# Final Reality

*Proyecto semestral*


**Nelson Baloian**
*Profesor*

**Matías Toro**
*Profesor*

**Beatriz Graboloza**
*Auxiliar*

**Francisco Muñoz**
*Auxiliar*

**Ignacio Slater**
*Auxiliar*

Santiago, Chile
2022-09-08
v2.0.0

**Resumen**

Uno de los principales objetivos del curso de Metodologías de diseño y programación es que lxs alumnxs aprendan a escribir programas de calidad utilizando buenas metodologías de programación. Para lograr este objetivo se le plantea un proyecto semestral dividido en 3 entregas en las que se espera ver aplicados los contenidos enseñados en el curso.

En este caso, el proyecto buscará simular un «caso real» en el que usted deberá programar un juego de acuerdo a un documento de requisitos. Los requisitos específicos serán explicados en el presente documento, pero para guiar el desarrollo de la solución se plantearan tareas periódicas opcionales donde cada entrega del proyecto será la acumulación de las tareas más pequeñas hasta la fecha de entrega. Las entregas parciales que compondrán cada una de las evaluaciones del proyecto se especificarán a través de *U-Cursos*.

Para comenzar con el proyecto se le entregará un código base sobre el que deberá construir su solución, teniendo total libertad de cambiar cuanto estime conveniente del código original mientras se mantengan buenas metodologías de diseño.

[0]Que las entregas parciales sean opcionales significa que no serán evaluadas individualmente ni se tomará en cuenta la fecha en que sean entregadas (esto se explicará en mayor detalle en la sección 4). Sin embargo, la evaluación de la tarea será la acumulación de las entregas parciales

# Índice

**1. Descripción del problema** **3**
1.1. Inventario . . . 3
1.2. Personajes . . . 3
1.3. Hechizos . . . 4
1.4. Enemigos . . . 4
1.5. Efectos adversos . . . 4
1.6. Turnos . . . 5

**2. Modelo de la solución** **5**
2.1. Arquitectura . . . 5
2.2. Implementación de los turnos . . . 5

**3. Código base** **7**
3.1. Enumeraciones . . . 7
3.2. Armas . . . 7
3.3. Personajes . . . 7
3.3.1. Personajes del jugador . . . 8

**4. Evaluación** **9**
4.1. Bonificaciones . . . 9
4.2. Requisitos adicionales . . . 9
4.3. Evaluación . . . 10
4.4. Recomendaciones . . . 10

# 1. Descripción del problema

El proyecto a realizar será crear un clon (simplificado) del combate de los juegos *Final Fantasy* desarrollado por Square Enix. A grandes rasgos para el combate el jugador cuenta con un grupo de personajes que puede manejar y un conjunto de enemigos que son controlados por la computadora.
A continuación se explicarán en detalle los requisitos que debe cumplir su programa al finalizar el proyecto. Para lo que sigue del documento se utilizarán **indistintamente** los términos jugador y usuario.

## 1.1. Inventario

El usuario debe tener un inventario con objetos que pueden ser equipados a sus personajes, estos tienen un nombre, ataque, peso y tipo.
Dependiendo de su tipo, los objetos podrán o no ser equipados a un personaje dado de acuerdo a la clase de dicho personaje como se especificará en la sección siguiente. Aparte de las clases a la que puede ser equipada cada arma todas tienen el mismo comportamiento. Los tipos de armas existentes son:

- Espada
- Hacha
- Cuchillos
- Bastones
- Arcos

Adicionalmente, los bastones tendrán también un daño mágico que se explicará más adelante

## 1.2. Personajes

El jugador tendrá a su disposición una cantidad fija de personajes que controlará, esta cantidad la definirá usted, pero debe asegurar que la cantidad de personajes sea exactamente esa. Cada personaje tiene un nombre, una clase, puntos de vida y defensa. Llamaremos *party* al grupo de personajes que controla el jugador.
Los personajes pueden agruparse en dos categorías, los personajes comunes y los magos.
Los personajes comunes son los que no pueden usar magias, en particular se componen de:

- Caballeros
- Ingenieros
- Ladrones

Por otro lado, los magos tienen un atributo adicional que llamaremos *mana* y que se utiliza para efectuar («pagar») hechizos. Los tipos son:

- Magos negros
- Magos blancos

Los tipos de armas que puede equiparse cada clase se muestran en el cuadro siguiente.
Tenga en consideración que un personaje debe tener exactamente un arma equipada para atacar, pero el arma con la que ataca puede ser cambiada en su turno por cualquier otra del inventario que cumpla los criterios presentados en el Cuadro 1.

| | Espada | Hacha | Cuchillos | Bastón | Arco |
|---|---|---|---|---|---|
| Caballero | Sí | Sí | Sí | No | No |
| Ingeniero | No | Sí | No | No | Sí |
| Ladrón | Sí | No | Sí | No | Sí |
| M. Negro | No | No | Sí | Sí | No |
| M. Blanco | No | No | No | Sí | No |

Cuadro 1: Personajes y tipos de armas que pueden equiparse.

