서 파일과 폴더를 포괄하는 추상객체입니다.

Entity 클래스를 상속하여 구현하였으며, (파일 또는 폴더의) 식별자, 개체 이름 및 상위 폴더에 대한 정보를 지닙니다. 더불어 XML 입출력 및 (개체 단위의) 단일 레코드 DB 입출력을 지원합니다.

❖ **FTFolder**

폴더를 표현하기 위한 클래스입니다.

기본적으로 **FTInstance** 가 상속되었으며, 이에 **SharedEntityArray**<FTInstance>를 더불어 상속함으로써 **FTFolder** 가 **FTInstance** 를 자식으로 지닐 수 있게 함으로써(Composition pattern), 폴더 구조를 표현할 수 있도록 구현하였습니다.

또한, **FTFolder** 는 생성자 메소드는 **FTFactory** 의 createFile()에 위임되었습니다. **FTFolder** 의 부모 클래스, **EntityGroup** 고유의 Factory method 인 createChild()를 재정의하지 마십시오.

❖ **FTFile**

파일을 표현하기 위한 추상 클래스입니다.

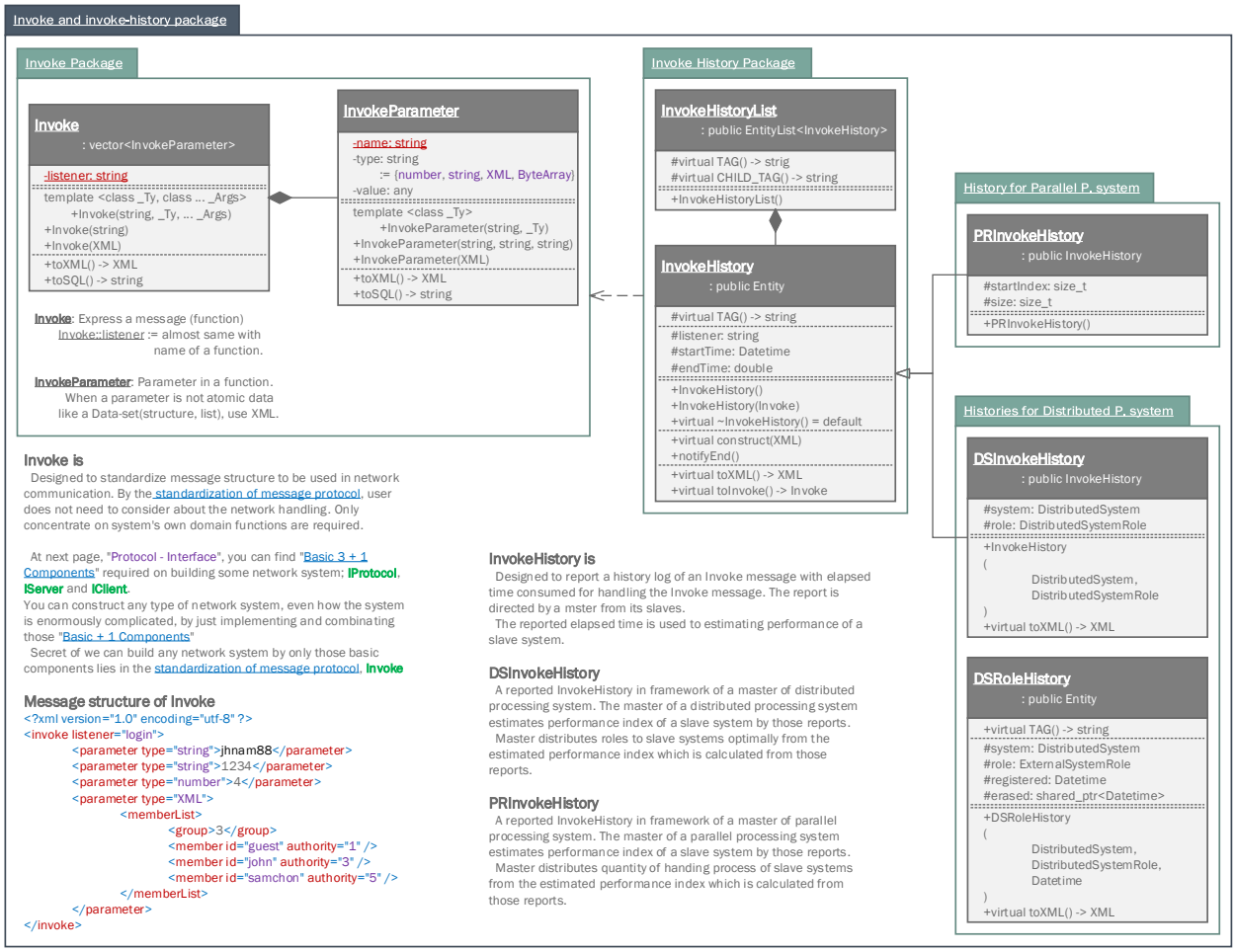
FTInstance 를 상속하였으며, 더불어 확장자 정보를 가지고 있습니다. 커스텀 파일 개체를 만들고자 하실 때, 이 **FTFile** 을 상속하여 구현하십시오.

- **FTTextFile**: 텍스트 파일을 표현하기 위한 추상 클래스
- **FTBinaryFile**: 바이너리 파일을 표현하기 위한 추상 클래스
- **비고**

  Nam-Tree 모듈의 **NTFile** 은 위 두 가지 경우에 모두 해당 안 됨.
  자체적인 포맷을 가지고 있음

2.3 Protocol

2.3.1 Invoke



Invoke 는, 삼촌 프레임워크의 네트워크 통신에 쓰이는, (표준화된) 메시지 프로토콜입니다.

삼촌 프레임워크에서 제공되는 모든 종류의 네트워크 라이브러리 및 예제는 이 Invoke 메시지 프로토콜을 따르고 있습니다.

```
<?xml version="1.0" encoding="utf-8" ?>
<invoke listener="메시지를 들을 대상, 거진 함수명">
        <parameter name="변수명" type="변수 타입">값</parameter>
        <parameter name="변수명" type="변수 타입">1234</parameter>
</invoke>
```

Invoke 는 위와 같이 XML 형태의 메시지 구조를 가집니다. Invoke 는 해당 메시지를 듣는 대상(객체 내지 메소드)이 누구인지, 해당 메시지의 파라미터들은 타입이 무엇이고 그 안의 값이 무엇인지 등을 담고 있습니다.

Invoke 에 속하는 파라미터의 값이 단일 원자값(int, double, string 등 숫자 혹은 글자)이 아닐 때는, 다음과 같은 표기법을 따르게 됩니다.

1) 파라미터가 binary data 일 때

```
<?xml version="1.0" encoding="utf-8"
<invoke listener="replyFile">
    <parameter name="name" type="string"> development_guide </parameter>
    <parameter name="extension" type="string"> pdf </parameter>
    <parameter name="data" type="ByteArray"> 4711343 </parameter>
</invoke>
01000010100101010101110101011010101111111110101010111011111010101
00010101010000010101011010000101010100000010101000010100101001010100
{...}
```

Invoke 메시지를 헤더 격으로써, 바이너리 데이터의 사이즈와 함께 먼저 전송합니다. 바이너리 데이터는 그 후에 전송합니다.

```
virtual void replyData(shared_ptr<Invoke> invoke) override
{
    if (invoke->getListener() == "replyFile")
    {
        string &name = invoke->at(0)->getValue<string>();
        string &extension = invoke->at(1)->getValue<string>();
        const ByteArray &byteArray
            = invoke->at(2)->referValue<ByteArray>();

        replyFile(name, extension, byteArray);
    }
}
```

2) 파라미터가 엔터티 오브젝트일 때

```
<?xml version="1.0" encoding="utf-8" ?>
<invoke listener="setDepartment">
    <parameter type="string">samchon</parameter>
    <parameter type="XML">
        <memberList department="programmer">
            <member id="john" name="John Doe" authority="3" />
            <member id="samchon" name="Jeongho Nam" authority="5" />
        </memberList>
    </parameter>
</invoke>
```

보내고자 하는 파라미터가 엔터티 오브젝트이거나 리스트 컨테이너의 형태일 때는, XML 포맷으로 변환하여 보내게 됩니다. 자세한 사항은 **Entity** 모듈을 참조해 주십시오.

```
virtual void replyData(shared_ptr<Invoke> invoke) override
{
    if (invoke->getListener() == "setDepartment")
    {
        string &master = invoke->at(0)->getValue<string>();

        MemberArray memberArray;
        memberArray.construct(invoke->at(2)->getvalueAsXML());

        setDepartment(master, memberArray);
    }
}
```

우리는 이와 같은 메시지 프로토콜의 표준화로 많은 이점을 누릴 수 있습니다.

그 중 가장 큰 이점은 그 어떤 네트워크, 아무리 거대하고 복잡한 네트워크 시스템을 만듦에 있어서도 네트워크 통신 기능부문을 중복 구현하지 않고, 오로지 네트워크 시스템 내의 각 객체들 간의 논리적인 연결관계에만 치중하여 구현할 수 있다는 것입니다.

마치 SW 클래스들을 다루듯 네트워크 객체들을 각각의 클래스로 손쉽게 표현할 수 있습니다. 이에 관한 자세한 내용은 **Interface** 부문에서 자세하게 다룹니다.

두 번재 이점은 바로 history log 를 관리함에 있습니다. 표준화된 메시지 구조는 곧 네트워킥 메시지 입출력 history 에 대한 정규화가 가능함을 의미하기도 합니다. 이를 통하여 각 네트워크 시스템의 메시지 전달 내역을 파악할 수도 있고, 더불어 각 네트워크 시스템의 성능을 측정하는 데에도 역시 용이합니다.

- **Invoke, InvokeParameter**

  삼촌 프레임워크의 표준 메시지 프로토콜을 표현할 수 있는 클래스입니다. 더불어 역으로 타 시스템으로부터 온 문자열 및 XML 오브젝트를 파싱하여, 표준 메시지 구조체로 변환시켜주기도 합니다.

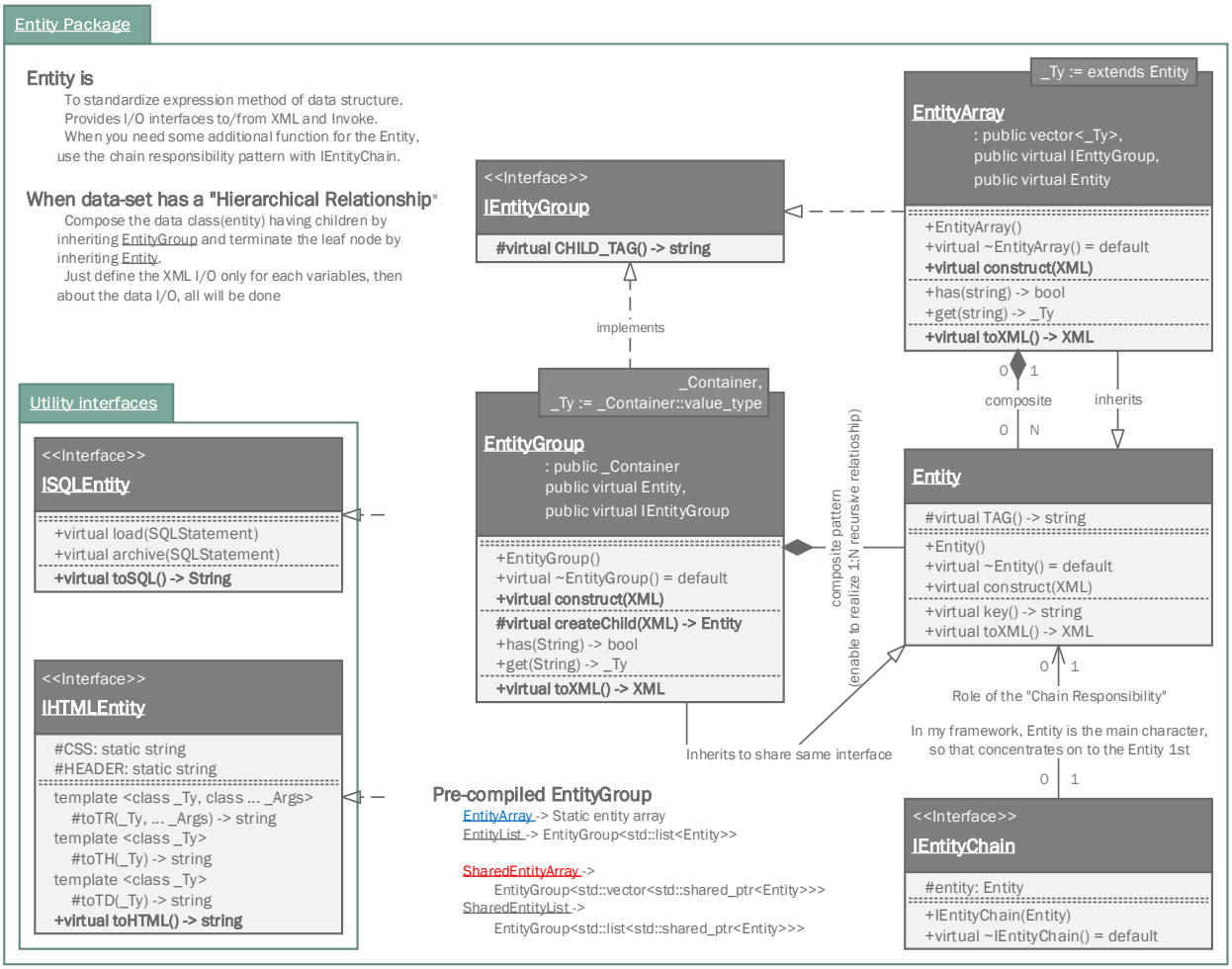
- **InvokeHistory**

  Slave 시스템에서 상위 Master 시스템으로 Invoke 메시지에 대한 내역 및 해당 메시지의 처리 시간을 보고하기 위해 설계된 오브젝트입니다. 삼촌 프레임워크의 분산 및 병렬처리 모듈에서 각 시스템의 성능지표를 측정하기 위해 사용되는 클래스입니다.

