원칙 1.

어렵지 않게 만들어야한다.

코드 리뷰 실력과 상관없이 쉽게 흐름을 찾고 한 부분에 대한 이해를 할 수 있어야한다. 흐름에 대한걸 만들 필요는 없지만 어차피 함수의 호출은 스택구조이며 스택에서 스택으로 이어지는 구조이기 때문에

원칙 2

코드만 보고도 해당 부분을 알 수 있어야한다.

함수로 보면 함수의 이름은 제목이고 내부는 내용이다. 작은 부분으로 나누어서 나와야한다. 무조건, 그것이 글을 읽는 사람을 생각하는것이다. 코드는 글이며 공통 언어일뿐이다. 그렇기 때문에 유추가 가능해야한다. 코드는 간결하지는 않아도 되지만 가독성은 있어야한다.

원칙 3

성능,유지보수성,사용성 을 잘지켜야한다.

프로그래밍시 고려해야하는 것들

성능

런타임의 프로그램에서는 성능을 고려해야한다. 느린 프로그램은 누구도 하지 않기때문이다. 여러가지 최적화 방법을 연구해보자.

유지보수성

유지보수성이란 무엇일까 유지보수한다라는 말은 이제 운영이라는 말과 일맥상통한다. 어떤 코드를 유지보수하기 위해서는 코드를 처음에 짤 때 잘짜야한다. 잘짜야한다라는것은 코드자체가 문서같이 읽혀야한다는 말이고 그말인 즉슨 가독성이라는 말이다. 가독성이 좋은 코드가 좋은 코드의 요소중 하나이고 유지보수성에 포함되는 말이다.   
가독성만 좋으면 되는 것도 아니다. 운용성또한 갖추어야한다. 운용성이란 운영의 편리함이다. 이러한 편리함이 좋아야 나중에 같은 코드를 수정하거나 재사용할 때 활용할 수 있다.  
단순성이라는 항목이 있다. 내가 가독성이라는 말을 했는데 단순성 과 일맥상통하는 부분이 많다. 하지만 약간 다른점도 있는데 단순하면 단순할수록 코드의 흐름을 좋게 만들어주는것이다. 해당부분은 추상화를 통해 얻어지는게 많다.  
정리해보자 유지보수성이란 뒤에 사람이 사용하기 좋게 하는것이고, 그러한 요소에 해당하는 것은 코드의 가독성,단순성,운용성이 라는것이다.  
데이터 중심 어플리케이션에서는 발전성이라는 것도 있는데 이를 좀더 쉬운언어로 풀어본것이다. 그니깐 쉬운코드를 만들어 해당 코드를 이해하기 쉽게 만들고, 단순하게 만들어서 코드의 흐름을 쉽게 볼 수 있게 만들고, 디버그 용도의 코드를 짜서 디버깅을 쉽게하라는 소리다.

확장성

확장성이란 유지보수성에 포함되기도 하지만 따로 카테고리를 뺀이유는 게임의 개발에서는 확장성이 중요하기 때문이다. 수정에는 닫혀있고 변화에는 열린구조를 띄는게 oop 의 핵심중에 핵심인데, 해당부분을 강하게 내놓은것이다. 확장성이 중요한 또다른 이유로는 업데이트를 할 때 필요하기때문이다. 확장성은 생각보다 많이 어려운 부분이다. 성능과 유지보수성을 고려하면 확장성은 닫히게 된다. 하지만 이것을 극복할 때 비로소 좋은 코드가 탄생하는 것이다. 단순하게 생각하면 이렇다. 어뎁터 패턴이나 프록시, 브릿지 패턴으로 무언가를 만든다고 하자. 그럼 성능에 큰 위배가 이루어지는데, 이를 직접 프록시가 인터페이스를 통해 컨트롤 한다고 하자. 혹은 분리된것이다,. 실질적인 실행코드는 다른곳에 있고 인터페이스만 해당 코드가 쓰이는것이다. 이러면 좋은게 빌드에 따라 다른 확장성을 부여한다. 하지만 이는 보호장치가 적은편이라 확장성이 줄어든다. 그럼 어떻게 해야할까? 그럴때는 경험이 중요하다. 어차피 가상테이블을 호출한다. 가상테이블이 몇번을 왔다갔다하는것보다 한번에 호출하는게 성능상 이점이 있다면 처음에 캐스팅을 시키는데 바로 접근할 수 있는 데이터로 변화를 시키는 것이다. 이러면 좋은게 캐스팅에 대한 비용을 줄이고, 인터페이스에 따라 작동을 해서 가상테이블을 통하지만 기능의 대리를 하는 녀석이 해당 기능을 전적으로 가지는 것일경우(이럴경우에는 프록시 패턴이겠지) 굉장히 성능에 이점이 생긴다. 하지만 흐름이 어려워진다.

