Memoria Lasertowerspp

Autor: Eduard Feicho

Resumen

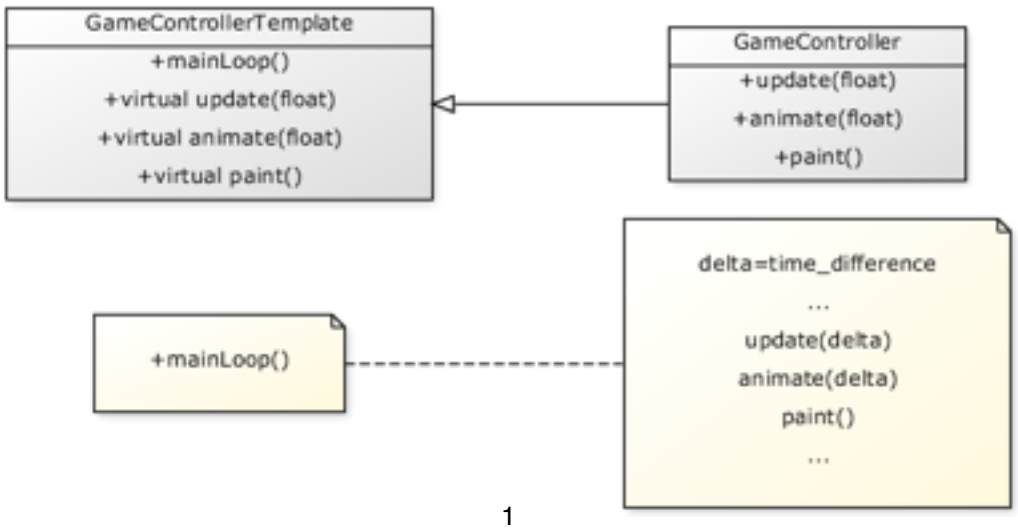
Lasertowerspp es un videojuego tipo "Tower Defense Game" basado en C++ y OpenGL. Depende de la librería OpenCV para leer imágenes.
La idea integral es que enemigos caminan a un camino y se puede construir torres para prevenir un ataque al fin del camino.

Index

1. Game Main Loop <Template>

Descripción: Aunque videojuegos pueden ser muy diferente por los jugadores, siempre tienen un lazo integral (main loop) similar, que es responsable de mantener el tiempo dentro cada lazo. Mientras cada lazo se dibuja el videojuego, así que se llama un frame. Un videojuego tiene que dibujar al menos 30 frames cada segundo para que nos parecen real los imágenes animados (mejor por ejemplo el doble 60fps). Por eso implementé el patrón *Template*, que ofrece una forma de game main loop con pasos fijos en el tiempo. El template llama tres funciones abstractos *update/animate/paint* que hay que implementar en el videojuego. Así el juego tiene una estructura basica.

Diagrama de clases:



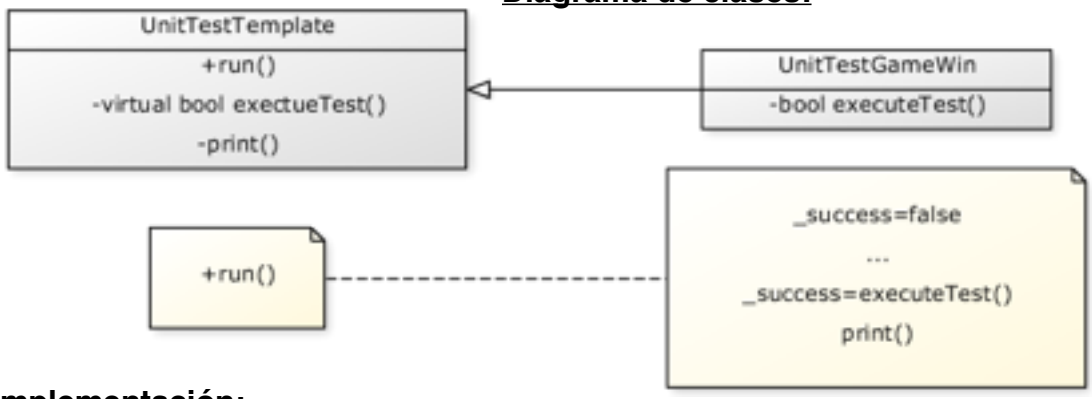
Implementación:

```
/**
 * class GameControllerTemplate provides a template for a basic game layout.
 * This includes a game main loop that can be used as an idle function.
 * Timing variables control the framerate. The update() function is called with
 * fixed timesteps for each frame.
 *
 * Fixed timesteps have benefits in collision detection, because the fixed update
 * gives an upper bound on the move in an update step. This way, one can assume
 * (or asure) that collisions don't recurse.
 *
 * <Template Pattern>
 *
 */
class GameControllerTemplate
{
public:

	GameControllerTemplate();

	// initialize the game
	virtual void initGame();

	// basic main loop for a game, using fixed timesteps
	void mainLoop();

	// The following functions are called from within the main loop.
	// These have to be implemented by an actual game.

	// update game model, for example move figures
	virtual void update(float delta) = 0;
	// update animation model, for example update sprite textures
	virtual void animate(float delta) = 0;
	// paint/render the scene
	virtual void paint() = 0;

	...
};

void GameControllerTemplate::mainLoop()
{
	time_current = glutGet(GLUT_ELAPSED_TIME);

	int time_deltaframe = time_current - time_lastframe;
	Input::keyOperations(time_deltaframe);

	// idle until a minimum time interval is reached (frame duration)
	if (time_deltaframe < min_time) {
		return;
	}
	time_lastframe = time_current;

	// count number of frames per second
	frames++;
	if (time_current-time_last_fps > 1000.0) {
		fps = frames * 1000.0 / (time_current-time_last_fps);
		time_last_fps = time_current;
		frames = 0.0;
	}

	// use an accumulator that accumulates the time passed since last update.
	// the number of how many times the time_step fits into the accumulator
	// represents the number of updates that need to be made with fixed time_step.
	accumulator += time_deltaframe * game_speed_factor;

	// render using fixed time delta
	while (accumulator >= time_step) {
		update(time_step);
		animate(time_step);
		paint();
		accumulator -= time_step;
	}
}
```

