The Journal of The Institute of Internet, Broadcasting and Communication (IIBC)
Vol. 23, No. 4, pp.183-188, Aug. 31, 2023. pISSN 2289-0238, eISSN 2289-0246

https://doi.org/10.7236/JIIBC.2023.23.4.183
JIIBC 2023-4-28

의존성 주입을 활용한 서바이벌 게임 API 설계 및 구현

Design and Implementation of the Survival Game API Using Dependency Injection

박인규*, 최규석**

InKyu Park*, GyooSeok Choi**

요 약 게임 오브젝트의 상속 및 여러 가지의 컴포넌트를 이용하여 시스템 아키텍처의 시각화, 우수한 코드 재사용 및 빠른 프로토타이핑이 가능하다. 반면에 객체는 게임 오브젝트와 컴포넌트 간의 높은 대기 시간, 정적 형변환과 널 포인터 등의 많은 참조에 의존하기 마련이다. 게임 제작시에 여러 클래스에 대한 객체의 의존도를 낮추고 코드의 재사용이 가능하도록 설계하는 것은 중요한 일이다. 따라서 의존성 주입(Dependency Injection)과 GoF(Gang of Four)가 제안한 디자인 패턴들을 적용하여 클래스의 모듈성을 더욱 높일 수 있도록 게임을 설계하였다. 이러한 의존성은 게임 오브젝트의 속성이고 주입은 초기화 패스에서만 발생하므로 게임 루프에서 성능 저하나 성능 페널티는 미세하다. 따라서 본 논문에서는 서바이벌 게임의 설계와 구현에 있어서 API(Application Programming Interface)를 효과적으로 재사용하는 효율적인 설계방법을 제안하였다.

Abstract Game object inheritance and multiple components allow for visualization of system architecture, good code reuse, and fast prototyping. On the other hand, objects are more likely to rely on high latency between game objects and components, static casts, and lots of references to things like null pointers. Therefore, It is important to design a game in such a way so that the dependency of objects on multiple classes could be reduced and existing codes could be reused. Therefore, we designed the game to make the classes more modular by applying Dependency Injection and the design patterns proposed by the Gang of Four. Since these dependencies are attributes of the game object and the injection occurs only in the initialization pass, there is little performance degradation or performance penalty in the game loop. Therefore, this paper proposed an efficient design method to effectively reuse APIs in the design and implementation of survival games.

Key Words : Dependency Injection, Design Pattern, Application Programming Interface, Unity

Ⅰ. 서 론

소프트웨어 개발은 비즈니스 요구 사항을 충족하고, 애플리케이션의 유효성과 유연성을 보장하는 과정이다. 게임 설계는 객체와 기능의 복잡한 조합과 상호작용을 포함한다. 따라서 게임 개발은 재미를 위한 사양과 로직

*정회원, 중부대학교 게임소프트웨어학과
**종신회원, 청운대학교 컴퓨터공학과 (교신저자)
접수일자 2023년 6월 15일, 수정완료 2023년 7월 15일
게재확정일자 2023년 8월 4일

Received: 15 June, 2023 / Revised: 15 July, 2023 /
Accepted: 4 August, 2023
*Corresponding Author: lionel@chungwoon..ac.kr
Dept. of Computer Science, Chungwoon Univ, Incheon, Korea

을 찾고 변경하는 어려운 과정이다. 출시 후에도 게임로직을 변경할 수 있으나 비즈니스와 엔지니어링 문제가 생길 수 있다.

모듈성(Modularity)은 클래스 간의 관계를 쉽게 변경할 수 있도록 설계하는 것이다. 클래스가 서로 직접 의존하면 결합도(Coupling)가 높고 응집력(Cohesion)이 낮아진다. IoC(Inversion of Control)는 이벤트를 사용하여 제어 흐름을 반전시켜 클래스를 분리하는 디자인 패턴이다. 의존성 관리가 잘못되면 클래스 간의 결합도가 높아져서 변경에 취약하고 경직적이 된다. 이는 모듈성과 유지보수성을 낮추고, 버그와 오류를 높인다[1,2].

