Pruebas unitarias en Python

Introducción

Las metodologías ágiles son consideradas como una evolución de la metodología clásica o de cascada. Una de las diferencias principales es la forma en la que plantean la fase de testing o prueba: para el desarrollo en cascada el testing es una fase que viene una vez terminada la fase de desarrollo, mientras que para el desarrollo ágil las pruebas son tan importantes o inclusive más que el código fuente.

Por definición cuando un programador se equivoca al codificar está introduciendo un error o *bug* al software (por ejemplo, utilizar *=* en lugar de *==*, equivocarse en el orden de los paréntesis o usar la variable equivocada). Dicho error provoca un defecto o *fault* en una sentencia que, al ser ejecutada, provoca una falla o *failure*. Esta falla, sin embargo, puede ser evidente o no. Si es evidente se vuelve un *incidente* que será reportado al observarse (por ejemplo, por el usuario final) pero puede ser que sea tan sutil que no sea notado en mucho tiempo.

Por consiguiente no cabe la menor duda de la conveniencia de encontrar errores tan pronto como sea posible, y las pruebas unitarias son la forma aceptada de hacerlo. Sin embargo, muchas veces se corre el peligro de no crearlos por problemas en el código, no ejecutarlos por ser lentos (ya sea por no estar optimizados y probar combinaciones innecesarias o por estar mal planteados), o no actualizarlos por ser débiles (dejan de compilar tan pronto como modificaciones son insertadas en el software por tener un alto acoplamiento, porque cuesta crear nuevas pruebas por tener baja cohesión, etc).

Una de las metodologías ágiles que intenta evitar dichos problemas es *TDD* o *Test-driven Development* (desarrollo guiado por pruebas) la cual enfatiza mantener el código en sincronía con las pruebas. Para ello se enfoca en hacer que el equipo de desarrollo interprete correctamente las especificaciones y las transforme en pruebas unitarias antes de iniciar la codificación. Ésta es la metodología que se usará en los ejemplos de este trabajo, intentando demostrar la importancia y conveniencia de las pruebas unitarias.

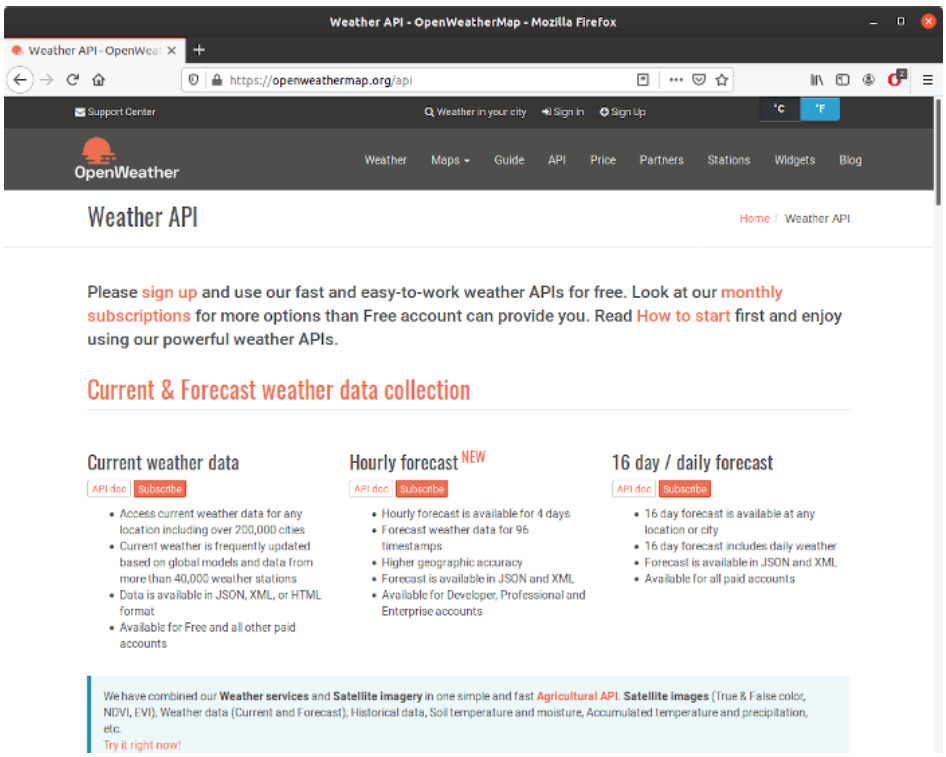
Ambiente de desarrollo

Para empezar con un ambiente limpio de desarrollo se pueden seguir las instrucciones de la documentación oficial, ejecutando `python3 -m venv unit-testing` para luego seleccionarla con `source unit-testing/bin/activate` en Linux (o `unit-testing\scripts\activate` en Windows). Este paso es opcional, ya que es totalmente posible utilizar un ambiente ya creado o la instalación original.

En cuanto a entorno de desarrollo, es posible ejecutar los ejemplos utilizando tanto línea de comando o interface gráfica (por ejemplo, Visual Studio Code). Los ejemplos aquí dados serán ejecutados por línea de comando por conveniencia, para poder mostrar el resultado de los comandos sin necesidad de incorporar imágenes.

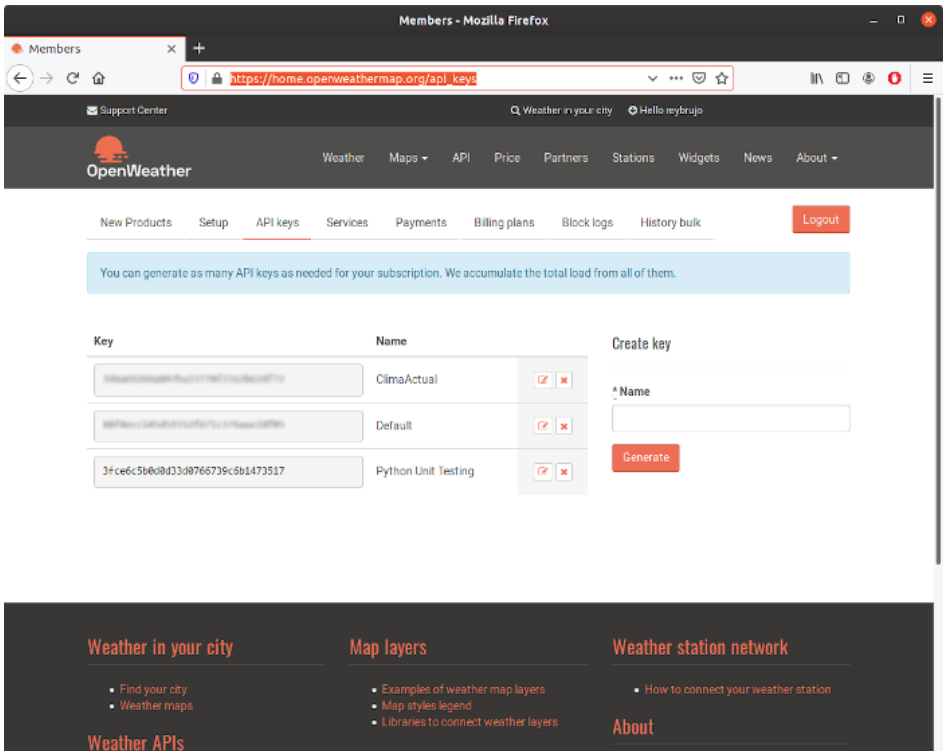
Registración de cuenta en OpenWeather

Para tener una idea de los datos que se pueden recibir del servidor es necesario verificar la API pública de *OpenWeather*, para lo cual hay que registrarse desde la sección de Weather API en el sitio.



Página con las instrucciones sobre como usar la API de clima.

Una vez accedido a la cuenta es posible crear una nueva API key (por ejemplo, *Python Unit Testing*). Esta *key* es necesaria para que puedan limitar la cantidad de veces que una aplicación accede a los datos.



Creación de una API key para poder acceder a los datos a través de la API.

En el plan gratuito tanto el clima actual como el pronóstico del tiempo para los próximos 5 días (en secciones de 3 horas, o sea 40 datos en total) pueden usarse con un máximo de 60 llamadas por minuto. Por ser una prueba de concepto se utilizará únicamente la *API* del clima actual.

Ejemplo de datos del clima actual

Los datos del clima actual se piden a través de la dirección: `https://api.openweathermap.org/data/2.5/weather?q={city name},{country code}&appid={apikey}`. El ejemplo que tiene el sitio se puede acceder desde aquí, y genera la siguiente respuesta:


```
{
    "coord": {
        "lon": -0.13,
        "lat": 51.51
    },
    "weather": [
        {
```

```
                "id": 300,
                "main": "Drizzle",
                "description": "light intensity drizzle",
                "icon": "09d"
            }
        ],
        "base": "stations",
        "main": {
            "temp": 280.32,
            "pressure": 1012,
            "humidity": 81,
            "temp_min": 279.15,
            "temp_max": 281.15
        },
        "visibility": 10000,
        "wind": {
            "speed": 4.1,
            "deg": 80
        },
        "clouds": {
            "all": 90
        },
        "dt": 1485789600,
        "sys": {
            "type": 1,
            "id": 5091,
            "message": 0.0103,
            "country": "GB",
            "sunrise": 1485762037,
            "sunset": 1485794875
        },
        "id": 2643743,
        "name": "London",
        "cod": 200
    }
```


Conociendo los datos disponibles se pueden seleccionar los que se van a mostrar durante la ejecución de la aplicación: el título del clima y su descripción (localizado en *weather)*, las temperaturas (dentro de *main)*, el nombre de la ciudad (en *name)*, las coordenadas (en *coord)* y el nombre del país (en la raíz).

Lineamientos principales de una prueba

De acuerdo a la metodología de desarrollo guiado por pruebas o *TDD* se empieza planeando la prueba unitaria antes de empezar a programar el software en sí. Para realizar pruebas unitarias utilizaremos la librería unittest.

Se crea un directorio llamado *clima* y dentro de ése un archivo llamado *clima_test.py* dentro del cual se escribirá la primera prueba unitaria:

clima_test.py

```
import unittest
import clima

class ClimaUnitTests(unittest.TestCase):
    def test_constructor_con_una_apikey_valida_deberia_incorporarla(self):
        # Arrange
        apikey = "abcdefg"

        # Act
        sut = clima.Clima(apikey)

        # Assert
        self.assertEqual(apikey, sut.apikey)

if __name__ == "__main__":
          unittest.main()
```

Como puede apreciarse se importa tanto el módulo *unittest* (que permitirá ejecutar las pruebas unitarias) y un módulo llamado *clima* (que aún no existe). Luego se define una clase llamada *ClimaUnitTests* que hereda de *unittest.TestCase*. Las pruebas unitarias se agrupan en casos de prueba o *test cases*, cada método dentro de la clase se volverá una prueba a ejecutar.