2. Unit Tests <Template>

Descripción: Una buena idea en desarrollo de software son hacer pruebas automáticas con la regla que durante del desarrollo no se puede poner código en el repositorio integral, si no pasan todas las pruebas automáticas correcto. Porque no tenía pruebas automáticas en XCode (es el entorno para desarrollar en Mac en Objective-C, pero funciona también con C++ mientras que faltan poco capacidad). Entonces usé el patrón *Template* otra vez para definir una prueba que se evalúa automáticamente.

Diagrama de clases:



Implementación:

```
/**
 * class UnitTestTemplate serves as a template for unit tests.
 * The template calls executeTest() and prints a message if execution did succeed or not.
 */
class UnitTestTemplate
{
public:
	// The constructor needs a unit test name to be displayed to the user
	UnitTestTemplate(string name) : _name(name) {}

	// unit test execution template
	void run() {
		_success = false;
		try {
			_success = executeTest();
		} catch (...) {
			_success = false;
		}
		dump();
	}

protected:
	// the actual code of the test.
	// each test has to implement it's own code and return if it succeeded or not.
	virtual bool executeTest() = 0;

private:
	// print a message if test succeeded or failed
	void print() {
		const string s = (_success) ? "succeeded." : "FAILED!";
		cout << " * Unit test " << _name << " " << s << endl;
	}

private:

	// variable to store if the unit test was successfull or not.
	bool _success;

	// name of the test to be displayed as an identifier to the user
	string _name;
};
```

3. Model-View-Controller (Game, Level)

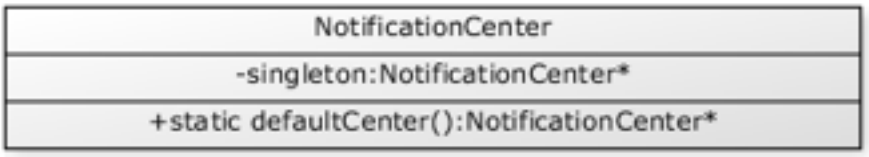
El proyecto sigue el principio Model-View-Controller. Es que, por ejemplo un juego tiene un estado cómo puntos adquirido totalmente, opciones, acceso a niveles del juego, etc. (Model). El lógico del juego contiene cómo funciona el juego, lo que debe hacer el jugador, fundamentalmente el principio del juego (Controller). Y para mostrar lo que pasa hay una vista, por ejemplo una ventana dibujando en OpenGL, que muestra el estado del jugador, los estados de los enemigos y los acciones en el videojuego (View). Se encuentra el lógico del juego en la carpeta "Controllers", los modeles en "Models" y los views en "Views". Por ejemplo el videojuego está separado en tres clases integrales, ***GameModel, GameView y GameController*** (el último implementa además GameTemplate). También el Level es un ejemplo con ***LevelModel, LevelView y LevelController.***

Comentario: El patrón MVC es importante también para el iPhone, porque una buena aplicación define modelos que el iPhone automáticamente almacena por la memoria cuando una aplicación tiene que pausar (por ejemplo una llamada interrumpe). Si la aplicación ya sigue el principio MVC será más fácil traducirlo. (y la idea es traducir el código luego para el iPhone).

4. NotificationCenter <Singleton, Observer>

Descripción: El NotificationCenter es un clase muy parecido a una con el mismo nombre que ya está disponible en es SDK del iPhone. La clase NotificationCenter tiene una instancia cómo **Singleton**. La función del NotificationCenter es tener una lista de **observadores** y enviar una notificación a ellos. La notificación puede contener un objeto arbitrario para enviar más información. Así se puede utilizar el patrón observador con una sola clase en una manera muy abierto, desde se puede enviar cómo objeto lo que se quiera. Por el momento lo está usado para enviar un los coordinados de un evento "click" del ratón a los observadores que se interesan.

