

Patrones de Diseño

Component
(primer paso hacia componentes)

TPV2
Samir Genaim

El Problema: I

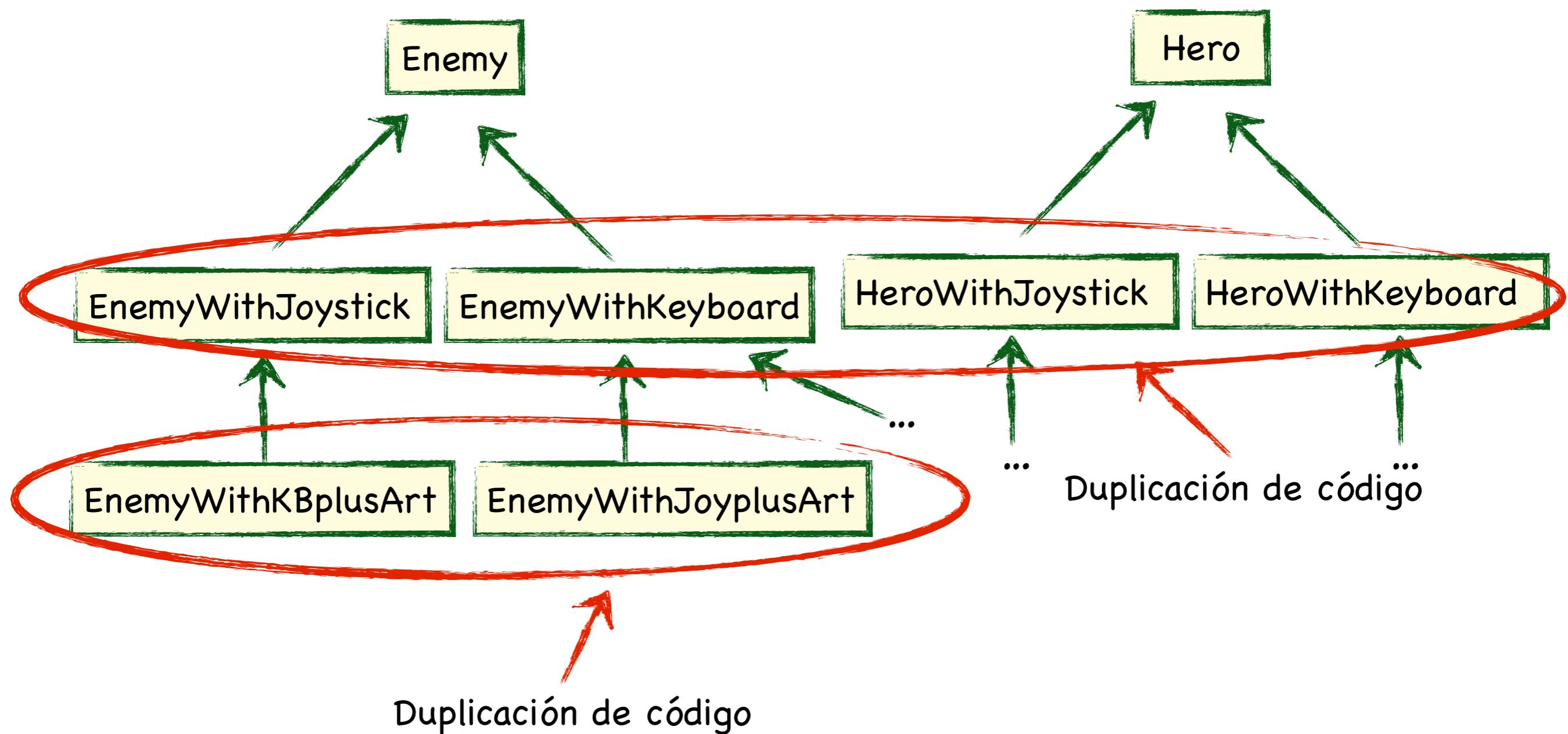
Una entidad de un video juego (entity) necesita “tocar” partes (dominios) conceptualmente distintos del juego como Entrada, Física, Gráfica, etc.

Mantener el código de varios dominios en la misma clase nos puede complicar el código, podemos acabar con código que refiere a muchos dominios y eso require que el programador sepa todo sobre esos dominios ...

Principio de diseño: mantener diferentes dominios en un programa separados uno del otro ...

El Problema: II

Usar herencia para definir objetos de juego con nuevos comportamientos nos puede obligar a duplicar el código, para evitarlo tenemos que cambiar la jerarquía de clases ... algo que no se debe hacer ...