La clase *ClimaUnitTests* tiene un método llamado *test_constructor_con_una_apikey_valida_deberia_incorporarla*.
Los nombres deben ser descriptivos, en este caso se utiliza primero *test* (el cual es obligatorio para que sea detectado como una prueba), luego el nombre del método a probar, luego una frase que utiliza "con" y la conjugación en potencial del verbo "deber". En otras palabras, *metodo_con_xxx_deberia_yyy*. En el ejemplo se quiere probar que al construir un objeto de la clase *Clima* y pasarle una *API key*, dicha *API key* se incorpora dentro del estado del objeto y por consiguiente se puede acceder a través de la propiedad *apikey*.

Aquí además se puede apreciar una característica de las pruebas unitarias: Se dividen en tres secciones: Arrange (arreglo, preparación o inicialización), Act (acto o ejecución) y Assert (aseveración, validación o comprobación). Aunque dependiendo del desarrollador se puede agregar un comentario al inicio de cada sección o dejar una línea en blanco entre ellas, lo importante es que se puedan distinguir perfectamente (especialmente la sección donde se realiza la ejecución de la línea a probar, que debería ser única. La sección de preparación usualmente tiene como máximo unas 10 líneas, si fuese más largo podría ser conveniente dividir la funcionalidad en secciones más

cortas. Y aunque normalmente se menciona que también debería existir una única aseveración, en realidad es más correcto pensar en "tantas aseveraciones como sean necesarias para probar que la prueba en sí no haya fallado". Por ejemplo, si se quiere probar que es posible crear una instancia de un objeto con verificar que la hipotética función *crear* devuelve algo que no sea nulo. Sin embargo, si la prueba consiste en verificar que la instancia creada por dicha hipotética función está inicializada correctamente, podrían ser necesarias una comprobación por cada atributo.

Finalmente se llama al método *unittest.main* para ejecutar las pruebas.

```
(unit-testing) usuario@desktop:~/src/python/clima$ python3 clima_test.py
Traceback (most recent call last):
  File "clima_test.py", line 2, in <module>
    import clima
ModuleNotFoundError: No module named 'clima'
```

Obviamente al intentar ejecutar el programa (ya sea desde línea de comando o con una interface gráfica) ocurrirá un error indicando que no existe el módulo "clima". Esto forma parte del planteo *TDD*: primero se debe provocar el error para partir sabiendo que el caso de prueba falla. El siguiente paso es escribir el mínimo código necesario para que la prueba sea exitosa, y por último refactorear el código para eliminar código duplicado. Esto normalmente se conoce como *red*, *green* y *yellow/blue/refactor* en la jerga: primero se provoca el error (estado "rojo"), luego se implementa la solución más simple (estado "verde"), y finalmente se procede a limpiar el código, a refactorearlo para mejorar u optimizar su funcionamiento (etapa "amarilla"). Ya que en la segunda etapa se está seguro que se ha implementado correctamente la solución es posible pasar a la tercer etapa; cualquier modificación que haga fallar a la/s prueba/s indicará que se debe volver al código anterior de la modificación o buscar la forma de repararlo. En otras palabras, no tiene sentido refactorear código que no funciona.

Aunque la prueba falló viendo el código de la misma se puede notar que ya está decidido el nombre de la clase (*Clima*), que debe tener un constructor que reciba al menos la *API key*, y que debe tener una propiedad que retorne dicho valor. En base a esas premisas es posible escribir el siguiente código en un archivo llamado *clima.py*:

clima.py

```
class Clima:
    def __init__(self, apikey: str):
```

```
        self.apikey = apikey
```

Esta es la implementación mínima elegida: Una clase que tiene un constructor que recibe la *API key* y la guarda en un miembro que puede ser accedido desde afuera. Otra implementación mínima válida para *TDD* es crear una propiedad que retorne exactamente el mismo valor "abcdefg", ya que permitiría pasar la prueba (de hecho, esa sería la implementación en la mayoría de los tutoriales o charlas dirigidas a novatos, por razones de brevedad es omitida).

Esta vez al ejecutar la prueba validará:

```
(unit-testing) usuario@desktop: ~/src/python/clima$ python3 clima_test.py
.
----------------------------------------------------------------------
Ran 1 test in 0.000s

OK
```

El punto que aparece en la segunda línea es la representación de la ejecución de la prueba. Un . o punto indica que se ejecutó con éxito, una E indicaría un error inesperado en tiempo de ejecución, y una F indicaría que alguna de las comprobaciones hechas resultó falsa. Como por el momento no hay nada que refactorear se puede agregar una nueva funcionalidad a la clase: si se le intenta pasar una *API key* vacía debería lanzar una excepción. Para ello primero se implementa una nueva prueba:

clima_test.py

```
    def test_constructor_con_una_apikey_vacia_deberia_lanzar_valueerror(self):
        with self.assertRaises(ValueError) as ve:
```

El borde en zigzag indica que no se está mostrando el código completo del archivo sino que se está mostrando solo una porción de código por lo que se deberá prestar atención al copiarlo.

Lo que se quiere es que al intentar crear una clase sin una *API key* no debería permitirlo y debería lanzar una excepción, así que se utiliza *with* para crear un contexto en el cual se atrapará únicamente la excepción *ValueError* y se intenta crear la clase. Si dicha excepción es lanzada la prueba continuará (ya que es lo esperado), si una excepción distinta es lanzada, o si no se lanza ninguna excepción, la prueba fallará. Al ejecutar la prueba:

```
(unit-testing) usuario@desktop:~/src/python/clima$ python3 clima_test.py
```

```
F.
======================================================================
FAIL: test_constructor_con_una_apikey_vacia_deberia_lanzar_valueerror
(__main__.ClimaUnitTests)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "clima_test.py", line 13, in
test_constructor_con_una_apikey_vacia_deberia_lanzar_valueerror
    sut = clima.Clima("")
AssertionError: ValueError not raised

----------------------------------------------------------------------
Ran 2 tests in 0.000s

FAILED (failures=1)
```

La prueba falla porque la comprobación esperaba que haya una excepción y no hubo ninguna al no haber sido codificada aún. Como siguiente paso es necesario agregar el mínimo código posible para que la prueba pase. En este caso se agrega una validación, si la *API key* está vacía se lanza la excepción *ValueError*. Luego se confirma que la prueba ahora pasa:

clima.py

```
class Clima:
    def __init__(self, apikey: str):
        if apikey == "":
            raise ValueError()

        self.apikey = apikey
```

```
(unit-testing) usuario@desktop:~/src/python/clima$ python3 clima_test.py
..
----------------------------------------------------------------------
Ran 2 tests in 0.000s

OK
```

Existen dos problemas extras que se pueden notar con respecto a la *API key*: no puede tener espacios al principio o al final, por lo que debería fallar si se intenta enviar " " por ejemplo como argumento, y que es posible modificar el valor de la *API key* utilizando la propiedad expuesta. Siguiendo los pasos anteriores:

1. Se crea la prueba unitaria que valide que ningún valor consistente en espacios únicamente pueda ser aceptado
2. Se implementa el código para hacerlo
3. Se refactorea la clase *Clima* para transformar la propiedad en sólo lectura

clima_test.py

```
    test_constructor_con_una_apikey_en_blanco_deberia_lanzar_valueerror(self):
                with self.assertRaises(ValueError) as ve:
                    sut = clima.Clima("    ")
```

```
(unit-testing) usuario@desktop:~/src/python/clima$ python3 clima_test.py
F..
======================================================================
FAIL: test_constructor_con_una_apikey_en_blanco_deberia_lanzar_valueerror
(__main__.ClimaUnitTests)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "clima_test.py", line 17, in
test_constructor_con_una_apikey_en_blanco_deberia_lanzar_valueerror
    sut = clima.Clima("    ")
AssertionError: ValueError not raised

----------------------------------------------------------------------
Ran 3 tests in 0.001s

FAILED (failures=1)
```

clima.py

```
class Clima:
    def __init__(self, apikey: str):
        self.__apikey = apikey.strip()

        if self.__apikey == "":
            raise ValueError()

    @property
    def apikey(self) -> str:
        return self.__apikey
```

```
(unit-testing) usuario@desktop:~/src/python/clima$ python3 clima_test.py
...
----------------------------------------------------------------------
Ran 3 tests in 0.000s

OK
```

Pero se ha introducido un posible problema: al pasar *None* como argumento habrá una excepción ya que se intentará ejecutar *strip* en un objeto nulo, por lo que se debe agregar una nueva prueba que verifique que *None* no sea aceptado.

clima_test.py

```
    def test_constructor_con_none_deberia_lanzar_valueerror(self):
        with self.assertRaises(ValueError) as ve:
```

```
(unit-testing) usuario@desktop:~/src/python/clima$ python3 clima_test.py
```

```
E...
======================================================================
ERROR: test_constructor_con_none_deberia_lanzar_valueerror
(__main__.ClimaUnitTests)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "clima_test.py", line 21, in
test_constructor_con_none_deberia_lanzar_valueerror
    sut = clima.Clima(None)
  File "/home/usuario/src/python/clima/clima.py", line 3, in __init__
    self.__apikey = apikey.strip()
AttributeError: 'NoneType' object has no attribute 'strip'

----------------------------------------------------------------------
Ran 4 tests in 0.000s

FAILED (errors=1)
```

La implementación más sencilla del constructor para que la prueba unitaria pase podría ser la siguiente:

clima.py

```
    def __init__(self, apikey: str):
        if apikey == None:
            raise ValueError()

        self.__apikey = apikey.strip()

        if self.__apikey == "":
```

```
(unit-testing) usuario@desktop:~/src/python/clima$ python3 clima_test.py
....
----------------------------------------------------------------------
Ran 4 tests in 0.000s

OK
```

Ahora bien, el código del constructor puede ser pulido un poco más por lo que se aprovecha la etapa de refactorización para hacerlo.

clima.py

```
class Clima:
    def __init__(self, apikey: str):
        self.__apikey = apikey.strip() if apikey is not None else ""
        if self.__apikey == "":
            raise ValueError()

    @property
    def apikey(self) -> str:
        return self.__apikey
```

```
(unit-testing) usuario@desktop:~/src/python/clima$ python3 clima_test.py
....
----------------------------------------------------------------------
Ran 4 tests in 0.000s

OK
```

Lo importante es priorizar la solución más simple posible que cubra exactamente la prueba unitaria ya que cada una debe enfocarse en un único aspecto del programa. No se debe codificar de más ni intentar avanzar más de lo que la prueba realmente requiera.