- **PRInvokeHistory, DRInvokeHistory**

  Slave 시스템으로부터 보고받은 InvokeHistory 를 Master 에서 처리할 때 쓰이는 객체입니다. 각각 분산처리 시스템 및 병렬처리 시스템의 Master 클래스에서 쓰이게 됩니다.

2.3.2 Entity



엔터티 모듈은 네트워크 통신에 쓰이는, 데이터 클래스(entity)의 표기법을 표준화하기 위해 설계된 모듈입니다. XML 로의 변환 및 역으로 XML 객체를 파싱하여 자신의 Entity 객체를 구성할 수 있는 메소드들을 제공합니다.

Entity 모듈을 이용하면, 네트워크 메시지를 주고 받음에 있어 통일된 데이터 표기법을 사용할 수 있게 됩니다. 따라서 사용자가 네트워크 통신에 있어서의 데이터의 출력 및 구성을 신경 쓸 필요가 없기에 매우 편리합니다.

하지만, XML 표기법은 상당한 오버헤드를 가집니다. 4 바이트면 표현이 가능한 integer 도 문자열로 변환이 되기에 자릿수만큼의 byte 가 필요하고, 각 데이터의 메타데이터를 표현함에 있어서도 많은 문자열과 byte 를 소모합니다. 네트워크 통신을 통한 신속한 데이터의 입출력이 중요한 엔터티의 경우, Entity 모듈이 아닌 **ByteArray** 클래스를 사용하십시오.

❖ **Entity**

단일 엔터티 객체를 표현할 때 쓰이는 클래스입니다.

- 현재의 Entity 가 XML 로 표현될 때, 쓰일 TAG 명, TAG() 메소드를 재정의 해 주십시오.
- 현재의 Entity 를 대표할 수 있는 식별자를 리턴하는 key() 메소드를 재정의 해 주십시오.
- XML 로부터 Entity 를 구성하는 construct() 메소드를 재정의 해 주십시오.
- 현재의 Entity 를 XML 오브젝트로 변환하는 toXML() 메소드를 재정의 해 주십시오.

❖ **IEntityGroup**

IEntityGroup 은 entity 객체를 담는 컨테이너에 쓰이는 인터페이스입니다. 하나의 Entity 오브젝트가 단일 엔터티 객체인지 아니면 컨테이너 셋인지 구분하는 데 유용하게 쓰입니다.

- if(dynamic_cast<IEntityGroup*>(&obj) != nullptr)
- 자식 entity 의 태그명, CHILD_TAG() 메소드를 재정의 해 주십시오.

❖ **EntityGroup**

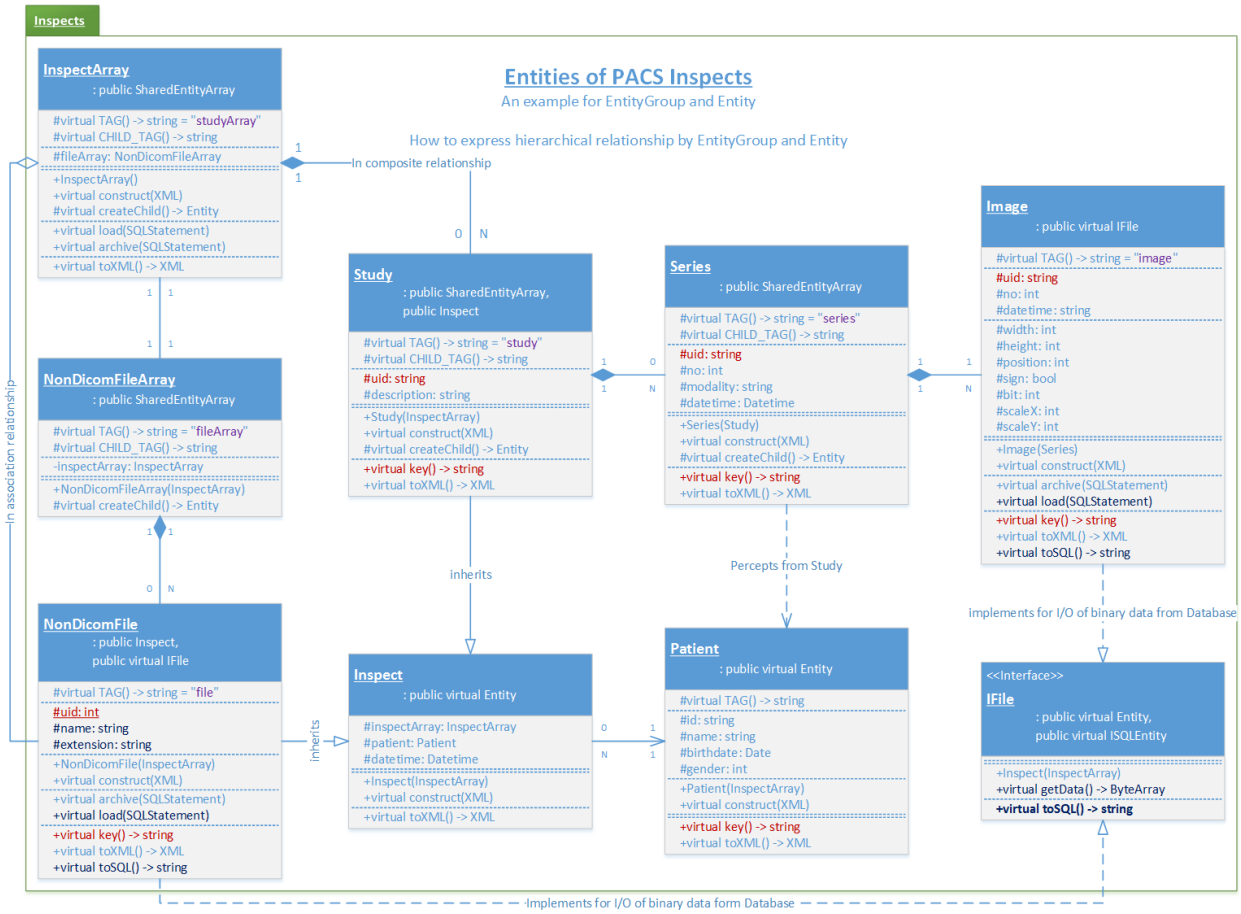
```
template
<
        typename _Container,
        typename _ETy = Entity, typename _Ty = _Container::value_type
>
class EntityGroup
```

계층 구조의 엔터티 객체를 표현할 때 쓰이는 템플릿 클래스로써, EntityGroup 도 entity 의 일종이며, 더불어 자식 entity 객체를 포인터의 형태로 가지고 있습니다. 1:N 의 계층 구조를 가지는 엔터티 관계를 표현하는 데 유용하며, 특히 트리구조(1:N 의 자가재귀관계)의 엔터티를 표현함에 있어서 탁월합니다 (Composite Pattern).

- 자식 entity 객체의 팩토리 메소드, createChild() 메소드를 재정의 해 주십시오.
- 자식 entity 의 태그명, CHILD_TAG() 메소드를 재정의 해 주십시오.
- **미리 정의된 EntityGroup 클래스**
  - UniqueEntityArray
    - 자식 entity 의 accessor 타입을 변경해주는 매크로가 존재합니다.
      - UNIQUE_ENTITY_ARRAY_ELEMENT_ACCESSOR_HEADER
      - UNIQUE_ENTITY_ARRAY_ELEMENT_ACCESSOR_BODY
  - UniqueEntityList
  - SharedEntityArray

- 자식 entity 의 accessor 타입을 변경해주는 매크로가 존재합니다.
  - SHARED_ENTITY_ARRAY_ELEMENT_ACCESSOR_HEADER
  - SHARED_ENTITY_ARRAY_ELEMENT_ACCESSOR_BODY

▪ SharedEntityList



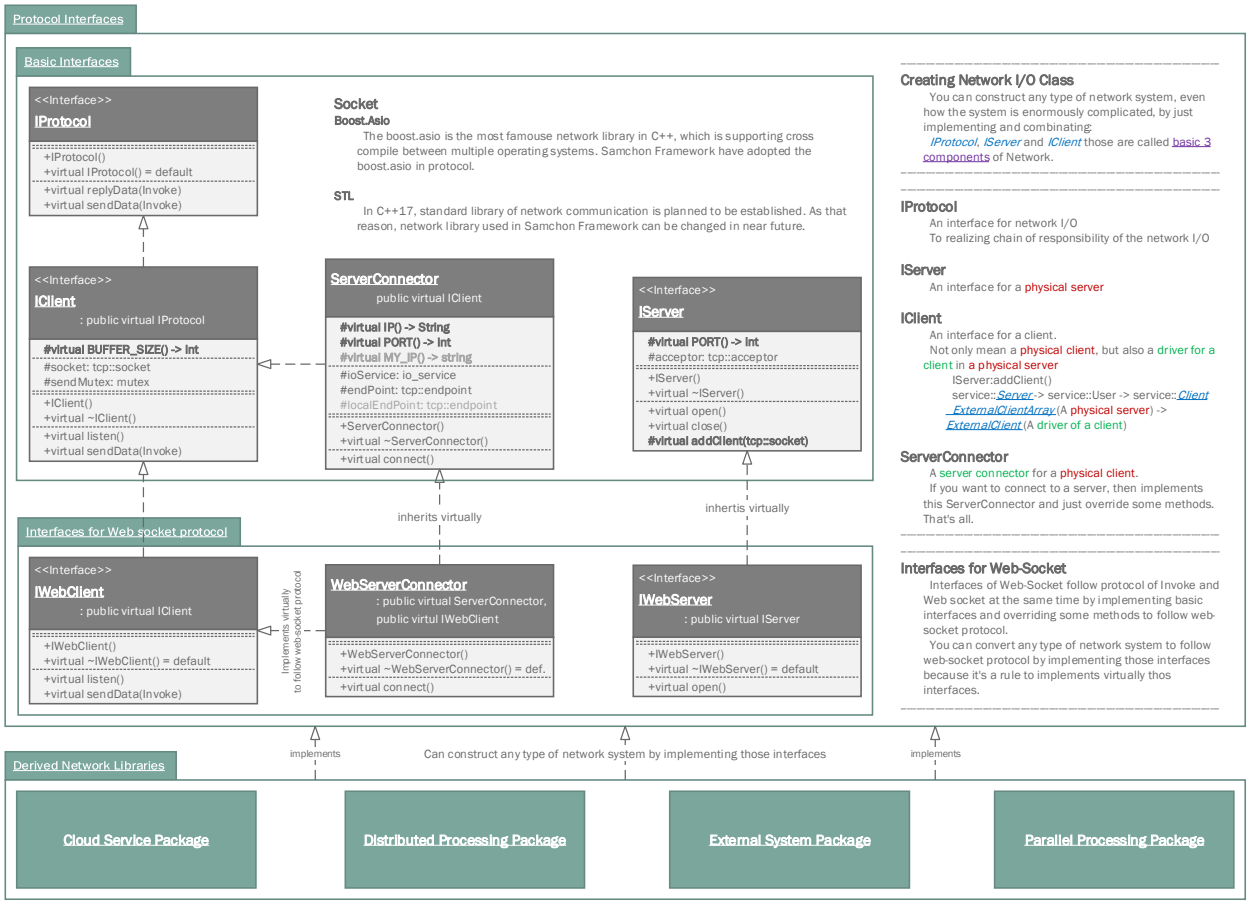
(위 클래스 다이어그램은, EntityGroup 과 Entity 를 사용하여 계층구조를 표현하는 사례입니다.)

❖ EntityArray

EntityGroup 은 자식 객체를 포인터로 가짐으로 인하여, 간접 참조의 형태를 띄기 때문에 iteration 이 다소 느릴 수 있습니다. EntityArray 는 이를 해결하기 위하여 자식 entity 를 직렬화된 배열의 형태로 가지는 컨테이너입니다.

하지만, 직렬화되었다는 말은, 각 자식 entity 에 할당되는 메모리의 양이 고정되었다는 의미로, EntityArray 에는 사전에 정의된 자식 entity 타입으로부터 파생된 entity 를 삽입할 수 없는 단점을 가지고 있기도 합니다.

2.3.3 Interface



삼촌 프레임워크에서는, 모든 종류의 네트워크 시스템을 단 3+1 개의 기초 인터페이스 (**IProtocol**, **IServer** 및 **IClient** + **ServerConnector**) 만을 가지고 구성할 수 있습니다. 만들고자 하는 네트워크 시스템이 클라우드 시스템이던, 복잡한 트리 구조와 수많은 slave 및 mediator(proxy)를 가진 분산처리 시스템이던 관계없이, 3+1 기초 인터페이스의 상속과 조합을 통하여 모두 구성이 가능합니다.

앞서의 **Invoke** 및 **Entity** 모듈 부문에서 서술되었듯이, 네트워크 메시지 프로토콜의 표준화와 데이터 표기법의 표준화로 인하여 도출될 수 있었던 3+1 기초 인터페이스로 인하여, 사용자는 각 네트워크 객체(개개의 서버 혹은 클라이언트)와의 통신에 대한 구현을 신경 쓰지 않고, 오로지 각 네트워크 객체들 간의 논리적인 역할에만 집중할 수 있게 됩니다.

