

# Patrones de Diseño

De GameObject a Componentes

El primer paso hacia componentes

TPV  
Samir Genaim



# El Problema: I

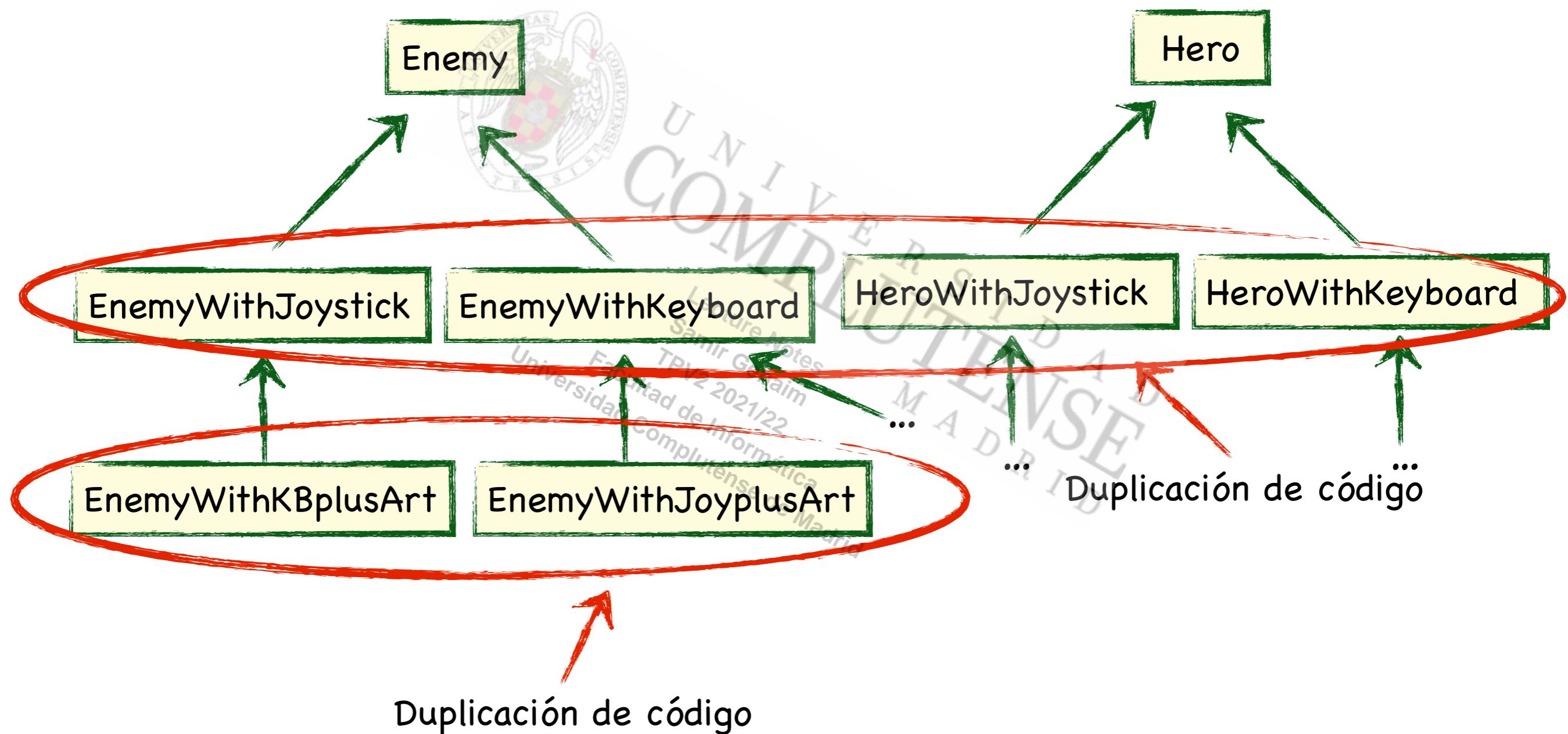
Una entidad de un video juego (entity) necesita “tocar” partes (dominios) conceptualmente distintos del juego como Entrada, Física, Gráfica, etc.