신뢰성

신뢰성이란 해당기능이 정상작동하는것이다. 문제없는 코드는 컴퓨터 구조를 안다면 쉽지 않다는 것을 안다. 캐싱미스라는 것은 싱글쓰레드 프로그램에서도 자주 나오는것이다. 캐시미스가 뜨면 os 는 캐시미스를 캐시히트를 시키기 위해 또 자원을 사용한다. 이것은 컴퓨터의 구조가 그런것이라 어쩔수없지만 해당 부분을 줄일 수는 있다. 참조 지역성이라고 하는 것을 사용하면 된다. 해당 내용은 알아서 찾아보고, 또한 부동소수점의 표현의 한계와 네트워크의 카오스 상태또한 그자체로 신뢰성이 없다고 봐야한다. 하지만 이러한 근본적 원인을 줄이는 것또한 프로그래머가 할일이며 추가적으로 내가 짠 논리가 다른곳에서는 제대로 작동을 안한다면 신뢰성에 큰 문제가 있는것이다. 그렇다. 신뢰성이란 결과의 일관됨을 말한다. 크로스 플랫폼프로그래밍이 어려운 이유이기도 하며 해당 부분은 고려할 사항이 많기에 어쩔 수 없이 어셈블리를 추적할 수밖에 없다.

플랫폼

개인정보는 컴포넌트 형태로 저장할것이다.왜냐하면 나중에 추가되어질 개인정보나 유저의 추가 정보를 쉽게 추가하기 위해서이다. 그래서 컴포넌트형태로 저장을 할것인데, 중요도 순으로 저장이 매겨져야한다. 중요한 정보는 메모리에 남기지 않는다. 남기더라도 보안을 위해 주기적으로 네트워크통신으로 확인을 한다.

보안의 종류로는 레드,오렌지,그린으로 나눈다.

레드는 개인정보중에서 가장 위험한 정보로 절대 램에 남기지 않고, 기능또한 연결하지 않는다. 무조건 특수한 보안 정책이 적용되어진 다른 프로세서에서 실행을 한다. 예시로는 현금으로 결제하는 부분이 있다. 해당부분의 결과는 원자성을 띄어야한다. 즉 cpu 락까지 걸정도로 세심해야한다.

오렌지는 게임환경에 치명적일 수 있는것에 적용한다. 예를들어 게임내 재화가 있다. 게임내에 재화를 램에는 저장을 하지만 네트워크로 주기적으로 확인을 해준다. 만약 실패했을 경우에는 해당부분을 락을 걸거나 그래야한다. 레드도 그렇고 오랜지또한 수정에 대한 이유가 존재해야한다. 그래서 모든 것에 receipt를 남기는 것이 중요하다.

그린은 개인 정보기는 한데 게임에 필요한 정보들이고 치명적이지 않는 정보들을 말한다. 예를 들어서 유저가 가지고 있는 진행상황이라든지 게임 닉네임 이라든지 위치? 뭐 그런것들을 의미한다. 이부분도 잘 나눠야한다. 공통의 기능이기는 한데 모듈별로도 가지고있어야하는 정보가 다르니 그것도 고려해야한다.

플랫폼에는 저런 정보들과 통신하고 데이터를 저장하고, 공통의 기능을 가진다.

계획을 세워보자

일단 버텍스를 수정할 수 있어야한다. 그럼 버텍스를 수정

버텍스를 수정하는게 맞나? 음…이게 맞나? 아니 생각해보니깐 그럴필요있나 싶네. 그런데 생각해보면 인스턴스로 사용하면 문제는 없긴한데 흠…

바닥은 점으로 해야한다. 그리고 인덱스 버퍼를 변경하는 방향으로 가고, 해당 정보를 새롭게 갱신하는 방법으로 가면 좋겠다. 객체의 표면에 그리드가 생긴다. 이거는 연속해서 있으면 해당 그리드는 합쳐진다. 합쳐지는 방식은 빌드 시스템에 있을것이다. 하여튼 이렇게 해당 영역에 그리드 혹은 닷넷이 생긴다.