마치 S/W 클래스를 설계해 나가듯이, 각 네트워크 객체들을 하나의 클래스로 여기시어 객체지향적인 관점에서 네트워크 시스템을 구현해 나가시면 됩니다.

- 기초 3+1 인터페이스의 상속은 모두 가상상속을 통해 구현하셔야 합니다.

❖ IProtocol

기초 3 인터페이스 중 하나로써 **Invoke** 메시지 체인에 관한 인터페이스입니다.

Chain of Responsibility 패턴을 사용, Invoke 메시지 전송 및 핸들링을 네트워크 드라이버나 혹은 그와 직-간접적으로 연결관계에 있는 객체에 전달할 수 있습니다.

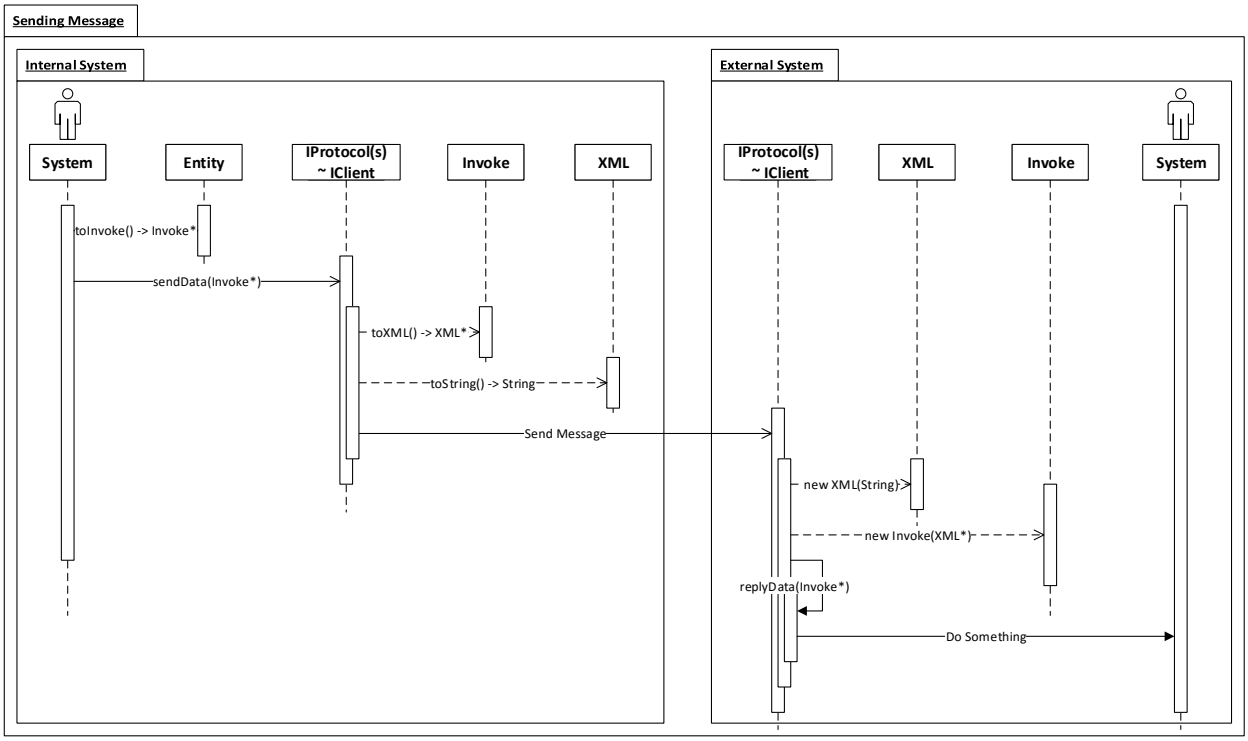
더불어 **IProtocol** 은 protocol 모듈 중, 가장 빈번하게 상속이 일어나는 객체이기도 합니다.

❖ IServer

기초 3 인터페이스 중 하나로써, 서버를 만들 때 사용하는 인터페이스입니다.

네트워크 시스템을 구현함에 있어, 본인 측의 시스템이 서버에 해당할 때, 이 IServer 를 상속하시면 됩니다.

- 열고자 하는 서버의 포트에 관한 메소드, PORT() 를 재정의 해 주십시오.
- 클라이언트의 접속에 대응하는, addClient() 메소드를 재정의 해 주십시오.



❖ **IClient**

기초 3 인터페이스 중 하나로써 클라이언트를 구현할 때 사용하는 인터페이스입니다.

하지만 **IServer** 와 달리, 반드시 본인 측의 시스템이 클라이언트일 때만 사용하는 인터페이스가 아닙니다. 서버 측에 접속한 (논리적) 클라이언트에 대한 객체를 구현할 때 역시 상속해야 할 인터페이스입니다.

- **IServer** 에 접속한 클라이언트에 대한 구현을 위한 용도로 **IClient** 를 상속하셨다면, **IClient** 의 파생 클래스의 생성자는, **IServer::**addClient() 와 페어를 이루게 됩니다.
- 외부 시스템에서 온 **Invoke** 메시지를 처리하는 함수, replyData() 를 재정의 해 주십시오.

❖ **ServerConnector**

기초 3+1 인터페이스 중 +1 에 해당하는 클래스로써, 문자 그대로 서버 접속기입니다. **ServerConnector** 는 **IClient** 를 상속하여 구현되었으며, 본인의 시스템이 외부 서버에 접속해야 할 때 상속하시면 됩니다.

혹여 접속하게 될 서버가 Samchon Framework 의 Invoke 메시지 프로토콜을 따르지 않거나 별도의 handshake 과정을 가진다면, 해당 서버의 프로토콜 및 handshake 과정에 대한 인터페이스를 **IClient** 를 상속하여 만드십시오. 그리고 **ServerConnector** 의 파생 클래스가 다시금 해당 인터페이스를 상속하면, 메시지 프로토콜이 바뀌게 됩니다.

- ServerConnector 는 **IClient** 를 상속하였기 때문에 replyData() 를 재정의 하셔야 합니다.
- 접속하고자 하는 서버의 주소에 관한 메소드 getIP(), getPort() 를 재정의 해 주십시오.

❖ **IWebServer, IWebClient**

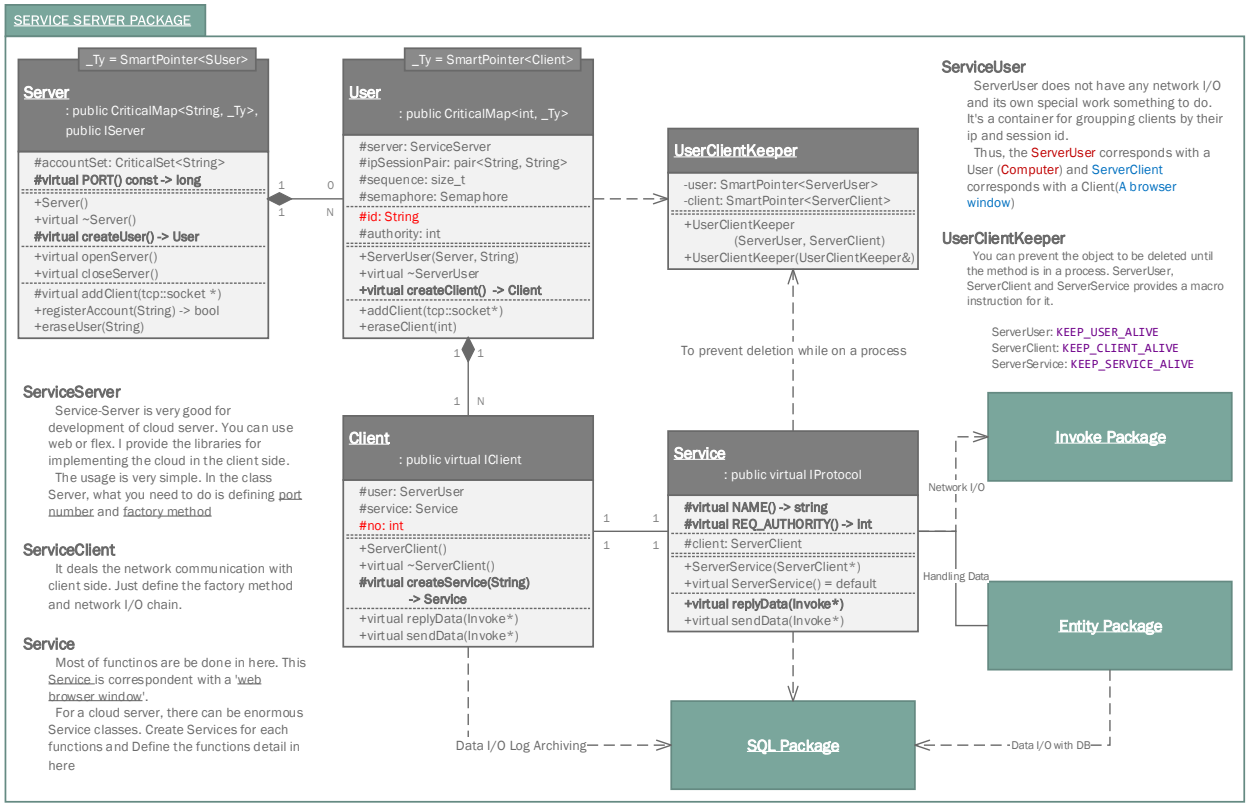
IServer 와 **IClient** 객체가 웹소켓의 프로토콜을 따라야 할 때 상속하게 되는 인터페이스입니다. 삼촌 프레임워크의 프로토콜을 따르는 그 어떠한 네트워크 시스템이라 할 지라도, 용도에 맞게끔 **IWebServer** 와 **IWebClient** 를 가상상속하게 되면, 프로토콜이 web-socket 을 따르게 됩니다.

❖ **WebServerConnector**

기존의 ServerConnector 에 IWebClient 를 가상상속시켜 웹소켓 프로토콜을 따르는 서버에 접속할 수 있게 반든 웹소켓 서버 컨넥터입니다.

여기까지 오셨다면, IServer 나 IClient 를 가상상속하여 프로토콜을 바꾼다는 것이 어떤 의미인지 이해가 되셨을 것입니다.

2.3.4 Cloud service



서비스 모듈은 클라우드 서버를 구현할 때 쓰이는 모듈입니다.

문자 그대로 서버를 의미하는 Server 와 해당 서버에 접속한 클라이언트를 지칭하는 Client 외에, 논리적인 단위로써, 사용자를 식별케 해 주는 User 및 클라이언트가 접속해 수행하고자 하는 작업을 의미하는 Service 객체가 있습니다.

- **Server:**

  문자 그대로 서버, 클라우드 서버를 의미합니다.

- **User**

  여러 클라이언트를 거느리는 사용자입니다. 하나의 유저가 여러 개의 브라우저를 띄울 수 있기에 이를 표현하기 위해 설계한 논리적 개념의 클래스입니다.

  User 객체는 sessionID 로 식별됩니다. 또한, 이 sessionID 는 브라우저(실제 물리적 클라이언트)에 통보되어, 브라우저는 이 sessionID 를 쿠키에 저장하고 매번 서버에 자신의 sessionID 를 알림으로써, 자신(Client)이 어느 User 에 속하는 지를 알리게 됩니다.

❖ **Client:**

서버에 접속한 클라이언트를 의미합니다.

Client 는 User 와 다대 1 의 관계를 맺고 있으며, 한 개의 Service 를 지닙니다.

❖ **Service:**

사용자가 수행하고자 하는 작업인 서비스를 대표하는 객체입니다.

Client 와 1:1 관계를 지니며, 클라우드 서버가 수행해야 할 본연의 기능(서비스)에 대해 구현되어있는 클래스입니다.

❖ **ServiceRole:**

서비스에 수행될 역할을 의미하는 논리 객체입니다.

서비스 간 세부 기능에 대해 변경사항 잦을 때, ServiceRole 을 이용하면 이를 유연하게 대처할 수 있습니다. Proxy pattern 의 logical proxy 에 해당합니다.

❖ **IPUserPair**

각각의 User 에게 세션 아이디를 발급하기 위한 클래스입니다.