기존의 경우에 몬스터를 공격해야 된다면 해당 몬스터를 참조해야 되고, 시간을 알려면 시간매니져을 참조해야 되고, 퀘스트를 하려면 퀘스트매니져을 참조해야 되고, 길드 원들의 정보를 볼 수 있으려면 길드 원 관리객체를 알아야하고 그리고 총알을 발사하면 총알을 참조해야 된다. 이와 같이 플레이어 객체가 여러 가지 객체에 대해 너무 높은 의존도(Dependency)를 가지고 있다. 이는 여러 클래스간의 결합도를 증가기켜 유지보수에 따른 많은 문제가 발생되어 효율적이라고 볼 수 없다[3,4].

본 논문에서는 Unity3D 환경에서 의존성 주입 프레임워크(Dependency Injection Frameworks)인 Zenject과 SOLID 원칙을 사용하여 Unity3D 서바이벌 게임을 위한 API를 설계하고 제작하였다. 유지보수 과정에서 게임의 변경 및 확장이 수월하도록 느슨하게 결합된 게임 구조를 만드는 것이다. 그래서 게임의 로직이 추가적으로 변경이 있을 시에 효과적으로 재사용 할 수 있음을 보인다.

II. 의존성 주입 및 설계패턴 관련연구

1. Unity3D의 참조와 싱글톤(Singleton) 패턴

Unity3D는 엔티티에 컴포넌트를 연결하여 엔티티에 새로운 동작을 추가하는 엔티티 프레임워크를 기반하고, 의존성 처리를 위한 툴 세트를 많이 제공하지 않는다. 또한 MonoBehaviour 클래스가 인터페이스를 구현하고 다른 MonoBehaviour 클래스가 해당 인터페이스에 종속되어 있어도 인스펙터(Inspector)가 인터페이스를 표시할 수는 없다. 따라서 의존성 탐색에 싱글톤 패턴과 GameObject.Find(), FindObjectOfType() 메서드를 제공하고 있다.

GameObject.Find 메서드는 장면(Scene)에서 오브젝트를 이름으로 찾는데 같은 이름이 있으면 첫 번째 것을 반환하고 문자열 이름은 런타임 오류를 발생시킬 수 있다. 또한 찾은 객체를 GetComponent 메서드로 패치해서 성능이 떨어지기 마련이다. 또한 FindObjectOfType 메서드는 장면 계층 구조에서 찾은 첫 번째 오브젝트를 반환한다. Object 클래스에서 파생된 유형만 허용하므로 특정한 인스턴스를 구현한 클래스의 인스턴스는 찾을 수 없다. 이 두 가지 방법 모두 런타임 중에 외부 요인으로 인해 반환 속성이 달라질 수 있어서 의존성 관리에 안정성이 떨어진다. 싱글톤 패턴은 클래스의 유일한 객체를 정적 필드에 저장하고 공용 정적 프로퍼티로 제공하기 때문에 전역 접근을 가능하게 하지만 모듈성, 테스트 용이성, 유연성에 문제가 있다.

2. 의존성 주입

의존성 주입은 모듈식 코드를 달성하기 위한 패턴으로 클라이언트 클래스가 의존성을 인스턴스화하거나 가져오지 않고 외부에서 생성자의 매개변수를 통하여 클라이언트 클래스에 의존성을 주입한다. 클라이언트 클래스는 특정 클래스에 직접 의존하지 않고 IServer 인터페이스에 의존하고, 인터페이스 추상화를 사용하면 클라이언트 클래스는 인터페이스의 구현 세부 사항을 인지 할 필요가 없다. 또한 제어의 반전을 이용하면 프로그램의 흐름 제어를 외부 라이브러리에 맡기고 작업을 구현하는 방식과 작업 수행 자체가 분리된다. 이렇게 하면 모듈을 제작할 때, 모듈과 외부 프로그램의 결합에 대해 고민할 필요 없이 모듈의 목적에 집중할 수 있으며, 모듈을 바꾸어도 다른 시스템에 부작용을 일으키지 않는다. 또한 의존성 주입과 같은 기법을 통해 의존하는 오브젝트를 인식하지 않고 유연하게 사용할 수 있다. 결국 인터페이스 추상화와 의존성 주입은 컴포넌트 구현의 교체와 확장, 테스트의 안정성과 유연성을 증진시킨다. 의존성 주입 컨테이너는 런타임에 유형 결정을 가능하게 한다.