¿Qué es el Patrón Component?

Es un patrón que nos ayuda a separar las responsabilidades de una entidad en clases separadas, según el dominio de la responsabilidad ...

Transforma las entidades en contenedores plug-and-play de componentes, la semántica de una entidad es la de sus componentes ...

Un programador de comportamiento de Entrada puede no saber nada sobre la parte Gráfica ...

Podemos cambiar los componentes de una entidad durante el juego cambiando su comportamiento ...

Evitamos jerarquías de clases incensaríais para definir nuevas entidades con comportamientos distintos

Bjorn - un objeto de un juego

```
class Bjorn : public GameObject {  
public:  
    Bjorn(...) : GameObject(...), velocity_(0), x_(0), y_(0) {  
    }  
  
    void handleInput(...);  
    void update(...);  
    void render(...);  
  
private:  
    static const int WALK_ACC = 1;  
  
    Volume volume_;  
  
    ...  
    Sprite spriteStand_;  
    Sprite spriteWalkLeft_;  
    Sprite spriteWalkRight_;  
};
```

Posición, velocidad, etc., atributos de GameObject

reciben todo lo necesario para llevar a cabo su tarea ...

No siempre es necesario distinguir estos métodos, bastaría con el update() para actualizar y dibujar el objeto, eso depende del juego. En nuestro GameObject hemos decidido separarlos.

Otros atributos relacionados con la parte gráfica, etc.

La lógica de Bjorn ...

El código de Bjorn abarca a varios dominios ...

```
void Bjorn::update(...) {  
    x_ += velocity_;  
    Collisions::resolveCollision(volume_, x_, y_, velocity_);  
}
```

```
void Bjorn::handleInput(...) {  
    switch (Controller::getJoystickDirection()) {  
        case DIR_LEFT:  
            velocity_ -= WALK_ACC;  
            break;  
  
        case DIR_RIGHT:  
            velocity_ += WALK_ACC;  
            break;  
    }  
}
```

```
void Bjorn::render(...) {  
    Sprite* sprite = &spriteStand_;  
    if (velocity_ < 0) {  
        sprite = &spriteWalkLeft_;  
    } else if (velocity_ > 0) {  
        sprite = &spriteWalkRight_;  
    }  
  
    graphics.draw(*sprite, x_, y_);  
}
```

La Interfaz Component

```
class Component {  
public:  
    Component() {}  
    virtual ~Component() {}  
    virtual void update(Bjorn *bjorn, ...) = 0;  
};
```

- ◆ La clase (o mejor dicho interfaz) Component es una encapsulamiento de comportamientos. El método update define el comportamiento concrete
- ◆ No hay que confundir el nombre update con lo que tuvimos antes, se puede llamar doSomething – es costumbre llamarlo update)
- ◆ Se puede definir una interfaz para cada tipo comportamiento (entrada, gráfica, física, lo hacemos con unas sola interfaz

Bjorn - sólo un Container

```
class Bjorn : public GameObject {  
public:  
    Bjorn(..., Component* ic, Component* pc, Component* gc):  
        GameObject(...),  
        input_(ic), physics_(pc), graphics_(gc) {  
    ...  
}  
    void handleInput(...) { input_->update(this, ...); }  
    void update(...) { physics_->update(this, ...); }  
    void render(...) { graphics_->update(this, ...); }  
    ...  
private:  
    ...  
    Component* input_;  
    Component* physics_;  
    Component* graphics_;  
};
```

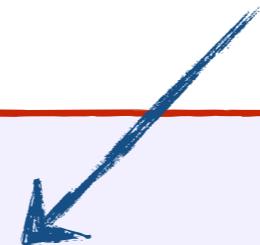
La constructora recibe los componentes

Simplemente delegan el trabajo a los componentes.

Atributos para los componentes

¿Quién es Bjorn?

Los componentes definen quién es Bjorn.



...

```
Bjorn* bjorn = new Bjorn( new SomeInputComp(...),  
                           new SomePhysicsComp(...),  
                           new SomeGraphicsComp(...) );
```

...