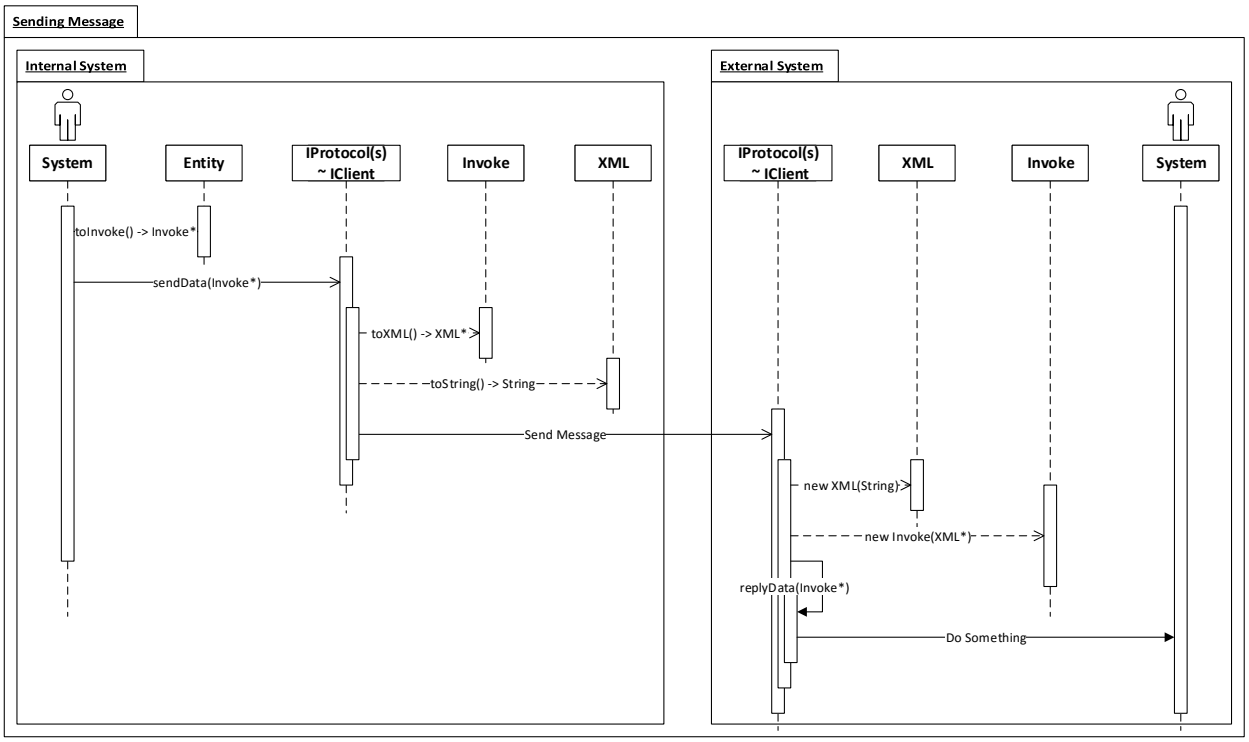
하지만, 때로는 세션 아이디를 변조하여 불법적으로 권한을 가지려는 사람도 있기 마련입니다. 이에 세션 아이디 발급자는 세션 아이디를 ip 와 매칭함으로써, 불법적인 세션 위조를 방지합니다.

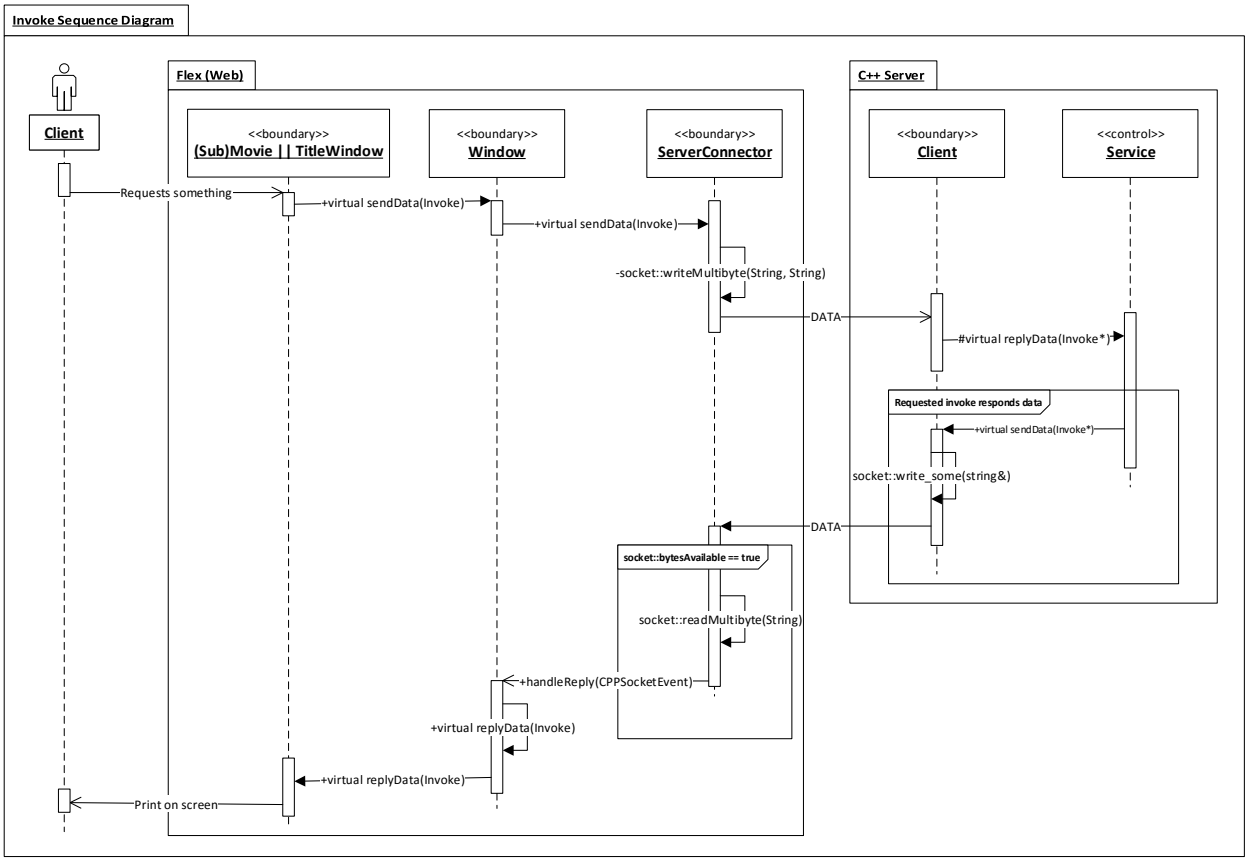
❖ **UserClientKeeper**

클라이언트가 접속을 끊어도, 이미 내린 명령에 대해서 작업이 계속 수행되어야 하는 경우가 종종 있습니다. 이럴 때, 연관된 User, Client 및 Service 객체의 소멸을 막기 위해 설계된 클래스입니다.

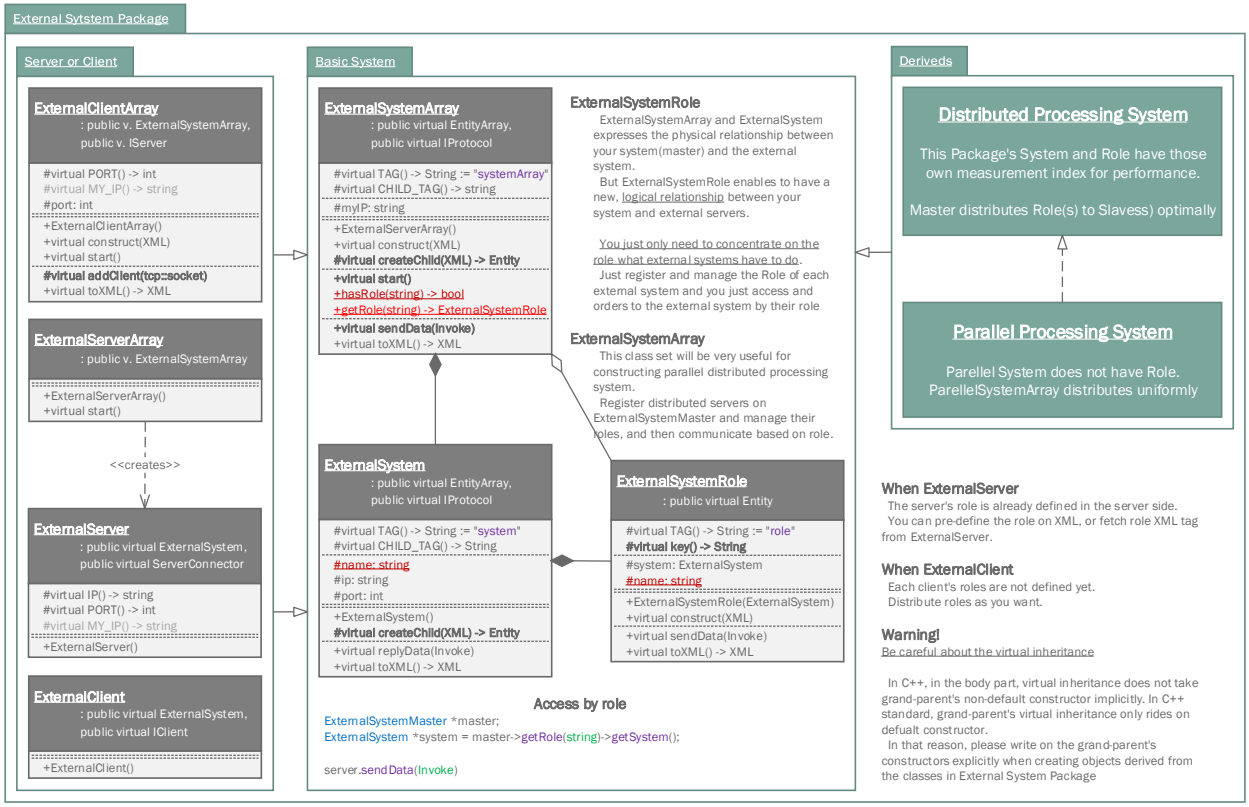
사용자가 직접 생성할 수는 없으되, User, Client 및 Service 클래스에서 각각 매크로 명령어를 사용하여 이 기능을 수행할 수 있습니다.

- **KEEP_USER_ALIVE**
- **KEEP_CLIENT_ALIVE**
- **KEEP_SERVICE_ALIVE**


Sending Message
Internal System
External System
System
Entity
IProtocol(s)
~ IClient
Invoke
XML
IProtocol(s)
~ IClient
XML
Invoke
System
toInvoke() -> Invoke*
sendData(Invoke*)
toXML() -> XML*
toString() -> String
Send Message
new XML(String)
new Invoke(XML*)
replyData(Invoke*)
Do Something


Invoke Sequence Diagram
Flex (Web)
C++ Server
Client
<<boundary>>
(Sub)Movie || TitleWindow
<<boundary>>
Window
<<boundary>>
ServerConnector
<<boundary>>
Client
<<control>>
Service
Requests something
+virtual sendData(Invoke)
+virtual sendData(Invoke)
-socket::writeMultibyte(String, String)
DATA
#virtual replyData(Invoke*)
Requested invoke responds data
+virtual sendData(Invoke*)
socket::write_some(string&)
DATA
socket::bytesAvailable == true
socket::readMultibyte(String)
+handleReply(CPPSocketEvent)
+virtual replyData(Invoke)
+virtual replyData(Invoke)
Print on screen

2.3.5 External System



외부 시스템 모듈은, 문자 그대로, 외부 네트워크 시스템과의 연동을 위해 설계된 추상 객체들입니다.

외부 시스템과의 네트워크 통신 부문에 대한 구현을 신경 쓰지 않고, 오로지 외부 시스템과 나의 시스템이 수행해야 할 역할, 그 역할 자체에만 집중할 수 있도록 해 줍니다.

심지어는, 어떠한 외부 시스템이 연결되어 있으며, 해당 외부 시스템이 무슨 역할을 수행한다라는 개념조차도 무시하고, 오로지 '어떠한 역할들이 있다' 라는 사실만을 인지한 채로 작업을 수행할 수도 있습니다.

외부 시스템 관리자인 ExternalSystemArray 가 외부 시스템인 ExternalSystem 을 통해서 데이터를 송수신 하는 것이 아닌, ExternalSystemRole 을 통하여 작업을 간접적으로 수행하는 것이 바로 그 방법인데, 이를 Proxy pattern 이라고 부릅니다.

- ❖ **ExternalSystemArray**

  외부 시스템들에 대한 상호작용을 총괄하는 관리자 클래스입니다.

  - ExternalServerArray: 외부 시스템들이 서버일 때. 현 시스템은 클라이언트
  - ExternalClientArray: 외부 시스템들이 클라이언트일 때, 관리자는 서버의 역할을 수행