Algo más que se puede apreciar es que, a medida que se avanza en el desarrollo del software, se van acumulando las pruebas que deben validar. Esto permite conocer inmediatamente cuando una nueva funcionalidad ha hecho que otra ya existente deje de funcionar. Aunque no de manera tan estricta como en *BDD* (*behaviour-driven development*, desarrollo guiado por comportamiento, donde todas las pruebas son representaciones de especificaciones), las pruebas en *TDD* son guías de cómo debería funcionar el software, por consiguiente mientras más pormenorizadas mejor.

Para empezar la comunicación entre la aplicación y el servidor es necesario el nombre de la ciudad. Para ello se agrega un nuevo argumento a la clase *Clima* y se la guarda dentro de una propiedad que será de lectura únicamente (similar a la propiedad *API key*):

clima_test.py

```
    def test_constructor_con_ciudad_valida_deberia_incorporarla(self):
        ciudad = "London"
        apikey = "abcdefg"

        sut = clima.Clima(apikey, ciudad)
```

```
(unit-testing) usuario@desktop:~/src/python/clima$ python3 clima_test.py
E....
======================================================================
ERROR: test_constructor_con_ciudad_valida_deberia_incorporarla
(__main__.ClimaUnitTests)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "clima_test.py", line 27, in
test_constructor_con_ciudad_valida_deberia_incorporarla
    sut = clima.Clima(apikey, ciudad)
TypeError: __init__() takes 2 positional arguments but 3 were given

----------------------------------------------------------------------
Ran 5 tests in 0.000s
```

```
FAILED (errors=1)
```

Sin embargo, al implementar la solución más simple se apreciará que, aunque la última prueba funcionará correctamente, las demás dejarán de funcionar.

clima.py

```
class Clima:
    def __init__(self, apikey: str, ciudad: str):
        self.__apikey = apikey.strip() if apikey is not None else ""
        if self.__apikey == "":
            raise ValueError()

        self.__ciudad = ciudad

    @property
    def apikey(self) -> str:
        return self.__apikey

    @property
    def ciudad(self) -> str:
        return self.__ciudad
```

```
(unit-testing) usuario@desktop:~/src/python/clima$ python3 clima_test.py
.EEEE
======================================================================
ERROR: test_constructor_con_none_deberia_lanzar_valueerror
(__main__.ClimaUnitTests)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "clima_test.py", line 21, in
test_constructor_con_none_deberia_lanzar_valueerror
    sut = clima.Clima(None)
TypeError: __init__() missing 1 required positional argument: 'ciudad'

======================================================================
ERROR: test_constructor_con_una_apikey_en_blanco_deberia_lanzar_valueerror
(__main__.ClimaUnitTests)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "clima_test.py", line 17, in
test_constructor_con_una_apikey_en_blanco_deberia_lanzar_valueerror
    sut = clima.Clima("    ")
TypeError: __init__() missing 1 required positional argument: 'ciudad'

======================================================================
ERROR: test_constructor_con_una_apikey_vacia_deberia_lanzar_valueerror
(__main__.ClimaUnitTests)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "clima_test.py", line 13, in
test_constructor_con_una_apikey_vacia_deberia_lanzar_valueerror
    sut = clima.Clima("")
TypeError: __init__() missing 1 required positional argument: 'ciudad'

======================================================================
ERROR: test_constructor_con_una_apikey_valida_deberia_incorporarla
```

```
(__main__.ClimaUnitTests)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "clima_test.py", line 8, in
test_constructor_con_una_apikey_valida_deberia_incorporarla
    sut = clima.Clima(apikey)
TypeError: __init__() missing 1 required positional argument: 'ciudad'

----------------------------------------------------------------------
Ran 5 tests in 0.000s

FAILED (errors=4)
```

Al agregar un nuevo argumento obligatorio al constructor (en este caso, la ciudad), las cuatro pruebas unitarias anteriores que únicamente especificaban la *API key* automáticamente dejan de funcionar hasta que sean actualizadas. Esta situación se volverá a repetir ya que se necesitan agregar más parámetros (por ejemplo, el lenguaje de la respuesta).

Patrón del constructor o *builder pattern*

Para solucionar este problema se implementará un patrón de diseño específico, el *builder* o Constructor (no confundir con el método *constructor* de una clase). Este patrón requiere la creación de una clase *ClimaBuilder* que permitirá crear un objeto *Clima* mientras oculta los parámetros obligatorios de su constructor de manera que no sea necesario modificar cada una de las pruebas unitarias al agregar nuevos parámetros al constructor de la clase.

climabuilder_test.py

```
import unittest
from clima import Clima
from climabuilder import ClimaBuilder

class ClimaBuilderUnitTests(unittest.TestCase):
    def test_con_apikey_con_una_apikey_valida_deberia_incorporarla(self):
        # Arrange
        apikey = "abcdefg"

        # Act
        builder = ClimaBuilder().con_apikey(apikey)

        # Assert
        self.assertEqual(apikey, builder.apikey)

    def test_con_ciudad_con_una_ciudad_valida_deberia_incorporarla(self):
        # Arrange
        ciudad = "London"

        # Act
```

```
        builder = ClimaBuilder().con_ciudad(ciudad)

        # Assert
        self.assertEqual(ciudad, builder.ciudad)

    def
test_constructor_con_valor_por_defecto_deberia_ser_BsAs_y_apikey_nulo(self):
        # Arrange
        ciudad = "Buenos%20Aires"

        # Act
        builder = ClimaBuilder()

        # Assert
        self.assertIsNone(builder.apikey)
        self.assertEqual(ciudad, builder.ciudad)

if __name__ == "__main__":
    unittest.main()



(unit-testing) usuario@desktop:~/src/python/clima$ python3 climabuilder_test.py
Traceback (most recent call last):
  File "climabuilder_test.py", line 3, in <module>
    from climabuilder import ClimaBuilder
ModuleNotFoundError: No module named 'climabuilder'
```

Nótese que estrictamente hablando se debería avanzar de a un método por vez, sin embargo es posible dar saltos siempre y cuando se tenga la experiencia suficiente como para anticipar posibles errores. En este caso, por una cuestión de brevedad, se muestra la totalidad de las pruebas de una vez.

A partir de este momento se debería crear primero una instancia de *ClimaBuilder*, definir la *API key* utilizando el método *con_apikey*, y luego indicar la ciudad con el método *con_ciudad*. Esos métodos son básicamente *setters*, guardando el valor en un atributo y retornando una referencia a sí mismo. Esto permite encadenar las llamadas de configuración una tras otra en una única expresión en lugar de tener una sentencia por cada *setter*. Finalmente una llamada al método *build* pasará todos los parámetros acumulados al constructor de *Clima*. Este patrón de diseño es ampliamente usado para "humanizar", para incorporar lenguaje natural al programar (además del patrón *builder*, es común ver la "humanización" con las aseveraciones tipo *assert* durante las pruebas unitarias, como por ejemplo en Fluent Check en Python o Fluent Assertions en .NET).

La prueba unitaria sólo verifica que el *builder* conserve los valores y los envíe al constructor de la clase que está creando correctamente, nada más. Por ejemplo, si en esta implementación se decidiese verificar que la *API key* sea válida (en otras palabras, que no esté vacía ni que sea nula) se utilizaría exactamente el mismo código que ya existe en el constructor de la clase *Clima*.

A continuación, la creación del *ClimaBuilder*.

climabuilder.py

```
from clima import Clima

class ClimaBuilder:
    def __init__(self):
        self.__apikey = None
        self.__ciudad = "Buenos%20Aires"

    def con_apikey(self, apikey: str):
        self.__apikey = apikey
        return self

    @property
    def apikey(self) -> str:
        return self.__apikey

    def con_ciudad(self, ciudad: str):
        self.__ciudad = ciudad
        return self

    @property
    def ciudad(self) -> str:
        return self.__ciudad

    def build(self):
        return Clima(self.__apikey, self.__ciudad)
```

```
(unit-testing) usuario@desktop:~/src/python/clima$ python3 climabuilder_test.py
...
----------------------------------------------------------------------
Ran 3 tests in 0.000s

OK
```

Cabe destacar que el constructor de *ClimaBuilder* define una ciudad por defecto, "Buenos Aires". *%20* es la codificación hexadecimal de un espacio y es la forma en la que se envía a través de una *url* o dirección de internet. Por otro lado no existe una *API key* por defecto, por consiguiente será obligatoria indicarla cada vez que se quiera construir una instancia de *Clima*.

Inyección de dependencias

La inyección de dependencias es la consecuencia de uno de los cinco principios fundamentales de la programación orientada a objetos moderna conocidos como S.O.L.I.D. La "D" del acrónimo, que representa al principio de inversión de la dependencia indica que se debe depender de abstracciones y no de

implementaciones, por ejemplo, si un objeto necesita crear una instancia de otro para poder realizar su procesamiento el patrón sugiere que se acepte la instanciación concreta de dicho objeto desde el exterior, mientras que internamente se la referencie con una interface. De esta manera será posible modificar el funcionamiento de la clase sin tener que modificarla (insidentalmente la "O" del acrónimo, principio de abierto/cerrado, se refiere a que las clases deben estar cerradas a modificaciones pero abiertas a extensiones).

Para poder verificar que el método *build* envía los parámetros correctos en el orden indicado al constructor se debe modificar *ClimaBuilder* para permitir la "inyección". Idealmente se debería recibir un factory que construya el objeto que se va a necesitar, sin embargo en este caso en específico alcanza con crear un método privado que reciba los mismos parámetros que el constructor de la clase *Clima* para crear instancia:

climabuilder.py

```
    def _crear_clima(self, apikey: str, ciudad: str) -> Clima:
        return Clima(apikey, ciudad)

    def build(self):
```

```
(unit-testing) usuario@desktop:~/src/python/clima$ python3 climabuilder_test.py
...
----------------------------------------------------------------------
Ran 3 tests in 0.000s

OK
```

Una vez verificado que las pruebas unitarias siguen pasando es posible agregar una nueva prueba que utilice un *mock* u objeto simulado para verificar que los datos enviados sean los correctos. Para ello se importa *patch* del módulo *unittest.mock* en la parte superior de *climabuilder_test.py*...

climabuilder_test.py

...para luego agregar la nueva prueba:

climabuilder_test.py

```
    def test_build_con_valores_deberia_crearlo_con_ellos(self):
        # Arrange
        apikey = "123abc"
```

```
        with patch.object(ClimaBuilder, '_crear_clima', return_value=None) as
mock_method:
            clima = ClimaBuilder().con_apikey(apikey).con_ciudad(ciudad).build()

            self.assertIsNone(clima)
            mock_method.assert_called_once_with(apikey, ciudad)



(unit-testing) usuario@desktop:~/src/python/clima$ python3 climabuilder_test.py
....
----------------------------------------------------------------------
Ran 4 tests in 0.000s

OK
```

Con with se crea un contexto en donde cualquier instancia nueva de *ClimaBuilder* va a tener su método *_crear_clima* reemplazado por uno falso o simulado que retornará *None*. Dentro de ese contexto se crea la instancia de *Clima* y se realizan dos verificaciones: que el valor retornado sea *None* (aunque no es necesario) y se verifica que el método simulado fue invocado una única vez con los valores de la *API key* y "Sydney", en ese orden.

