Práctica 2 parte I: Super cars refactored

Fecha de entrega: 8 Noviembre

Objetivo: Uso de herencia y abstracción

1. Introducción

¿Cómo podemos extender nuestra aplicación de manera robusta? Es decir, ¿cómo podemos añadir nuevas funcionalidades sin que se rompan las que ya están funcionando?. En particular en nuestro juego podríamos pensar diferentes extensiones:

- Nuevos objetos de juego. Por ejemplo, añadiendo powerups u obstáculos móviles, como camiones.
- Nuevos comandos. Por ejemplo, un comando Shoot para disparar a los obstáculos.
- Nuevos estados de juego. Por ejemplo, diferentes niveles, o incluso un estado de juego Pause.

A lo largo de las prácticas veremos cómo hacer crecer nuestro juego con diferentes extensiones. Pero, antes de extender nuestra aplicación primero tenemos que refactorizar nuestro código. Refactorizar consiste en cambiar la estructura del código sin cambiar su funcionalidad. En cada sección veremos un cambio.

2. GameObject

Nos hemos dado cuenta de que tanto el Car como los obstáculos comparten mucha funcionalidad. Ambos tienen una posición en el tablero, un collider, un método render... La idea es crear una jerarquía de objetos que hereden de un objeto básico GameObject. Que tendrá la siguiente cabecera:

```
class GameObject : public Collider{

    Point2D<double> pos;
    int w, h;

protected:
    Game *game;
    Texture *texture;

    void drawTexture(Texture* texture);
public:

    GameObject(Game *game): game(game){};
    virtual ~GameObject(){};

    virtual void draw()=0;
    virtual void drawDebug();
    virtual void update()=0;
    virtual bool toDelete(){return false;}

    virtual void onEnter(){};
    virtual void onDelete(){};

    void setPosition(double x, double y);
    void setDimension(double width, double height);

    int getWidth() {return w;};
    int getHeight() {return h;};

    int getX() {return pos.getX();};
    int getY() {return pos.getY();};
    virtual SDL_Rect getCollider();
    virtual SDL_Rect getCenter();

    bool collide(SDL_Rect other);
};
```

GameObject es una clase abstracta, es decir no se pueden crear objetos de esta clase, solo podemos instanciar las clases derivadas: Car o Rock (en la primera práctica le habíamos llamando Wall)

Tenemos métodos virtuales, no virtuales, y virtuales puros:

- Los métodos virtuales puros los tendrán que implementar las clases derivadas, no tienen ninguna implementación por defecto.
- Los métodos virtuales tienen un comportamiento definido pero los pueden sobreescribir las clases derivadas.
- Los métodos no virtuales los heredan tal cual las clases derivadas.

En nuestro juego deberemos reorganizar el código el coche y de los obstáculos para que hereden lo máximo posible de GameObject.

2.1. Object Container

Tanto el coche como los obstáculos son GameObjects. La idea es que el Game guarde una referencia al coche para pasarle los comandos y un vector de GameObjects donde están

el resto de los objetos. Lo importante es que yna vez introducimos un GameObject al juego el Game ya no sabe qué es. Para almacenar los objetos vamos a crear una clase wrapper Game Object Container con la siguiente cabecera:

```
class GameObjectContainer {
    vector<GameObject*> gameObjects;
public:
    GameObjectContainer(){};
    ~GameObjectContainer();
    void update();
    void draw();
    void drawDebug();
    void add(GameObject *gameObject);
    void removeDead();
    vector<Collider *> getCollisions(GameObject *g);
    bool hasCollision(GameObject *g);
};
```

El Game utiliza el Container para delegar acciones a los GameObjects. Así pues en el Game llamaremos al draw del GameObjectContainerque a su vez llamará al draw de cada uno de los GameObjects.