3. 의존성 주입 프레임워크

의존성 주입 프레임워크인 IoC 컨테이너 또는 DI 컨테이너는 객체의 의존성을 다른 객체에 주입하는 소프트웨어이다. 의존성은 다양한 바인딩을 설정하여 컨테이너를 구성함으로써 결정되기 때문에 클래스가 인터페이스에 종속될 경우에 바인딩은 해당 의존성을 충족하기 위해 주입되는 인스턴스를 정의하게 된다[5,6,7].

Unity3D를 대상으로 하는 몇 가지 DI 컨테이너가 있

는데 Unity3D 전용은 아니지만, 대부분은 Sebastian Mandala가 수행한 연구를 기반으로 의존성을 증명하기 위한 것이었다. 모든 프레임워크는 서로 매우 유사하며 MonoBehaviour 스크립트 주입을 지원하고 모두 작업은 동일하고 기능과 문법만 약간 다르다.

Zenject은 특히 Unity3D를 대상으로 하는 의존성 주입 컨테이너로서 Niantic Labs의 포켓몬 고와 같은 Android 및 iOS용 모바일 게임에서 Zenject을 사용하고 있다. Zenject는 Update(), LateUpdate(), FixeddUpadte() 라는 MonoBehaviour 클래스 기능을 일반 클래스로 가져올 수 있는 ITickable, ILateTickable, IFixedTickable 인터페이스를 제공한다. MonoBehavior를 상속하는 클래스에는 생성자가 없지만, Zenject 프레임워크의 [Inject] 속성을 사용하면 종속성을 클래스에 주입할 수 있다.

또한 객체에 대한 초기화 로직을 생성하는 기능으로 IInitializable 인터페이스가 있다. 초기화는 객체 그래프 생성 도중에 발생하여 문제를 일으킬 수 있으므로 생성자에서 실행은 불가하여 다른 인터페이스와 동일한 방식으로 사용할 수 있다. 전체 오브젝트 그래프가 생성된 후에 초기화가 실행된다. 추가 기능으로 장면이 변경되거나 애플리케이션이 닫히거나 객체가 소멸된 후 C#의 IDisposable 인터페이스를 이용하여 메모리를 정리할 수도 있다[8,9].

또한 생성자 주입은 이식성과 순환 의존성이 좋지만 Unity3D의 MonoBehaviour에는 적합하지 않다. 따라서 메서드 주입이 MonoBehaviour에 권장되며 IoC 컨테이너와 함께 사용되고, 필드 또는 속성 주입은 의존성이 많으면 모듈성이 떨어진다. 그러나 의존성 주입을 과도하게 사용하면 클래스 간의 모듈성이 떨어질 수 있고, 클래스와 인터페이스의 구조를 개선하고 SOLID 원칙을 적용하여 의존성 주입을 효과적으로 활용해야 한다[10,11,12].

Ⅲ. API 설계 및 구현

1. 게임 초기화

의존성 주입이 주입된 장면을 통하여 SceneContext의 GameInstaller가 MonoInstaller, GameSettingsInstaller가 ScriptableObjectInstaller에 등록되어 일부 매개변수를 보관하고 여러 가지의 패턴을 준비하는 데 사용된다. GameInstaller에서는 그림 1과 같이 EnemySpawner, EnemyFacade, Bullet, Explosion, Audio플레이어, Game Restart등의 바인딩(Binding)을 통한 의존성을 주입한다.