Ya con el *builder* creado y probado es posible refactorear las pruebas unitarias en *clima_test.py* para utilizarlo y evitar que vuelvan a fallar al agregar nuevos parámetros al constructor de *Clima*.

clima_test.py

```
import unittest
from climabuilder import ClimaBuilder
from clima import Clima

class ClimaUnitTests(unittest.TestCase):
    def test_constructor_con_una_apikey_valida_deberia_incorporarla(self):
        apikey = "abcdefg"

        sut = ClimaBuilder().con_apikey(apikey).build()

        self.assertEqual(apikey, sut.apikey)

    def test_constructor_con_una_apikey_vacia_deberia_lanzar_valueerror(self):
        with self.assertRaises(ValueError) as ve:
            sut = ClimaBuilder().con_apikey("").build()

    def
test_constructor_con_una_apikey_en_blanco_deberia_lanzar_valueerror(self):
        with self.assertRaises(ValueError) as ve:
            sut = ClimaBuilder().con_apikey("    ").build()

    def test_constructor_con_none_deberia_lanzar_valueerror(self):
        with self.assertRaises(ValueError) as ve:
            sut = ClimaBuilder().con_apikey(None).build()

    def test_constructor_con_ciudad_valida_deberia_incorporarla(self):
```

```
        ciudad = "London"
        apikey = "abcdefg"

        sut = ClimaBuilder().con_apikey(apikey).con_ciudad(ciudad).build()
        self.assertEqual(ciudad, sut.ciudad)

if __name__ == "__main__":
    unittest.main()
```

```
(unit-testing) usuario@desktop:~/src/python/clima$ python3 clima_test.py
.....
----------------------------------------------------------------------
Ran 5 tests in 0.000s

OK
```

Cabe destacar que si no existiesen las pruebas unitarias de *ClimaBuilder* se podría considerar que las pruebas unitarias en *clima_test.py* no son del todo correctas ya que estarían probando tanto que se pueda crear un objeto de *ClimaBuilder*, que los métodos *con_apikey* y *con_ciudad* estén guardando los datos dentro de los atributos correctos y que el método *build* esté pasando la información de forma correcta a la instancia de la clase *Clima*. Esto se asemeja más a una *prueba de integración*, la cual será explicada en una sección posterior.

Detroit vs Londres

En la última prueba se verificó que el resultado fuese nulo y que se haya llamado a *_crear_clima* con determinados valores, exponiendo las dos escuelas que existen en *TDD*: la Detroit (o clásica) y la escuela Londres (o *mockist*, literalmente "burlones" pero en este contexto, "simuladores"). La escuela clásica indica que únicamente se deben verificar los resultados: si el resultado es correcto se asume que el funcionamiento también. La escuela londinense indica que es necesario verificar que la mensajería entre los objetos sea la correcta, no tanto el resultado: de nada sirve llegar al resultado correcto si se llaman funciones que no son necesarias o si se las utiliza en el orden incorrecto. A eso que tiene sentido la escuela clásica contrapone que verificar el funcionamiento interno de un objeto fomenta el acoplamiento, ya que si se modificase la función que se utiliza internamente por reorganización de código sería necesario modificar las pruebas cuando con su planteo no deberían hacerlo.

Existe cuatro tipos de objetos "dobles" (como en "doble de película de acción") que ayudan en la ejecución de pruebas. De esos cuatro hay dos, los bobos (o *dummies*) que no ejecutan lógica ni

devuelven nada y los *stubs* que no ejecutan lógica pero devuelven valores predeterminados son los preferidos del estilo Detroit. Implican que se deben escribir clases extras que implementan las interfaces y mediante inyección de dependencias se las proveen a los objetos siendo testeados. Los otros dos tipos de dobles verifican interacciones y son los preferidos de la escuela londinense: los espías y los *mocks*. Los espías son *stubs* que graban determinada información, por ejemplo, los argumentos que fueron dados a los métodos, la cantidad de veces que cada método se llamó y el orden en que se llamaron, por ejemplo. Finalmente los *mocks* son espías que pueden verificar en tiempo real las llamadas y lanzar una excepción si se llamó con diferentes argumentos, o si se llama más de una vez.

Muchos *frameworks* de pruebas unitarias llaman a cualquiera de esos cuatro tipos *mocks* pero permite configurar si se desea que no haga nada (como un *dummy*), que retorne un valor predeterminado (como un *stub*), que recuerde datos sobre la interacción (como los espías) o si se desea que lance una excepción tan pronto como algo que no estaba planeado pase (como un *mock* literal).

Como siempre es preferible permanecer agnóstico: mientras que la mayoría de las pruebas pueden seguir el estilo Detroit, existen casos donde el estilo londinense es preferible.

Test dirigido por datos

En las pruebas unitarias de *ClimaBuilder*, hasta ahora existía algo así:

climabuilder_test.py

```
def test_con_ciudad_con_una_ciudad_valida_deberia_incorporarla(self):
    # Arrange
    ciudad = "London"

    # Act
    builder = ClimaBuilder().con_ciudad(ciudad)

    # Assert
```

Sin embargo, el nombre del método no es del todo correcto: Como los métodos de *ClimaBuilder* son simples *setters*, deberían aceptar cualquier valor, sea éste válido o inválido. Una opción sería crear una nueva prueba por cada valor inválido, por ejemplo:

climabuilder_test.py

```
    def test_con_ciudad_con_una_ciudad_nula_deberia_incorporarla(self):
        # Arrange
        ciudad = None

        # Act
        builder = ClimaBuilder().con_ciudad(ciudad)

        # Assert
        self.assertEqual(ciudad, builder.ciudad)

    def test_con_ciudad_con_una_ciudad_vacia_deberia_incorporarla(self):
        # Arrange
        ciudad = ""

        # Act
        builder = ClimaBuilder().con_ciudad(ciudad)

        # Assert
```

Pero además deberíamos armar una prueba para " " (que también es vacío), o " ", o tal vez números en lugar de un nombre, etc. Es posible llegar a considerar recorrer una lista con los valores inválidos, realizando una verificación por cada iteración:

climabuilder_test.py

```
    def test_con_ciudad_con_una_ciudad_invalida_deberia_incorporarla(self):
        # Arrange
        ciudades = [ None, "", " ", "    ", "\t" ]

        # Act
        for ciudad in ciudades:
            builder = ClimaBuilder().con_ciudad(ciudad)

            # Assert
```

pero se estarían realizando varios actos y verificaciones en una única prueba cuando debería ser un único acto, además de que si cada prueba necesitara una inicialización (usualmente un código que debe ejecutarse antes y/o después de cada prueba) dicho código sólo se ejecutaría una vez por prueba. Para evitar estos problemas es posible utilizar la librería ddt (*Data-driven Tests*, o pruebas guiadas por datos) para "alimentar" con datos a cada prueba. Primero es necesario instalar el módulo con `pip3 install ddt`, luego importarlo y utilizar el decorador *@ddt* en la clase de prueba:

climabuilder_test.py

```
import unittest
```

```
from unittest.mock import patch
from clima import Clima
from climabuilder import ClimaBuilder

@ddt
class ClimaBuilderUnitTests(unittest.TestCase):
```

y el decorador *@data* en el método junto con los distintos valores que alimentarán al metodo de prueba. Finalmente, para que este último los incorpore es necesario agregar un parámetro nuevo para recibirlo en el método de prueba:

climabuilder_test.py

```
    @data(None, "", "    ", "abcdefg")
    def test_con_ciudad_con_cualquier_ciudad_deberia_incorporarla(self, ciudad:
str):
        # Act
        builder = ClimaBuilder().con_ciudad(ciudad)

        # Assert
```

```
(unit-testing) usuario@desktop:~/src/python/clima$ python3 clima_test.py
........
----------------------------------------------------------------------
Ran 8 tests in 0.000s

OK
```

Sabiendo como utilizar *@ddt* es posible escribir las pruebas unitarias para cada uno de los parámetros que se pueden enviar a la *API.*

En la página de *OpenWeather* se indica que además de la ciudad es posible enviar como argumentos el país, la unidad de temperatura, o en lugar del nombre de la ciudad es posible pasar el id de la misma, o sus coordenadas.

En este momento es posible refactorear los métodos de prueba existentes para que utilicen *@data* y, al mismo tiempo, eliminar *test_con_ciudad_con_una_ciudad_valida_deberia_incorporarla* ya que su caso ahora será probado por *test_con_ciudad_con_cualquier_ciudad_deberia_incorporarla*.
Una de las ventajas de tener nombres descriptivos es saber si la funcionalidad de un método esta repetida en otro con sólo leerlo.

climabuilder_test.py

```
import unittest
from unittest.mock import patch
from ddt import ddt, data, unpack
```

```
from clima import Clima
from climabuilder import ClimaBuilder

@ddt
class ClimaBuilderUnitTests(unittest.TestCase):

    @data(None, "", "  ", "abcdefg")
    def test_con_apikey_con_cualquier_apikey_deberia_incorporarla(self,
cualquier_apikey: str):
        # Arrange / Act
        builder = ClimaBuilder().con_apikey(cualquier_apikey)

        # Assert
        self.assertEqual(cualquier_apikey, builder.apikey)

    @data(None, "", "  ", "abcdefg")
    def test_con_ciudad_con_cualquier_ciudad_deberia_incorporarla(self,
cualquier_ciudad: str):
        # Arrange / Act
        builder = ClimaBuilder().con_ciudad(cualquier_ciudad)

        # Assert
        self.assertEqual(cualquier_ciudad, builder.ciudad)

    @data(None, "", "  ", "abcdefg")
    def test_en_pais_con_cualquier_pais_deberia_incorporarlo(self,
cualquier_pais: str):
        # Arrange / Act
        builder = ClimaBuilder().en_pais(cualquier_pais)

        # Assert
        self.assertEqual(cualquier_pais, builder.pais)

    @data(None, "", "  ", "abcdefg")
    def
test_en_codigo_postal_con_cualquier_codigo_postal_deberia_aceptarlo(self,
cualquier_codigo_postal: str):
        # Arrange / Act
        builder = ClimaBuilder().en_codigo_postal(cualquier_codigo_postal)

        # Assert
        self.assertEqual(cualquier_codigo_postal, builder.codigo_postal)

    @data((0, 0), (-1, 10), (13, -1), (-8, -6))
    @unpack
    def test_en_coordenadas_con_cualquier_coordenada_deberia_incorporarla(self,
cualquier_latitud: float, cualquier_longitud: float):
        # Arrange
        cualquier_coordenada = (cualquier_latitud, cualquier_longitud)
        builder = ClimaBuilder()

        # Act
        builder = builder.en_coordenadas(cualquier_latitud, cualquier_longitud)

        # Assert
        self.assertEqual(cualquier_coordenada, builder.coordenadas)

    def test_en_fahrenheit_con_fahrenheit_flag_deberia_incorporarlo(self):
        # Arrange
        builder = ClimaBuilder()

        # Act
        builder = builder.en_fahrenheit()
```

```
            # Assert
            self.assertEqual("fahrenheit", builder.grados)

        def test_en_celsius_con_celsius_flag_deberia_incorporarlo(self):
            # Arrange
            builder = ClimaBuilder()

            # Act
            builder.en_celsius()

            # Assert
            self.assertEqual("celsius", builder.grados)

        def test_en_kelvin_con_kelvin_flag_deberia_incorporarlo(self):
            # Arrange
            builder = ClimaBuilder()

            # Act
            builder.en_kelvin()

            # Assert
            self.assertEqual("kelvin", builder.grados)

        @data(-33, 0, 1, 10, 1000)
        def test_con_id_con_cualquier_id_deberia_incorporarlo(self, cualquier_id:
    int):
            # Arrange
            builder = ClimaBuilder()

            # Act
            builder.con_id(cualquier_id)

            # Assert
            self.assertEqual(cualquier_id, builder.id)

        @data("", "  ", "es", "english")
        def test_en_idioma_con_cualquier_idioma_deberia_incorporarlo(self,
    cualquier_idioma: str):
            # Arrange
            builder = ClimaBuilder()

            # Act
            builder.en_idioma(cualquier_idioma)

            # Assert
            self.assertEqual(cualquier_idioma, builder.idioma)
```