비스듬이 볼 때 카메라가 바라보는 각에서 백은 어떻게 해야할까 생각해보자.

모듈이란

패턴의 선택

생성패턴

해당 패턴들을 코드를 쓰는 사람에게 적합한 패턴이다. 코드를 쓸 때 생기는 문제점을 해결하기 위한 패턴이기 때문에 해당 패턴으로 코드를 작성하면 유연성과 재사용성을 확장시킬 수 있는 유지 보수, 생산성, 관리 부분이다. 성능은 특정 패턴만 고려한다.

성능 2/5

직접 생산하는 것이아닌 어떤 클래스 혹은 객체를 통해서 생산하는 것이기 때문에 성능은 고려사항이 아니다

유지보수성 5/5

생산패턴에 해당하는 것의 주된 목적은 신뢰성과 유지보수성이기 때문에 그렇다. 해당 패턴들을 사용하면 추후에 관리적인 측면에서 쉽게 접근하고 객체를 생산할 수 있다.

생산성 3/5 -> 개발단계 1/5, 운영단계 5/5

복잡성이 생각보다 높아서 유지보수하는 인터페이스는 좋아도 생산성은 떨어진다. 하지만 추후에 유지보수 단계에서는 생산성이 올라가기 때문에 처음 구축비용이 높다라고 이해해도 좋다

확장성 3/5

어떤 생성패턴이든간에 확장을 할때는 좋다. 왜냐하면 해당 코드는 직접 생산부분이 분리되어있기때문이다. 하지만 특정 구조의 어떤걸 만들기 위해서는 무조건적으로 코드를 작성해야하는 패턴또한 존재하기 때문에 3점이다.

신뢰성 5/5

일단 해당 패턴들로 어떤 객체를 만들면 만들어진다라는 확정은 있다. 운영이나 업데이트 단계에서도 마찬가지로 생산만을 위한 것이기 때문에 코드만 망가지지 않았다면 해당 코드는 신뢰할만 하다.

팩토리 메서드

상황 :

추상팩토리

빌더

생성자방식

Old 빌더 방식

프로토타입

싱글턴

구조패턴

구조패턴은 빌더와 다르게 전체적인 구조를 말한다. 그리고 해당 구조는 확장성을 메인으로 한다.

성능 2/5

중간자를 거쳐가는 경우가 많아서 성능은 사실 별로다. 하지만 경우에 따라서는 제일 빠른 방법이 될수도 있다. 그렇기에 2점

유지보수성 5/5

유지보수는 끝장나는게 애초에 구조패턴들이 해당부분을 고려해서 만든것이다. 심지어 옛날객체를 불러와 만드는 경우도 있다. 그런것또한 가능하게 만들어준다.

생산성 2/5 개발단계 1/5 테스트단계 3/5 운영단계 4/5

어떤 객체를 거의 복사하여 기능의 인터페이스들을 만들어야 하는 경우도 있다. 그렇기에 시간이 배로걸리기도 한다. 대부분의 구조가 그렇다. 심지어 테스트 단계때 해당 구조 때문에 디버깅에 복잡도만 늘어나기도 한다. 또한 운영단계에서 문제가 생기는 부분이 해당 구조일 수 도있다.

확상성 5/5

애초에 유지보수와 확장성을 고려한게 디자인패턴이고 무엇보다 구조패턴은 그것을 가장 중점으로 둔것이다. 그렇기에 확장성 면에서 구조패턴을 사용한다면 확장에 문제는 거의 없다. 물론 ocp 를 잘 지켰을때의 이야기다.

신뢰성 3/5

신뢰성은 업데이트를 거치면서 제대로 작동을 안할 수 도있다. 혼자작업하는게 아닌경우에는 더 그런데, 해당 구조로 중간자를 만들었을 경우에는 더더욱 신뢰성은 떨어지지만 작동은 하게 만들 수도 있는게 구조패턴이기 때문에 잘 고려해야한다.

어뎁터

어뎁터 패턴은 호환되지 않는 것을 완전히 다른 객체로 감싸서 같은 동작을 시키는 것이다. 어뎁터 또한 추상화 되어있어서 해당 부분을 실행하는 것은 공통의 작동으로 실행시킬 수 있다.