Bjorn - Componentes de Entrada

```
class JoystickInputComponent: public Component {  
public:
```

Joystick

```
class KeyboardInputComponent: public Component {  
public:
```

Keyboard

```
void update(Bjorn *bjorn, ...){  
    switch (Controller::getPressedKey()) {  
        case UP:  
            bjorn->setVel(bjorn->getVel() - WALK_ACCELERATION);  
            break;  
        case DOWN:  
            bjorn->setVel(bjorn->getVel() + WALK_ACCELERATION);  
            break;  
    }  
}
```

```
class DemoInputComponent: public Component {  
public:
```

```
void update(Bjorn *bjorn, ...){  
    // do some AI o move bjorn  
    ...  
}
```

```
};
```

Demo

Bjorn - Física

```
class BjornPhysicsComponent: public Component {  
public:  
    void update(Bjorn *bjorn, ...) {  
        // do some physics ...  
    }  
    ...  
};
```

Física 1

```
class BjornAdvPhysicsIComponent: public Component {  
public:  
    void update(Bjorn *bjorn, ...) {  
        // do some advanced physics ...  
    }  
    ...  
};
```

Física 2

Bjorn - Gráfica

```
class BjornGraphicsComponent: public Component {  
public:  
    void update(Bjorn *bjorn, ...) {  
        // do some graphics ...  
    }  
    ...  
};
```

Gráfica 1

```
class BjornAdvGraphicsIComponent: public Component {  
public:  
    void update(Bjorn *bjorn, ...) {  
        // do some advanced graphics ...  
    }  
    ...  
};
```

Gráfica 2

No sólo Bjorn ...

```
class Container: class GameObject {  
public:
```

Podemos generalizar el concepto a un Container

```
Container(..., Component* ic, Component* pc, Component* gc) :  
    GameObject(...),  
    input_(ic), physics_(pc), graphics_(gc) {  
}
```

```
void handleInput(...) {  
    input_->update(this, ...);  
}
```

```
void update(...) {  
    physics_->update(this, ...);  
}
```

```
void render(...) {  
    graphics_->update(this, ...);  
}
```

```
private:  
    Component* input_;  
    Component* physics_;  
    Component* graphics_;  
};
```

El constructor recibe los componentes

Simplemente delegan el trabajo a los componentes.

Atributos para los componentes

Interfaces de Componentes

```
class Component {  
public:  
    Component() {}  
    virtual ~Component() {}  
    virtual void update(Bjorn *bjorn, ...) = 0;  
};  
                                Container* c
```

Bjorn se ha desaparecido

No hace falta la clase Bjorn!!

```
GameObject* createBjorn() {  
    return new Container( new SomeInputComp(...),  
                         new SomePhysicsComp(...),  
                         new SomeGraphicsComp(...) );  
}
```



Se define en la
factoría

```
...  
GameObject* bjorn = createBjorn();  
...
```

Esto se puede hacer porque Bjorn no necesita más información de lo que tenemos en GameObject. Algunas veces necesitamos definir una clase heredando de Container para poder añadir atributos, etc. En ese caso los componentes normalmente usan `static_cast` para poder usar esos atributos — ver comunicación entre componentes

Cambiar Componentes

```
class Container: class GameObject {  
public:  
    ...  
    void setInputComponent(Component* ic) {  
        input_ = ic;  
    }  
    void setPhysicsComponent(Component* pc) {  
        physics_ = pc;  
    }  
    void setRenderComponent(Component* gc) {  
        graphics_ = gc;  
    }  
private:  
    Component* input_;  
    Component* physics_;  
    Component* graphics_;  
};
```

Se usa para cambiar componentes.
También para configurar el container en lugar de pasar los componentes a la constructora.

```
Container* bjorn = createBjorn();
```

```
...  
bjorn->setInputComponent( new JoystickInputComponent(...) );  
...
```

Varios componentes de cada tipo

```
class Container: class GameObject {  
public:  
    ...  
    void handleInput(...) {  
        for(Component* ic : input_) ic->update(this, ...);  
    }  
    void update(...) {  
        for(Component* pc : physics_) pc->update(this, ...);  
    }  
    void render(...) {  
        for(Component* gc : graphics_) gc->update(this, ...);  
    }  
};
```

delegar la petición a todos

```
void addIComponent(Component* ic) { input_.add(ic); }  
void addPComponent(Component* pc) { physics_.add(pc); }  
void addRComponent(Component* gc) { graphics_.add(gc); }
```

private:

```
vector<Component*> input_;  
vector<Component*> physics_;  
vector<Component*> graphics_;  
};
```