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) {  
    }  
}
```

Posición, velocidad, etc., atributos de GameObject

```
void handleInput(...);  
void update(...);  
void render(...);
```

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

```
private:  
    static const int WALK_ACC = 1;
```

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

```
Volume volume_;
```

```
...
```

```
Sprite spriteStand_;  
Sprite spriteWalkLeft_;  
Sprite spriteWalkRight_;
```

Otros atributos relacionados con la parte gráfica, etc.

```
};
```

# La lógica de Bjorn ...

El código de Bjorn abarca a varios dominios, el programador tiene que saberlo todo!

```
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_);  
}
```

# Separar los dominios de Bjorn

```
class Component {  
public:  
    Component() {}  
    virtual ~Component() {}  
};
```

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

Interfaz para manejo de entrada

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

Interfaz para manejo de física

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

Interfaz para manejo de gráfica

# Bjorn - sólo un Container

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

*Salvo autorización expresa, los materiales  
entregados a estudiantes para su uso personal  
durante la carrera sólo se podrán utilizar  
para el estudio de la asignatura correspondiente  
en la Universidad Complutense de Madrid.  
La publicación o distribución posterior  
(incluida la divulgación en redes sociales  
o servicios de compartir información en Internet)  
puede vulnerar la normativa de protección  
de datos y/o la de propiedad intelectual y  
generar responsabilidad de la persona infractora.  
Si encuentras este material en otro sitio web  
que no tenga la extensión ucm.es,  
avísalos en denunciacontenido@ucm.es*

La constructora recibe los componentes

Simplemente delegan el trabajo a los componentes.

Atributos para los componentes

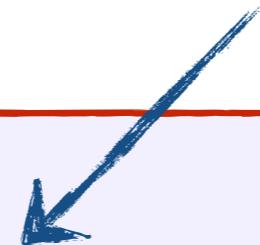
# ¿Quién es Bjorn?

Los componentes definen quién es Bjorn.

...

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

...



UNIVERSITAS  
COMPLUTENSIS  
ATENIENSIS  
LECTURE NOTES  
Samir Genaim  
TPV2 2021/22  
Facultad de Informática  
Universidad Complutense de Madrid

# Bjorn - Componentes de Entrada

```
class JoystickInputComponent: public InputComponent {  
public:  
    virtual void handleInput(Bjorn* bjorn, ...) {  
        switch (Controller::getJoystickDirection()) {  
            case DIR_LEFT:  
                bjorn->setVel(bjorn->getVel() - WALK_ACCELERATION);  
                break;  
            case DIR_RIGHT:  
                bjorn->setVel(bjorn->getVel() + WALK_ACCELERATION);  
                break;  
        }  
    }  
};  
...
```

Joystick

```
} class KeyboardInputComponent: public InputComponent {  
public:  
    virtual void handleInput(Bjorn *bjorn, ...) {  
        switch (Controller::getPressedKey()) {  
            case UP:  
                bjorn->setVel(bjorn->getVel() - WALK_ACCELERATION);  
                break;  
            case DOWN:  
                bjorn->setVel(bjorn->getVel() + WALK_ACCELERATION);  
                break;  
        }  
    }  
};  
...
```

Keyboard

```
class DemoInputComponent: public InputComponent {  
public:  
    virtual void handleInput(Bjorn *bjorn, ...) {  
        // do some AI o move bjorn  
        ...  
    }  
};  
...
```

Demo

# Bjorn - Física y Gráfica

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

Física 1

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

Física 2

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

Gráfica 1

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

Gráfica 2

# No sólo Bjorn ...

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

podemos generalizar el concepto  
a un Container

```
Container(..., InputComponent* ic, PhysicsComponent* pc, GraphicsComponent* gc) :  
    GameObject(...),  
    input_(ic), physics_(pc), graphics_(gc) {
```

El constructor recibe  
los componentes

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

Simplemente delegan el  
trabajo a los componentes.