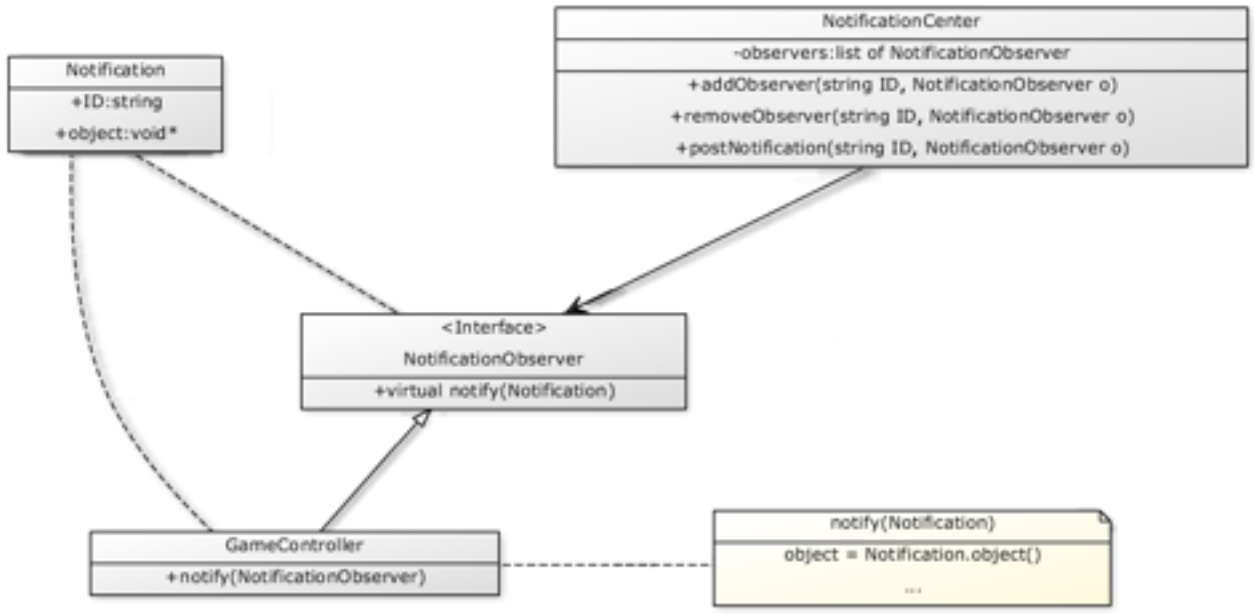
Diagrama de clases (Singleton):



Implementación (Singleton):

```
/**
 * class NotificationCenter can be used to send instant event notifications to
 * objects that implement the NotificationObserver interface.
 * Using a Singleton pattern, the class can be referenced anywhere in the code in a modular way.
 */
class NotificationCenter
{
public:
	// Singleton pattern instance
	static NotificationCenter* defaultCenter() {
		static NotificationCenter* singleton = NULL;
		if (singleton == NULL) {
			singleton = new NotificationCenter();
		}
		return singleton;
	}
	...
}
```

Diagrama de clases (Observer):



Implementación (Observer):

Aquí se ve la implementación del NotificationCenter. Los observadores se pueden añadirse a la lista de los observadores con las funciones *addObserver* y *removeObserver*. El sujeto notifica los observadores con el método *postNotification*.

```
class NotificationCenter
{
	...
public:

	// Observer pattern: add an observer for given notification
	void addObserver(NotificationObserver* observer, string notificationID) {
		observers.push_back(NotificationObserverElement(notificationID, observer));
	}

	// Observer pattern: remove observer for given notification
	void removeObserver(NotificationObserver* observer, string notificationID) {
		for (vector<NotificationObserverElement>::iterator it = observers.begin();
			it != observers.end();
			  it++) {
			if (it->notificationID == notificationID && it->observer == observer) {
				observers.erase(it);
				break;
			}
		}
	}

	/* Observer pattern: post a notification using
	 * notificationID as identificator and
	 * passing a pointer to an object of interest
	 */
	void postNotification(string notificationID, void* object) {
		for (vector<NotificationObserverElement>::iterator it = observers.begin();
			it != observers.end();
			  it++) {
			if (it->notificationID == notificationID) {
				it->observer->notify(Notification(notificationID, object));
			}
		}
	}

private:

	vector<NotificationObserverElement> observers;

};
```

Al principio un clase que implementa el observador se registra con la funciona *addObserver* por el sujeto y un evento con distinto identificación

```
GameController::GameController() {
      ...
      NotificationCenter::defaultCenter()->addObserver(this, MOUSE_CLICK_NOTIFICATION);
}
```

El sujeto envía un mensaje a los observadores. Por ejemplo el código que trata un "click" de la ratón envía las coordenadas 2d del "click" en un vector 3d relativo al centro de la ventana.

```
void Input::botonRaton( int boton, int estado, int x, int y )
{
      if (boton == GLUT_LEFT_BUTTON) {
                if (estado == GLUT_DOWN) {
                        x -= screen_width / 2.0;
                        y -= screen_height / 2.0;

                        last_x = x;
                        last_y = y;

                        MyVector* v = new MyVector(x,y,0);
                        NotificationCenter::defaultCenter()-
>postNotification(MOUSE_CLICK_NOTIFICATION, v);
                        delete v;

                }
      }
}
```

...