Enemy, Bullet와 Explosion은 유니티의 Instantiate()을 이용하지 않고, Factory를 기반으로 메모리 풀을 이용하여 객체를 동적으로 관리한다. 플레이어에서도 입력, 이동, 피해, 방향과 슈팅의 의존성을 주입하고, Enemy의 경우에 상태처리기와 더불어 평소상태, 공격상태, 이동상태, 사망과 회전의 처리기를 주입한다.

```
using Zenject;
using UnityEngine;
public class GameInstaller : MonoInstaller{
Container.BindInterfacesAndSelfTo<EnemySpawner>().AsSingle();
Container.BindFactory<float,float,EnemyFacade,EnemyFacade.Factory>()
.FromPoolableMemoryPool<float,float,EnemyFacade,
                    EnemyFacadePool>(poolBinder => poolBinder
.WithInitialSize(5)
.FromSubContainerResolve()
.ByNewPrefabInstaller<EnemyInstaller>(_settings.EnemyFacadePrefab)
.UnderTransformGroup("Enemies"));
Container.BindFactory<float, float, BulletTypes, Bullet, Bullet.Factory>()
.·····
Container.BindFactory<Explosion, Explosion.Factory>()
.·····
}
```

그림 1. GameInstaller의 스크립트
Fig. 1. GameInstaller's script

2. Facade 및 Adapter 패턴

Facade 패턴을 적용하여 적의 상태, 위치, 속도, 죽음, 생성과 플레이어의 위치, 회전, 사망을 효율적으로 관리할 수 있다. 런타임에 스파이크가 발생하지 않도록 5명의 적을 생성하고 씬 계층 구조의 루트에서 EnemyObject 아래에 각각의 적을 배치한다. 또한 믹스인(Mixin)이라는 개념을 사용하여 상속(Inheritance)되는 대신에 구성(Composition)을 사용하여 기능 단위가 별도의 클래스로 생성되고 이러한 클래스의 인스턴스는 부모 클래스에서 사용되고, 이벤트를 수신하여 특정 오브젝트가 언제 장면에서 게임 객체가 완전히 제거되는지 알 수 있다.

클래스에 대한 인터페이스 구현은 각 메서드, 프로퍼티 및 이벤트를 해당 의존성으로 전달한다. 이는 제어의 반전(IoC)을 지향하게 되며 부모 클래스에 대한 많은 상용구(boilerplate) 코드를 생성할 수 있지만, 클래스의 동작 논리를 변경할 수 있는 기능도 제공한다. 그림 2는 믹스인의 부모 클래스(EnemyFacade)와 구현 가능한

IPoolable<float, float, IMemoryPool>, IDisposable를 보여준다. 또한 Player 와 Enemy의 RigidBody에 대한 힘(Force)이나 토크(Torque) 그리고 오디오의 Play() 메서드에는 Adapter패턴을 사용하였다.

```
public class EnemyFacade : MonoBehaviour, IPoolable<float,
float, IMemoryPool>, IDisposable
{
   private EnemyView _view;
   private EnemyTunables _tunables;
   private EnemyDeathHandler _deathHandler;
   private EnemyStateManager _stateManager;
   private EnemyRegistry _registry;
   private IMemoryPool _pool;

   [Inject]
   public void Construct(
      EnemyView view,
      EnemyTunables tunables,
      EnemyDeathHandler deathHandler,
      EnemyStateManager stateManager,
      EnemyRegistry registry)
   {
       _view = view;
       _tunables = tunables;
       _deathHandler = deathHandler;
       _stateManager = stateManager;
       _registry = registry;
   }
   …
   public void OnSpawned(float accuracy, float speed,
IMemoryPool pool)
   {
       _pool = pool;
       _tunables.Accuracy = accuracy;
       _tunables.Speed = speed;
       _registry.AddEnemy(this);
   }
   …
}
```

그림 2. EnemyFacade 클래스
Fig. 2. EnemyFacade class

3. State Pattern

EnemyStates에는 Idle, Attack과 Follow의 상태에 따은 Factory을 활용하여 객체를 동적으로 생성하였다. 또한 IEnamyState의 EnterState(), ExitState(), Update()와 FixedUpdate() 인터페이스를 가지고 고 다형성을 구현하여 인터페이스 분리 원칙(Interface Segregation Principle)을 준수한다. EnemyStateManager 클래스의 ChangeState() 메서드를 통하여 요청된 게임 상태를 StateHandler를 통하여 변환하고 있다.