Aunque hablaremos de las colisiones más adelante vemos en el header que tenemos los dos siguientes métodos:

- getCollisions que nos da todas las colisiones entre un objeto y el reto.
- hasCollision nos devuelve si un objeto tiene alguna colisión con otro (podemos utilizar el método anterior para programar este). Esta función la utilizaremos para evitar que se solapen los objetos cuando los creamos y los metemos en el juego.

2.2. Object Generator

Siempre es buena práctica tener un objeto que cree otros objetos, este tipo de patrón de diseño se llama Factory. No vamos a estudiar el patrón como tal pero vamos a seguir su misma filosofía.

```
class GameObjectGenerator {

    Point2D<int> static generateRandomPosition(Game *game, GameObject *o);
    void static addInRandomPosition(Game *game, GameObject *o);

    public:
    static void generate(Game *game, int N_ROCKS = 0){
        for(int i  = 0; i  < N_ROCKS; i++)
            addInRandomPosition(game, new Rock(game));
      ...
    }
}
```

- Tenemos un bucle que va creando objetos y los mete el juego.
- generateRandomPosition se usa para generar una posición aleatoria en la carretera. Tenemos que segurarnos que el objeto no sale cortado.

- Antes de añadir un objeto al juego deberemos comprobar que no se solapa (colisiona) con ningún otro objeto. Eso se hace en addInRandomPosition. Recordemos que si el objeto que vamos a meter solapa con otro entonces lo descartamos, no le buscamos otra nueva posición.
- El método generate es estático, por lo que no necesitamos crear un objeto de GameObjectGenerator, es suficiente con invocarlo con el nombre de la clase, por ejemplo la siguiente llamada generará hasta un máximo de 15 rocas:

```
GameObjectGenerator::generateLevel(this, 15);
```

2.3. Extension 1: PowerUp

Para comprobar que esta primera refactorización ha ido bien vamos a crear un nuevo objeto de juego: powerup. Es igual que el obstáculo pero en lugar de matar al coche le da una nueva vida o power. Usaremos la siguiente textura con tamaño $40x40$ píxeles. El power del coche es inicialmente 3 si se choca con una roca baja uno y si coge un corazón sube 1.

Figura 1: PowerUps

2.4. Good and bad objects

En la siguiente práctica vamos a tener más tipos de objetos. Así que para poder manipular los objetos buenos (powerups, coins...) de los malos (rocks, truck,...) vamos hacer dos clases abstractas auxiliares GoodObjects y BadObjects. En la imagen puedes ver la jerarquía de clases donde ya incluye las futuras extensiones. El Car heredará directamente GameObject pero ni bueno ni malo.

```
class BadObject : public GameObject {
protected:
    bool alive;
public:
    static int instances;
    BadObject(Game *game): GameObject(game) {
        alive  = true;
    };
```

```
    ~BadObject() override = default;

    void update() override;
    bool toDelete() override;
    void onEnter() override;
    void onDelete() override;
    void static reset();
};
```

Por el momento solo vamos a utilizar estas clases para contar cuántos objetos de cada uno tenemos en la carretera. Para ello vamos a utilizar una variable estática (que es como una variable global que comparten todos los objetos de la clase), cada vez que metemos un objeto en el juego la incrementamos y cada vez que desaparece del juego la decrementamos. Para acceder al número de instancias usaremos esta llamada BadObject::instances. Esta es la implementación.

```
void BadObject::onEnter(){
    instances += 1;
};

void BadObject::onDelete(){
    instances -= 1;
};

void BadObject::update(){
};

void BadObject::reset(){
    instances = 0;
};

bool BadObject::toDelete() {
    return !alive || game->isRebased(this);
}
int BadObject::instances = 0;
```

- Vemos en la figura, que se mostrará la información de cuántos objetos buenos y malos hay en la pista en cada momento.
- Sin nos fijamos en el código deberemos hacer un método en game isRebased para saber si el objeto ha sido adelantado por el coche, en cuyo caso lo deberemos eliminar.
- Cuando eliminamos un objeto llamamos a su método onDelete que es el encargado de disminuir su número de instancias.
- El código de GoodObject, por el momento, será igual que el de BadObject.