Usar un array/vector/list para cada tipo de componente

Configuración de Bjorn

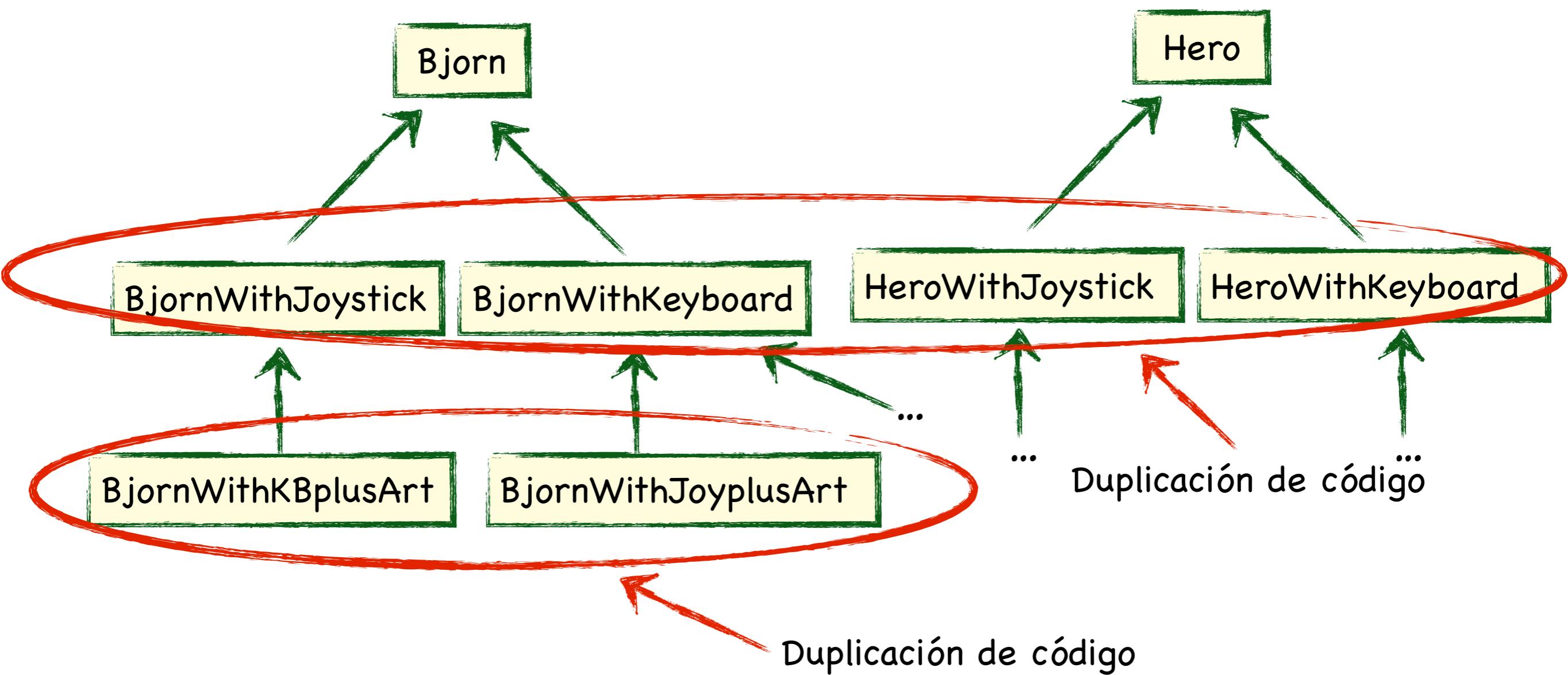
```
GameObject* createBjorn() {  
    GameObject* bjorn = new Container(...);  
  
    bjorn->addIComponet( new SomeInputComp1(...) );  
    bjorn->addIComponet( new SomeInputComp2(...) );  
    bjorn->addPomponet( new SomePhysicsComp1(...) );  
    bjorn->addPComponet( new SomePhysicsComp2(...) );  
    bjorn->addRComponet( new SomeGraphicsComp(...) );  
  
    ...  
  
    return bjorn;  
}
```

Interfaces de Componentes

```
class Component {  
protected:  
    Container* c_;  
public:  
    Component() {}  
    virtual ~Component() {}  
    void setContainer(Container* c) { c_ = c; };  
    virtual void init(...) = 0;  
    virtual void update(...) = 0;  
};
```

```
class Container: class GameObject {  
    ...  
    void handleInput(...) {  
        for(Component* ic : input_) ic->update(...);  
    }  
    ...  
    void addIComponent(Component* ic) {  
        input_.add(ic);  
        ic->setContainer(this);  
        ic->init(...);  
    }  
};
```