특히 Enemy의 Follow동작을 AI(Artificial Intelligence) 기반에서 실행하기 위하여 EnemyStateController를 통하여 Moving_looking_for_target(), Chasing_target(), Backing _up_looking_for_target(), Stopped_turning _left(), Stopped_ turning_right(), Paused_looking_ for_target()의 상태 제어를 위하여 IsObstacleAhead(), TurnTowardTarget(), LookAroundFor()와 CanSee() 메서드를 적용하였다. 모두 Target에 해당하는 Player 객체가 의존성 주입에 기반하고 있다.

4. Observer Pattern

Player와 Enemy가 죽는 경우를 감지하여 적절한 조치를 취해야 하는 경우에 클래스간의 결합을 줄이기 위하여 그림 3과 같이 Zenject의 Signals을 이용한다. 신호 버스에서 신호를 직접 구독할 수 있을 뿐만 아니라 SignalBus.Subscribe을 통해서 핸들링 클래스에 신호를 직접 바인딩할 수도 있다. 이렇게 하면 이벤트를 이용하는 것보다 더 효과적이고 구독자나 발행자 어느 쪽에서도 감지가 되지 않고 독립성을 유지할 수 있다.

```
public class PlayerDiedSignalObserver
{
    public void OnPlayerDied()
    {
        Debug.Log("Fired PlayerDiedSignal");
    }
}
public class GameSignalsInstaller :
Installer<GameSignalsInstaller>
{
    public override void InstallBindings()
    {
        SignalBusInstaller.Install(Container);
        Container.DeclareSignal<EnemyKilledSignal>();
        Container.DeclareSignal<PlayerDiedSignal>();
        Container.BindSignal<PlayerDiedSignal>()
                        .ToMethod<PlayerDiedSignalObserver>
                         (x => x.OnPlayerDied).FromNew();
        Container.BindSignal<EnemyKilledSignal>().ToMethod(
                         () => Debug.Log("Fired
EnemyKilledSignal"));
        }
    }
}
```

그림 3. GameSignalsInstaller 스크립트
Fig. 3. GameSignalsInstaller script

5. User Interface

사용자 인터페이스(UI)는 매우 단순하고 작으며 복잡한 부분이 포함되어 있지 않다. 하지만 UI는 외부 레이어로서 제어의 반전을 보여주고 있다. 즉, UI만이 내부 컴포넌트인 healthLabelBound, killLabelBounds에 대해 OnEnemyKilledSignal()과 PlayerDiedSignal()을 구독하여 자체적으로 업데이트된다.

Ⅳ. 서바이벌 게임의 구현

본 절에서는 제안된 기법을 통해서 설계된 서바이벌 게임의 기본 템플릿으로 한 명의 Player가 경기장을 돌아다니며 많은 Enemy와 싸우는 액션 게임의 구현 환경을 설명하고 구현된 게임의 테스트 예를 보여준다. 그림 4는 Zenject을 통한 의존성 주입과 디자인 패턴 원리를 적용한 게임의 전체적인 구조를 나타낸다. 특히 EnemyStateController에서는 AI기능에 초점을 맞춘 기본 상태 머신(State Machine)이다. 움직이는 물체와 같이 회전하고 앞뒤로 움직이며 장애물을 피하고 Player를 찾아가는 동작을 제어할 수 있다.

그림 5는 게임이 시작된 직후의 모습을 보여준다. PlayerDamageHandler()와 PlayerDiedSignalObserver 클래스의 PlayerDiedSignal에 의하여 업데이트되는 Player의 개수와 체력의 상태를 보여준다.