3. Collision

Las colisiones se calculan justo después de mover el coche:

- Calculamos las nueva posición del coche.

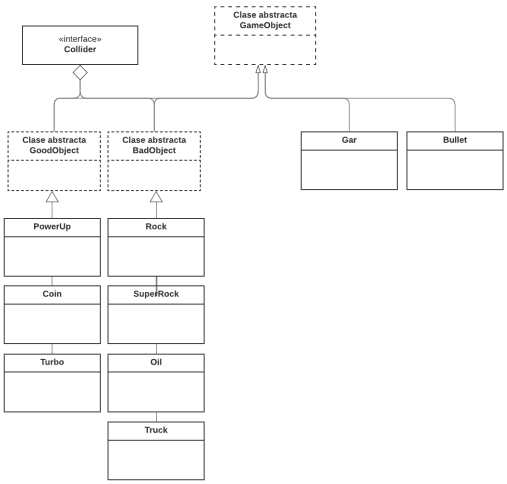

Figura 2: Class Hierarchy

- Pedimos al game los objetos con los que están colisionando. Puede no ser ninguno a haber varios.
- Calculamos el efecto de las colisiones.

Veamos el esqueleto del código:

```
void Car::update() {

    //   Calculate the new position
    ...

    setPosition (newX, newY);

    // The car asks the game who is colliding with him.

    vector<GameObject *> collisions = game->getCollisions(this);

    for(auto c: collisions ){
      // Calculate the effect  of every collision
      ..
     }
}
```

Cuando solo manejamos objetos abstractos gestionar las colisiones no es obvio. Si el coche se choca un GameObject, a priori no sabemos si es una roca o un powerup, por lo tanto no sabemos si hay que sumarle vida o restarla. Para solucionar esto podríamos hacer un down-casting, es decir hacer un casting a una clase derivada, como en el siguiente ejemplo:

```
void Car::update() {
  ...
  for(auto c: collisions ){
    Rock rock* = dynamic_cast<Rock*>(c);
```

```
    if (rock != nullptr) {
      rock->setAlive(false);
      this->decreasePower();
      this->speed = 0;
      continue;
    }
    PowerUp powerUp* = dynamic_cast<PowerUp*>(c);
    if (powerUp != nullptr) {
      powerUp->setAlive(false);
      this->coins += 1;
      continue;
    }
  }
}
```

Con dynamic_cast le *preguntamos* a C++ si el objeto con el que colisiona es un rock o un powerup. Esta opción es **INCORRECTA** porque contraviene la idea de robustez. Si hacemos esto ya no podemos extender de manera transparente nuestro código porque cada vez que añadamos un nuevo objeto de juego deberemos modificar la clase Car. Lo que queremos es que la incorporación de nuevas funcionalidades sea *plug-and-play*. Así pues el uso de down-castings en el contexto de esta asignatura está terminantemente prohibido. En la siguiente sección veremos una solución mejor.

3.1. Callbacks

Para resolver el problema de las colisiones vamos utilizar la noción de callback. Todos los objetos de juego van a implementar (heredar de) un interfaz Collider.

```
class Collider {
public:
    virtual bool receiveCarCollision(Car *car){
        return false;
    };
};
```

Cada GameObject va a implementar cual es la consecuencia de que el coche colisiones con él. Por defecto no pasa nada, por eso devuelve false. Tanto rock como powerup deberán sobre-escribir el método receiveCarCollision.

Ahora nuestra gestión de colisiones quedará así:

```
void Car::update() {

    ...

    vector<Collider *> collisions  = game->getCollisions(this);

    for(auto c: collisions )
        c->receiveCarCollision(this);
}
```

El resultado de la colisión se hace con el callback receiveCarCollision. Vemos el coche

se pasa a sí mismo como argumento para que el objeto con el que colisiona resuelva el resultado de la colisión.