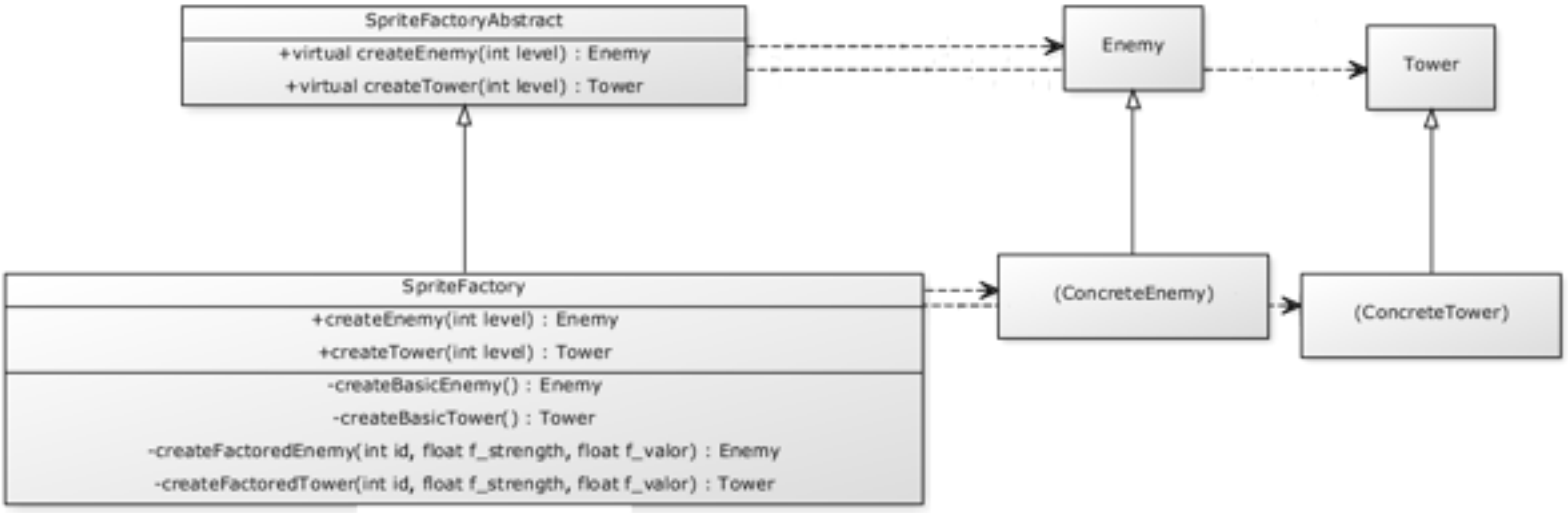
Después se va a enviar la notificación a todos los observadores. Un observador que será interesado en este notificación va a identificarla con el nombre (la identificación) del evento. Por ejemplo en el código el GameController se interesa a la ratón para construir torres en el videojuego cuando el usuario haga click.

```
void GameController::notify(Notification notification)
{
	if (notification.getObject() == NULL) {
		return;
	}

	// Mouse click event
	if (notification.getID() == MOUSE_CLICK_NOTIFICATION) {
		// A 2D vector of (x,y) coordinate is expected where the click happened
		MyVector* v = (MyVector*)notification.getObject();

		// Translate window coordinates to level field coordinates
		int x;
		int y;

		const int field_size = _model->getLevel()->getFieldSize();
		MyVector coord_relative_to_board = MyVector(
			v->x() + _model->getLevel()->getWidth()  / 2*field_size,
			v->y() + _model->getLevel()->getHeight() / 2*field_size,
			0
		);

		x = coord_relative_to_board.x() / field_size;
		y = coord_relative_to_board.y() / field_size;
		y = _model->getLevel()->getHeight()-y-1;

		if (_model->getLevel()->isEmpty(x,y)) {
			Tower* tower = SpriteFactory::createLaserTower();
			if (_model->getPlayer()->getMoney() >= tower->getCost()) {
				_model->getPlayer()->removeMoney( tower->getCost() );
				_levelController->buildTower(x,y,tower);
			} else {
				delete tower;
			}
		}

		_levelController->selectField(x,y);
	}
}
```

5. SpriteFactory <Factory>

Descripción: Los enemigos y torres se construye usando método fábrica. Es típico por el juego que hay un tipo de enemigo y cada ola será más fuerte. Lo mismo con los torres: Cuando el jugador ha ganado más dinero puede construir torres más fuertes. Sin embargo de la perspectiva del creador no hay que saber cómo se crea un objeto concretamente. Así que uso el método fábrica para ocultar los detalles de la creación.