Hay un último método ya existente que debe ser modificado. El método que verifica los valores por defecto debe verificar también que los grados estén en *kelvin*, que el país sea Argentina y que el idioma sea inglés.

climabuilder_test.py

```
        def
    test_constructor_con_creacion_deberia_tener_idioma_ciudad_grados_pais(self):
            # Arrange
            idioma = "en"
```

```
        pais = "ar"
        grados = "kelvin"

        # Act
        builder = ClimaBuilder()

        # Assert
        self.assertEqual(pais, builder.pais)
        self.assertEqual(ciudad, builder.ciudad)
        self.assertEqual(idioma, builder.idioma)
        self.assertEqual(grados, builder.grados)
```

Si se intentase correr las pruebas en estos momentos habrían 25 errores sobre 33 pruebas. Con los casos de prueba listos para *ClimaBuilder* es posible implementarlo en *climabuilder.py*.

climabuilder.py

```
from clima import Clima

class ClimaBuilder:
    def __init__(self):
        self.__apikey = None
        self.__id = None
        self.__coordenadas = None
        self.__codigo_postal = None
        self.__grados = "kelvin"
        self.__pais = "ar"
        self.__ciudad = "Buenos%20Aires"
        self.__idioma = "en"

    def con_apikey(self, apikey: str):
        self.__apikey = apikey
        return self

    @property
    def apikey(self) -> str:
        return self.__apikey

    def con_ciudad(self, ciudad: str):
        self.__ciudad = ciudad
        return self

    @property
    def ciudad(self) -> str:
        return self.__ciudad

    def en_pais(self, pais: str):
        self.__pais = pais
        return self

    @property
    def pais(self) -> str:
        return self.__pais

    def en_codigo_postal(self, codigo: str):
        self.__codigo_postal = codigo
        return self

    @property
```

```
    def codigo_postal(self) -> str:
        return self.__codigo_postal

    def en_fahrenheit(self):
        self.__grados = "fahrenheit"
        return self

    def en_celsius(self):
        self.__grados = "celsius"
        return self

    def en_kelvin(self):
        self.__grados = "kelvin"
        return self

    @property
    def grados(self) -> str:
        return self.__grados

    def en_idioma(self, idioma: str):
        self.__idioma = idioma
        return self

    @property
    def idioma(self) -> str:
        return self.__idioma

    def con_id(self, id: int):
        self.__id = id
        return self

    @property
    def id(self) -> int:
        return self.__id

    def en_coordenadas(self, latitud: float, longitud: float):
        self.__coordenadas = (latitud, longitud)
        return self

    @property
    def coordenadas(self) -> (float, float):
        return self.__coordenadas

    def _crear_clima(self, apikey: str, ciudad: str) -> Clima:
        return Clima(apikey, ciudad)

    def build(self):
        return self._crear_clima(self.__apikey, self.__ciudad)


(unit-testing) usuario@desktop:~/src/python/clima$ python3 clima_test.py
.............................
----------------------------------------------------------------------
Ran 33 tests in 0.000s

OK
```

Además la verificación debe ser actualizada para verificar que se creó la clase con todos los argumentos y no sólo con la *API key* y la ciudad como era originalmente. Para ello hay que agregar el siguiente método:

climabuilder_test.py

```
    @data(("123abc", "Sydney", "au", 1334, "SY1233", (-1, 9), "australian
english", "fahrenheit"))
    @unpack
    def test_build_con_valores_deberia_crearlo_con_ellos(self, apikey: str,
ciudad: str, pais: str, id: str, cp: str, coordenadas: (float, float), idioma:
str, grados: str):
        # Arrange
        latitud, longitud = coordenadas

        with patch.object(ClimaBuilder, "_crear_clima", return_value=None) as
mock_method:
            builder = (
                ClimaBuilder().con_apikey(apikey)
                              .con_ciudad(ciudad)
                              .en_pais(pais)
                              .con_id(id)
                              .en_codigo_postal(cp)
                              .en_coordenadas(latitud, longitud)
                              .en_idioma(idioma)
            )

            if grados == "fahrenheit":
                builder.en_fahrenheit()
            elif grados == "celsius":
                builder.en_celsius()
            else:
                builder.en_kelvin()

            # Act
            clima = builder.build()

            # Assert
            self.assertIsNone(clima)
            mock_method.assert_called_once_with(apikey, id, ciudad, pais, cp,
```

En Python se puede usar paréntesis en la asignación de *builder* para poder alinear los métodos de *ClimaBuilder* uno debajo del otro. También se podría escribir todo en una línea pero normalmente es inconveniente tener que mover el cursor más allá del final de la pantalla para ver todos los métodos que se están llamando.

```
(unit-testing) usuario@desktop:~/src/python/clima$ python3 climabuilder_test.py
.F.............................
======================================================================
FAIL:
test_build_con_valores_deberia_crearlo_con_ellos_1___123abc____Sydney____au___13
34___SY1233_____1__9____australian_english____fahrenheit__
(__main__.ClimaBuilderUnitTests)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "/home/usuario/src/python/unit-testing/lib/python3.7/site-packages
/ddt.py", line 145, in wrapper
    return func(self, *args, **kwargs)
  File "climabuilder_test.py", line 152, in
test_build_con_valores_deberia_crearlo_con_ellos
```

```
    mock_method.assert_called_once_with(apikey, id, ciudad, pais, cp, (latitud,
longitud), idioma, grados)
  File "/usr/lib/python3.7/unittest/mock.py", line 884, in
assert_called_once_with
    return self.assert_called_with(*args, **kwargs)
  File "/usr/lib/python3.7/unittest/mock.py", line 873, in assert_called_with
    raise AssertionError(_error_message()) from cause
AssertionError: Expected call: _crear_clima('123abc', 1334, 'Sydney', 'au',
'SY1233', (-1, 9), 'australian english', 'fahrenheit')
Actual call: _crear_clima('123abc', 'Sydney')

----------------------------------------------------------------------
Ran 34 tests in 0.001s

FAILED (failures=1)
```

El error indica que se esperaba que la instancia de *Clima* se creara con nueve parámetros pero que solamente se crea con dos, la *API key* y la ciudad. Para solucionarlo habría que modificar la instancia del constructor de *Clima* para que reciba todos esos argumentos, sin embargo algunos de ellos son incompatibles: la ciudad se puede indicar por id, coordenadas, nombre de ciudad (más país), o código postal (más país).

Existen dos formas simples de solucionar este problema: la primera es hacerlos mutuamente exclusivos (si se indica el nombre de una ciudad se deberían borrar las coordenadas, el id y el código postal, por ejemplo), la segunda es permitir que el usuario ingrese cualquier dato (incluyendo los mutuamente exclusivos) y que luego la clase *Clima* determine el orden de prioridades, por ejemplo buscar primero la ciudad, después el código postal, etc.

El primer paso es modificar en *ClimaBuilder* tanto *_crear_clima* como *build* para que pasen al constructor de *Clima* todos los argumentos en el orden correcto.

climabuilder.py

```
    def _crear_clima(self, apikey: str, id: int, ciudad: str, pais: str, cp:
str, coordenadas: (float, float), idioma: str, grados: str) -> Clima:
        return Clima(apikey, id, ciudad, pais, cp, coordenadas, idioma, grados)

    def build(self):
        return self._crear_clima(self.__apikey, self.__id, self.__ciudad,
self.__pais, self.__codigo_postal, self.__coordenadas, self.__idioma,
```

```
(unit-testing) usuario@desktop:~/src/python/clima$ python3 climabuilder_test.py
.................................
----------------------------------------------------------------------
Ran 34 tests in 0.001s

OK
```