```
private:  
    InputComponent* input_;  
    PhysicsComponent* physics_;  
    GraphicsComponent* graphics_;
```

Atributos para los componentes

# Interfaces de Componentes

```
class Component {  
public:  
    Component() {}  
    virtual ~Component() {}  
};
```

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

Interfaz para manejo de entrada

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

Interfaz para manejo de física

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

Interfaz para manejo de gráfica

# Bjorn se ha desaparecido

No hace falta la clase Bjorn!!

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

Se define en la factoría

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 setComponent(InputComponent* ic) {  
        input_ = ic;  
    }  
    void setComponent(PhysicsComponent* pc) {  
        physics_ = pc;  
    }  
    void setComponent(GraphicsComponent* gc) {  
        graphics_ = gc;  
    }  
private:  
    InputComponent* input_;  
    PhysicsComponent* physics_;  
    GraphicsComponent* graphics_;  
};
```

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

```
Container* bjorn = createBjorn();  
...  
bjorn->setComponent( new JoystickInputComponent(...) );  
...  
bjorn->setComponent( new KeyBoardInputComponent(...) );  
...
```

# Varios componentes de cada tipo

```
class Container: class GameObject {  
public:  
    Container(...) : GameObject(...) { ... }  
  
    void handleInput(...) {  
        for(InputComponent* ic : input_) ic->handleInput(this, ...);  
    }  
    void update(...) {  
        for(PhysicsComponent* pc: physics_) pc->update(this, ...);  
    }  
    void render(...) {  
        for(GraphicsComponent* gc: graphics_) gc->render(this, ...);  
    }  
  
    void addComponent(InputComponent* ic) { input_.add(ic); }  
    void addComponent(PhysicsComponent* pc) { physics_.add(pc); }  
    void addComponent(GraphicsComponent* gc) { graphics_.add(gc); }  
  
    void removeComponent(InputComponent* ic) { ... }  
    void removeComponent(PhysicsComponent* pc) { ... }  
    void removeComponent(GraphicsComponent* gc) { ... }  
  
private:  
    vector<InputComponent*> input_;  
    vector<PhysicsComponent*> physics_;  
    vector<GraphicsComponent*> graphics_;
```