브릿지

브릿지 패턴은 추상화가 되어진 부분과 실제 구현되어진 즉 자식들이 다른건데, 추상화부분또한 추상화 되어있기에 완전히 다른 작동을 가능케 해준다. 추상화되어진 인터페이스와 실제 추상화 되어진 작동부는 서로 독립적이다.  
추상 인터페이스를 만든다. 추상 인터페이스는 추상 디바이스와 독립적이지만 리모컨 같은것이다. 추상 디바이스를 만든다. 추상 디바이스는 구체적인 실행을담당하는 부분으로 추상 인터페이스에서 실행할 수 있어야한다.   
이렇게 만들어진 추상 인터페이스를 확장시켜 추상 디바이스 기능을 적절하게 사용하게 만든다. 추가적인 기능이 필요하면 해당 구분이 가능하게 만들면 된다.

복합체

그냥 트리 노드같은거다. 그런데 최상단의 추상클래스가 있고, 그것을 composite 하는 특수 노드가 있는것뿐이다. 복잡하게 얽힌 것을 재귀적으로 처리할 때 사용하는거 같은데 한번 생각해보자.

데코레이터

이거도 어뎁터처럼 wrapper 클래스인데 다른건 인터페이스가 컨텐츠 클래스와 같다는것이다. 그리고 무엇보다 합성이 가능하도록 한다.

퍼사드

그냥 복잡한거를 정리해서 인터페이스로 사용하는 것이다. 나중에 많이 복잡할경우 사용하자

플라이웨이트

고정구조 혹은 고정부분은 냅두고 인스턴스 내용만 바꾸는 것이다. 메테리얼 생각하자. 그래도 한번더 설명하자면 A 집합의 교집합 요소(값)는 A 요소들 전부가 가질 필요없고, 차집합의 요소들만 A 집합들이 가지면 된다는것이다. 물론 교집합의 요소들을 가르키는 포인터는 필요하다. 포인터도 제거할 수는 있다. 물론 어떻게 만드냐에 따라 다른것뿐이지만 핵심은 그렇다.

프록시

들고오는데 무거운 자료나 연결이 필요한 자료들은 프록시를 통해 처리를 지연시킬 수 있다. 그러니깐 메쉬를 수정하는 작업은 굉장히 무거운 작업인데 이를 해당 메쉬의 프록시를 가져온후 특정부분을 바꿔라라는 명령을 내리면 프록시는 이를 저장했다가 뭐 비동기적으로 틱에 나눠서 처리하든가 하겠지. 물론 캐싱과 지연만 존재의 이유는 아니다. 예를 들자면 접근은 빨라야하는데 처리는 느려도 되는경우는 결제 처리같은게 있다. 그럼 프록시에 캐싱해놨다가(램에 저장) 저장되지 않는 함수형 프로그래밍처리로 프록시의 값을 처리하고 돌려주는 방법을 생각해볼 수 있다. 기본적으로 비동기가 핵심 코드이다.

행동 패턴

성능 3/5

구현 방식에 따라서 룩업테이블,가상테이블 을 고려하여 구현하면 성능 최적화가 이루어지기도 한다.

유지보수성 4/5

기본적으로 유지보수성이 뛰어난게 디자인패턴들의 특징이긴 하나 서비스가 변화하거나 할 때 문제가 생기는 패턴들이 존재한다.

생산성

확상성

신뢰성

책임연쇄

파이프라인으로 분할 처리한다는 말이다. 분할 처리를 하는데 핵심 목표는 중간에 커트하거나 분기하는게 주된 목적이다. 행동패턴의 가장큰 장점은 이러한 것에 성능향상을 도모 할 수 있다는 것인데, 여기서 룩업테이블 식으로 핸들컨트롤을 할 수 있다. 처리가 실패하는 부분에만 if 를 걸어주고 그외에는 무조건 처리하게 만들면 cpu-scheduling 에 도움을 줄 수 있다.물론 if 가 없는것도 생각을 할 수있는데, 이럴땐 계산의 결과로 if 를 대신하는 고도의 방법이 필요하다. 진짜 여러 번 호출되는 부분의 경우에는 이렇게 처리할 수 있는데 ,if 가 100개인데 한번만 실행되는것과 if 가 1개인데 100번 실행되는것은 사실 성능차이가 난다. 물론 어떤 처리인지가 중요하긴한데, 단순 계산일경우 if 1개인게 좋지만 복잡한 계산(나눗셈이나 행렬계산 이나 float 나 double 계산 같은거)일경우 차라리 if 가 100개인게 좀더 좋은 성능을 낸다고 한다. 물론 이건 거의 의미없지만 이렇다는것만 생각하자.  
하여튼 돌아와서 책임연쇄는 파이프라이닝이고 분기가 들어가서 책임연쇄라는 특징을 가지게 된것이다  
핵심키워드는 핸들러와 next 이다.