### 1.3. Hechizos

Además de sus armas, los magos pueden utilizar hechizos con distintos efectos. Existen dos tipos de magias, blanca y negra que pueden ser utilizadas por los magos de ese mismo color (por simplicidad puede asumir que un mago blanco nunca tendrá hechizos de magia negra y viceversa).
Los hechizos existentes son:

- **Trueno (M. Negra):** Reduce la vida del oponente en `magicDamage` al enemigo y tiene un 30 % de probabilidad de paralizarle.
  *Costo*: 15 mana.
- **Fuego (M. Negra):** Reduce la vida del oponente en `magicDamage` al enemigo y tiene un 20 % de probabilidad de quemarlo.
  *Costo*: 15 mana.
- **Cura (M. Blanca):** Cura a un aliado el 30 % de sus puntos de vida máximos.
  *Costo*: 15 mana.
- **Veneno (M. Blanca):** Envenena a un enemigo.
  *Costo*: 40 mana.
- **Parálisis (M. Blanca):** Paraliza a un enemigo.
  *Costo*: 25 mana.

Donde `magicDamage` es el daño mágico del arma equipada al personaje (note que solamente los bastones tendrán daño mágico, para el resto de las armas será su trabajo decidir cuál sería la mejor forma de resolver este problema). De forma similar a las armas los magos pueden tener varios hechizos a su disposición para utilizar, en caso de decidir usar alguno debe seleccionarse exactamente uno.

### 1.4. Enemigos

En los combates, además de los personajes que controlará el jugador habrá uno o más enemigos. Cada enemigo tendrá un nombre, puntos de vida, defensa, ataque y peso. Los enemigos **no pueden** equiparse objetos ni usar hechizos.
A diferencia de los personajes, la cantidad de enemigos no es fija, pero debe definirse una cantidad máxima de enemigos que pueden participar en el combate

### 1.5. Efectos adversos

En el juego existirán tres tipos de efectos adversos, estos efectos son causados por el efecto de algún hechizo utilizado por el jugador. Los efectos son:

- **Parálisis**: Cuando un enemigo está paralizado no puede atacar. El efecto de la parálisis dura un turno.
- **Envenenado**: Los enemigos envenenados pierden $\frac{\texttt{magicDamage}}{3}$ puntos de vida al comienzo de cada uno de sus turnos. `magicDamage` es el daño mágico del arma que tenía equipado el personaje que envenenó al enemigo al momento de usar su habilidad.
- **Quemado**: Los enemigos quemados pierden $\frac{\texttt{magicDamage}}{2}$ puntos de salud al comienzo de cada uno de sus turnos. magicDamage se define igual que para el envenenamiento.

### 1.6. Turnos

En el contexto del juego definiremos como turnos los espacios en que el jugador esté utilizando a uno de sus personajes (seleccionando o realizando una acción) además de los momentos en los que los enemigos «decidan» qué hacer.
En el turno de cada personaje el jugador puede cambiar el arma o las magias de éste tantas veces como quiera, pero para terminar el turno **debe** atacar o utilizar un hechizo. Cuando se ataca o usa un hechizo el turno del personaje termina **inmediatamente** luego de que se realice la acción. El caso de los enemigos es análogo.
El orden de los turnos se maneja individualmente por personaje, esto quiere decir que cada personaje (del jugador o enemigo) tendrá una forma de definir cuándo le toca. Esto va a implicar que el orden de los turnos no necesariamente será el mismo siempre.
El orden de los turnos se definirá utilizando el peso del arma equipada al personaje o enemigo correspondiente y creando una tarea que se ejecutará luego de que pasen (al menos) $t_i$ segundos desde un momento $t_i^0$, con:

$$t_i^0\text{: el momento en que el personaje i termina su turno.}$$

$$t_i : \frac{\text{weight}}{10}$$

El detalle de cómo solucionar este problema se presentará en la sección siguiente.