Esta solución, aunque a priori es menos evidente, es mucho más robusta que la anterior ya que, ahora, para cada nuevo tipo de objeto que incorporemos al juego no tendremos que modificar la clase Car. El nuevo objeto tendrá que implementar el resultado de la colisión.

4. Pattern Command

Al igual que con los GameObjects queremos ser capaces de meter nuevas acciones al juego sin modificar la estructura del juego, para ello vamos a usar el conocido patrón de diseño Command[1].

La idea principal es:

- Cada una de las acciones se encapsula como una clase.
  - Tenemos una clase abstracta Command.
  - Tenemos una serie de comandos concretos: MoveCommand, AccelerateCommand, QuitCommand,... que encapsula la acción correspondiente.
- Tenemos un CommandFactory que se encarga de generar los comandos concretos.
- El Controller es genérico, solo usa la clase abstracta Command por lo que no sabe qué está ejecutando en cada momento.

Veamos cada uno de las partes:

4.1. Command

Todos los comandos tienen exactamente la misma estructura:

```
class Command {
protected:
    string  info_string;
    Game *game;
public:
    Command(){};
    virtual ~Command()=default;
    virtual bool parse(SDL_Event &event)=0;
    virtual void execute()=0;
    void bind(Game *game){
        this->game = game;
        game->appendHelpInfo(info_string);
    }
};
```

- Cada comando, dado un evento, es el encargado de ver si ese evento le corresponde a él, usando el método parse.
- Cada comando ejecuta su acción concreta sobre el juego con el método execute.
- El método bind sirve para guardar la referencia al game y para pasarle su información al help

[1] No vamos a ver el patrón con detalle, simplemente lo vamos a usar de manera intuitiva

4.2. Concrete Commands

Cada acción que hagamos va estar encapsulada en una clase, veamos el ejemplo del comando más sencillo:

```
class QuitCommand : public Command{

public:
    const string INFO_STRING = "[ESC] to quit";

    QuitCommand(){
        info_string = INFO_STRING;
    };
    ~QuitCommand()=default;
    bool parse(SDL_Event &event) override;
    void execute() override;
};


bool QuitCommand::parse(SDL_Event &event) {
    if( event.type == SDL_QUIT)
        return true;
    if(event.type == SDL_KEYDOWN){
        SDL_Keycode key = event.key.keysym.sym;
        if(key == SDLK_ESCAPE)
            return true;
    }
    return false;
}

void QuitCommand::execute() {
    game->setUserExit();
}
```

Todo esto lo teníamos lo teníamos antes en el controller pero ahora cada comando es encargado de su propia funcionalidad. Los comandos solo acceden al Game, nunca a las clases que encapsula: Car, GameObjectContainer,....

4.3. Controller

En esta versión el código del método handleEvents del controlador va a ser mínimo. Básicamente pide a la factoría el comando encargado del evento y lo ejecuta.

```
void ViewController::handleEvents() {
    SDL_Event event;
    while (SDL_PollEvent(&event) ){
        Command *command = commandFactory->getCommand(event);
        if(command != nullptr){
            command->execute();
            break;
        }
    }
}
```

4.4. Command Factory

La factoría de comandos es la encargada de almacenar los comandos disponibles y de dárselos al controlador. Su código es el siguiente:

```
class CommandFactory {
    vector<Command *> availableCommands;
    Game *game;

public:
    CommandFactory(Game *g){
        game = g;
        game->clearHelp();
    }

    ~CommandFactory(){
        for(auto c: availableCommands)
            delete c;
    }

     Command *getCommand(SDL_Event &event){
        for(auto c: availableCommands){
            if(c->parse(event)) return c;
        }
        return nullptr;
    }

    void add(Command *c){
        c->bind(game);
        availableCommands.push_back(c);
    }
};
```

Lo interesante de este patrón es que para añadir cualquier nueva funcionalidad será suficiente con hacer una nueva clase que herede de Command y añadirla al CommandFactory.