❖ **ExternalSystem**

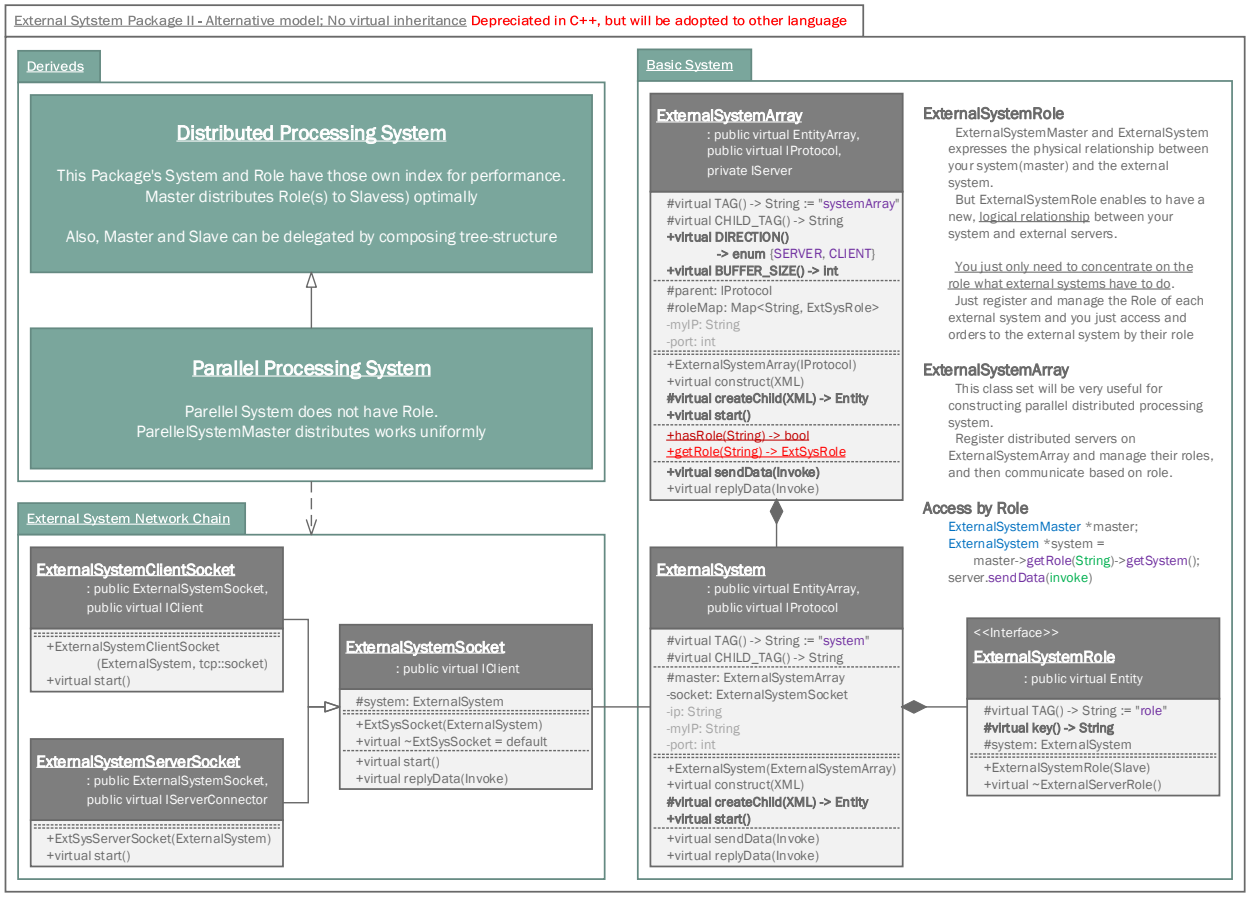
외부 시스템과의 네트워크 통신을 담당하는 클래스

- ExternalServer: 외부 시스템이 서버일 때
- ExternalClient: 외부 시스템이 클라이언트일 때

❖ **ExternalSystemRole**

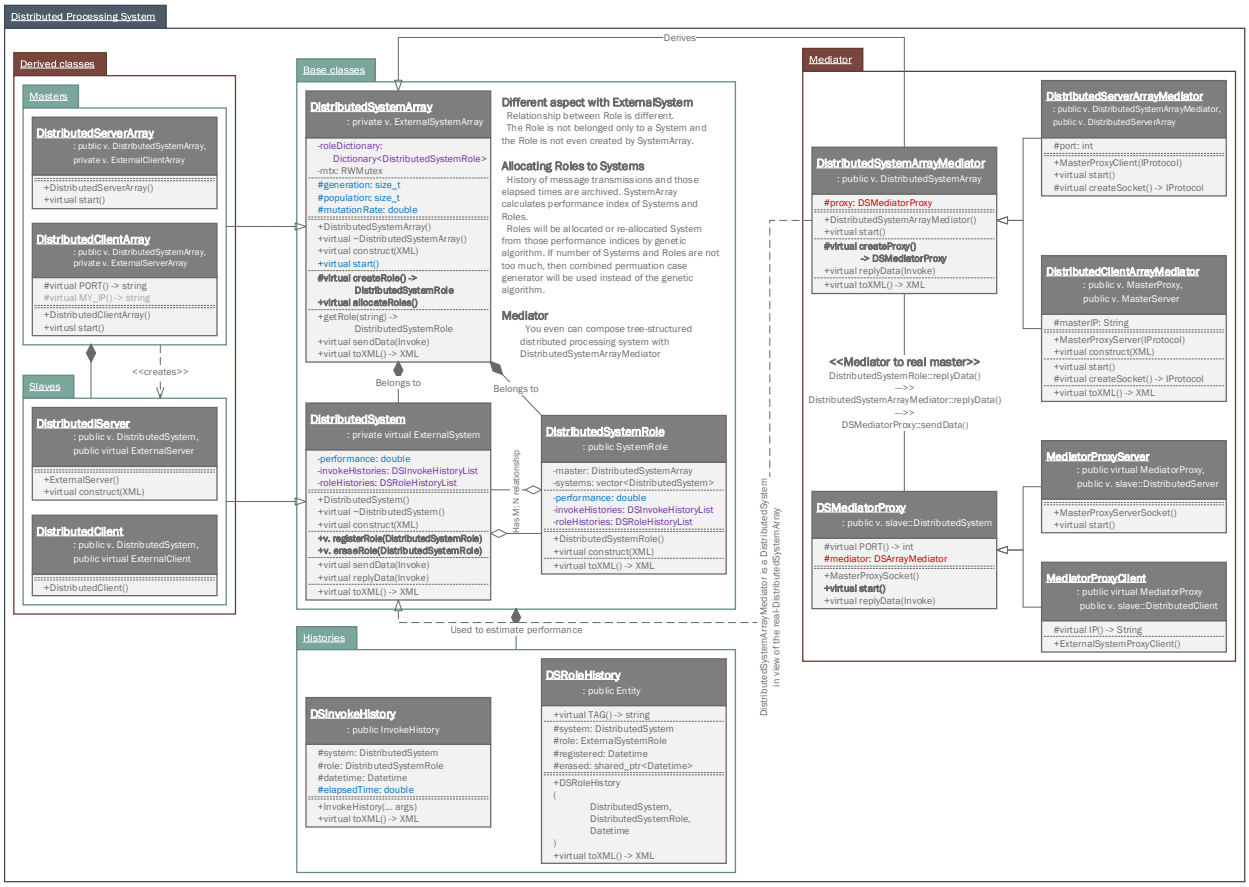
외부 시스템에 할당되는, 해당 외부 시스템이 수행해야 할 작업에 대한 명세 클래스입니다.

ExternalSystemArray 에서 ExternalSystem 이 아닌 이 ExternalSystemRole 을 통하여 간접적으로 데이터를 송수신 하시게끔 코드를 짜신다면, 외부 시스템의 명세에 대해 사용자가 알 필요도 없게 됩니다. 이와 같은 경우를 Proxy pattern 이라고 합니다.



- **가상 및 이중상속을 지원하지 않는 언어들을 위한 external system 모듈 설계도**

2.3.6 Distributed Processing System



Master 의 Distributed system 모듈은, 역할 기반의 분산처리시스템을 구현하기 위한 모듈입니다.

다수의 역할(**DistributedSystemRole**, 이하 **Role**)을 각 슬레이브 시스템(**DistributedSystem**, 이하 **Slave**)의 성능에 맞추어, 적합한 수준의 성능 분산이 이루어지도록 할당해 주어, 최적의 분산처리를 이룰 수 있도록 해 줍니다.

각 **System** 의 성능 및 **Role** 의 요구 성능치는, 각기 자신에게 주어지는 Invoke 메시지의 처리 소요시간(elapsed time)에 근거하여 평가됩니다. External system 모듈과 다르게 Distributed system 모듈은 **Role** 이 특정 **Slave** 시스템에 고정되어있지 않고, 성능지표를 통해 수시로 다른 **Slave** 로 할당됩니다. 더불어 **Role** 은 단 하나의 **Slave** 가 아닌, 여러 **Slave** 에 걸쳐 할당될 수 있습니다.

- **InvokeHistory**: Slave 측에서의 소요시간 보고
- **DSInvokeHistory**: Master 측에서의 소요시간 집계

만일, **DistributedSystemArray** 가 단 하나의 Role 만을 가지거나, 혹은 단 하나의 Role 도 가지지 않는다면, 이 때는 모든 작업이 각 Slave 의 성능 지표 및 유휴 상태에 따라 고르게 분산되게 됩니다.

❖ **DistributedSystemArray**

외부의 (Slave) System 과 Role 에 대한 상호작용을 총괄하는 관리자 클래스입니다.

ExternalSystemArray 를 상속하여 만들어졌으되, ExternalSystemArray 와는 달리, Role 과의 관계가 aggregation 이 아닌 composition 으로써 System 뿐 아니라 Role 또한 자신이 직접 관리합니다. Role 할당 역시 이 DistributedSystemArray 가 관장하게 됩니다.

수시로 System 과 Role 의 성능지표를 측정하여, 성능 분산(deviation)의 정도가 특정 기준치(표준편차, standard deviation) 치를 넘어섰을 때, 각 System 의 Role 을 회수하여 다시금 최적의 분산이 이루어지도록 배치합니다.

- DistributedServerArray: 외부 분산처리시스템이 서버. 현 시스템은 클라이언트
- DistributedClientArray: 외부 분산처리시스템이 클라이언트. 현 시스템은 서버

❖ **DistributedSystem**

외부 분산처리 시스템과의 네트워크 통신을 담당하는 클래스입니다.

ExternalSystem 을 상속하여 만들어졌으며, ExternalSystem 에 더하여 별도의 Invoke 메시지 처리내역과 성능지표 및 Role 할당내역을 가지고 있습니다. 또한, 부모 클래스인 ExternalSystem 과는 다르게, DistributedSystem 은 Role 과 composition 이 아닌 aggregation 의 관계를 가지고 있습니다.

즉, DistributedSystemRole 은 DistributedSystem 에 종속되지 않습니다. 더불어 Role 수시로 소속 System 이 변할 수 있으며, 하나의 System 이 아닌 여러 System 에 할당될 수도 있습니다.

- DistributedServer: 외부 분산처리 시스템이 서버
- DistributedClient: 외부 분산처리 시스템이 클라이언트

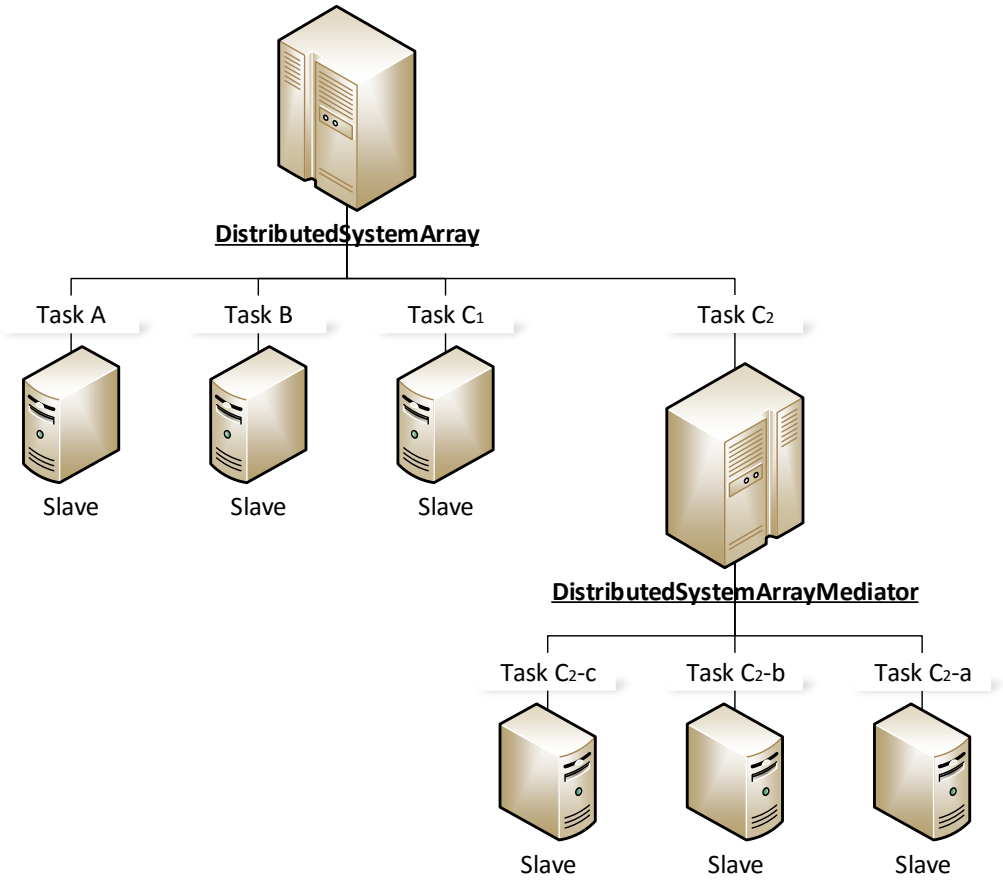
❖ **DistributedSystemRole**

분산처리 시스템에 할당되는, 해당 분산처리 시스템이 수행해야 할 작업에 대한 명세입니다.

ExternalSystemRole 을 상속하여 만들어졌으되, 별도의 Invoke 메시지 처리내역과 성능지표 및 System 할당내역을 가지고 있습니다. 또한, 부모 클래스인 ExternalSystemRole 과 다르게, System 과는 aggregation 관계를 가져 종속되지 아니하며, 더불어 여러 System 에 걸쳐 할당될 수도 있습니다.

DistributedSystemRole 이 여러 System 에 더불어 할당된 경우에는, 각 System 의 성능지표와 유휴 작업상태를 고려하여 가장 빠른 처리시간이 기대되는 System 에 Invoke 메시지를 보내게 됩니다.

❖ **DistributedSystemArrayMediator**



현 시스템이 트리 구조의 분산처리 시스템 중, 중간 노드에 해당할 때 사용하는 클래스입니다. 즉, 상위의 시스템 입장에서는 Slave 에 해당하며, 하부로도 다수의 Slave 를 거느리는 Master 일 때 사용합니다.

현 시스템을 기준으로 상위의 Master 와 하부의 Slave 사이를 중개해주는 중개자(Mediator, proxy) 의 일종으로써, DistributedSystemArray 을 상속함으로써 Master 의 역할을 지니며 DistributedSlaveSystem 을 상속하여 만들어진 DSArrayMediatorProxy 를 지님으로써 Slave 의 역할 또한 겸합니다.

