

# Patrones de Diseño

## Entity-Component-System (Parte II)

TPV2  
Samir Genaim

# Objetivos

- ✦ Ya tenemos el primer diseño de juego basado en Entity-Component (sin System de momento)
- ✦ Tenemos control sobre la gestión de la memoria, addComponent y addEntity crean y borran los objetos correspondientes, y queremos aprovecharlo!
- ✦ Queremos mejorar el uso de la memoria dinámica, permitiendo el uso de ObjectPool, ObjectFactor, etc.
- ✦ Lo vamos a hacer usando una abstracción sobre la creación y destrucción de objetos — Factorías de Entidades, Componentes, etc.
- ✦ En el nuevo diseño, el usuario puede decir que factoría usar para crear e componente, la entidad, etc.

# Factorías

- ✦ Ya hemos visto factorías en varios contextos
- ✦ En el contexto actual, factoría de un tipo T es una clase con:
  - ✓ métodos estáticos con el nombre **construct** que devuelven un puntero a un object de tipo T - cada método recibe información distinta, como se fueran constructoras ...
  - ✓ un método con el nombre **destroy** que recibe un puntero (creado anteriormente con la factoría) y destruye el objeto correspondiente (lo borra, lo devuelve al pool, etc).
- ★ Cómo se crea y se destruye el objeto no nos interesa

# Factoría usando new y delete

```
template<typename T>
class DefFactory {
public:
    DefFactory() {}
    virtual ~DefFactory() {}
```

```
    template<typename ...Targs>
    inline static T* construct(Targs&& ...args) {
        return new T(std::forward<Targs>(args)...);
    }
```

```
    inline static void destroy(T* p) {
        delete p;
    }
};
```

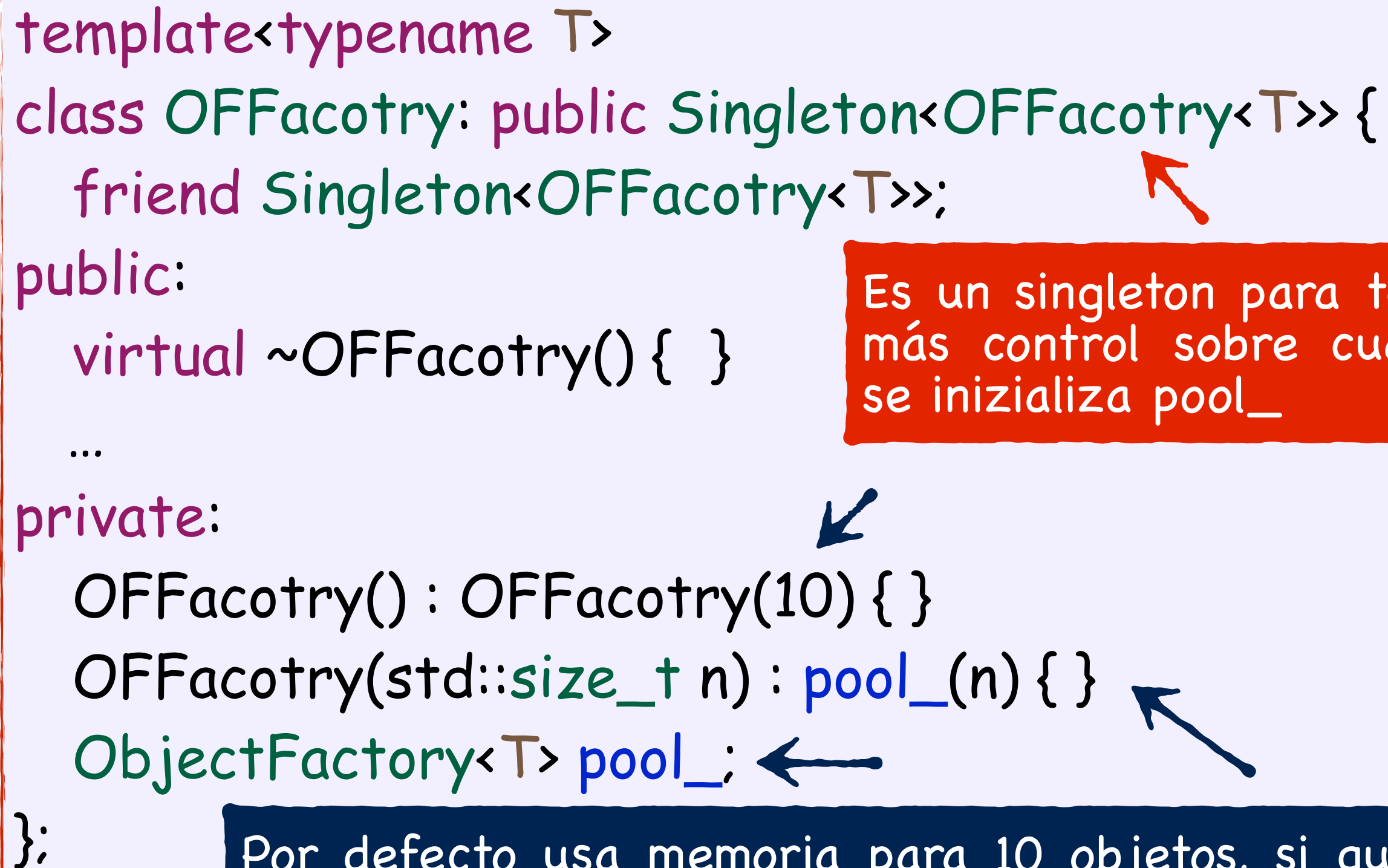
Una factoría muy sencilla,  
simplemente usa new y  
delete