Queda por modificar el constructor de la clase *Clima* para que reciba todos los argumentos correctamente. Como se van a agregar varios atributos a dicha clase lo que corresponde es primero agregar en *clima_test.py* las pruebas unitarias para verificar que al construir un objeto con un valor determinado se guarde correctamente el valor. Utilizando *ddt*:

clima_test.py

```
import unittest
from ddt import ddt, data
from climabuilder import ClimaBuilder
from clima import Clima

@ddt
class ClimaUnitTests(unittest.TestCase):
    def test_constructor_con_una_apikey_valida_deberia_incorporarla(self):
        apikey = "abcdefg"

        sut = ClimaBuilder().con_apikey(apikey).build()

        self.assertEqual(apikey, sut.apikey)

    def test_constructor_con_una_apikey_vacia_deberia_lanzar_valueerror(self):
        with self.assertRaises(ValueError) as ve:
            sut = ClimaBuilder().con_apikey("").build()

    def
test_constructor_con_una_apikey_en_blanco_deberia_lanzar_valueerror(self):
        with self.assertRaises(ValueError) as ve:
            sut = ClimaBuilder().con_apikey("    ").build()

    def test_constructor_con_none_deberia_lanzar_valueerror(self):
        with self.assertRaises(ValueError) as ve:
            sut = ClimaBuilder().con_apikey(None).build()

    def test_constructor_con_ciudad_valida_deberia_incorporarla(self):
        ciudad = "London"
        apikey = "abcdefg"

        sut = ClimaBuilder().con_apikey(apikey).con_ciudad(ciudad).build()
        self.assertEqual(ciudad, sut.ciudad)

    @data(None, -11, 0, 10, 2172797)
    def test_constructor_con_cualquier_id_deberia_incorporarlo(self,
cualquier_id: int):
        # Arrange
        cualquier_id = id

        # Act
        sut = ClimaBuilder().con_apikey("abcdefg").con_id(cualquier_id).build()

        # Assert
        self.assertEqual(cualquier_id, sut.id)

    @data(None, "", "ar", "uy")
    def test_constructor_con_cualquier_pais_deberia_incorporarlo(self,
cualquier_pais: str):
        # Act
        sut =
ClimaBuilder().con_apikey("abcdefg").en_pais(cualquier_pais).build()
```

```
            # Assert
            self.assertEqual(cualquier_pais, sut.pais)

        @data(None, "", "13233")
        def test_constructor_con_cualquier_codigo_postal_deberia_incorporarlo(self,
    cualquier_cp: str):
            # Act
            sut =
    ClimaBuilder().con_apikey("abcdefg").en_codigo_postal(cualquier_cp).build()

            # Assert
            self.assertEqual(cualquier_cp, sut.codigo_postal)

        def test_constructor_con_cualquier_coordenadas_deberia_incorporarlo(self):
            # Arrange
            latitud, longitud = (34.6037, 58.3816)

            # Act
            sut = ClimaBuilder().con_apikey("abcdefg").en_coordenadas(latitud,
    longitud).build()

            # Assert
            self.assertEqual((latitud, longitud), sut.coordenadas)

        @data(None, "", "sp")
        def test_constructor_con_cualquier_idioma_deberia_incorporarlo(self,
    cualquier_idioma: str):
            # Act
            sut =
    ClimaBuilder().con_apikey("abcdefg").en_idioma(cualquier_idioma).build()

            # Assert
            self.assertEqual(cualquier_idioma, sut.idioma)

        def test_constructor_con_fahrenheit_deberia_incorporarlo(self):
            # Act
            sut = ClimaBuilder().con_apikey("abcdefg").en_fahrenheit().build()

            # Assert
            self.assertEqual("fahrenheit", sut.grados)

        def test_constructor_con_celsius_deberia_incorporarlo(self):
            # Act
            sut = ClimaBuilder().con_apikey("abcdefg").en_celsius().build()

            # Assert
            self.assertEqual("celsius", sut.grados)

        def test_constructor_con_kelvin_deberia_incorporarlo(self):
            # Act
            sut = ClimaBuilder().con_apikey("abcdefg").en_kelvin().build()

            # Assert
            self.assertEqual("kelvin", sut.grados)

    if __name__ == "__main__":
        unittest.main()
```

Y luego se modifica la clase *Clima* para aceptar todos los argumentos en el constructor, además de agregar las propiedades de lectura:

clima.py

```
class Clima:
    def __init__(self, apikey: str, id: int, ciudad: str, pais: str, cp: str,
coordenadas: (float, float), idioma: str, grados: str):
        self.__apikey = apikey.strip() if apikey is not None else ""
        if self.__apikey == "":
            raise ValueError()

        self.__id = id
        self.__ciudad = ciudad
        self.__pais = pais
        self.__codigo_postal = cp
        self.__coordenadas = coordenadas
        self.__idioma = idioma
        self.__grados = grados

    @property
    def apikey(self) -> str:
        return self.__apikey

    @property
    def ciudad(self) -> str:
        return self.__ciudad

    @property
    def id(self) -> int:
        return self.__id

    @property
    def pais(self) -> str:
        return self.__pais

    @property
    def codigo_postal(self) -> str:
        return self.__codigo_postal

    @property
    def coordenadas(self) -> (float, float):
        return self.__coordenadas

    @property
    def idioma(self) -> str:
        return self.__idioma

    @property
    def grados(self) -> str:
        return self.__grados
```

```
(unit-testing) usuario@desktop:~/src/python/clima$ python3 climabuilder_test.py
..................................
----------------------------------------------------------------------
Ran 34 tests in 0.001s

OK
(unit-testing) usuario@desktop:~/src/python/clima$ python3 clima_test.py
........................
----------------------------------------------------------------------
Ran 24 tests in 0.001s

OK
```

En este momento es recomendable modificar a *test_build_con_valores_deberia_crearlo_con_ellos* en *climabuilder_test.py* para que sea alimentado por una cantidad de datos suficiente como para verificar cada una de las combinaciones válidas posibles:

climabuilder_test.py

```
    @data(
        ( 2172797, None, None, None, None, None, "english", "fahrenheit" ),
        ( None, "Sydney", "au", None, None, None, "australian english", "kelvin"
),
        ( None, None, None, None, 34.6037, 58.3816, "sp", "celsius" ),
        ( None, None, None, "92730", None, None, "en", "fahrenheit" )
    )
    @unpack
    def test_build_con_varios_valores_deberia_crearlo_con_ellos(self, id: int,
ciudad: str, pais: str, cp: str, latitud: float, longitud: float, idioma: str,
grados: str):
        # Arrange
        apikey = "abcdefg"

        with patch.object(ClimaBuilder, "_crear_clima", return_value=None) as
mock_method:
            builder = (
                ClimaBuilder().con_apikey(apikey)
                              .con_id(id)
                              .con_ciudad(ciudad)
                              .en_pais(pais)
                              .en_codigo_postal(cp)
                              .en_coordenadas(latitud, longitud)
                              .en_idioma(idioma)
            )

            if grados == "fahrenheit":
                builder.en_fahrenheit()
            elif grados == "celsius":
                builder.en_celsius()
            else:
                builder.en_kelvin()

            # Act
            clima = builder.build()

            # Assert
            self.assertIsNone(clima)
            mock_method.assert_called_once_with(apikey, id, ciudad, pais, cp,
```

```
(unit-testing) usuario@desktop:~/src/python/clima$ python3 climabuilder_test.py
.....................................
----------------------------------------------------------------------
Ran 37 tests in 0.002s

OK
```

A simple vista las pruebas unitarias de *Clima* son similares, verifican que el estado interno de ambas clases se mantenga

correctamente después de setearlo. Es común que clases compartan atributos y por consiguiente que existan pruebas unitarias que se parezcan entre ellas.

Preparación de la API

Para acceder a la API de *OpenWeather* se utilizan mensajes *GET* que se envían a través de una *url*. Al tener todos los valores necesarios para construir dicha *url* al instanciar *Clima* es posible calcular la dirección a la que se debe conectar. Una vez calculada puede exponerse a través de una propiedad de sólo lectura como al resto de las propiedades.

Para comenzar se importa en *clima_test.py* la funcionalidad de *unpack*:

clima_test.py

y luego se agrega la nueva prueba:

clima_test.py

```
    @data(
        ( 2172797, None, None, None, (None, None), "", "fahrenheit",
"http://api.openweathermap.org/data/2.5/weather?appid=abcdefg&id=2172797&
units=imperial" ),
        ( None, "Sydney", "au", None, (None, None), "fr", "kelvin",
"http://api.openweathermap.org/data/2.5/weather?appid=abcdefg&q=Sydney,au&
lang=fr" ),
        ( None, None, None, None, (34.6037, 58.3816), "es", "celsius",
"http://api.openweathermap.org/data/2.5/weather?appid=abcdefg&lat=34.6037&
lon=58.3816&lang=es&units=metric" ),
        ( None, None, None, "94040", (None, None), "ja", "fahrenheit",
"http://api.openweathermap.org/data/2.5/weather?appid=abcdefg&zip=94040&lang=ja&
units=imperial" ),
        ( None, None, "ar", "94040", (None, None), "ja", "fahrenheit",
"http://api.openweathermap.org/data/2.5/weather?appid=abcdefg&zip=94040,ar&
lang=ja&units=imperial" )
    )
    @unpack
    def test_url_con_datos_deberia_crear_url(self, id: int, ciudad: str, pais:
str, cp: str, coordenadas: (float, float), idioma: str, grados: str, url: str):
        # Arrange
        apikey = "abcdefg"

        # Act
        clima = Clima(apikey, id, ciudad, pais, cp, coordenadas, idioma, grados)

        # Assert
```

```
(unit-testing) usuario@desktop:~/src/python/clima$ python3 clima_test.py
.....................EEEEE
======================================================================
ERROR:
test_url_con_datos_deberia_crear_url_1__2172797__None__None__None___None__None__
______fahrenheit____http___api_openweathermap_org_data_2_5_weather_appid_abcdefg
_id_2172797_units_imperial__ (__main__.ClimaUnitTests)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "/home/usuario/src/python/unit-testing/lib/python3.7/site-packages
/ddt.py", line 145, in wrapper
    return func(self, *args, **kwargs)
  File "clima_test.py", line 116, in test_url_con_datos_deberia_crear_url
    self.assertEqual(url, clima.url)
AttributeError: 'Clima' object has no attribute 'url'

======================================================================
ERROR:
test_url_con_datos_deberia_crear_url_2__None___Sydney____au___None___None__None_
___fr____kelvin____http___api_openweathermap_org_data_2_5_weather_appid_abcdefg_
q_Sydney_au_lang_fr__ (__main__.ClimaUnitTests)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "/home/usuario/src/python/unit-testing/lib/python3.7/site-packages
/ddt.py", line 145, in wrapper
    return func(self, *args, **kwargs)
  File "clima_test.py", line 116, in test_url_con_datos_deberia_crear_url
    self.assertEqual(url, clima.url)
AttributeError: 'Clima' object has no attribute 'url'

======================================================================
ERROR:
test_url_con_datos_deberia_crear_url_3__None__None__None__None___34_6037__58_381
6____es____celsius____http___api_openweathermap_org_data_2_5_weather_appid_abcde
fg_lat_34_6037_lon_58_3816_lang_es_units_metric__ (__main__.ClimaUnitTests)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "/home/usuario/src/python/unit-testing/lib/python3.7/site-packages
/ddt.py", line 145, in wrapper
    return func(self, *args, **kwargs)
  File "clima_test.py", line 116, in test_url_con_datos_deberia_crear_url
    self.assertEqual(url, clima.url)
AttributeError: 'Clima' object has no attribute 'url'

======================================================================
ERROR:
test_url_con_datos_deberia_crear_url_4__None__None__None___94040____None__None__
__ja____fahrenheit____http___api_openweathermap_org_data_2_5_weather_appid_abcde
fg_zip_94040_lang_ja_units_imperial__ (__main__.ClimaUnitTests)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "/home/usuario/src/python/unit-testing/lib/python3.7/site-packages
/ddt.py", line 145, in wrapper
    return func(self, *args, **kwargs)
  File "clima_test.py", line 116, in test_url_con_datos_deberia_crear_url
    self.assertEqual(url, clima.url)
AttributeError: 'Clima' object has no attribute 'url'

======================================================================
ERROR:
test_url_con_datos_deberia_crear_url_5__None__None___ar____94040____None__None__
__ja____fahrenheit____http___api_openweathermap_org_data_2_5_weather_appid_abcde
```

```
fg_zip_94040_ar_lang_ja_units_imperial__ (__main__.ClimaUnitTests)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "/home/usuario/src/python/unit-testing/lib/python3.7/site-packages
/ddt.py", line 145, in wrapper
    return func(self, *args, **kwargs)
  File "clima_test.py", line 116, in test_url_con_datos_deberia_crear_url
    self.assertEqual(url, clima.url)
AttributeError: 'Clima' object has no attribute 'url'

----------------------------------------------------------------------
Ran 29 tests in 0.001s

FAILED (errors=5)
```