## 2. Modelo de la solución

Para guiar la solución de este problema, se plantearan objetivos pequeños en forma de *entregas parciales* que buscarán enfrentar una problemática a la vez. Dichas tareas tendrán fecha de entrega pero será opcional entregar en la fecha señalada y no serán evaluadas con nota. Solamente será evaluado que al momento de entregar la tarea se cumplan todos los objetivos planteados en las *entregas parciales*.

### 2.1. Arquitectura

La resolución de este proyecto se hará siguiendo el patrón arquitectónico *Modelo-Vista-Controlador*, donde primero se implementará el *modelo*, luego el *controlador* y por último la *vista*. Este patrón se explicará en más detalle en el transcurso del curso, pero en el contexto del proyecto estos componentes serán como se explica a continuación.

**Modelo** Para la primera parte se le solicitará que cree todas las entidades necesarias que servirán de estructura base del proyecto y las interacciones posibles entre dichas entidades. Las entidades en este caso se refieren a los elementos que componen el juego.

**Vista** Se le pedirá también que cree una interfaz gráfica simple para el juego que pueda responder al input de un usuario y mostrar toda la información relevante del juego en pantalla.

**Controlador** Servirá de conexión lógica entre la vista y el modelo, se espera que el controlador pueda ejecutar todas las operaciones que un jugador podría querer efectuar, que entregue los mensajes necesarios a cada objeto del modelo y que guarde la información más importante del estado del juego en cada momento.

### 2.2. Implementación de los turnos

Para manejar el orden de los turnos se utilizará una **cola** (FIFO) y un **scheduler de tareas**. El *scheduler de tareas* servirá de «temporizador» para decidir cuándo se debe notificar que llegó el turno de un personaje. Cuando se cumpla el tiempo definido en el temporizador se agregará al personaje al final de la cola.
Tenga en cuenta que mientras el juego espera que el usuario tome una decisión el temporizador seguirá corriendo, razón por la que se necesita la cola.
Los turnos, entonces, se implementarán como sigue:

[0]https://www.github.com/CC3002-Metodologias/apunte-y-ejercicios/wiki/Modelo-Vista-Controlador

1. Se toma al primer personaje en la cola.
    - Si es un personaje controlado por el jugador se espera a que éste tome una decisión sobre qué acción realizar con el personaje.
    - Si es un enemigo selecciona un personaje del jugador de manera aleatoria y realiza un ataque.
2. Se saca al personaje de la cola.
3. Se inicia un temporizador acorde al peso.
4. Si hay más personajes en la cola vuelve a 1.
5. Si la cola está vacía espera vuelva a contener personajes y luego vuelve a 1.

Note que dada la naturaleza de las tareas puede darse el caso de que dos personajes quisieran agregarse a la cola al mismo tiempo, por esta razón utilizaremos la estructura `LinkedBlockingQueue` que provee *Java* que soluciona este problema.
A continuación se muestra las implementaciones de ambas mecánicas.

```
public void waitTurn() {
  scheduledExecutor = Executors.newSingleThreadScheduledExecutor();
  if (this instanceof PlayerCharacter player) {
    scheduledExecutor.schedule(
        /* command = */ this::addToQueue,
        /* delay = */ player.getEquippedWeapon().getWeight() / 10,
        /* unit = */ TimeUnit.SECONDS);
  } else {
    var enemy = (Enemy) this;
    scheduledExecutor.schedule(
        /* command = */ this::addToQueue,
        /* delay = */ enemy.getWeight() / 10,
        /* unit = */ TimeUnit.SECONDS);
  }
}

private void addToQueue() {
  try {
    turnsQueue.put(this);
  } catch (Exception e) {
    e.printStackTrace();
  }
  scheduledExecutor.shutdown();
}
```

Además, se entrega un pequeño programa que ilustra el funcionamiento de esto último.

```
BlockingQueue<GameCharacter> queue = new LinkedBlockingQueue<>();
Random rng = new Random();
for (int i = 0; i < 10; i++) {
  // Gives a random speed to each character to generate different waiting times
  var weapon = new Weapon("", 0, rng.nextInt(50), WeaponType.KNIFE);
  var character = new Thief(Integer.toString(i), 10, 10, queue);
  character.equip(weapon);
  character.waitTurn();
}
// Waits for 6 seconds to ensure that all characters have finished waiting
Thread.sleep(6000);
```

```
while (!queue.isEmpty()) {
  // Pops and prints the names of the characters of the queue to illustrate the turns
  // order
  System.out.println(queue.poll().toString());
}
```

Tenga en cuenta que no necesitará (ni se le recomienda) utilizar métodos adicionales a los que se muestran en el ejemplo para implementar el proceso de *espera* de los turnos. Del ejemplo, el método `Queue#add(Object)` agrega un elemento a la cola (*push*) y `Queue#poll(Object)` saca y retorna el primer elemento de la cola (*pop*).

