**2.2 Otros requisitos**

Además de los requisitos anteriores, se tienen en cuenta otra serie de requisitos que se indican en la siguiente tabla:

|  |  |
| --- | --- |
| **REQ.** | **DESCRIPCIÓN** |
| RI01 | La interfaz sobre la que se realizarán las pruebas será NetBeans |
| RD01 | Se probarán las herramientas sobre el código php disponible en el repositorio del grupo (calculadora) |
| RR01 | Los tests se realizarán sin tener ningún otro servicio en el PC, ya que podría afectar en los tiempos |

Glosario de términos:

RF: requisito funcional

RI: requisito de interfaz

RD: requisito de datos

RR: requisito de rendimiento

**4. Proyecto de implementación de un prototipo del sistema utilizando PHPUnit**

A continuación, se describe toda la documentación necesaria para entender el desarrollo de los test realizados con PHPunit, las pruebas realizadas durante el proceso de implementación y la configuración requerida para poder probar el proyecto en cualquier otro equipo.

**4.1 Documentación de diseño**

Hemos creado en primer lugar un código en php de una calculadora con las funciones principales (suma, resta, multiplicación y división), la cual nos sirve para realizar el cometido de esta práctica, testing automático para php, y en base a ese código hemos realizado las pruebas con PHPUnit.

Hemos diseñado los siguientes casos de prueba para comprobar que cada una de las funciones realiza lo que se espera de ella. Comprobando con estos test que se realiza correctamente la suma, la resta, la multiplicación y la división, controlando en este último caso la división entre cero como una excepción.

* TestSumar
* TestRestar
* TestMultiplicar
* TestDividir

**4.2 Documentación de construcción**

* TestSumar

Para realizar la función de suma el prototipo que se ha seguido ha sido el siguiente uso de código PHP que suma los números a y b:

function sumar($a, $b) {

        return ($a + $b);

    }

Para realizar su testing, hemos cinco funciones que realizan una suma de dos números dados y comprueba que el dato introducido sea correcto al dado como se puede ver en el caso del siguiente ejemplo que suma 0+0 y como resultado tiene que proporcionar el valor de 0:

public function testSumar() {

        $this->assertEquals(

                0, $this->object->sumar(0, 0)

        );

    }

* TestRestar

Para realizar la función de restar dos números se ha utilizado el siguiente fragmento de código que resta el número a menos el número b:

function restar($a, $b) {

        return ($a - $b);

    }

En cuanto al testing de esta operación, se han elaborado cuatro casos en los que dados dos números se comprueba que al realizar la resta el resultado obtenido es el resultado deseado, como por ejemplo en el fragmento de código siguiente que resta 1-2 dando como resultado –1.

public function testRestar4() {

        $this->assertEquals(

                -1, $this->object->restar(1, 2)

        );

    }

* TestMultiplicar

La función de multiplicar se ha realizado usando el siguiente código que multiplica el primer valor a por el segundo valor b:

function multiplicar($a, $b) {

        return ($a \* $b);

    }

Para realizar el testing de la función de multiplicar, se han elaborado cuatro casos que realizan la propia multiplicación de dos números comprobando que su resultado es correcto, como puede observarse el en código siguiente que multiplicando 1x1 obtenemos el resultado 1.

public function testMultiplicar3() {

        $this->assertEquals(

                1, $this->object->multiplicar(1, 1)

        );

    }

* TestDividir

La última función de la calculadora es la división de dos números, teniendo en cuenta que el denominador de dicha división no puede ser 0, ya que en tal caso no sería posible realizar la operación y nos mostraría un error diciendo que la división por cero no es posible.

function dividir($a, $b) {

        if ($b == 0) {

            throw new \InvalidArgumentException("Division by zero is not possible");

        }

        return ($a / $b);

    }

En cuanto al testing de esta última operación se han realizado un caso, incluyendo el posible error de dividir entre 0:

public function testDividir2() {

        $this->object->dividir(1, 0);

    }

Y realizando otros tres casos, observando que el resultado obtenido al dividir dos números introducidos es correcto:

public function testDividir3() {

        $this->assertEquals(

                1, $this->object->dividir(1, 1)

        );

    }

**6. Comparación de las dos implementaciones**

Teniendo en cuenta los criterios de comparación definidos en el apartado 3 sobre la implementación de cada uno de los prototipos, se ha fijado la evaluación de los criterios en la implementación usando ambas herramientas.

**6.1 Evaluación de los criterios en la implementación usando PHPUnit**

|  |  |
| --- | --- |
| **CRITERIO** | **EVALUACIÓN** |
| C1 Dificultad instalación SW | 3 horas |
| C2 Recursos necesarios | 3 herramientas |
| C3 Tiempo de respuesta | 0,000 segundos |
| C4 Información disponible | 2 |
| C5 Implementación intuitiva | 2 |

**7. Comparación de la implementación de las tecnologías**

|  |  |  |  |
| --- | --- | --- | --- |
| **CRITERIOS** | **PHPUnit** | **Codeception** | **COMENTARIOS** |
| C1 Dificultad instalación SW | 3 horas | 4 horas | Se ha tardado una hora más en Codeception, mayor complejidad. |
| C2 Recursos necesarios | 3 herramientas | 3 herramientas | Necesitamos la misma cantidad de herramientas- |
| C3 Tiempo de respuesta | 0,000 segundos | 0,007 segundos | En el código PHP creado se tarda más en realizarse las pruebas en Codeception, al ser un código tan pequeño los tiempos son mínimos, tanto es así, que en PHPUnit nos da 0,000 segundos. |
| C4 Información disponible | 2 | 1 | Se ha encontrado algo más de información en PHPUnit, pero aun así la cantidad de información encontrada ha sido muy reducida y difícil de encontrar. |
| C5 Implementación intuitiva | 2 | 1 | Como era más difícil encontrar la información en Codeception, ante un problema nos ha costado más resolverlo, y encima hemos tenido más problemas con Codeception para hacer funcionar sus tests. |

**8. Conclusiones**

Tal y como se explicó en el trabajo anterior PHPUnit y Codeception presentan funcionalidades similares.

Codeception es más aconsejable para aplicaciones grandes ya que la velocidad que ofrece para pruebas complejas es mayor. También ofrece una mayor usabilidad a la hora de entender los test de prueba. Sin embargo, el esfuerzo requerido para la puesta en marcha y la configuración es mucho mayor. Esto es debido a que tiene mayor complejidad, pero también a que hay mucha menos documentación disponible.

En este caso en concreto, la aplicación que se ha desarrollado es pequeña y los tests requeridos no implican una dificultad excesiva por lo que la mejor opción es PHPUnit