커맨드

이건 예상해볼 수 있겠지만 액션 혹은 트랜잭션이라는 이름으로도 불리는 패턴이다. 요청을 주면 그걸 저장했다가(프록시) 하나하나 개별 실행으로 실행하는 하지만 undo 나 삭제 취소가 가능해야한다. 책임연쇄와 조금 다른점이 이것이다. 하나하나 처리한다. 이다.   
가장 쉽게 볼 수 있는 예시는 그래픽작업이다. 메쉬의 버택스를 옮기는 작업은 그 자체로 그냥 벡터의 이동일 뿐이다. 그걸 쌓고 다음에 그 버텍스에 색을 넣는것도 개별작업이다. Undo 하고 나서 색을 넣고 이동해도 같은 결과이다. 이것이 개별 작동하는 커맨드 이다.  
하여튼 커맨드 패턴은 기본적으로 요청을 바로 처리하는게 아니라 다른 로직이 처리하고 그러한 처리를 요청으로 처리하는 것이 핵심이다.   
핵심 키워드는 invoke 와 command, request,receive,pay\_load 이다.

반복자

그냥 iterator 임

중재자

컴포넌트간의 복잡한 연결관계를 해당 패턴으로 만든 클래스에 위임함으로써 독립성을 유지하게 하는 행동패턴임

메멘토

메멘토는 스냅샷을 남기고 그 스냅샷을 변경하는 클래스를 또 따로두어서 컨트롤 하는것이다. 즉 원본의 값에는 접근을 할 수 없는 것이다. 내부에서 보안을 신경쓸 때 사용할 수 있는것이지만 램에 냅두는 것 자체가 보안상에 문제가 있는 방식이다. 물론 해킹을 방지하기 위해 마스크를 두어 만들어도 되기는 하는데 이게 생각보다 성능상에 문제가 생기는 방법이기도하다. 검증을 서버에서 하는데 음…이거 괜찮은데? 근데 이거는 표시가 유저에게 되는순간 안좋긴하겠다. 왜냐하면 역으로 추측이 가능해지기 때문에 그렇다.

옵서버

옵서버 옵저버 하여튼 그냥 델리게이트 서브스크라이브 이거가 핵심인거다. 누군가가 대신 계속 확인을 해주는거다

상태

상태전략은 FSM 이다.

전략

이건 FSM 과 가장 유사하긴한데 전략을 갈아끼우는 것이다. 작동은 비슷해 보이지만 FSM 은 상태에 대해 각 부분이 서로알고있지만 전략패턴은 그걸 서로 모른다는것이다. 알게도 할 수는 있는데, 전이보단 진짜 완전히 다른 상태를 만들고 싶을 때 사용한다.

템플릿 메서드

비지터

네이밍규칙

단단하면서 무언가 뭉쳐서 하나의 공간을 만드는 것

1. Mountain
   1. Hill
      1. Knoll
      2. Mound
         1. Clod
            1. Soil

무언가 흐르거나 점점 진행하는 것

1. Ocean
   1. Riverl
      1. Creek
         1. Spill
            1. drop

유동성 있는 어떤게 뭉치는 것

1. dew
   1. Puddle
      1. Pond
         1. Reservoir
            1. Lake

모든 인테리어 물체는 추상 팩토리를 기반으로 한다. 만들땐 추상팩토리를 쓰는데,

모듈은 브릿지 패턴으로 만든다.

유저인포시그날의 처리는 책임연쇄패턴으로 만든다.