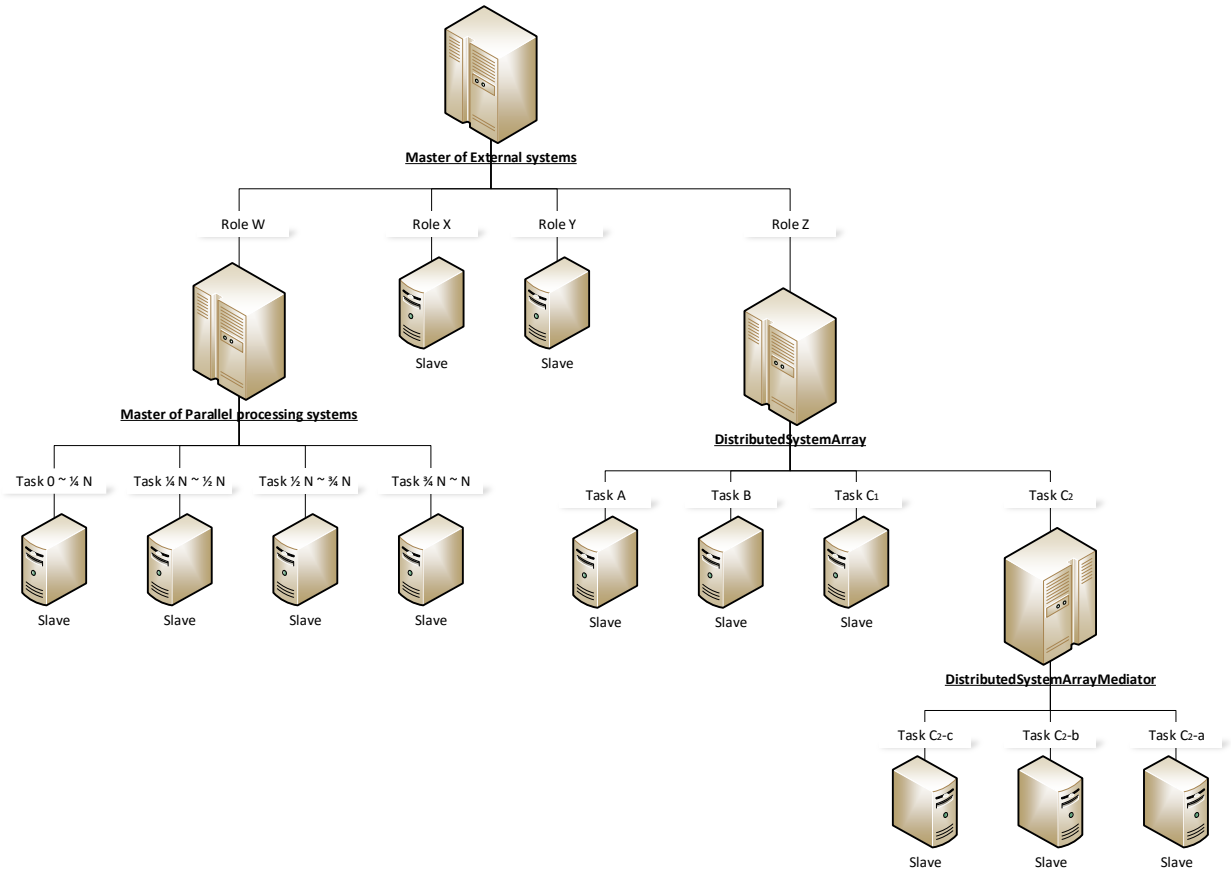
- **DistributedServerArrayMediator**
  - Master 로써: 하부 시스템이 서버, 현 시스템은 클라이언트
  - Slave 로써: 현 시스템이 서버, 상위 Master 시스템은 클라이언트
- **DistributedClientArrayMediator:**
  - Master 로써: 하부 시스템이 클라이언트. 현 시스템은 서버
  - Slave 로써: 현 시스템이 클라이언트, 상위 Master 시스템은 서버

❖ **DSArrayMediatorProxy**

상위 Master 와 통신하기 위한, DistributedSystemArrayMediator 에서 사용하는 객체입니다.

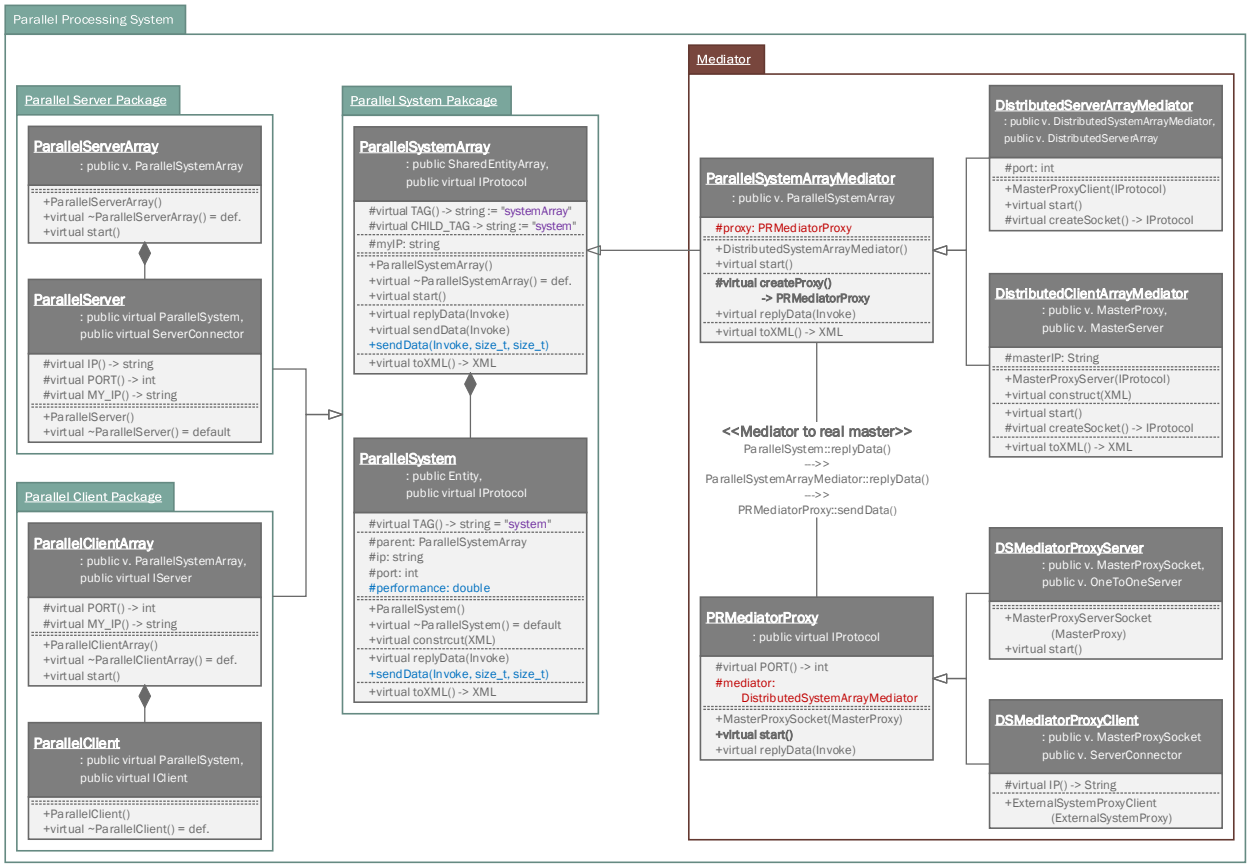
상위 Master 와 현 시스템 기준 하부 Slave 시스템들 간의 proxy 역할을 합니다.

- **DSMediatorProxyServer**
  - Slave 관점에서의 현 시스템이 Server
  - DistributedServerArrayMediator 에서 사용
- **DSMediatorProxyClient**
  - Slave 관점에서의 현 시스템이 Client
  - DistributedClientArrayMediator 에서 사용



- **External system 모듈 및 파생 모듈들의 극적인 활용 사례**

2.3.7 Parallel Processing System



Master 의 Parallel system 모듈은 각 작업을 특정한 수량으로 쪼개어 각 네트워크 시스템 단위로 병렬처리를 하고자 설계된 모듈입니다. Open-MP 의 #pragma omp parallel for 을 네트워크 단위에서 구현했다고 보시면 됩니다.

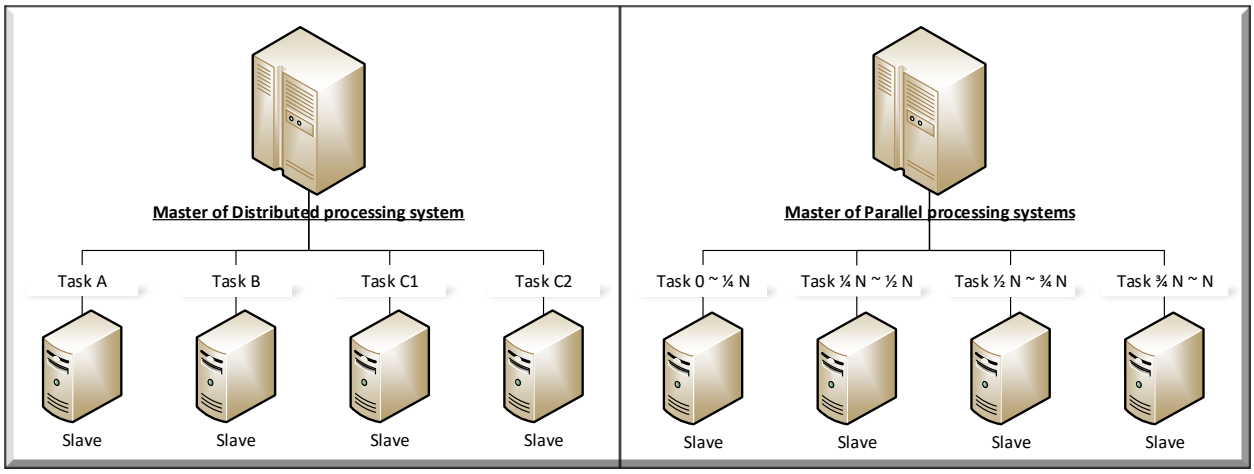
마스터(ParallelSystemArray, 이하 Master)는 각 슬레이브 시스템(ParallelSystem, 이하 Slave)의 성능을 평가하여 각 Slave 별로 적절한 수량의 작업을 나눠주어 병렬처리가 이루어지도록 합니다. 각 Slave 의 성능은, 주어진 작업량 대비 소요시간에 근거하여 평가됩니다.

Parallel system 모듈은 특히, 수학이나 통계와 같은 계산작업이 매우 거대하여 여러 컴퓨터를 두고 네트워크를 통하여 이를 병렬처리를 하고자 할 때 매우 유용합니다.

- **InvokeHistory**: Slave 측에서의 소요시간 보고
- **PSInvokeHistory**: Master 측에서의 소요시간 집계

Parallel system 모듈은 external system 모듈을 상속하여 제작되었지만, 병렬처리에서 각 시스템은 모두 수행하는 역할이 같기에 따로이 Role 을 가질 필요가 없습니다. 또한, distributed system 처럼 각 시스템의 성능을 평가하여 지표로써 기록하지만, 그처럼 복잡한 구조를 가지지도 않습니다.

- **Distributed system 모듈과 Parallel system 모듈의 차이점 비교**



❖ **ParallelSystemArray**

외부의 병렬처리 시스템 드라이버(ParallelSystem)을 관리하는 클래스입니다.

ExternalSystemArray 를 상속하여 만들어졌으며. ParallelSystemArray 는 여기에 한 작업(Invoke with quantity)에 대해 segment 별로 나누어 병렬처리를 맡기는 별도의 sendData()[7] 메소드가 정의되어 있습니다.

위 메소드를 호출하면, 각 병렬처리 시스템(ParallelSystem)의 성능지표를 토대로, 각 시스템에 적절한 수량만큼의 작업을 분배(segmentation), 최적의 병렬처리를 가능하게끔 해 줍니다.

- ParallelServerArray: 외부 병렬처리 시스템이 서버. 현 시스템은 클라이언트
- ParallelClientArray: 외부 병렬처리 시스템이 클라이언트. 현 시스템은 서버

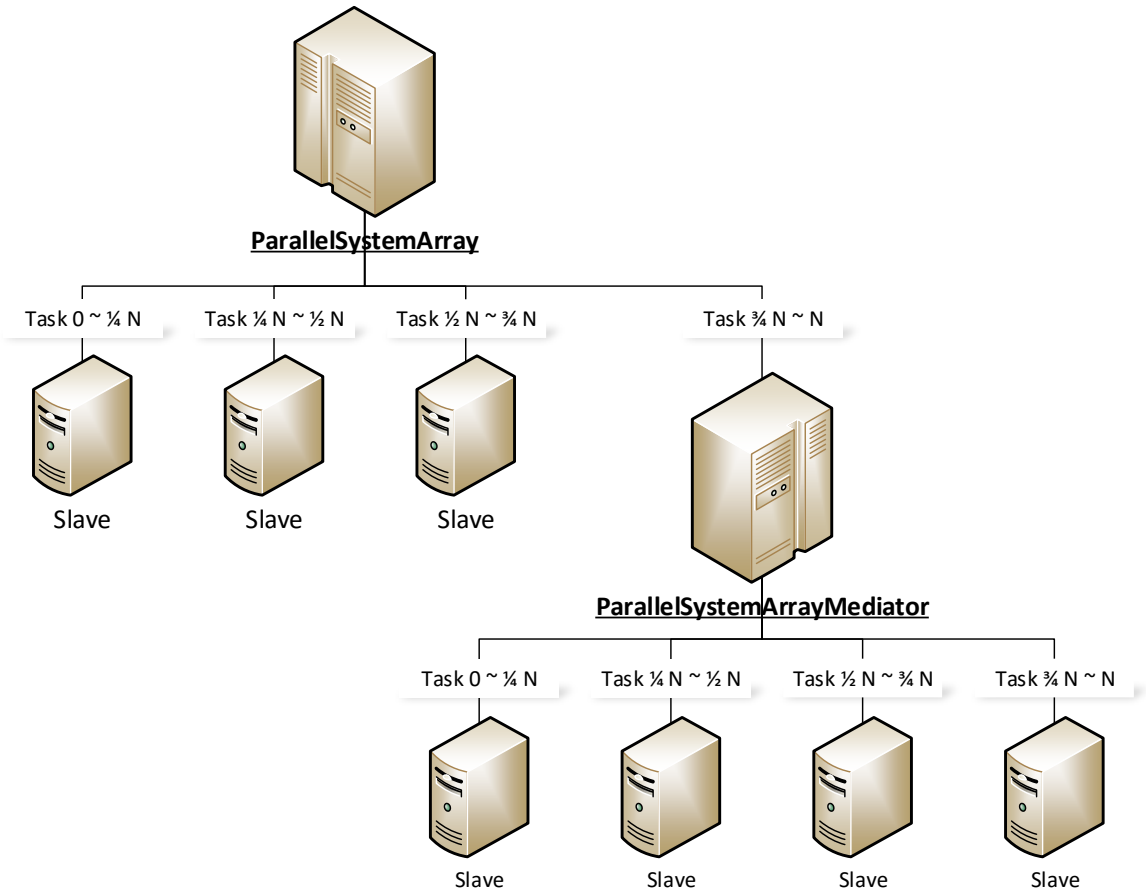
❖ **ParallelSystem**

외부 병렬처리 시스템과의 네트워크 통신을 담당하는 클래스입니다.