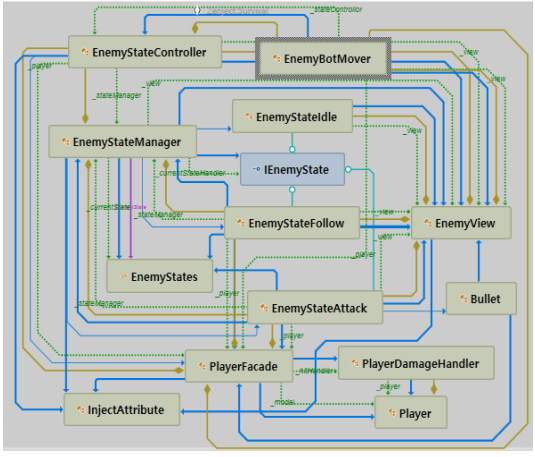

그림 4. 게임의 구조
Fig. 4. Game architecture

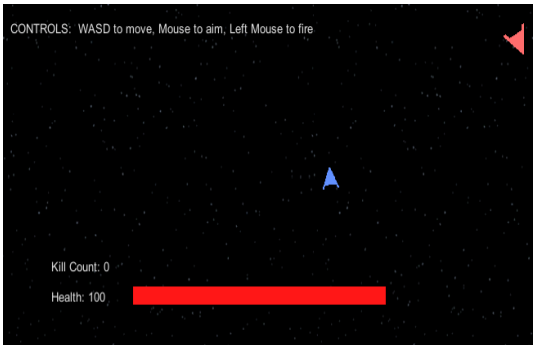

그림 5. 게임 시작 직후의 스크린 샷
Fig. 5. Screenshot right after the game starts

그림 6에서는 Player는 총알을 발사하고 이동 버튼으로 방사형태로 이동할 수 있고, Enemy의 경우도 Player의 추적을 위하여 EnemyFollow의 강화된 추적제어를 통하여 속도, 위치, 회전을 이용하여 Player를 공격한다. 특히 회전에 의한 총알의 면이 그림 6과 같이 여러 가지의 상태를 가진다는 것을 알 수 있다. Factory를 이용하여 Enemy, Bullet와 Explosion의 동적 객체를 관리하기 위하여 유니티의 메모리 풀을 이용하였다.

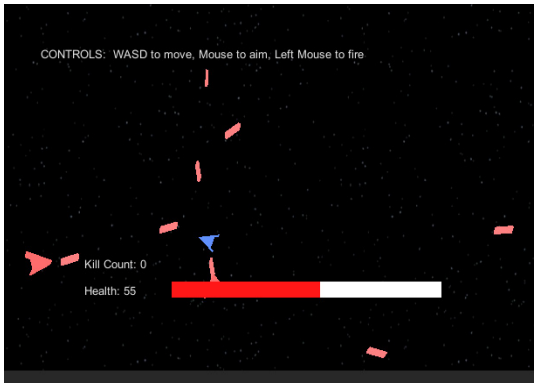

그림 6. 게임 플레이에서 탄환 회전의 스크린 샷
Fig. 6. Screenshot of bullet rotation while playing

Ⅴ. 결 론

본 논문에서는 의존성 주입과 디자인 패턴을 적용하여 구현한 서바이벌 게임을 통하여 구현된 게임의 로직을 수정하거나 새로운 기능을 추가 할 경우에 적용 가능한 설계 기법을 제안하였다. 또한 의존성 주입을 통한 게임의 API 설계에서 SOLID원리를 지향한 디자인 패턴을 적용하였고 생성패턴, 구조패턴과 행위패턴등이 적용 가능함을 보였다.

의존성 주입과 디자인 패턴의 적용을 통하여 데이터와 로직을 분리하였고 기존기능의 제거나 새로운 기능의 추가에서도 클래스간의 결합이 적어서 API의 재사용성의 유지보수가 양호함을 보였다. 따라서 데이터(Model)와 로직(Controller)을 분리해주는 소위 MVC 패턴과 같은 방법을 사용하는 것이 현명하다. 따라서 유니티의 경우에 모델의 변화를 뷰가 감지해야 되고, 뷰에서 발생하는 이벤트를 컨트롤러에게 전달해야하며 각각의 컨트롤러끼리는 통신이 원활해야 하는데 이러한 조건을 충족시키는 방법이 의존성 주입이다.