El siguiente paso es agregar el código dentro del constructor de *Clima* para generar la *url* y crear la propiedad con la cual se podrá consultar su valor:

clima.py

```
class Clima:
    def __init__(self, apikey: str, id: int, ciudad: str, pais: str, cp: str,
coordenadas: (float, float), idioma: str, grados: str):
        self.__apikey = apikey.strip() if apikey is not None else ""
        if self.__apikey == "":
            raise ValueError()

        self.__id = id
        self.__ciudad = ciudad
        self.__pais = pais
        self.__codigo_postal = cp
        self.__coordenadas = coordenadas
        self.__idioma = idioma
        self.__grados = grados
        self.__url = self._build_url()

    def _build_url(self):
        url = f"http://api.openweathermap.org/data/2.5/weather?appid=
{self.__apikey}"
        if self.__id is not None:
            url = f"{url}&id={self.__id}"
        elif self.__ciudad is not None:
            url = f"{url}&q={self.__ciudad}"

            if self.__pais is not None:
                url = f"{url},{self.__pais}"
        elif self.__codigo_postal is not None:
            url = f"{url}&zip={self.__codigo_postal}"

            if self.__pais is not None:
                url = f"{url},{self.__pais}"
        elif self.__coordenadas is not None:
            url = f"{url}&lat={self.__coordenadas[0]}&
lon={self.__coordenadas[1]}"

        if self.__idioma is not None and self.__idioma != "" and self.__idioma
!= "en":
            url = f"{url}&lang={self.__idioma}"
```

```
        if self.__grados is not None:
            if self.__grados == "fahrenheit":
                url = f"{url}&units=imperial"
            elif self.__grados == "celsius":
                url = f"{url}&units=metric"

        return url

    @property
    def url(self) -> str:
        return self.__url
```

```
(unit-testing) usuario@desktop:~/src/python/clima$ python3 clima_test.py
.........................
----------------------------------------------------------------------
Ran 29 tests in 0.001s

OK
```

Se sobreentiende que para llegar a esta versión de *_build_url* se requirieron varios ciclos de prueba fallida e implementación de código (al menos una iteración por cada juego de datos para crear las diferentes combinaciones de *urls*). Por una cuestión de brevedad nuevamente se ha simplificado el proceso mostrando la versión final.

Existen dos problemas inmediatos: si tanto la ciudad como el código postal, las coordenadas y el id son nulos, no es posible calcular la temperatura por lo que en ese caso se debe lanzar una excepción. Por otro lado, las coordenadas no pueden tener un elemento (latitud o longitud) nulo, en cuyo caso también es necesario lanzar una excepción. Se agregan dos pruebas que cubran ambos escenarios.

clima_test.py

```
    def
test_constructor_con_ciudad_codigo_postal_coordenadas_id_nulas_deberia_lanzar_va
lueerror(self):
        # Arrange
        apikey = "abcdefg"

        # Act / Assert
        with self.assertRaises(ValueError) as ve:
            sut = Clima(apikey, None, None, None, None, None, None, None)

    def
test_constructor_con_latitud_o_longitud_nulas_y_ninguna_otra_locacion_deberia_la
nzar_valueerror(self):
        # Arrange
        apikey = "abcdefg"
        url_servicio = "http://api.openweathermap.org/data/2.5/weather"

        # Act / Assert
        with self.assertRaises(ValueError) as ve:
```

```
None)


(unit-testing) usuario@desktop:~/src/python/clima$ python3 clima_test.py
.F..................F.........
======================================================================
FAIL:
test_constructor_con_ciudad_codigo_postal_coordenadas_id_nulas_deberia_lanzar_va
lueerror (__main__.ClimaUnitTests)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "clima_test.py", line 124, in
test_constructor_con_ciudad_codigo_postal_coordenadas_id_nulas_deberia_lanzar_va
lueerror
    sut = Clima(apikey, None, None, None, None, None, None, None)
AssertionError: ValueError not raised

======================================================================
FAIL:
test_constructor_con_latitud_o_longitud_nulas_y_ninguna_otra_locacion_deberia_la
nzar_valueerror (__main__.ClimaUnitTests)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "clima_test.py", line 134, in
test_constructor_con_latitud_o_longitud_nulas_y_ninguna_otra_locacion_deberia_la
nzar_valueerror
    sut = Clima(apikey, None, None, None, None, (None, None), None, None)
AssertionError: ValueError not raised

----------------------------------------------------------------------
Ran 31 tests in 0.001s

FAILED (failures=2)
```

La implementación en la clase agrega los respectivos chequeos y excepciones:

clima.py

```
    def _build_url(self):
        url = f"http://api.openweathermap.org/data/2.5/weather?appid=
{self.__apikey}"
        if self.__id is not None:
            url = f"{url}&id={self.__id}"
        elif self.__ciudad is not None:
            url = f"{url}&q={self.__ciudad}"

            if self.__pais is not None:
                url = f"{url},{self.__pais}"
        elif self.__codigo_postal is not None:
            url = f"{url}&zip={self.__codigo_postal}"

            if self.__pais is not None:
                url = f"{url},{self.__pais}"
        elif self.__coordenadas is not None:
            if None in self.__coordenadas:
                raise ValueError()

            url = f"{url}&lat={self.__coordenadas[0]}&
```

```
        else:
            raise ValueError()

        if self.__idioma is not None and self.__idioma != "" and self.__idioma
!= "en":
            url = f"{url}&lang={self.__idioma}"

        if self.__grados is not None:
            if self.__grados == "fahrenheit":
                url = f"{url}&units=imperial"
            elif self.__grados == "celsius":
                url = f"{url}&units=metric"

        return url
```

```
(unit-testing) usuario@desktop:~/src/python/clima$ python3 clima_test.py
.............................
----------------------------------------------------------------------
Ran 31 tests in 0.001s

OK
```

Para permitir la conexión a distintas direcciones (por ejemplo, para el clima actual existen dos conexiones posibles, a un servidor de prueba y al servidor de producción) se implementará una propiedad que permita ingresar la *url* base del servicio a utilizar.

Se agrega una prueba en *climabuilder_test.py* para verificar que dicha *url* se guarde correctamente:

climabuilder_test.py

```
    @data(None, "", "    ", "http://www.example.com",
"http://api.openweathermap.org/data/2.5/weather")
    def test_con_servicio_con_cualquier_url_deberia_incorporarlo(self,
cualquier_url: str):
        # Arrange
        builder = ClimaBuilder()

        # Act
        builder.con_servicio(cualquier_url)

        # Assert
```

```
(unit-testing) usuario@desktop:~/src/python/clima$ python3 climabuilder_test.py
.....EEEEE.............................
======================================================================
ERROR: test_builder_con_cualquier_url_deberia_incorporarlo_1_None
(__main__.ClimaBuilderUnitTests)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "/home/usuario/src/python/unit-testing/lib/python3.7/site-packages
/ddt.py", line 145, in wrapper
    return func(self, *args, **kwargs)
```

```
  File "climabuilder_test.py", line 165, in
test_con_servicio_con_cualquier_url_deberia_incorporarlo
    builder.con_servicio(cualquier_url)
AttributeError: 'ClimaBuilder' object has no attribute 'con_servicio'

======================================================================
ERROR: test_builder_con_cualquier_url_deberia_incorporarlo_2_
(__main__.ClimaBuilderUnitTests)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "/home/usuario/src/python/unit-testing/lib/python3.7/site-packages
/ddt.py", line 145, in wrapper
    return func(self, *args, **kwargs)
  File "climabuilder_test.py", line 165, in
test_builder_con_cualquier_url_deberia_incorporarlo
    builder.con_servicio(cualquier_url)
AttributeError: 'ClimaBuilder' object has no attribute 'con_servicio'

======================================================================
ERROR: test_builder_con_cualquier_url_deberia_incorporarlo_3_____
(__main__.ClimaBuilderUnitTests)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "/home/usuario/src/python/unit-testing/lib/python3.7/site-packages
/ddt.py", line 145, in wrapper
    return func(self, *args, **kwargs)
  File "climabuilder_test.py", line 165, in
test_builder_con_cualquier_url_deberia_incorporarlo
    builder.con_servicio(cualquier_url)
AttributeError: 'ClimaBuilder' object has no attribute 'con_servicio'

======================================================================
ERROR:
test_builder_con_cualquier_url_deberia_incorporarlo_4_http___www_example_com
(__main__.ClimaBuilderUnitTests)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "/home/usuario/src/python/unit-testing/lib/python3.7/site-packages
/ddt.py", line 145, in wrapper
    return func(self, *args, **kwargs)
  File "climabuilder_test.py", line 165, in
test_builder_con_cualquier_url_deberia_incorporarlo
    builder.con_servicio(cualquier_url)
AttributeError: 'ClimaBuilder' object has no attribute 'con_servicio'

======================================================================
ERROR:
test_builder_con_cualquier_url_deberia_incorporarlo_5_http___api_openweathermap_
org_data_2_5_weather (__main__.ClimaBuilderUnitTests)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "/home/usuario/src/python/unit-testing/lib/python3.7/site-packages
/ddt.py", line 145, in wrapper
    return func(self, *args, **kwargs)
  File "climabuilder_test.py", line 165, in
test_builder_con_cualquier_url_deberia_incorporarlo
    builder.con_servicio(cualquier_url)
AttributeError: 'ClimaBuilder' object has no attribute 'con_servicio'

----------------------------------------------------------------------
Ran 42 tests in 0.002s

FAILED (errors=5)
```

A continuación se verifica que el valor por defecto del servicio en el *builder* es el correcto. De esta manera se evitará tener que especificarlo cada vez que se quiera instanciar *Clima*.

climabuilder_test.py

```
    def
test_constructor_con_creacion_deberia_tener_idioma_ciudad_grados_pais_url(self):
        # Arrange
        idioma = "en"
        ciudad = "Buenos%20Aires"
        pais = "ar"
        grados = "kelvin"
        url_servicio = "http://api.openweathermap.org/data/2.5/weather"

        # Act
        builder = ClimaBuilder()

        # Assert
        self.assertEqual(pais, builder.pais)
        self.assertEqual(ciudad, builder.ciudad)
        self.assertEqual(idioma, builder.idioma)
        self.assertEqual(grados, builder.grados)
```