Diagrama de clases:



Comentario: Porque el videojuego está en un estado básico todavía no hay los clases ConcreteEnemy y ConcreteTower. Hasta ahora solo hay un tipo de enemigo/torre y lo que cambia concretamente son las características. De tal manera la diagrama muestra la estructura planificada y, cuando se necesita tipos diferente de enemigos y torres, el principio no cambia.

Implementación:

La interfaz abstracto para crear enemigos y torres es bastante simple. Por el momento depende solo de un parámetro, el nivel del torre/enemigo.

```
/**
 * class SpriteFactoryAbstract is an abstract factory
 *
 * <Factory>
 */
class SpriteFactoryAbstract {
public:
      SpriteFactoryAbstract() {}

      virtual Enemy* createEnemy(int level) = 0;
      virtual Tower* createTower(int level) = 0;
};
```

Concretamente la fabrica crea un tipo de enemigo/torre y dependiendo del nivel multiplica unas de las características con factores asimétricas. Hay un factor f_strength que multiplica las características de la fuerza. Hay otro factor f_valor que multiplica las características monetario.

```
/**
 * class SpriteFactory creates and configures sprite objects.
 *
 * <Factory>
 */
class SpriteFactory : public SpriteFactoryAbstract {
public:
      SpriteFactory() {};

      virtual Enemy* createEnemy(int level);
      virtual Tower* createTower(int level);

private:
      Enemy* createBasicEnemy();
      Enemy* createFactoredEnemy(int id, float f_strength, float f_valor);

      Tower* createBasicTower();
      Tower* createFactoredTower(int id, float f_strength, float f_valor);
};

...

Enemy* SpriteFactory::createEnemy(int level)
{
      Enemy* result = NULL;

      if (level == 1) {
               result = createFactoredEnemy(ENEMYID_LASER, 1.0, 1.0);
      } else if (level == 2) {
               result = createFactoredEnemy(ENEMYID_LASER2, 2.5, 1.5);
      } else if (level == 3) {
               result = createFactoredEnemy(ENEMYID_LASER3, 4.0, 2.0);
      }
      return result;
}

Enemy* SpriteFactory::createFactoredEnemy(int id, float f_strength, float f_valor)
{
      Enemy* sprite = createBasicEnemy();

      sprite->setID(id);
      sprite->setAttackPower( sprite->getAttackPower() * f_strength);
      sprite->setStamina(     sprite->getStamina()     * f_strength);
      sprite->setStaminaMax(  sprite->getStaminaMax()  * f_strength);
      sprite->setSpeed(       sprite->getSpeed()       * f_strength);
      sprite->setCost(        sprite->getCost()        * f_valor);
      sprite->setValue(       sprite->getValue()       * f_valor);

      return sprite;
}

Enemy* SpriteFactory::createBasicEnemy()
{
      Enemy* sprite = new Enemy();

      sprite->setID(ENEMYID_LASER);
      sprite->setAttackPower(10);
      sprite->setStamina(100);
      sprite->setStaminaMax(100);
      sprite->setCost(50);
      sprite->setValue(25);

      return sprite;
}
...
```

6. Bad Smells: Long Code

Descripción: Cómo el lógico del videojuego se ha desarrollado más y más, al final he encontrado un método con mucho código que parece cómo un "*bad smell*" tipo "*long code*". Lo que pasa abstractamente se puede leer en los comentarios, pero todavía hay mucho código que usa más que una pagina de la pantalla visible. Así que he hecho refactorización tipo "extract method". Los métodos nuevos son privados.

Código antes y después:

Before

```
void GameController::update(float delta)
{
	// if the game is paused, no need for update
	if (game_pause || game_over || game_won) {
		return;
	}

	// move figures
	vector<Enemy*>* enemies = _model->getLevel()->getEnemies();
	// Sort enemies such that the first one is the one most near to the exit
	sort(enemies->begin(), enemies->end(), Enemy::compare_path_distance);

	// generate new enemies and continue rounds
	if (enemies->size() == 0 && _model->getEnemyGenerator()->isRoundFinished()) {
		_model->getEnemyGenerator()->nextRound();
	} else {
		vector<Enemy*> *new_enemies = _model->getEnemyGenerator()-
>generate( delta );
		_model->getLevel()->addEnemies( new_enemies );
		delete new_enemies;

		_model->getEnemyGenerator()->continueRound( delta );
	}

	// move all enemies along their path
	for (vector<Enemy*>::iterator it = enemies->begin(); it != enemies->end(); it++)
	{
		Enemy* enemy = *it;
		if (!enemy->isAlive()) {
			continue;
		}
		_levelController->moveSpriteAlongPath( enemy, delta );
	}
	...
}
```