결론적으로 게임 소프트웨어의 설계에 의존성 주입을 사용하고 디자인 패턴을 적용하는 것이 시스템의 구현에서도 유지 보수성과 확장성을 향상시키는 결과를 보장한다고 사료된다.

References

[1] Ancheta, F. 2015. SOLID Review: Liskov Substitution Principle. Blog post on Runtime Era website, 5 March 2015. Accessed on 13 December 2016.
DOI: https://doi.org/10.1201/9781315148076-16

[2] Chambers, J. 2016. Upgraded Mono/.Net in Editor on 5.5.0b9. Forum post on Unity3D website's forums, 27 October 2016. Accessed on 13 February 2017.
DOI: https://doi.org/10.2991/ifmca-16.2017.55

[3] Jenkov, J. 2014. Dependency Injection Benefits. Article on Jenkov's website, 26 May 2014. Accessed on 27 February 2017.
DOI: https://doi.org/10.1007/s12480-014-0041-1

[4] Martin, R. 2014. SOLID Principles of Object Oriented & Agile Design. Presentation at Yale School of Management. Watchable on Youtube. Accessed on 25 February 2017.
DOI: https://doi.org/10.1007/978-1-4842-8245-8_1

[5] Mandalà, S. 2012a. Inversion of Control with Unity3D. Blog post on Seba's Lab website, 30 September 2012. Accessed on 12 December 2016.
DOI: https://doi.org/10.1145/2619195.2656311

[6] Mandalà, S. 2012b. Inversion of Control with Unity3D. Blog post on Seba's Lab website, 14 November 2012. Accessed on 2 March 2017.
DOI: https://doi.org/10.1007/978-1-4302-0623-1_4

[7] Martin, R. 2014. SOLID Principles of Object Oriented & Agile Design. Presentation at Yale School of Management. Watchable on Youtube. Accessed on 25 February 2017.
DOI: https://doi.org/10.1007/978-1-4842-4366-4_1

[8] Skrchevski, B. 2015. High Cohesion, Loose Coupling. Post at The Bojan's Blog 8.4.2015. Accessed on 12 December 2016.
DOI: https://doi.org/10.1002/9781118785317.weom110183

[9] Zenject Documentation. 2016. File on Zenject's GitHub repository, 2 December 2016. Accessed on 3 January 2017.
DOI: http://dx.doi.org/10.18419/opus-10459

[10] "Seong Oun Hwang", Development of Rhythm Game based on Object-Oriented Software Engineering, JIIBC, vol.8, no.4, pp. 105-110, 2008.
DOI: http://dx.doi.org/10.18419/opus-10459

[11] "Gil Hwan Heo, Seung Young Lee", A Study on Development of Integrated Logistics Support with Virtual Reality, JKAIS, v.19, no.12, pp.90-98, 2018.
DOI: http://dx.doi.org/10.5762/KAIS.2018.19.12.90

[12] "Hong Jin Park", Trend Analysis of Korea Papers in the Fields of Artificial Intelligence Machine Learning and Deep Learning, KIIECT, vol.13, no.4, pp. 283-292, 2020.
DOI: http://dx.doi.org/10.5762/KAIS.2018.19.10.424

저 자 소 개

박 인 규(정회원)

- 제10권 5호 참조
- 현 중부대학교 게임S/W학과 교수
- 주관심분야 : 데이터마이닝, 퍼지집합, 러프집합

최 규 석(종신회원)

- 제9권 6호 참조
- 1991~1996 : (주)SK텔레콤 중앙연구원 책임연구원
- 1997 ~ 현재 : 청운대학교 컴퓨터공학과 교수
- 주관심분야 : 인공지능, 이동통신, 빅데이터, ITS

※ 본 논문은 2023학년도 청운대학교 학술연구조성비에 의하여 지원되었음.