ExternalSystem 을 상속하여 만들어졌으며, ExternalSystem 에 더하여 별도의 Invoke 메시지 처리내역과 성능지표를 가지고 있습니다. 또한, ParallelSystem 은 동일한 작업에 대해 segment 별로 나누어 병렬처리를 하기 위한 목적으로 제작되었기에, 모든 ParallelSystem 은 모두 같은 역할을 갖습니다. 때문에 각 ParallelSystem 에 별도의 Role(ExternalSystemRole)이 정의될 필요는 없습니다.

- ParallelServer: 외부 병렬처리 시스템이 서버
- ParallelClient: 외부 병렬처리 시스템이 클라이언트

[7] ParallelSystemArray::sendData(Invoke, size_t, size_t)



❖ **ParallelSystemArrayMediator**

현 시스템이 트리 구조의 분산처리 시스템 중, 중간 노드에 해당할 때 사용하는 클래스입니다. 즉, 상위의 시스템 입장에서는 Slave 에 해당하며, 하부로도 다수의 Slave 를 거느리는 Master 일 때 사용합니다.

현 시스템을 기준으로 상위의 Master 와 하부의 Slave 사이를 중개해주는 중개자(Mediator, proxy) 의 일종으로써, ParallelSystemarray 을 상속함으로써 Master 의 역할을 지니며 ParallelSlaveSystem 을 상속하여 만들어진 PSArrayMediatorProxy 를 지님으로써 Slave 의 역할 또한 겸합니다.

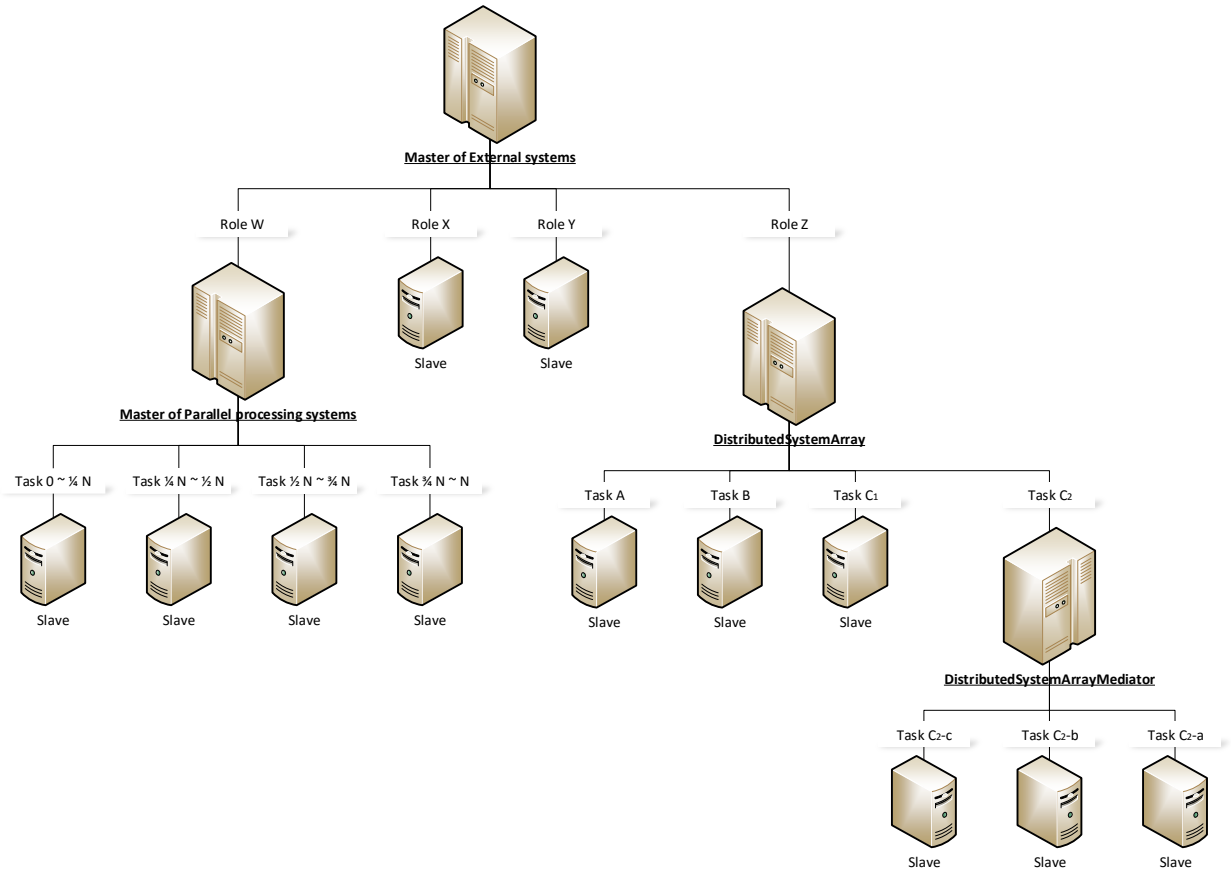
- **DistributedServerArrayMediator**
  - Master 로써: 하부 시스템이 서버, 현 시스템은 클라이언트
  - Slave 로써: 현 시스템이 서버, 상위 Master 시스템은 클라이언트
- **DistributedClientArrayMediator:**
  - Master 로써: 하부 시스템이 클라이언트. 현 시스템은 서버
  - Slave 로써: 현 시스템이 클라이언트, 상위 Master 시스템은 서버

❖ **PSArrayMediatorProxy**

상위 Master 와 통신하기 위한, ParallelSystemArrayMediator 에서 사용하는 객체입니다.

상위 Master 와 현 시스템 기준 하부 Slave 시스템들 간의 proxy 역할을 합니다.

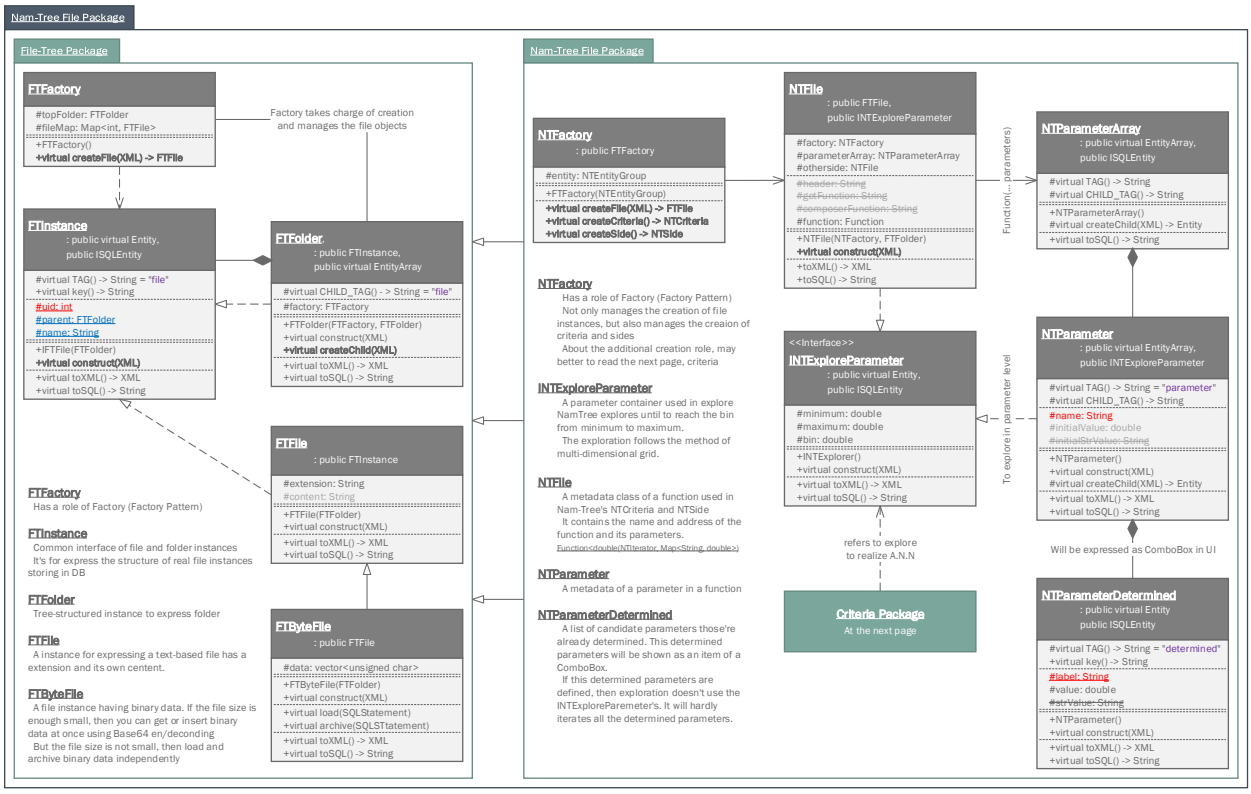
- **PSMediatorProxyServer**
  - Slave 관점에서의 현 시스템이 Server
  - ParallelServerArrayMediator 에서 사용
- **PSMediatorProxyClient**
  - Slave 관점에서의 현 시스템이 Client
  - ParallelClientArrayMediator 에서 사용



2.4 Nam-Tree

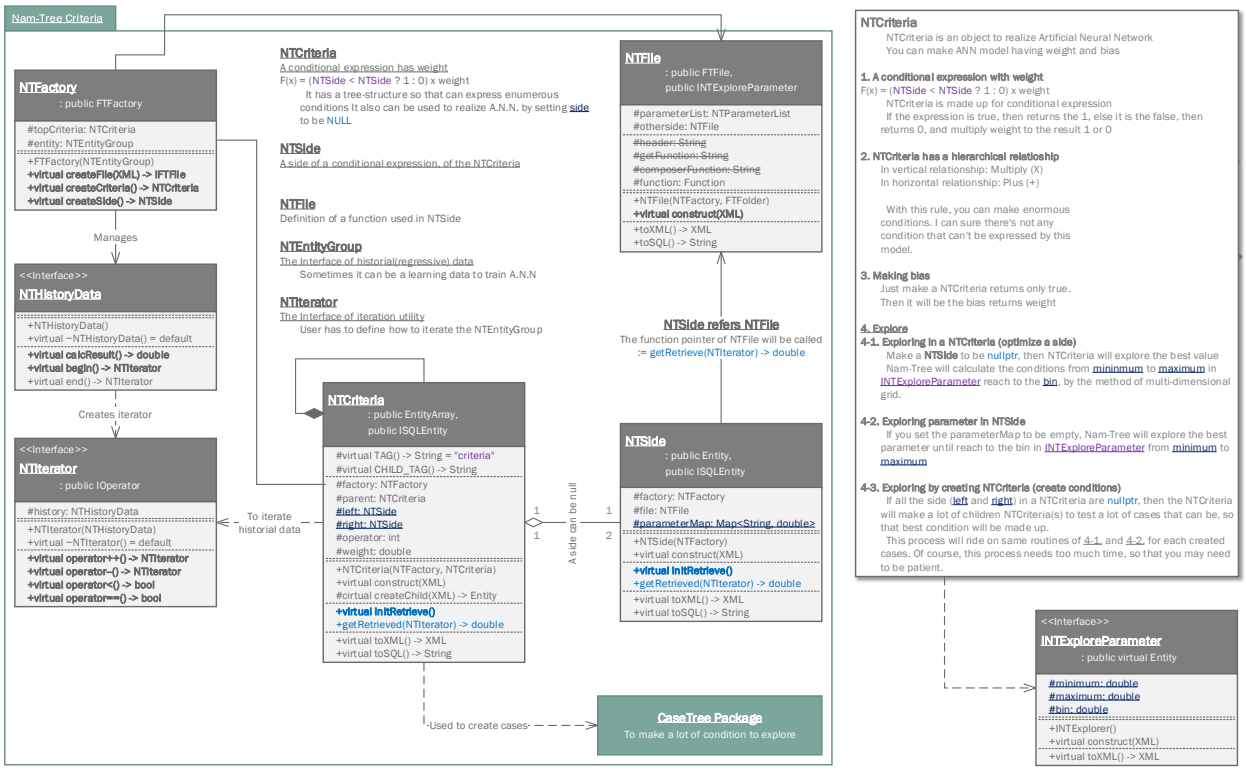
2.4.1 Conception

2.4.2 File instances



- ❖ NTFactory
- ❖ NTFile
- ❖ NTParameter
- ❖ NTParameterDetermined
- ❖ INTExploreParameter