delegar la petición a todos

Añadir

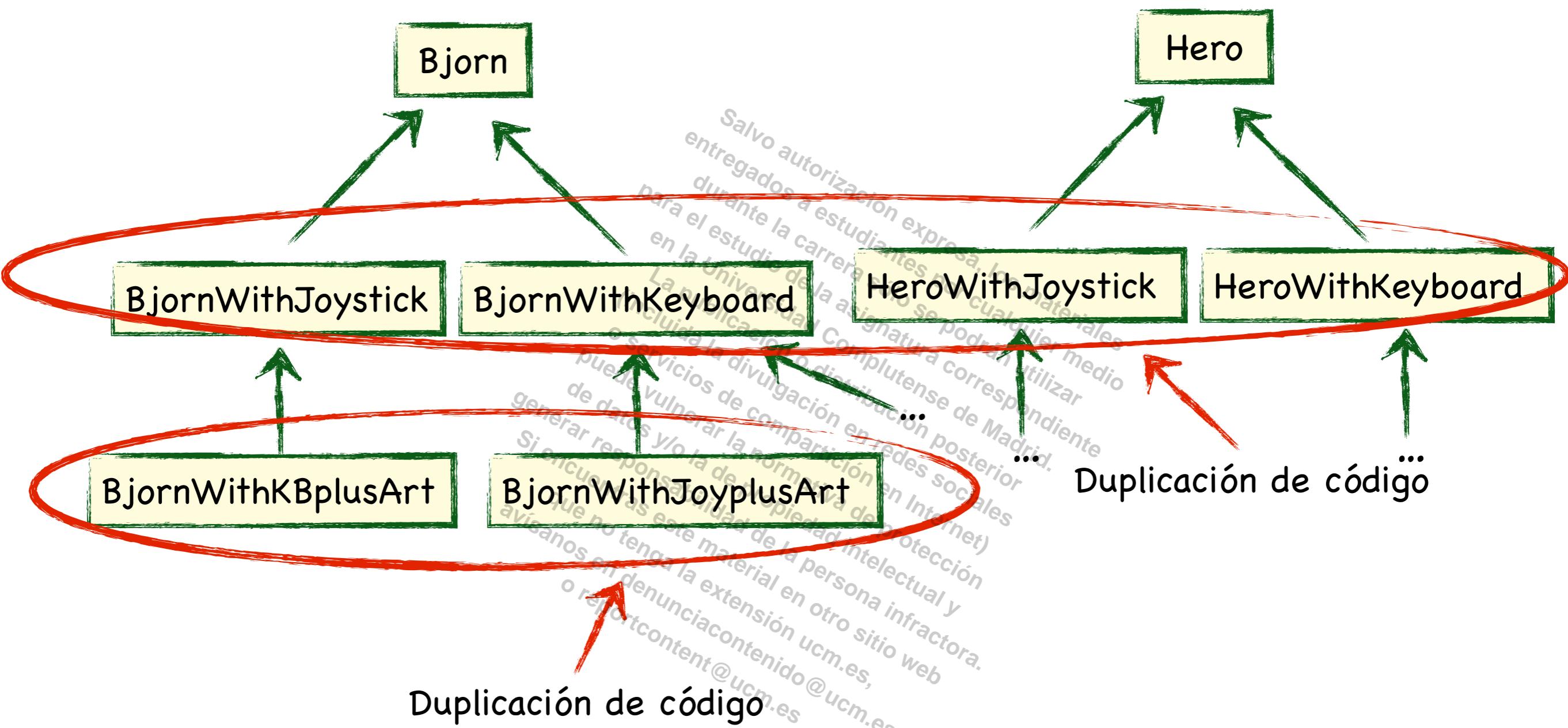
Quitar

Usar un array o alguna Colección para cada tipo de componente

# Configuración de Bjorn

```
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(...) );  
  
    ...  
  
    return bjorn;  
}
```