After

```
class GameController : public GameControllerTemplate, public NotificationObserver
{
public:
	// update game model, for example move figures
	virtual void update(float delta);
private:
	void updateMoveFigures(float delta);
	void updateGenerator(float delta);
	void updateTowers(float delta);
	void updateEnemiesDead(float delta);
	void updateEnemiesFinished(float delta);
	void updateCheckGameWon();
};

void GameController::update(float delta)
{
	// if the game is paused, no need for update
	if (game_pause || game_over || game_won) {
		return;
	}

	updateGenerator(delta);		// generate new enemies and continue rounds
	updateMoveFigures(delta);	// move figures
	updateTowers(delta);		// make towers attack enemies
	updateEnemiesDead(delta);	// count dead enemies and add their monetary value
	updateEnemiesFinished(delta); // remove health of player for enemy at the exit
	updateCheckGameWon();		// check if game is won
}
```

Prueba de refactorización:

Si se quiere probar el lógico integral que se ha cambiado, normalmente haría que probar todas las funciones especialmente. Hice una aproximación de este prueba con dos pruebas: "*UnitTestGameWin*" y "*UnitTestGameLose*". *UnitTestGameLose* prueba si se pierde el videojuego cuando no se construye ninguno torre. *UnitTestGameWin* prueba si se gana el videojuego cuando se construye bastante torres demasiado fuertes. Las pruebas no ven el cambio de la refactorización porque llaman a GameController::update(). Estas pruebas son aproximaciones del lógico de verdad, porque la prueba de perder prueba los siguientes partes del lógico del juego:

- si bastante enemigos están generado
- si los enemigos llegarán al fin
- si los enemigos que lleguen al fin reducen la vida del jugador
- si el variable game_lost está puesto al final

La prueba de ganar prueba similarmente

- si los torres ataquen enemigos (de la otra prueba sabemos que hay bastante enemigos)
- si los enemigos mueren
- si el variable game_won está puesto al final.

Aquí se ve el código de las pruebas:

```
class UnitTestGameLose : public UnitTestTemplate
{
public:

      UnitTestGameLose()
      : UnitTestTemplate("UnitTestGameLose") {
      }

protected:

      virtual bool executeTest() {
                // Execute the game update loop thousand times with the default game,
                // which should result in game lose.

                game_pause = false;
                game_over = false;
                game_won = false;

                // create game
                LevelModel* level = new LevelModel();
                GameController* game = new GameController(level);
                game->setTestmode();

                // execute game
                float time_steps = 1000.0/60.0;
                float time_accumulator = time_steps * 10000;
                while (time_accumulator > 0) {
                        game->update(time_steps);
                        time_accumulator -= time_steps;
                }

                delete game;

                return game_over;
      }
};
```

```
class UnitTestGameWin : public UnitTestTemplate
{
public:

	UnitTestGameWin()
	: UnitTestTemplate("UnitTestGameWin") {
	}

protected:

	virtual bool executeTest() {
		// Execute the game update loop thousand times with the default game,
		// except that more than enough towers have been built before,
		// which should result in game win.

		game_pause = false;
		game_over = false;
		game_won = false;

		// create game
		LevelModel* level = new LevelModel();
		GameController* game = new GameController(level);
		game->setTestmode();
		LevelController* levelController = new LevelController(level, NULL);

		// create more than enough powerful towers
		SpriteFactoryAbstract* spriteFactory = new SpriteFactory();
		levelController->buildTower( 3, 4, spriteFactory->createTower(3) );
		levelController->buildTower( 4, 4, spriteFactory->createTower(3) );
		levelController->buildTower( 5, 4, spriteFactory->createTower(3) );
		levelController->buildTower( 6, 4, spriteFactory->createTower(3) );
		levelController->buildTower( 3, 5, spriteFactory->createTower(3) );
		levelController->buildTower( 4, 5, spriteFactory->createTower(3) );
		levelController->buildTower( 5, 5, spriteFactory->createTower(3) );
		levelController->buildTower( 6, 5, spriteFactory->createTower(3) );

		// execute game
		float time_steps = 1000.0/60.0;
		float time_accumulator = time_steps * 10000;
		while (time_accumulator > 0) {
			game->update(time_steps);
			time_accumulator -= time_steps;
		}

		delete game;
		delete levelController;
		delete spriteFactory;

		return game_won;
	}
};
```

7. Bad Smells: Large Class

Desde los torres y enemigos son las figuras integrales del videojuego, al final tenían mucho código. El clase torre por ejemplo contenía código para mantener el estado en el juego (fuerza, aspectos monetario) y estados cómo figura gráfica. Además contenía lógico para atacar enemigos. Al final parecían cómo clases demasiado grandes.
Se podría separar los clases siguiendo el principio Model-View-Controller. Hay aspectos de MVC en el código, pero también quería agrupar el código en clases modulares. No era tan claro cómo en los clases "GameModel/GameView/GameController" o "LevelModel/LevelController/LevelView". Fundamentalmente he separado variables y funciones de la clase y los puesto en otros clases. Así que la interfaz del clase todavía es lo mismo y no me parecía importante a hacer una prueba que la interfaz es lo mismo. (Lo se sabe ya cuando el código compilado). Explico los cambios en el código:

A un lado el modelo de una figura en el juego tiene dos aspectos: Un aspecto es monetario, es decir crear un torre cuesta dinero, luego se podrá venderlo también. Otros aspectos son la fuerza y la vida de las figuras. Por eso he separado dos modelos, *CostValueModel* y *GameFigureModel* de que hereda una figura. Es decir he usado el método de refactoriación "extract class". Por ejemplo los aspectos monetarios:

Before

```
class Enemy : virtual public Sprite
{
	...
public:
	int cost;
	int value;
	...
};
```

After

```
class Enemy : virtual public Sprite, public CostValueModel
{
	...
};

/**
 * class CostValueModel is a model for monetary items in the game that have a cost and value.
 * For example a tower costs money to be built and has a reduced value when sold again.
 * The value of an enemy is added to the player's money stock when an enemy is defeated.
 */
class CostValueModel
{
public:
	CostValueModel(int cost = 0, int value = 0)
	: _cost(cost), _value(value)
	{
	}
	int getCost() {
		return _cost;
	}
	int getValue() {
		return _value;
	}
	void setCost(int cost) {
		_cost = cost;
	}
	void setValue(int value) {
		_value = value;
	}

private:
	unsigned int _cost;
	unsigned int _value;

};
```

Y aquí los otros estados de la figura:

Before

```
class Sprite
{
	...
public:
	unsigned int id;

	MyVector position;
	MyVector position_path;

	unsigned int speed;
	unsigned int radius;

	float stamina_max;
	float stamina;
	...
};
```

After

```
class Sprite : public GameFigureModel
{
	...
};

class GameFigureModel
{
public:
	GameFigureModel()
	: _speed(0), _radius(0), _stamina(-1), _stamina_max(-1) {}

	unsigned int getID() {
		return _id;
	}
	void setID(unsigned int ID) {
		_id = ID;
	}

	float getStamina() {
		return _stamina;
	}
	void setStamina(float stamina) {
		_stamina = stamina;
	}
	...
private:

	unsigned int _id;

	MyVector _position;
	MyVector _position_path;

	unsigned int _speed;
	unsigned int _radius;

	float _stamina_max;
	float _stamina;
};
```

Es decir hay aspectos de extraer variables a otra clase, cambiar el nombre de variables y introducir métodos cómo "Getter" y "Setter".

Con el respecto de la apariencia (view) me parecía sano hacer algo que se llama un "*Protocol"* en Objective-C o "*Interface"* en Java. Es decir que la función para dibujar la figura (void draw()) sea definido en un clase virtual "*InterfaceDrawable*". Así creo que sigo el concepto de "lose coupling", que significa que no se necesite de saber algo del objeto, solo hay que saber que hay el protocol (aquí una función para dibujar).
...

La refactorización de este parte entonces era la siguiente:

Before

```
class Sprite
{
	...
public:
	virtual void draw();
	...
};
```

After

```
class Sprite : public InterfaceDrawable
{
	...
public:
	virtual void draw();
	...
};
```

Además quería que las figuras pueden atacarse. Por eso he separado las funciones relacionadas que estaban en el fichero Sprite.h en los clases *Attack* y *AttackController*. Es decir que por ejemplo había un clase Attack en Sprite.h y ahora está en su propio fichero Attack.h. Además el lógico de un ataque:

Before

```
class Sprite
{
	...
public:
	int countActiveAttacks();
	void continueActiveAttacks(float delta);
	Attack* createAttack(Sprite* source, Sprite* target, float target_distance);
	void removeAttack( Sprite* target );
	unsigned int getAttackPower();
	void setAttackPower(int attack_power);
	unsigned int attack_power;
	unsigned int numAttacks;
	Attack* attacks;

	void setNumAttacks(int num_attacks);
	...
};
```

After

```
class Enemy : virtual public Sprite, public AttackController
{
	...
};
class Tower : virtual public Sprite, public AttackController
{
	...
};

class AttackController
{
public:
	AttackController(int num_attacks = 1)
	: _attacks(NULL) {
		setNumAttacks( num_attacks );
	}

	int countActiveAttacks();
	void continueActiveAttacks(float delta);
	Attack* createAttack(GameFigureModel* source, GameFigureModel* target, float
target_distance);
	void removeAttack( GameFigureModel* target );
	unsigned int getAttackPower();
	void setAttackPower(int attack_power);
	unsigned int getNumAttacks();
	void setNumAttacks(int num_attacks);
private:

	unsigned int _attack_power;
	Attack* _attacks;
	unsigned int _num_attacks;
};
```

8. Bad Smells: Code Duplication

Descripción: Aquí muestro un ejemplo de duplicación de código en el proyecto. La creación de enemigos había hecho en tres métodos similares, porque hay tres tipos de enemigos y cada uno es poco más fuerte. Lo que pasó cada vez era crear un enemigo básico y multiplicar sus características con un factor variable. Así que se necesita solo una función que depende del factor cómo parámetro y los tres tipos de enemigos se crea con este función, usando un factor diferente.