유저인포시그날에 정보를 전달할땐 커맨드 패턴으로 한다. 그러나 메멘토로 보안을 신경쓴다. 혹은 요청만 존재하고 커맨트패턴은 쓰지 않는다. 아니면 메멘토로 만들고나서 스냅샷을 저장하고 커멘드 패턴으로 정보를 처리하는 방식을사용하자  
정리하자면 오렌지에 저장되는건 유저의 고유 해쉬정보, 해쉬정보는 스냅샷하지 않고 특수한 경로를 통해서만 처리하도록한다. 하여튼 일정한 주기별로 스냅샷을 찍고 이전의 스냅샷과의 차이점을 서버에 보내서 무결성을 검사한다. 즉 스냅샷은 이순간에 3개가 생기지만 상관은 없다. 그렇게 보낸 스냅샷이 올바른 스냅샷일 경우에는 유저의 스냅샷은 거들이지 않는다. 하지만 문제가 있는 스냅샷의 경우에는 레드를 보내서 락을 걸고 커맨더 리스트를 보내서 상황을 본다.   
스냅샷의 경우에는 그린에서 더 많이 사용이 될것같다.

하우징모듈

복합체로 구성?!

결국 최하단의 floor 가 공간을 뿌려놓으면 되는거아님? 그리고 이제 빌드 시스템이 해당 floor 를 참조하면 되는거고 그럼 탐색기를 만들어서 상태에 따라 움직이게?!

탐색기의 경우에는 전략패턴으로?!

* 플랫폼
  + 유저인포시그날
    - * 레드 패시브

유저 개인정보 및 램에도 저장되면 안되는 정보.

함수에서 바로 처리가 되어야하며 요청을 받으면 락을 걸어야한다.

요청을 받으면 요청에 대해서만 정해진 커멘드가 전달이 되며 서버에서 처리되고 나서 리턴을 받는다. 받고나서 원하는 결과면 바로 다른요청으로 가린다.

현질을 여기서 관리하며 유저의 개인정보와 관련된 민감한 정보는 여기서 관리를 하지만 책임에 대한부분은 좀더 회의를 진행후에 결정한다.

* + - * 오렌지 액티브

게임내에 있는 게임화폐를 관리한다. 레드만큼 유저에 가깝지는 않지만 게임 내부에서 관리해야하는 정보는 여기서 관리한다. 즉 게임에 치명적인 부분은 여기서 관리해아한다는 말이다.

액티브인 이유는 램에는 저장이 되지만 지속적인 연결을 요구하며 치트를 원천적으로 막는 부분이다. 게임 화폐 결제 또한 오렌지의 영역에 속하며 계산은 서버에서 해준다. 서버에서 유저의 화폐정보를 가지고있다가 결제내역으로 들어온 결과와 비교후 내역을 보내주고 결제가 이루어진다. 게임을 원천적으로 막지는 않지만 다른 작동은 막아야한다.

요청을 받으면 연결을 한다.

표현을 주로한다. 숫자를 표시하거나 하는 기능은 하지만 실질적으로 처리는 서버에서 해준다. 표시는 램에 저장되고 표시하라는 함수가 존재해야겠지?

* + - * 그린 올
        + 게임내에 자원같은것들을 말한다. 무결성 확인은 오렌지보다는 적지만 행위를 저장함으로써 이상행동을 지속적으로 모니터링하는 요소로 사용된다.
        + 락을 걸지 않고 로그만을 지속적으로 내부검사와 서버검사를 겸하는 역할을 한다.
        + 로그를 남긴다는 점에서 램에 상주하는게 문제이기 때문에 특수한 해쉬로써만 저장을 한다.
        + 행위는 FSM 으로 만들어져서 진행이된다.
  + 모듈퓨즈->팩 파워시스템
    - 모듈을 실행하는 주체이다. 실행만 하는 것뿐만아니라 무결성 검사를 레드 오렌지 그린 순으로 진행을 한다. 모듈이 실행될때는 이렇게 실행이 된느데 음…이러면 너무느린가? 하여튼 브릿지 패턴으로 만들어진 기능없이 오렌지와 그린을 가지는 데이터센터가 있으며 아래에 서술한다.
  + 데이터센터
    - 오렌지와 그린을 프록시로써 가지고있으며
* 모듈
  + 모듈은 일반적으로 어떤 독립적인 기능이다. 하지만 퓨즈없이는 작동하지 않으며 유저의 데이터센터에 요청을 받아 요청한 녀석에게 값을 넘기는 역할을 한다. 이러면 속도가 느려지지 않을까 할 수 있는데, 성능을 요하는 부분에서는 많은 데이터(변동되어도 문제는 안되는것들)를 미리 캐싱하는 방법을 사용한다.
    - 예시로 생각한 것은 유저가 물건을 배치하는걸 클릭하면 물체가 먼저 자신의 영역을 닷넷에 보낸다. 이는 실시간으로 이루어지며 움직임이 들어갔을때만 호출을 한다. 최초에는 한번 호출하는데 닷넷의 영역으로 역 트렌스폼을 해주고 그 영역에서 점과 obb 를 해준다. 이는 아주 쉬운작동인데, 3가지 단계가 될거같다.
      * 마우스의 위치로 어느 닷 레이어의 영역을 활성화 시킬지 결정
      * 레이어와 물체사각형을 충돌
        + 물체의 역transform 으로 변환후에 하는 방법