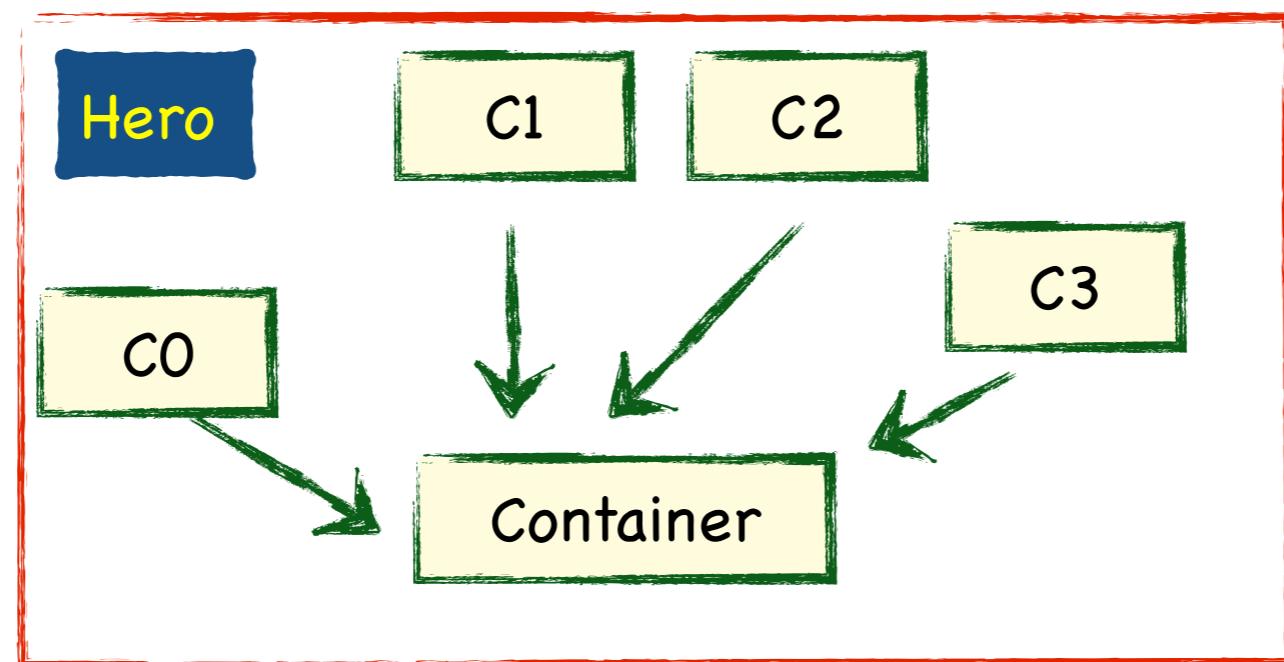
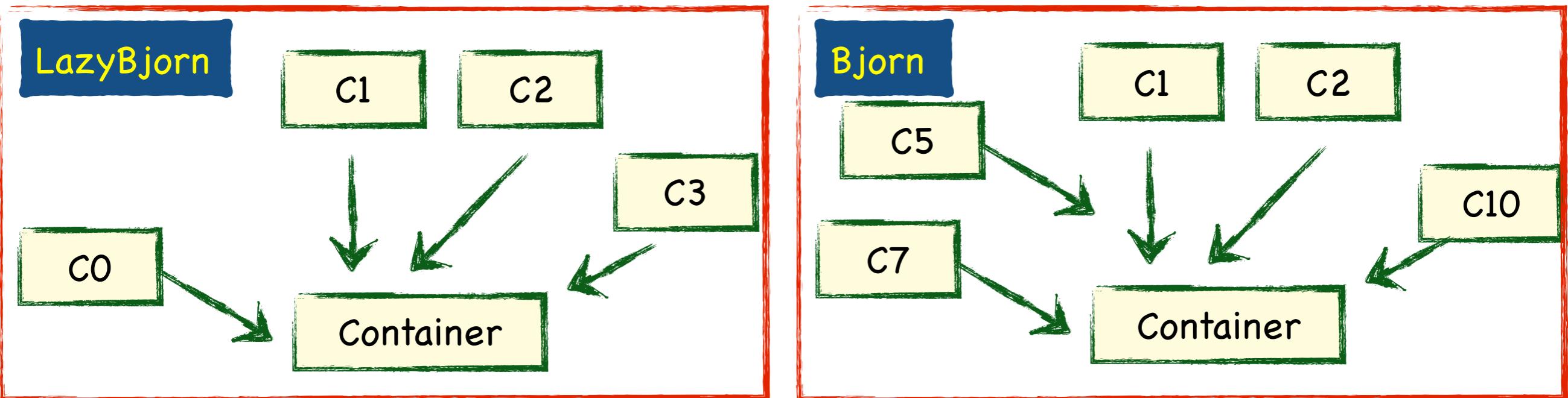
Composición vs. Herencia



Usar herencia para definir objetos de juego con nuevos comportamientos nos puede llevar a duplicar el código, para evitarlo tenemos que cambiamos la jerarquía de clases ... algo que no se debe hacer ...