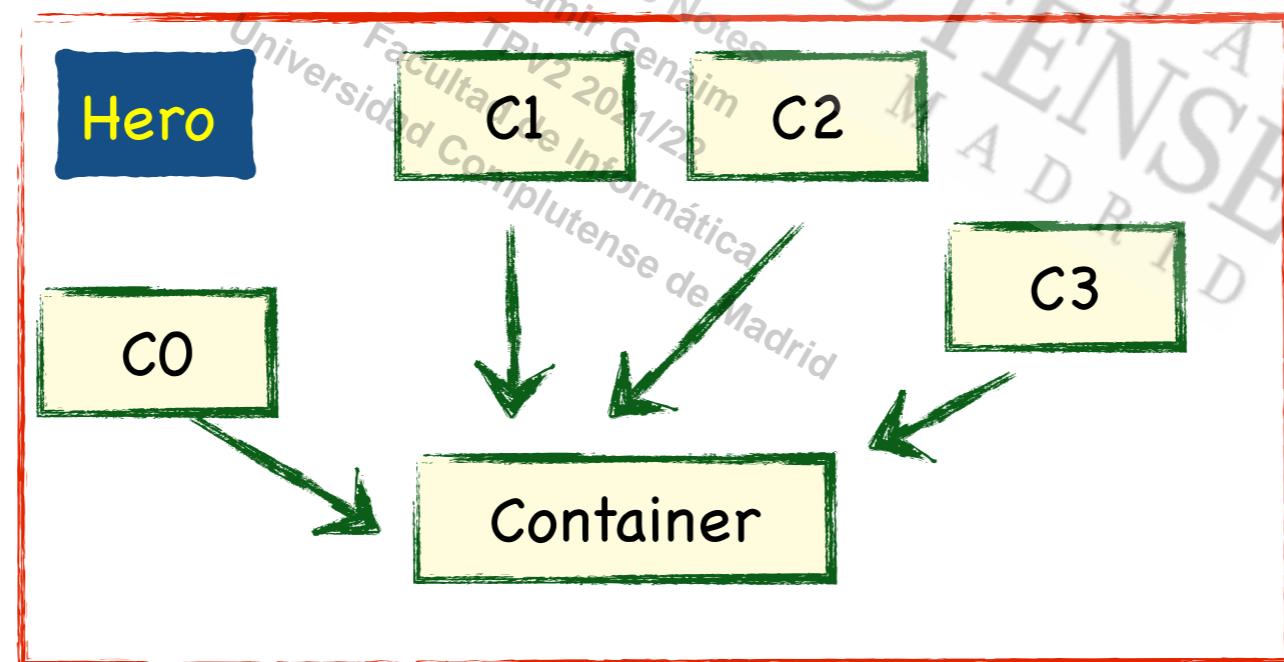
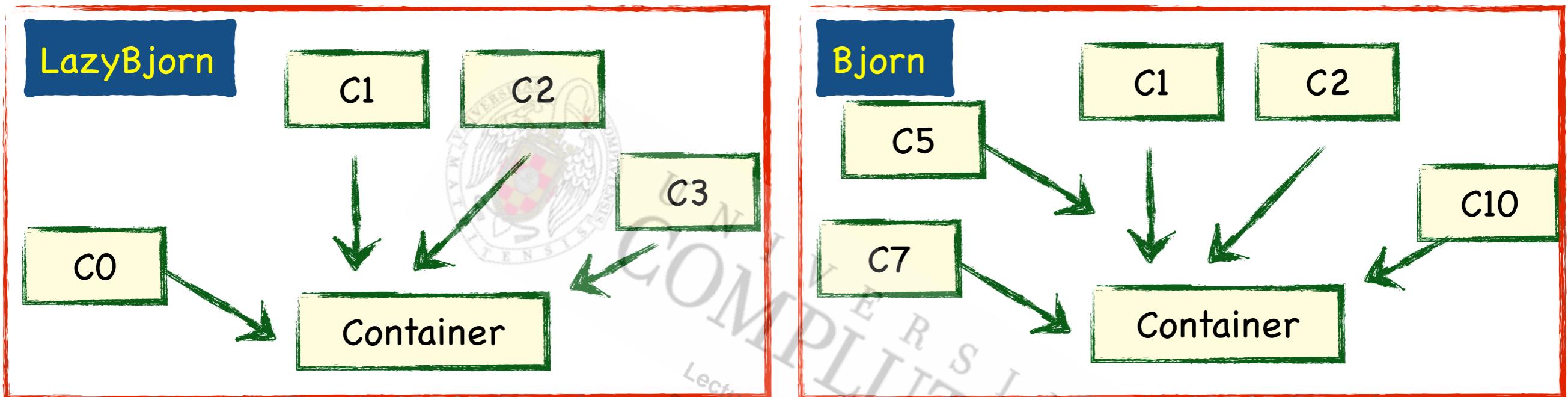
# Composición vs. Herencia



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 ...