장점은 aabb 를 사용할 수 있다는 점

단점은 점은 2차원 배열인데 그걸 회전시키면 locality 를 못받을 수 있다. 아니 못받는다. 그리고 무엇보다 일차원적으로 점과 원을 충돌시키는데(그래야 스냅이 걸림) 그러한 해당 영역을 또 변환 해야한다는 점에서 이게 유효한 방법일까라는 생각이 든다. 총 45도 각도로 움직이니깐 사실 해당 부분을 캐싱은 할 수 있다. 예를 들어서

* + - * 해당 점의 변화가 가능하면 변화

Class Life cycle

InitXXX

RoutineXXX

FinalXXX

Time Semantic

xxxBefore

xxxMain

xxxAfter

class type

platform == station

module == pak

context == desc[ription]

parameter syntax

array == XXXarray

빌더패턴-빌더방식과 생성자 방식은 다르다.

빌더패턴을 적용하고싶으면

생성자를 만드는 것처럼 해야한다.

큰일났네…메크로도 안통하고 그럼 결국 extension 을 사용해야한다는건데 어떻게하지…

스냅이랑 충돌을 나누자.

스냅은

만약 마우스가 움직인다면

마우스에 물체가 달려있는지 확인. 마우스객체를 추상화 해서 가져오자. 가져와서 움직임이 커졌을때 반응시킨다.

반응이 일어나면 마우스 포인터를 전사시키고 그 전사시킨 부분에서 원을 만들고 위치를 대략적으로 잡은후 근처의 정점들을 가져오고 아 피킹위치에서 마우스 움직인 위치의 크기를 재서 계산해야겠구나. 하여튼 이렇게 근처를 잡은게 생기면 그 위치로 순간이동 이게 피킹.

그럼 마우스가 움직일 때

먼저 물체가 달려있는지 확인

근처의 전사시킬 레이어 확인. 이 레이어는 전체 맵에서 가져온다. 그니깐 해당 영역을 전체적으로 전사시킨 녀석으로 그냥 계산시키는거임. 혹은 루트에서 확인하자. 계층적으로 올라가는거 너무 느린거 같다. 영역을 칠하는건 음…계층적으로 할까? ㅋㅋㅋ으…시발

지금 직면해 있는문제

계층적으로 올라가는 것을 추상화 하는게 vt 를 지나서 호출되기 때문에 속도에 조금의 손해가 생기는데, 이게 만약에 진짜 만약에 자주 일어나는게 아니라면? 이렇게 추상화하는게 좋지 않을까 와 그래도 성능을 위해서 완전 특수화하는 것으로 사용하자가 있다. 즉 완전 특수화할것인가 계층화 할것인가

계층화를 할거면 어떻게 할것인가이다. 모든 것을 하나의 노드의 계층구조로 표현할 것인가? 아니면 좀 다르게 만들것인가 이다. 솔직히 많이 나누면 나눌수록 코딩할 때 불편하고 유지보수에 불리해진다. 그럼 어떻게 해야한다는건가.

그냥 하나의 노드로 통합

인터페이스로 모든 사이클을 빼는게 좋은가.

장점: 관리가 편리해진다. 코드짤때 명확하다. 해당부분의 인터페이스 상속되어있으면 해당 인터페이스가 있구나 하고 사용하면된다.

단점: 복잡해지고, 어쨌거나 해당 인터페이스만빼서 사용할게 아니라면 구지? 라는게 되는데 생각해보니 처음 소팅이나 그그런 해줄 때 한번만 해당 인터페이스를 상속해놓고 있다면 그런애들만 묶어서 처리할 수 있지않나? 계층적으로 되어있어서 공통적으로 관리가 가능하고 서치가 가능은하지만 기능적으로 계산해줄땐 그러한 애들만 묶어서 최대한 그렇게 관리?! 이거 좋다!