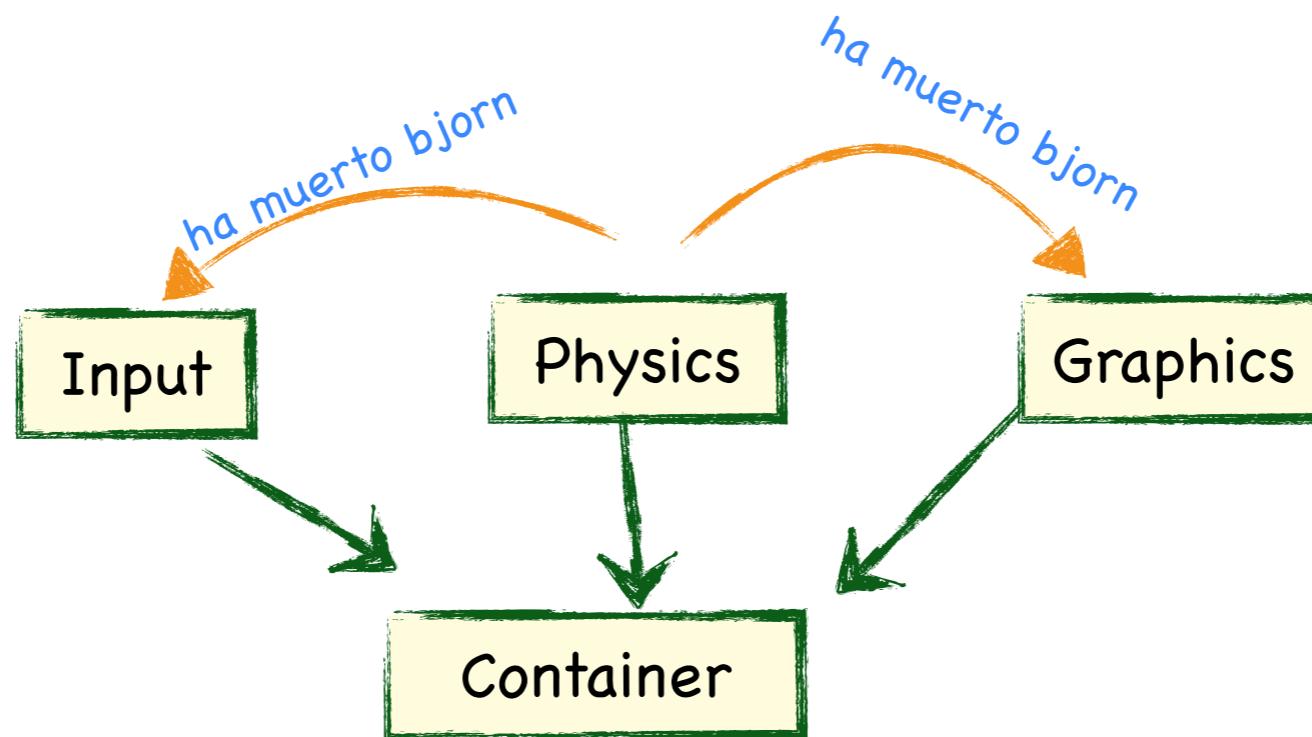
Composición (Has) vs. Herencia (Is)

Usando componentes, se puede definir varias objetos de juego de manera más fácil y sin duplicar el código ...



Comunicación entre Componentes

Muchos componentes necesitan compartir información entre ellos. Por ejemplo, el componente de física decide que Bjorn se muere, ¿cómo avisamos al componente de gráfica esa información para poder dibujar un Bjorn muerto?