pasa los argumentos  
a la constructora

# Factoría usando ObjectFactory

```
template<typename T>
class OFFacotry: public Singleton<OFFacotry<T>> {
    friend Singleton<OFFacotry<T>>;
public:
    virtual ~OFFacotry() { }
    ...
private:
    OFFacotry() : OFFacotry(10) { }
    OFFacotry(std::size_t n) : pool_(n) { }
    ObjectFactory<T> pool_;
};
```



Es un singleton para tener más control sobre cuando se inicializa pool\_

Por defecto usa memoria para 10 objetos, si queremos mas hay que inicializar al principio del juego llamando a init(n) (ver Singleton.h)

# Factoría usando ObjectFactory

```
template<typename ...Targs>
inline static T* construct(Targs &&...args) {
    return OFFacotry<T>::instance()->construct_(std::forward<Targs>(args)...);
}

inline static void destroy(T *p) {
    OFFacotry<T>::instance()->destroy_(p);
}

template<typename ...Targs>
inline static T* construct_(Targs &&...args) {
    pool_.construct(std::forward<Targs>(args)...);
}

inline static void destroy_(T *p) {
    pool_.destroy(p);
}
```


construct y destroy llaman a construct\_ y destroy\_ de la instancia

construct\_ y destroy\_ de la instancia usan las del pool\_

Si no se define como singleton, se pueden eliminar construct\_ y destroy\_ (copiando su código a construct y destroy)

# Factoría usando ObjectPool

```
class StarsPool: public Singleton<StarsPool> {  
    friend Singleton<StarsPool>;  
public:  
    virtual ~StarsPool() {}  
    ...  
private:  
    StarsPool() : StarsPool(10) {}  
    StarsPool(std::size_t n) : pool_(n) {  
        for (Entity *e : pool_.getPool()) {  
            e->addComponent<Transform>();  
            ...  
        }  
    }  
  
    ObjectPool<Entity> pool_;  
};
```