통합하고 기능을 인터페이스로 나누고, 태그로 소팅을한다.

기능별로 태그를 나눠서 한번더 소팅을 하든가 그런다. 저장용량은 많이 늘어나겠지만 성능을 생각해서 내부 테이블을 만드는것으로 간다.

다시

유저가 처음에 접속을 한다. 접속을 해서 스테이션이 접속한 자의 상태를 파악하고 팩을 실행시킨다. 팩이 실행하면 절대자부분에서 노드를 불러온다.불러오는 방식은 서버든 뭐든 상관없음. 해당 노드는 실행화면의 메인 노드이며 뭐 보안 실행하고 자신의 자식들의 상태에맞는 비지터를 만든다. 여기에 팩토리 메소드 패턴이 사용이 될것이다.(해당부분은 커스텀이 되는게 아니라 정해진 걸해야하기때문이다) 이렇게 만들어진 비지터는 이제 해당 노드의 실행파트에서 실행이 될것이다, 만약 하우스 팩이었고 하우스 노드중 site 에 해당했다면 site

참고로 해당 시스템은 액터와 분리되어진 시스템에 해당한다. 서로 양방향 참조는 하지만 그렇다고 strong 하지는 않게 만들어야한다. 액터의 명세서? 같은느낌이다. 그러면 사람이 그 명서세를 보고 행동을 하는거지.

비지터가 해당 노드 안에 있는 객체들을 방문한다. 방문전에 적절한 비지터를 넣기도한다. 하여튼 해당 코드는 리프에서 이루어지는게 아니다. 음… 새로운걸 만듦, 부쉬라고 부쉬노드에서 자신의 자식들에게 뭔가를 작동하게 한다. 거의 이것만 쓸거같은데 그럼 정리하자면 이렇다

리프 : 단일 기능. 예를들자면 위에 뭔가를 올릴수없는 물체나 뭐 전체를 순회하는데 if 용도나 behavior 처럼 단일이 뭉쳐서 하나의기능을 출력할 때 사용한다. 그럴땐 blackBoard 가필요하겠네

컴포지트: 단순하게 물체를 여러 개 가지고있으며 자신의 자식들을 실행시키는 기능만 가진다. 물론 어느것만 실행할지 같은 것은 리프에서 정하면 이제 실행하는 것. 뭉쳐있을텐데

부쉬 : 컴포지트 + 리프로 여러가지면에서 성능이 뛰어날것으로 보인다. 확장성은 적어보이는 것인데, 이걸 만든 이유는 물체가 배치되면 서로 상관관계가 있으면서 무언가를 실행해야하는데 그걸 리프로 넘길수는 없기때문이다. 그래서 리프 + 컴포지트임

컴포지트가 고민이 되네

스테이션을 어디선가 만들어준다. 그 스테이션의 init 를 해준다

해당 init 에는 뭐가있을까. Command 세트들을 만들고 observer 를 만들어준다.

command 는 어떤 명령들을 처리할 녀석들이다.

공통적으로 부지는 썸네일이 보여야한다. 그렇다면 그러한 정보는 부지가 가지는게 좋다. 해당 부지에 어떤 태그가 달렸고 물체는 무엇인지

부지를 선택하는 기능

층을 선택하는 기능

층을 새롭게 만드는 기능

층내부 물체들을 가지고있어야한다.-> 노드가 가지고있음

물건을 놓는 기능

물건을 놓는 기능

지금 내가 선택 할 수 있는 물건이 보이는 기능

선택한 물체의 복사체를 만드는기능

물체를 생성하는 기능

물체가 마우스에 올라가있는 기능

지금 내 마우스와 충돌한 레이어를 확인하는 기능

물체를 배치하면 주변의 물체와 통신하는 기능

물건의 색을 변경하는 기능

물체에 접근하고 메테리얼을 변경하는 기능

메테리얼을 묶는 기능

그럼 레이어도 물체로 취급하자.그럼 그 물체위에 올릴 수 있는 시멘틱이 있으면 올릴 수 어야한다. 그럼 레이어는 랜더를 빼기는 하는데, 그럼 랜더는 필수가 아니란거다.

레이어

가상화 이건어떻게 해야할까. 흠...하여튼 이게 있긴해야함.

실체화 준비 데이터를 커멘더에 넘겨준다.