# 3. Código base

Es importante notar que si bien el código entregado funciona será su trabajo **juzgar si su diseño es apropiado**, por lo cual tiene total libertad de modificarlo a conveniencia.

## 3.1. Enumeraciones

Para representar los tipos de armas se utilizan enumeraciones, puede pensar en una enumeración como una clase que se compone únicamente de constantes.

```
public enum WeaponType {
  SWORD, AXE, KNIFE, STAFF, BOW
}
```

## 3.2. Armas

Las armas están definidas por la clase `Weapon`. Esta clase es bastante simple, formada solamente de un constructor, los *getters* y *setters* necesarios, el método `equals(Object)` y la función de *hashing*. Note que la firma del método `equals` recibe un parámetro de tipo `Object`, esto **debe ser así** ya que ese es el método que será llamado por los tests al hacer `assertEquals`.

## 3.3. Personajes

Todos los personajes del juego implementan la interfaz **GameCharacter** y por lo tanto deben implementar las siguientes funcionalidades:

```
void waitTurn();
String getName();
int getCurrentHp();
int getMaxHp();
int getDefense();
void setCurrentHp(int hp) throws InvalidStatValueException;
```

La clase abstracta `AbstractCharacter` implementa directamente la interfaz y da definiciones para las funcionalidades comunes de todos los personajes. Es altamente probable que tenga que modificar esta clase a lo largo del proyecto, en ese caso, debe tener especial cuidado con los llamados a `scheduledExecutor` y `turnsQueue`. Esas variables están pensadas para usarse como están en el código base, puede moverlas de lugar (incluso a otra clase) pero en ningún caso deberá borrarlas y el orden de llamadas debe mantenerse de la misma forma que se entrega. A continuación se explicará un poco más en detalle la función que cumplen éstas.
La línea:

```
scheduledExecutor = Executors.newSingleThreadScheduledExecutor();
```

[0]Tenga cuidado con los nombres que le de a sus clases. La clase `Character` es parte de la librería estándar de *Java* así que no puede crear clases con ese nombre en su proyecto.