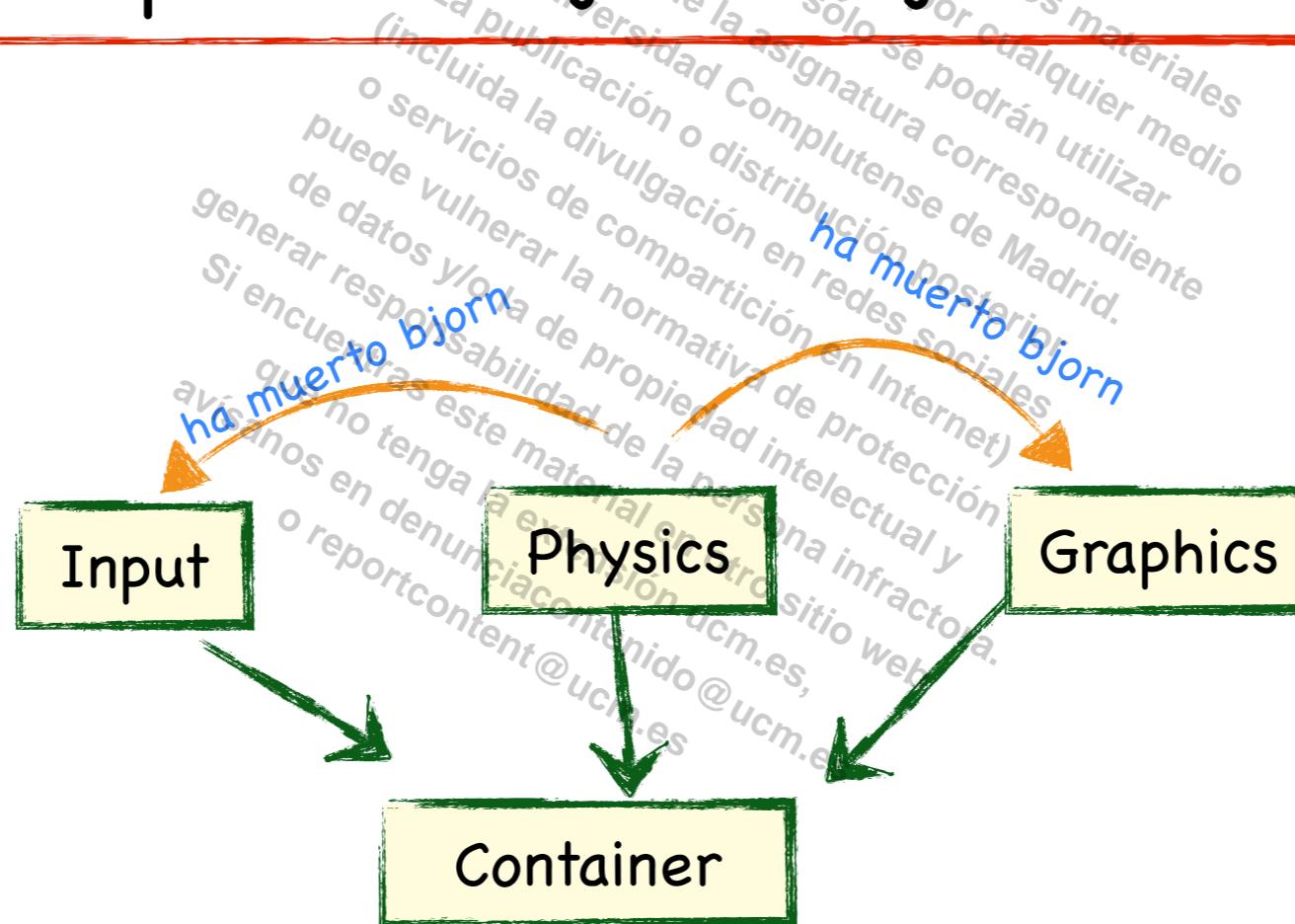
# Composición vs. Herencia

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 tienes acceso a esos atributos ...

```
class BjornPhysicsComponent: public PhysicsComponent {  
public:  
    virtual void update(Container* c, ...) {  
        ...  
        static_cast<Bjorn*>(c)->setDead(true);  
    }  
};
```

```
class BjornGraphicsComponent: public GraphicsComponent {  
public:  
    virtual void update(Container *c, ...) {  
        ...  
        if ( static_cast<Bjorn*>(c)->isDead() ) { ... } else { ... }  
    }  
};
```