Usando atributos del container

```
class Bjorn : public Container {  
public:  
    void setDead(bool dead);  
    void isDead() { return dead_; }  
private:  
    bool dead_;  
};
```

Bjorn es un Container con más información sobre su estado. Todos los componentes tienen acceso a esos atributos ...

```
class BjornGraphicsComponent: public Component {  
public:  
    void update(...) {  
        ... if ( static_cast<Bjorn*>(c_)->isDead() ) { ... } else { ... }  
    } ...  
};
```

```
class BjornPhysicsComponent: public Component {  
public:  
    void update(...) {  
        ... static_cast<Bjorn*>(c_)->setDead(true);  
    } ...  
};
```

Usando mensajes ...

```
class Component {  
public:  
    virtual Component() {}  
    virtual ~Component() {}  
    virtual void receive(const Message& m) {}  
    ...  
};
```

Un método para recibir mensajes. Por defecto no hace nada ...

```
class Container: public GameObject {  
public:  
    ...  
    void localSend(const Message& m) {  
        for(Component* ic : input_) ic->receive(m);  
        for(Component* pc : physics_) pc->receive(m);  
        for(Component* gc : graphics_) gc->receive(m);  
    }  
    ...  
};
```

Un método para enviar mensajes a todos los componentes ...

¿Qué es un mensaje?

```
enum MessageType {  
    BJORN_IS_HAPPY,  
    BJORN_IS_DEAD,  
    SHOOT  
};
```

```
struct Message {  
    Message(MessageType type) : type_(type) {  
    }  
    MessageType type_;  
};
```

```
struct Shoot: public Message {  
    Shoot(Vector2D pos, Vector2D dir) :  
        Message(SHOOT), pos_(pos), dir_(dir) {  
    }  
    Vector2D pos_;  
    Vector2D dir_;  
};
```

Un mensaje puede ser cualquier struct que lleva bastante información sobre el evento que ha ocurrido ...

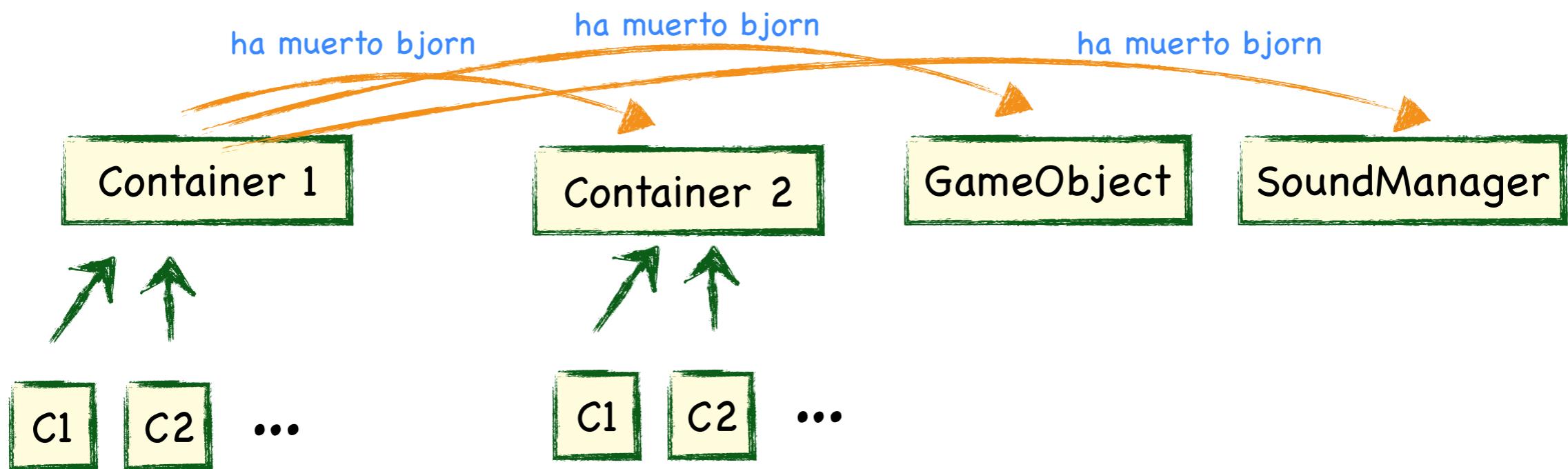
Enviar y recibir mensajes ...

```
class BjornPhysicsComponent: public Component {  
public:  
    void update(...) override {  
        ...  
        c->localSend(Message(BJORN_IS_DEAD))  
        ...  
    }  
}
```

```
class BjornGraphicsComponent: public Component {  
private:  
    Texture* pic_;  
public:  
    void update(...) override { ... graphics.draw(pic_); ... }  
    void receive(Container*c, const Message& m) override {  
        switch (m.type_) {  
            case BJORN_IS_DEAD:  
                pic_ = &deadBjornTexture;  
                break;  
            case BJORN_IS_HAPPY:  
                pic_ = &happyBjornTexture;  
        }  
    }  
};
```

Comunicación entre Objetos

Se puede usar la misma idea de mensajes para que los objetos (de todos tipos) comunican entre ellos usando mensajes, en lugar de tener referencias uno al otro.