Los comandos se *registran* en la factoría usado el método add, que se llama desde el ViewController antes de comenzar el juego:

```
    commandFactory->add(new MoveCommand());
    commandFactory->add(new AccCommand());
    commandFactory->add(new DebugCommand());
    commandFactory->add(new HelpCommand());
    commandFactory->add(new QuitCommand());
    game->startGame();
```

En esta práctica tendremos los siguientes comandos:

- El comando Move es el encargado de mover el coche arriba y abajo.
- El comando Acc es el encargado de acelerar y frenar.
- El comando Debug es el encargado de mostrar/ocultar el modo debug, en el que pintamos los coliders.

- El comando Help es el encargado mostrar/ocultar la ayuda.
- El comando Quit es el encargado de forzar la salida del juego.

5. Friend Class: InfoBar

En la práctica 1 la información del estado de juego la renderizábamos en el propio Game. Vamos a crear una clase que se encargue exclusivamente de esto: InfoBar. Esta clase va a tener que acceder a mucha información privada del Game vamos a hacerla friend class.

Hay en pocas ocasiones en las que está recomendado romper la encapsulación de una clase, accediendo a su implementación, esta sería una excepción ya que estamos utilizando InfoBar para liberar al Game de la tarea de pintar su propia información. Es una clase meramente utilitaria, no modifica el estado del juego. Veamos su cabecera:

```
class Infobar {
    Game *game;
public:
    Infobar(Game *game):game(game){}
    void drawInfo();
    void drawHelp();
    void drawState();
};
```

- drawInfo muestra la información durante el juego.
- drawState muestra el estado del juego (playing, menu,...)
- drawHelp muestra la ayuda. Puede estar activa o no.

6. Resumen

La siguiente imagen muestra la arquitectura de alto nivel y las dependencias entre las clases.

Una vez terminada la refactorización el comportamiento del juego debe ser exactamente el mismo que en la primera práctica salvo por los powerups. En la siguiente práctica añadiremos nuevas funcionalidades.

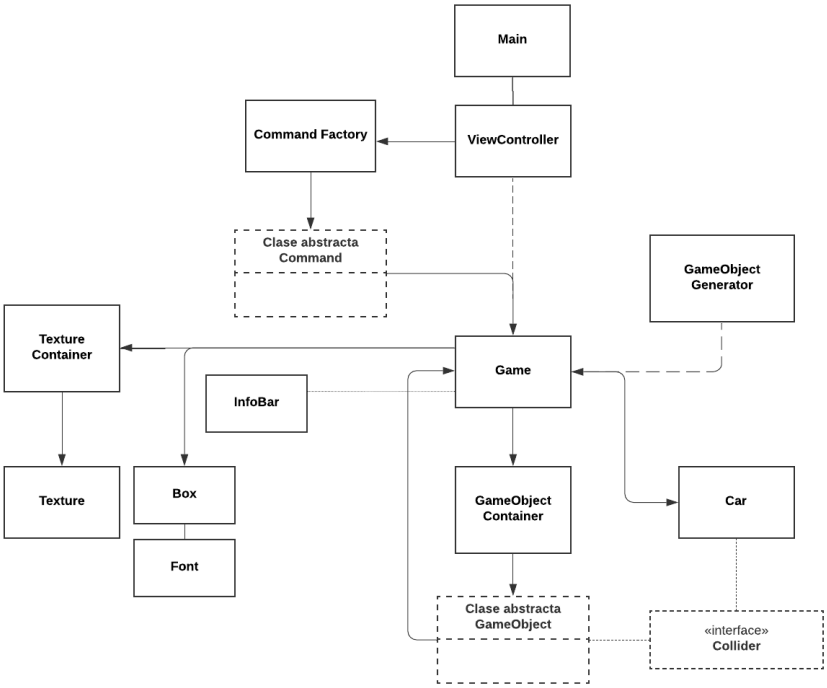

Figura 3: Arquitectura de alto nivel