Es un singleton para tener más control sobre cuando se inicializa pool\_



La constructora inicializa los objetos del pool, p.ej., añadiendo componentes



# Factoría usando ObjectPool

```
inline static Entity* construct(...) ...  
inline static void destroy(...) ...
```

llaman a construct\_ y  
destroy\_ exactamente  
como antes

```
inline Entity* construct_(double x, double y, ...) {  
    Entity *e = pool_.getObj();  
    if (e != nullptr) {  
        Transform *tr = GETCMP1_(Transform);  
        tr->setPos(x, y);  
        e->setActive(true);  
        ...  
    }  
    return e;  
}
```

construct\_ pide objetos al  
pool y los (re)inicializa. No  
crea objetos ...

```
inline void destroy_(Entity *p) {  
    pool_.relObj(p);  
}
```


destroy\_ devuelve el  
objeto al pool




# addComponent con Factorías

```
using uptr_cmp =  std::unique_ptr<Component, std::function<void(Component*)>>;
```

Los parámetros de tipo incluyen la factoría (por defecto DefFactory<T>)

```
template<typename T,  
        typename FT=DefFactory<T>,   
        typename ...Targs>
```

```
inline T* addComponent(Targs&&...args) {  
    T *c = FT::construct(std::forward<Targs>(args)...);  
    uptr_cmp uPtr(c, [](Component*p) {  
        FT::destroy(static_cast<T>(p));   
    });  
    ...  
}
```

Usa la factoría para construir

el unique\_ptr ejecute este deleter, y así devuelve el objeto a la factoría

# addComponent - ejemplo de uso

Como antes, crea un Transform usando la factoría por defecto ...



```
e->addComponent<Transform>();
```

```
e->addComponent<Transform, OFFactory<Transform>>();
```



Crea un Transform usando la factoría OFFactory<Transform>

Si cambiamos la factoría por defecto a OFFactory<T>, siempre usamos ObjectFactory, sería mejor porque reduce las llamadas a new y delete y los componentes estarán en memoria contigua

# addEntity con Factorías

Usamos unique\_ptr con custom deleter

```
using uptr_cmp =  
std::unique_ptr<Entity, std::function<void(Entity*)>>;
```

Los parámetros de tipo incluyen la factoría (por defecto DefFactory<T>)

```
template<typename FT = DefFactory<Entity>, typename ...Targs>  
inline Entity* addEntity(Targs &&...args) {  
    Entity *e = FT::construct(std::forward<Targs>(args)...);  
    storeEntity(uptr_cmp(e, [](Entity *p) { FT::destroy(p); }));  
    return e;  
}
```

```
void EntityManager::storeEntity(uptr_ent&& e) {  
    e->setEntityMngr(this);  
    ents_.push_back(std::move(e));  
}
```

Usa la factoría para construir y el deleter del unique\_ptr lo devuelve a la factoría

# addEntity - ejemplos de uso (I)

Se puede seguir usando como antes, se usa la factoría por defecto.

```
Entity *e = entityManager->addEntity();
```

...

Si cambiamos la factoría por defecto a OFFactory<T>, siempre usamos ObjectFactory, sería mejor porque reduce las llamadas a new y delete y las entidades estarán en memoria contigua

# addEntity - ejemplos de uso (II)

Al pulsar una tecla añadimos 10 estrellas al juego (las estrellas mueren cuando se cumpla alguna condición)

```
void GameCtrl::update() {  
    if (InputHandler::instance()->keyDownEvent()) {  
        for( int i=0; i<10; i++) {  
            double x = ...;  
            double y = ...;  
            ...  
            entity_->getEntityMngr()->addEntity<StarsPool>(x,y,...);  
        }  
    }  
}
```

# Resumen

- ◆ Hemos mejorado la gestión de la memoria permitiendo el uso de factorías
- ◆ El uso de factorías nos permite usar ObjectPool y ObjectFactory
- ◆ En el caso de factorías basadas en ObjectPool o ObjectFactory, mejor inicializar al principio del juego con el tamaño adecuado

```
OFFactory<Entity>::init(100);  
OFFactory<Transform>::init(100);  
StarsPool::init(50);
```

```
...
```