Los componentes pueden usar sus containers para enviar mensajes a otros containers, y los containers al recibir un mensaje pueden enviarlos a sus componentes

¿Qué es un mensaje?

Primero generalizamos el concepto de mensajes para que tenga atributos de origen y destino – cada objeto va a tener (si es necesario) un identificador no necesariamente único ...

```
enum ObjectId {  
    None,  
    Broadcast, // valor especial para decir TODOS ...  
    Bjorn,  
    Enemy  
};  
  
struct Message {  
    Message(MessageType type, ObjectId sender, ObjectId dest) :  
        type_(type), sender_(sender), dest_(dest) {  
    }  
    MessageType type_;  
    ObjectId sender_;  
    ObjectId dest_  
};
```

Los componentes usan ahora este tipo de mensajes, también para comunicaciones locales. Para el atributo `sender_` pueden usar el identificador de su Container ...

Interfaz Única para todos ...

Cualquiera que quiere recibir mensajes tiene que tener un método receive, es decir todos tienen que implementar alguna interfaz común ...

```
class Observer {  
public:  
    Observer() : id_(None) {}  
    virtual ~Observer();  
  
    msg::ObjectId getId() { return id_; }  
    void setId(ObjectId id) { id_ = id; }  
  
    virtual void receive(const Message& msg) = 0;  
  
private:  
    ObjectId id_;  
};
```

Servicio de mensajería ...

```
class SDLGame {  
public:  
...  
void send(const Message& msg) {  
    for(Observer* o : obs_) {  
        if( msg.dest_ == Broadcast || msg.dest_ == o->getId()) {  
            o->receive(sender, msg);  
        }  
    }  
}  
  
void addObs(Observer* o) { obs_.push_back(o); }  
...  
private:  
...  
vector<Observer*> observers_;
```

En algún punto con acceso global, p/ej., en SDLGame, definimos métodos para poder registrarse y mandar mensajes ...

El método send envía el mensaje recibido ...

Lista de los que quieren recibir mensajes. Para poder recibir mensajes hay que registrarse!

Preparar GameObject y Container

```
class GameObject : virtual public Observer {  
public:  
    ...  
    virtual void receive(const Message& msg) {};  
}
```

Por defecto no hace nada al recibir un mensaje

```
class Container : public GameObject {  
public:  
    ...  
    virtual void receive(const Message& msg) {  
        localSend(sender, msg);  
    }  
}
```

Por defecto envía el mensaje a sus componentes

```
virtual void gloablSend(const Message& msg) {  
    getGame()->send(sender, msg);  
}  
...  
};
```

Los componentes usan este método para enviar mensajes globales, (si SDLGame es singleton no hace falta)

Registrarse ...

```
virtual GameObject* createBjorn() {  
    GameObject* bjorn = new Container(...);
```

```
bjorn->addComponet( new SomeInputComp1(...) );  
bjorn->addComponet( new SomeInputComp2(...) );  
bjorn->addComponet( new SomePhysicsComp1(...) );  
bjorn->addComponet( new SomePhysicsComp2(...) );  
bjorn->addComponet( new SomeGraphicsComp(...) );
```

```
...
```

```
bjorn.setId(Bjorn);  
getGame()->addObs(bjorn);  
...
```

```
}
```

```
return bjorn;
```

Asignar un identificador si es necesario ...

Registrarse para recibir mensajes, para enviar no hace falta registrarse ...

Resumen

- ◆ Lo que hemos hecho es sacar los comportamientos de un GameObject fuera mediante el encapsulamiento de comportamientos (la interfaz Component)
- ◆ Ahora un objeto del juego (entidad) es simplemente una colección de componentes
- ◆ Con este diseño evitamos jerarquías complicadas de clases, herencia, etc. — Composición vs. Herencia.
- ◆ Comunicación entre componentes a través del container
- ◆ Comunicación en general usando mensajes
- ◆ **Siguiente objetivo:** todavía tenemos la clase GameObject! Para qué? Hay que eliminarla y mover los datos a los componentes