# Referencia directa

```
class KeyboardInputComponent: public InputComponent {  
public:  
  
    KeyboardInputComponent(PhysicsComponent* state) {  
        state_ = state;  
    }  
  
    virtual void handleInput(Container*bjorn, ...)  
    switch (Controller::getPressedKey()) {  
        case UP:  
            state_->speedUp(WALK_ACCELERATION);  
            break;  
        case DOWN:  
            state_->slowDown(WALK_ACCELERATION);  
            break;  
    }  
}  
  
private:  
    static const int WALK_ACCELERATION = 1;  
    PhysicsComponent* state_;  
};
```

Pasar un componente a otro

Comunicar la información directamente a otro componente

# Usando mensajes ...

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

Un método para recibir mensajes de otros componentes. Por defecto no hace nada ...

```
class Container: public GameObject {  
public:  
    ...  
    void localSend(const Message& m) {  
        for(InputComponent* ic : input_)  
            if (ic != sender) ic->receive(m);  
        for(PhysicsComponent* pc : physics_)  
            if (pc != sender) pc->receive(m);  
        for(GraphicsComponent* gc : graphics_)  
            if (gc != sender) gc->receive(m);  
    }  
    ...  
};
```

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

# ¿Qué es un mensaje?

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

```
enum MessageType {  
    BJORN_IS_HAPPY,  
    BJORN_IS_DEAD,  
    SHOOT  
};  
  
struct Message {  
    Message(MessageType type)  
    {  
        MessageType type_;  
    }  
  
    struct Shoot: public Message {  
        Shoot(Vector2D pos, Vector2D dir)  
        :  
            Message(SHOOT), pos_(pos), dir_(dir) {  
    }  
  
        Vector2D pos_;  
        Vector2D dir_  
    };  
};
```

Salvo autorización expresa, los materiales entregados a estudiantes por cualquier motivo durante la carrera sólo se podrán utilizar en la Universidad Complutense de Madrid, para el estudio de la asignatura correspondiente y en la Universidad Complutense de Madrid, La publicación o distribución posterior (incluida la divulgación en redes sociales o servicios de compartición en Internet) o servicios de protección de datos y/o la de propiedad intelectual y puede vulnerar la normativa de protección de datos y/o la de propiedad intelectual y que no tenga la extensión .pdf o .rtf. Si encuentras este material en otro sitio web, avisanos en denunciacontent@ucm.es o reporta contenido@ucm.es

# Enviar y recibir mensajes ...

```
class BjornPhysicsComponent: public PhysicsComponent {  
public:  
    virtual void update(Container* c, ...) {  
        ...  
        c->localSend(Message(BJORN_IS_DEAD))  
    }  
    ...  
};  
  
class BjornGraphicsComponent: public GraphicsComponent {  
private:  
    Texture* pic_;  
public:  
    virtual void update(Container *c, ...) {  
        graphics.draw(pic_);  
    }  
    ...  
    virtual void receive(const Message& m) {  
        switch (m.type_) {  
            case BJORN_IS_DEAD:  
                pic_ = &deadBrjonTexture;  
                break;  
            case BJORN_IS_HAPPY:  
                pic_ = &happyBrjonTexture;  
        }  
    }  
};
```

# Interfaz Component - otro diseño

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

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

Salvo autorización expresa, los materiales entregados a estudiantes por cualquier medio durante la carrera sólo se podrán utilizar para el estudio de la asignatura correspondiente en la Universidad Complutense de Madrid. La publicación y distribución posterior (incluida la digitalización en redes sociales o servicios de compartición en Internet) genera vulnerabilidad de la propiedad intelectual y generará responsabilidad de la persona infractora. Si tuviéramos este material en otro sitio web o reportártelo a [avisoalalma@ucm.es](mailto:avisoalalma@ucm.es), o reportártelo a [denuncia@ucm.es](mailto:denuncia@ucm.es).

# 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 usando su container
- ◆ Comunicación directa entre componentes
- ◆ Comunicación usando mensajes