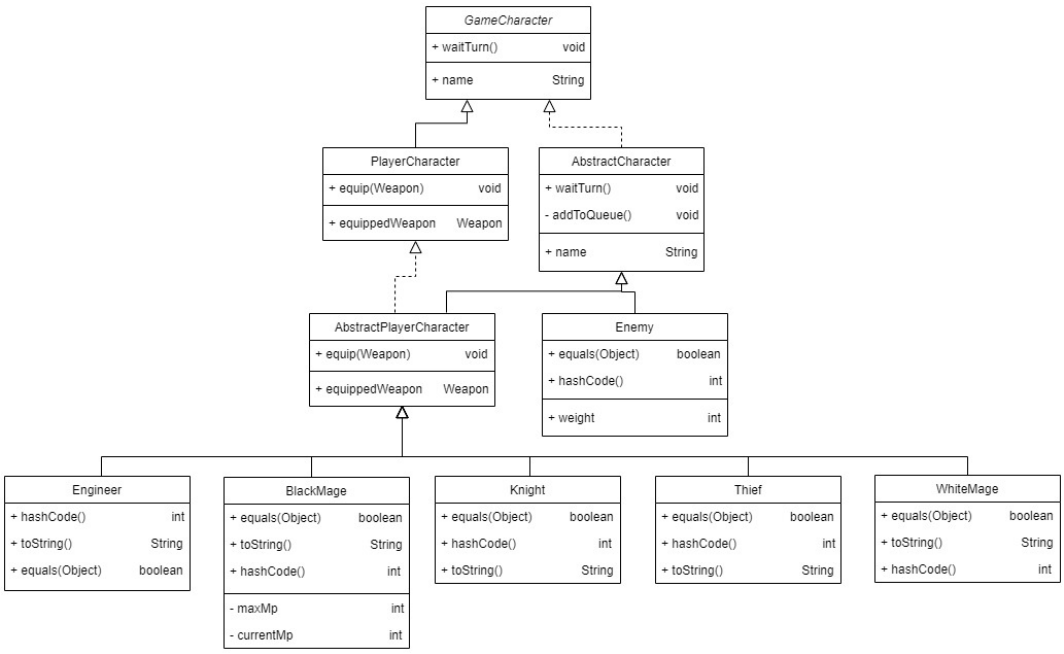


Figura 1: Clases de personajes

crea un nuevo «timer» asociado al objeto que lo creó. Al ir asociado al objeto significa que cada personaje tendrá su propio *scheduler* independiente de los otros. No debe preocuparse de problemas de concurrencia ni paralelismo como *data races*, *dead-locks*, etc.
Luego, la instrucción:

```
scheduledExecutor.schedule(
    /* command = */ this::addToQueue,
    /* delay = */ player.getEquippedWeapon().getWeight() / 10,
    /* unit = */ TimeUnit.SECONDS);
```

le dice al *scheduler* que ejecute el método `addToQueue()` luego de que pasen `player.getEquippedWeapon().getWeight() / 10` segundos.
El caso de los enemigos es análogo.
Por último, en las llamadas a

```
turnsQueue.add(this);
scheduledExecutor.shutdown();
```

primero se agrega al personaje a la cola de turnos y luego se termina la ejecución del «temporizador» hasta que se comience uno nuevo con `Executors.newSingleThreadScheduledExecutor()`. No es necesario que se preocupe de eliminar el temporizador original luego de apagarlo ya que el *Garbage collector* de *Java* se encargará de eso.

### 3.3.1. Personajes del jugador

Para representar a los personajes del jugador se utilizan la interfaz `PlayerCharacter` y la clase abstracta `AbstractPlayerCharact`
La única funcionalidad adicional que vienen implementadas en el código base es la de equipar armas (la cual se realiza con el método `equip(Weapon)`) y la de obtener el arma equipada (la cual se realiza con el método `getEquippedWeapon()`).

```
@Override
public void equip(Weapon weapon) {
```

```
  this.equippedWeapon = weapon;
}

@Override
public Weapon getEquippedWeapon() {
  return equippedWeapon;
}
```

# 4. Evaluación

## 4.1. Bonificaciones

Además del puntaje asignado por cumplir con los requisitos anteriores, puede recibir puntos adicionales (*que se sumarán a su nota total*) implementando lo siguiente:

- **Entregas parciales** (0.4 pts.): De acuerdo a las entregas que haya hecho de las entregas parciales puede recibir una bonificación de hasta 0.4 pts. Entregue las entregas parciales solamente si considera que están completas o altamente completas. Entregas parciales demasiado incompletas puede significar un descuento en el puntaje.[1]
- **Manejo de excepciones** (0.3 pts.): Se otorgará puntaje por la correcta utilización de excepciones para manejar casos de borde en el juego. Recuerde que atrapar *runtime exceptions* es una mala práctica, así como arrojar un error de tipo `Exception` (esto último ya que no es un error lo suficientemente descriptivo). Bajo ninguna circunstancia una de estas excepciones debiera llegar al usuario.

TAREA 3 **Interfaz gráfica avanzada** (0.5 pts.): Si su interfaz gráfica cumple más de los requisitos mínimos podrá recibir una bonificación de hasta 0.5 pts. (esto quedará a criterio del ayudante que le revise).

## 4.2. Requisitos adicionales

Que su programa funcione no es suficiente, se espera además que éste presente un buen diseño, siendo esta la característica a la que se le dará **mayor importancia** en la revisión de sus entregas.
Sus programas además deberán estar bien testeados, por lo que **NO SE REVISARÁN** las funcionalidades que no tengan un test correspondiente que pruebe su correcto funcionamiento. Para revisar esto, también es de suma importancia que el trabajo que entregue pueda ejecutarse (que su código compile), en caso contrario no podrá revisarse la funcionalidad ni *coverage* y por ende no tendrá puntaje en este aspecto.
Otro aspecto que se tendrá en cuenta al momento de revisar sus tareas es que estas estén bien documentadas, siguiendo los estándares de documentación para Java, Kotlin y Scala.
Todo su trabajo debe ser subido al repositorio privado que se le entregó a través de *Github Classroom*. La modalidad de entrega será mediante un resumen en **formato TXT** entregado mediante *u-cursos* que tenga el formato:

```
Nombre: <Su nombre>
PR: <URL del Pull Request>
```