2.4.3 Criteria



NTCriteria

NTSide

NTIterator

NTHistory

3 JS guidance

3.1 Common

3.1.1 Outline

3.1.2 Protocol

3.1.3 Movie

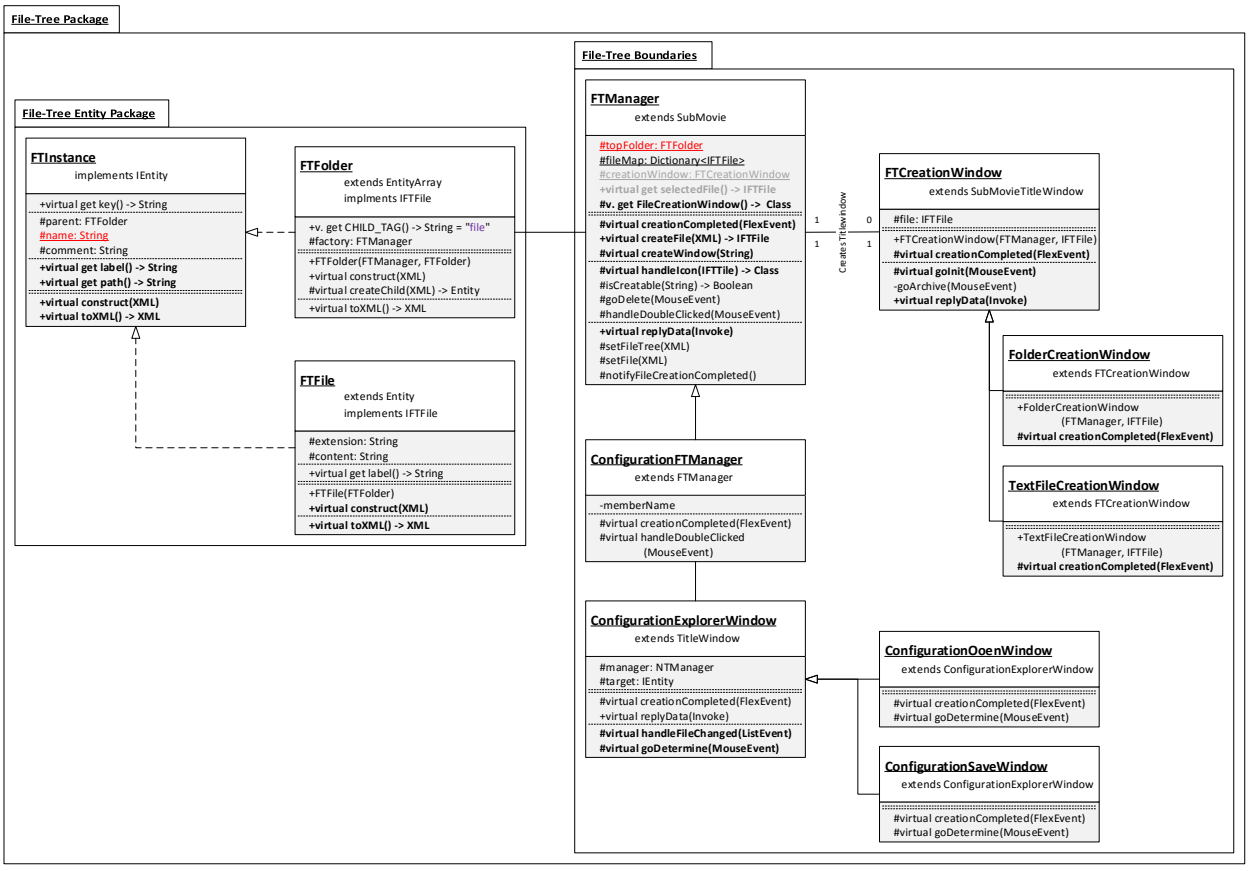
3.2 TypeScript

3.2.1 Library

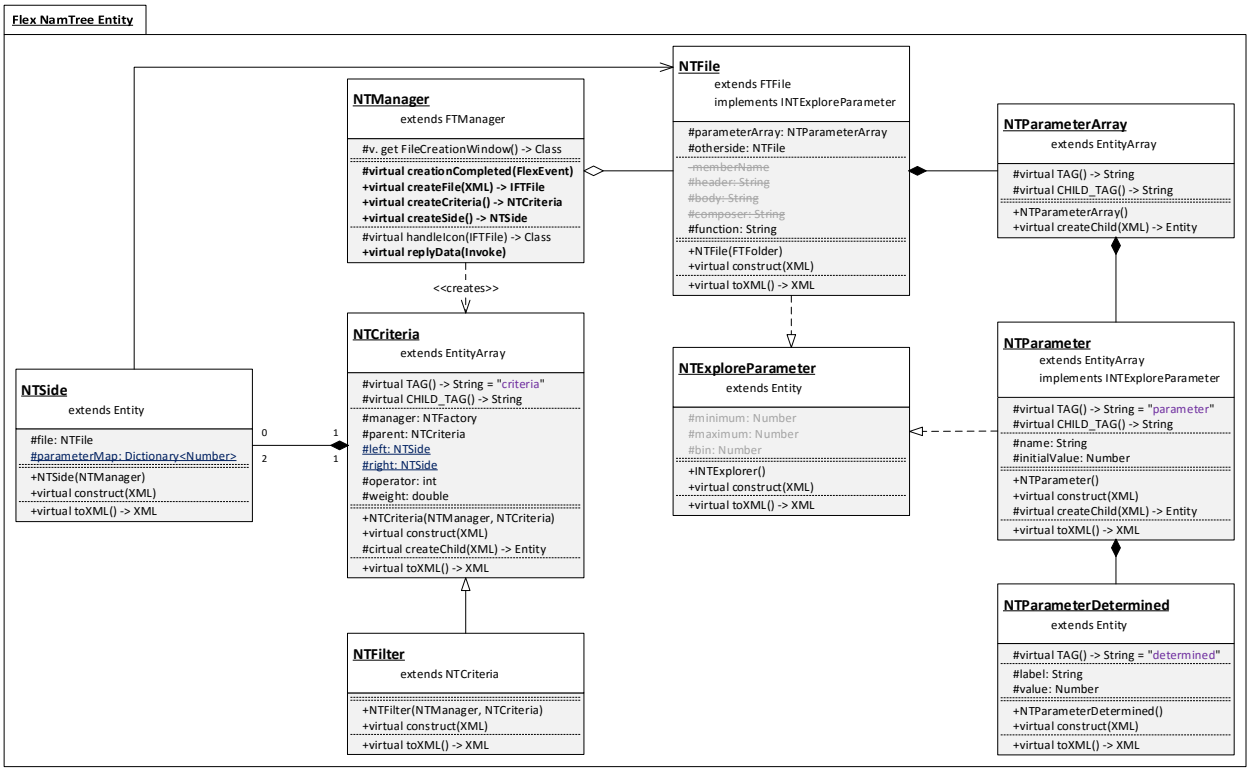
- **Like STL**
  - Vector
  - Map
  - MapIterator
- **XML**
  -

3.3 Flex

3.3.1 File-Tree

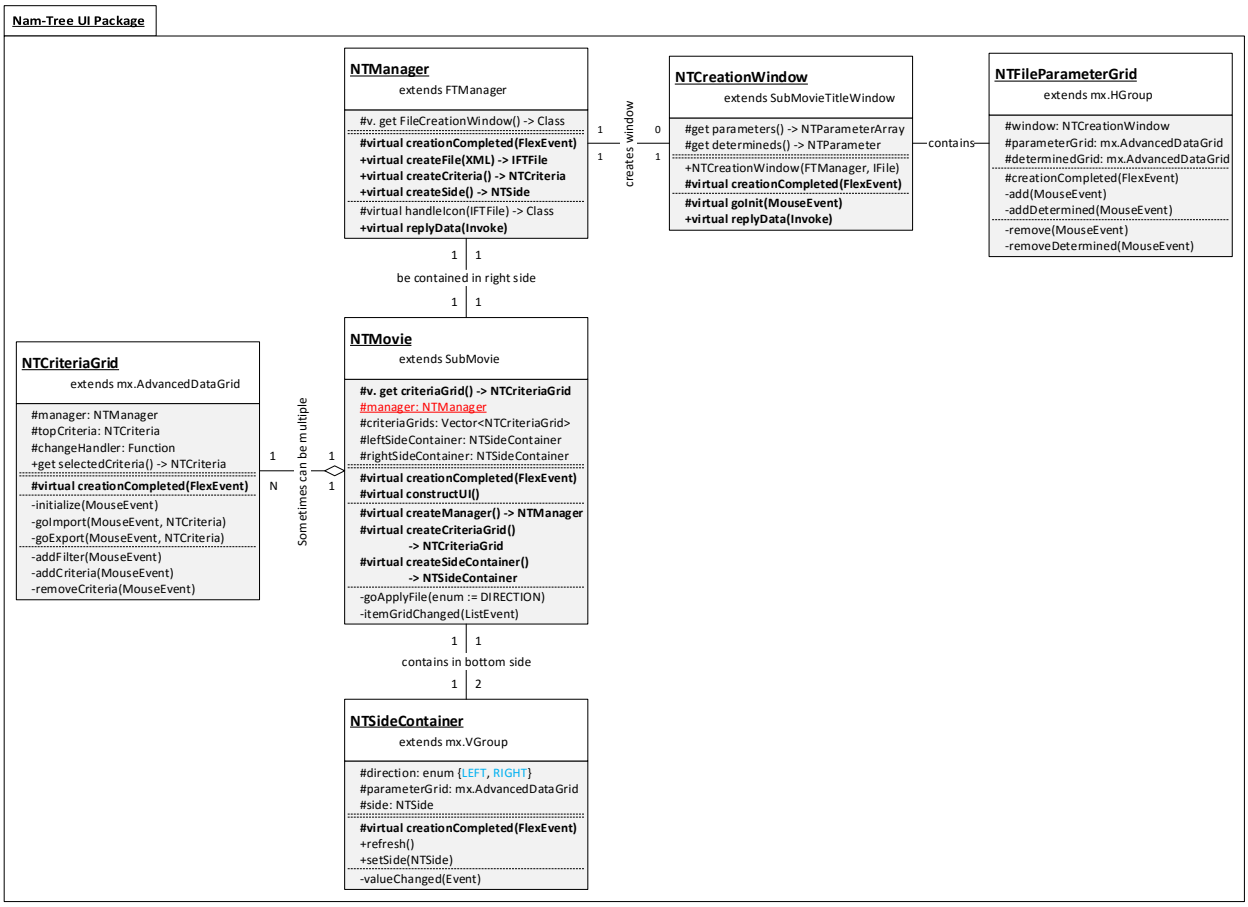


3.3.2 Nam-Tree


Flex NamTree Entity
NTFile
extends FTFile
implements INTExploreParameter
#parameterArray: NTParameterArray
#otherside: NTFile
-memberName
#header: String
#body: String
#composer: String
#function: String
+NTFile(FTFolder)
+virtual construct(XML)
+virtual toXML() -> XML
NTManager
extends FTManager
#v. get FileCreationWindow() -> Class
#virtual creationCompleted(FlexEvent)
+virtual createFile(XML) -> IFTFile
+virtual createCriteria() -> NTCriteria
+virtual createSide() -> NTSide
#virtual handleIcon(IFTFile) -> Class
+virtual replyData(Invoke)
<<creates>>
NTParameterArray
extends EntityArray
#virtual TAG() -> String
#virtual CHILD_TAG() -> String
+NTParameterArray()
+virtual createChild(XML) -> Entity
NTCriteria
extends EntityArray
#virtual TAG() -> String = "criteria"
#virtual CHILD_TAG() -> String
#manager: NTFactory
#parent: NTCriteria
#left: NTSide
#right: NTSide
#operator: int
#weight: double
+NTCriteria(NTManager, NTCriteria)
+virtual construct(XML)
#cirtual createChild(XML) -> Entity
+virtual toXML() -> XML
NTExploreParameter
extends Entity
#minimum: Number
#maximum: Number
#bin: Number
+INTExplorer()
+virtual construct(XML)
+virtual toXML() -> XML
NTParameter
extends EntityArray
implements INTExploreParameter
#virtual TAG() -> String = "parameter"
#virtual CHILD_TAG() -> String
#name: String
#initialValue: Number
+NTParameter()
+virtual construct(XML)
#virtual createChild(XML) -> Entity
+virtual toXML() -> XML
NTSide
extends Entity
#file: NTFile
#parameterMap: Dictionary<Number>
+NTSide(NTManager)
+virtual construct(XML)
+virtual toXML() -> XML
0
2
1
1
NTFilter
extends NTCriteria
+NTFilter(NTManager, NTCriteria)
+virtual construct(XML)
+virtual toXML() -> XML
NTParameterDetermined
extends Entity
#virtual TAG() -> String = "determined"
#label: String
#value: Number
+NTParameterDetermined()
+virtual construct(XML)
+virtual toXML() -> XML

JS guidance

Flex - Nam-Tree



4 Appendix

4.1 Projects using Samchon Framework

4.1.1 Samchon Simulation

4.1.2 Hansung timetable

4.1.3 OraQ

4.2 Developers

4.2.1 Jeongho Nam

4.2.2 Participate in Samchon Framework

4.3 Version history

4.3.1 Samchon Library

4.3.2 Samchon Framework

4.3.3 Plans for next generation of Samchon Framework