Código antes y después:

Before

```
Enemy* SpriteFactory::createEnemy1()
{
	Enemy* sprite = createBasicEnemy();

	sprite->setID(ENEMYID_LASER);
	sprite->setAttackPower( sprite->getAttackPower() * 1.0);
	sprite->setStamina(     sprite->getStamina()     * 1.0);
	sprite->setStaminaMax(  sprite->getStaminaMax()  * 1.0);
	sprite->setSpeed(       sprite->getSpeed()       * 1.0);
	sprite->setCost(        sprite->getCost()        * 1.0);
	sprite->setValue(       sprite->getValue()       * 1.0);

	return sprite;
}

Enemy* SpriteFactory::createEnemy2()
{
	Enemy* sprite = createBasicEnemy();

	sprite->setID(ENEMYID_LASER2);
	sprite->setAttackPower( sprite->getAttackPower() * 2.5);
	sprite->setStamina(     sprite->getStamina()     * 2.5);
	sprite->setStaminaMax(  sprite->getStaminaMax()  * 2.5);
	sprite->setSpeed(       sprite->getSpeed()       * 2.5);
	sprite->setCost(        sprite->getCost()        * 1.5);
	sprite->setValue(       sprite->getValue()       * 1.5);

	return sprite;
}


Enemy* SpriteFactory::createEnemy3()
{
	Enemy* sprite = createBasicEnemy();

	sprite->setID(ENEMYID_LASER3);
	sprite->setAttackPower( sprite->getAttackPower() * 4.0);
	sprite->setStamina(     sprite->getStamina()     * 4.0);
	sprite->setStaminaMax(  sprite->getStaminaMax()  * 4.0);
	sprite->setSpeed(       sprite->getSpeed()       * 4.0);
	sprite->setCost(        sprite->getCost()        * 2.0);
	sprite->setValue(       sprite->getValue()       * 2.0);

	return sprite;
}
```

After

```
Enemy* SpriteFactory::createEnemy(int level)
{
	Enemy* result = NULL;

	if (level == 1) {
		result = createFactoredEnemy(ENEMYID_LASER, 1.0, 1.0);
	} else if (level == 2) {
		result = createFactoredEnemy(ENEMYID_LASER2, 2.5, 1.5);
	} else if (level == 3) {
		result = createFactoredEnemy(ENEMYID_LASER3, 4.0, 2.0);
	}
	return result;
}

Enemy* SpriteFactory::createFactoredEnemy(int id, float f_strength, float f_valor)
{
	Enemy* sprite = createBasicEnemy();

	sprite->setID(id);
	sprite->setAttackPower( sprite->getAttackPower() * f_strength);
	sprite->setStamina(     sprite->getStamina()     * f_strength);
	sprite->setStaminaMax(  sprite->getStaminaMax()  * f_strength);
	sprite->setSpeed(       sprite->getSpeed()       * f_strength);
	sprite->setCost(        sprite->getCost()        * f_valor);
	sprite->setValue(       sprite->getValue()       * f_valor);

	return sprite;
}
```

...

Prueba de refactorización:

La prueba de la refactorización usa el generador de los enemigos para generar enemigos sin saber cómo se lo hace. Así puede probar si la generación todavía es correcto después de se ha cambiado los métodos en la fábrica. Aquí muestro el código de la prueba:

```
class UnitTestEnemyGenerator : public UnitTestTemplate
{
public:

	UnitTestEnemyGenerator()
	: UnitTestTemplate("UnitTestEnemyGenerator") {
	}

protected:

	virtual bool executeTest() {

		// Generate enemies and check if the correct ones were generated using only ID

		LevelModel* level = new LevelModel();
		GameController* game = new GameController(level);
		game->setTestmode();
		LevelController* levelController = new LevelController(level, NULL);

		// execute generator
		float time_steps = 1000.0/60.0;
		float time_accumulator = time_steps * 10000;
		while (time_accumulator > 0) {

			// generate and add enemies
			vector<Enemy*> *new_enemies = game->getModel()->getEnemyGenerator()-
>generate( time_steps );
			game->getModel()->getLevel()->addEnemies( new_enemies );
			delete new_enemies;

			game->getModel()->getEnemyGenerator()->continueRound( time_steps );

			time_accumulator -= time_steps;
		}

		// compare IDs of generated enemies with expected IDs
		int expected_enemies[] = {
			ENEMYID_LASER,  ENEMYID_LASER,  ENEMYID_LASER2, ENEMYID_LASER2,
			ENEMYID_LASER2, ENEMYID_LASER2, ENEMYID_LASER3, ENEMYID_LASER3,
			ENEMYID_LASER3, ENEMYID_LASER3, ENEMYID_LASER3, ENEMYID_LASER3,
ENEMYID_LASER3, ENEMYID_LASER3, ENEMYID_LASER3, ENEMYID_LASER3
		};

		int i=0;
		bool success = true;
		for (vector<Enemy*>::iterator it = game->getModel()->getLevel()->getEnemies()-
>begin(); it != game->getModel()->getLevel()->getEnemies()->end(); it++) {
			success = success && (expected_enemies[i++] == (*it)->getID());
		}

		delete game;
		delete levelController;

		return success;
	}
};
```