Además su repositorio deberá contener un archivo **README.md** que contenga todas las instrucciones necesarias para ejecutar su programa, todos los supuestos que realice y una breve explicación del funcionamiento y la lógica de su programa.
Para la tarea deberá crear su propia rama para trabajar y subir todo su código en esta y, cuando tenga una entrega lista para ser revisada, realizar un **pull request**. Tenga en cuenta que si hace *push* de otros *commits* en esa rama se actualizará el PR, por lo que siempre debe crear una nueva rama para seguir trabajando.

[1] Esto es solamente para asegurarnos de que no se aprovechen e intenten tener los puntos adicionales sin haber trabajado.

## 4.3. Evaluación

- **Código fuente (4.0 puntos):** Este ítem se divide en 2:
  - **Funcionalidad (1.5 puntos):** Se analizará que su código provea la funcionalidad pedida. Para esto, se exigirá que testee las funcionalidades que implementó[2]. **Si una funcionalidad no se testea, no se podrá comprobar que funciona y, por lo tanto, NO SE ASIGNARÁ PUNTAJE por ella.**
  - **Diseño (2.5 puntos):** Se analizará que su código provea la funcionalidad implementada utilizando un buen diseño.
- **Coverage (1.0 puntos):** Sus casos de prueba deben crear diversos escenarios y contar con un *coverage* de al menos 90 % de las **líneas**. No está de más decir que sus tests deben *testear* algo (es decir, no ser tests vacíos o sin *asserts*).
- **Javadoc y estilo (1.0 puntos):** Cada clase, interfaz y método público debe ser debidamente documentado y su código no debe arrojar ningún warning al ser revisado con el *linter*:
  - **Java:** *CheckStyle* bajo el estándar de Google.
  - **Scala:** *Scalastyle*
  - **Kotlin:** *Ktlint*

  Se descontará 0,01 puntos por cada falta.

## 4.4. Recomendaciones

Como se mencionó anteriormente, no es suficiente que su tarea funcione correctamente. Este curso contempla el diseño de su solución y la aplicación de buenas prácticas de programación que ha aprendido en el curso. Dicho esto, no se conforme con el primer diseño que se le venga a la mente, intente ver si puede mejorarlo y hacerlo más extensible.

**No comience su tarea a último momento.** Esto es lo que se dice en todos los cursos, pero es particularmente importante/cierto en este. Si usted hace la tarea a último minuto lo más seguro es que no tenga tiempo para reflexionar sobre su diseño, y termine entregando un diseño deficiente o sin usar lo enseñado en el curso. Haga la documentación de su programa en inglés (no es necesario). La documentación de casi cualquier programa *open-source* se encuentra en este idioma. Considere esta oportunidad para practicar su inglés. Se les pide encarecidamente que las consultas referentes a la tarea las hagan por el **foro de U-Cursos**. En caso de no obtener respuesta en un tiempo razonable, pueden hacerle llegar un correo a los auxiliares o ayudantes.

Por último, el orden en el que escriben su programa es importante, se le sugiere que para cada funcionalidad que quiera implementar:

1. Cree los *tests* necesarios para verificar la funcionalidad deseada, de esta manera el enfoque está en como debería funcionar ésta, y no en cómo debería implementarse. Esto es muy útil para pensar bien en cuál es el problema que se está buscando resolver y se tengan presentes cuales serían las condiciones de borde que podrían generar problemas para su implementación.
2. Escriba la firma y la documentación del método a implementar, de esta forma se tiene una definición de lo que hará su método incluso antes de implementarlo y se asegura de que su programa esté bien documentado. Además, esto hace más entendible el código no solo para alguna persona que revise su programa, sino que también para el mismo programador.
3. Por último implemente la funcionalidad pensando en que debe pasar los tests que escribió anteriormente y piense si estos tests son suficientes para cubrir todos los escenarios posibles para su aplicación, vuelva a los pasos 1 y 2 si es necesario.



[2]Se le recomienda enfáticamente que piense en cuales son los casos de borde de su implementación y en las fallas de las que podría aprovecharse un usuario malicioso.

[2]Entender un programa mal documentado que haya escrito uno mismo después de varios días de no trabajar en él puede resultar bastante complicado.