Y se agrega la *url* del servicio a los datos de la prueba:

climabuilder_test.py

```
    @data(
        ( 2172797, None, None, None, None, None, "english", "fahrenheit",
"http://api.openweathermap.org/data/1.0/weather" ),
        ( None, "Sydney", "au", None, None, None, "australian english",
"kelvin", "http://api.openweathermap.org/data/1.5/weather" ),
        ( None, None, None, None, 34.6037, 58.3816, "sp", "celsius",
"http://api.openweathermap.org/data/2.0/weather" ),
        ( None, None, None, "92730", None, None, "en", "fahrenheit",
"http://api.openweathermap.org/data/2.5/weather" ),
    )
    @unpack
    def test_build_con_varios_valores_deberia_crearlo_con_ellos(self, id: int,
ciudad: str, pais: str, cp: str, latitud: float, longitud: float, idioma: str,
grados: str, url_servicio: str):
        # Arrange
        apikey = "abcdefg"

        with patch.object(ClimaBuilder, "_crear_clima", return_value=None) as
mock_method:
            builder = (
                ClimaBuilder().con_apikey(apikey)
                              .con_id(id)
                              .con_servicio(url_servicio)
                              .con_ciudad(ciudad)
                              .en_pais(pais)
                              .en_codigo_postal(cp)
                              .en_coordenadas(latitud, longitud)
                              .en_idioma(idioma)
            )
```

```
            if grados == "fahrenheit":
                builder.en_fahrenheit()
            elif grados == "celsius":
                builder.en_celsius()
            else:
                builder.en_kelvin()

            # Act
            clima = builder.build()

            # Assert
            self.assertIsNone(clima)
            mock_method.assert_called_once_with(apikey, id, ciudad, pais, cp,
(latitud, longitud), idioma, grados, url_servicio)
```

Ahora se implementa la *url* en la clase *ClimaBuilder* y se modifica *_crear_clima* y *build* para agregar el argumento extra:

climabuilder.py

```
class ClimaBuilder:
    def __init__(self):
        self.__apikey = None
        self.__id = None
        self.__coordenadas = None
        self.__codigo_postal = None
        self.__grados = "kelvin"
        self.__pais = "ar"
        self.__ciudad = "Buenos%20Aires"
        self.__idioma = "en"
        self.__url_servicio = "http://api.openweathermap.org/data/2.5/weather"

    def con_servicio(self, url_servicio: str):
        self.__url_servicio = url_servicio
        return self

    @property
    def url_servicio(self) -> str:
        return self.__url_servicio

    def _crear_clima(self, apikey: str, id: int, ciudad: str, pais: str, cp:
str, coordenadas: (float, float), idioma: str, grados: str, url_servicio: str)
-> Clima:
        return Clima(apikey, id, ciudad, pais, cp, coordenadas, idioma, grados,
url_servicio)

    def build(self):
        return self._crear_clima(self.__apikey, self.__id, self.__ciudad,
self.__pais, self.__codigo_postal, self.__coordenadas, self.__idioma,
```

```
(unit-testing) usuario@desktop:~/src/python/clima$ python3 clima_test.py
.........................................
----------------------------------------------------------------------
Ran 42 tests in 0.001s

OK
```

Se agrega el nuevo argumento al constructor de la clase *Clima* y se la expone a través de una propiedad:

clima.py

```
class Clima:
    def __init__(self, apikey: str, id: int, ciudad: str, pais: str, cp: str,
coordenadas: (float, float), idioma: str, grados: str, url_servicio: str):
        self.__apikey = apikey.strip() if apikey is not None else ""
        if self.__apikey == "":
            raise ValueError()

        self.__id = id
        self.__ciudad = ciudad
        self.__pais = pais
        self.__codigo_postal = cp
        self.__coordenadas = coordenadas
        self.__idioma = idioma
        self.__grados = grados
        self.__url_servicio = url_servicio
        self.__url = self._build_url()

    @property
    def url_servicio(self) -> str:
```

Por último se modifica *_build_url* dentro de la clase *Clima* para que no tenga la dirección del servidor fija:

clima.py

```
    def _build_url(self):
        url = f"{self.__url_servicio}?appid={self.__apikey}"
        if self.__id is not None:
            url = f"{url}&id={self.__id}"
        elif self.__ciudad is not None:
            url = f"{url}&q={self.__ciudad}"

            if self.__pais is not None:
                url = f"{url},{self.__pais}"
        elif self.__codigo_postal is not None:
            url = f"{url}&zip={self.__codigo_postal}"

            if self.__pais is not None:
                url = f"{url},{self.__pais}"
        elif self.__coordenadas is not None:
            if None in self.__coordenadas:
                raise ValueError()

            url = f"{url}&lat={self.__coordenadas[0]}&
lon={self.__coordenadas[1]}"
        else:
            raise ValueError()

        if self.__idioma is not None and self.__idioma != "" and self.__idioma
!= "en":
            url = f"{url}&lang={self.__idioma}"

        if self.__grados is not None:
```

```
            if self.__grados == "fahrenheit":
                url = f"{url}&units=imperial"
            elif self.__grados == "celsius":
                url = f"{url}&units=metric"

        return url
```

Dentro de *clima_test.py* existen tres pruebas que llaman al constructor de *Clima* directamente en lugar de usar *ClassBuilder*, éstas deben ser modificadas para agregar la dirección del servicio a usar:

clima_test.py

```
    @data(
        ( 2172797, None, None, None, (None, None), "", "fahrenheit",
"http://api.openweathermap.org/data/2.5/weather?appid=abcdefg&id=2172797&
units=imperial" ),
        ( None, "Sydney", "au", None, (None, None), "fr", "kelvin",
"http://api.openweathermap.org/data/2.5/weather?appid=abcdefg&q=Sydney,au&
lang=fr" ),
        ( None, None, None, None, (34.6037, 58.3816), "es", "celsius",
"http://api.openweathermap.org/data/2.5/weather?appid=abcdefg&lat=34.6037&
lon=58.3816&lang=es&units=metric" ),
        ( None, None, None, "94040", (None, None), "ja", "fahrenheit",
"http://api.openweathermap.org/data/2.5/weather?appid=abcdefg&zip=94040&lang=ja&
units=imperial" ),
        ( None, None, "ar", "94040", (None, None), "ja", "fahrenheit",
"http://api.openweathermap.org/data/2.5/weather?appid=abcdefg&zip=94040,ar&
lang=ja&units=imperial" )
    )
    @unpack
    def test_url_con_datos_deberia_crear_url(self, id: int, ciudad: str, pais:
str, cp: str, coordenadas: (float, float), idioma: str, grados: str, url: str):
        # Arrange
        apikey = "abcdefg"
        url_servicio = "http://api.openweathermap.org/data/2.5/weather"

        # Act
        clima = Clima(apikey, id, ciudad, pais, cp, coordenadas, idioma, grados,
url_servicio)

        # Assert
        self.assertEqual(url, clima.url)

    def
test_constructor_con_ciudad_codigo_postal_coordenadas_id_nulas_deberia_lanzar_va
lueerror(self):
        # Arrange
        apikey = "abcdefg"
        url_servicio = "http://api.openweathermap.org/data/2.5/weather"

        # Act / Assert
        with self.assertRaises(ValueError) as ve:
            sut = Clima(apikey, None, None, None, None, None, None, None,
url_servicio)

    def
test_constructor_con_latitud_o_longitud_nulas_y_ninguna_otra_locacion_deberia_la
```

```
        # Arrange
        apikey = "abcdefg"
        url_servicio = "http://api.openweathermap.org/data/2.5/weather"

        # Act / Assert
        with self.assertRaises(ValueError) as ve:
            sut = Clima(apikey, None, None, None, None, (None, None), None,
None, url_servicio)
```

```
(unit-testing) usuario@desktop:~/src/python/clima$ python3 clima_test.py
.............................
----------------------------------------------------------------------
Ran 31 tests in 0.001s

OK
```

Falta agregar una nueva validación: Si el constructor recibe una dirección nula o vacía debería lanzar una excepción del tipo *ValueError* similar a como lo hace con *API key*.

clima_test.py

```
    @data(None, "", "    ")
    def
test_constructor_con_un_url_servicio_invalido_deberia_lanzar_valueerror(self,
url_servicio: str):
        with self.assertRaises(ValueError) as ve:
            sut =
```

```
(unit-testing) usuario@desktop:~/src/python/clima$ python3 clima_test.py
......................FFF........
======================================================================
FAIL:
test_constructor_con_un_url_servicio_invalido_deberia_lanzar_valueerror_1_None
(__main__.ClimaUnitTests)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "/home/usuario/src/python/unit-testing/lib/python3.7/site-packages
/ddt.py", line 145, in wrapper
    return func(self, *args, **kwargs)
  File "clima_test.py", line 140, in
test_constructor_con_un_url_servicio_invalido_deberia_lanzar_valueerror
    sut =
ClimaBuilder().con_apikey("abcdefg").con_servicio(url_servicio).build()
AssertionError: ValueError not raised

======================================================================
FAIL: test_constructor_con_un_url_servicio_invalido_deberia_lanzar_valueerror_2_
(__main__.ClimaUnitTests)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "/home/usuario/src/python/unit-testing/lib/python3.7/site-packages
/ddt.py", line 145, in wrapper
    return func(self, *args, **kwargs)
  File "clima_test.py", line 140, in
test_constructor_con_un_url_servicio_invalido_deberia_lanzar_valueerror
```

```
    sut =
ClimaBuilder().con_apikey("abcdefg").con_servicio(url_servicio).build()
AssertionError: ValueError not raised

======================================================================
FAIL:
test_constructor_con_un_url_servicio_invalido_deberia_lanzar_valueerror_3_____
(__main__.ClimaUnitTests)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "/home/usuario/src/python/unit-testing/lib/python3.7/site-packages
/ddt.py", line 145, in wrapper
    return func(self, *args, **kwargs)
  File "clima_test.py", line 140, in
test_constructor_con_un_url_servicio_invalido_deberia_lanzar_valueerror
    sut =
ClimaBuilder().con_apikey("abcdefg").con_servicio(url_servicio).build()
AssertionError: ValueError not raised

----------------------------------------------------------------------
Ran 34 tests in 0.001s

FAILED (failures=3)
```

Se agrega la validación para que el constructor de *Clima* lance la excepción si la dirección del servidor está vacía o es nula:

clima.py

```
class Clima:
    def __init__(self, apikey: str, id: int, ciudad: str, pais: str, cp: str,
coordenadas: (float, float), idioma: str, grados: str, url_servicio: str):
        self.__apikey = apikey.strip() if apikey is not None else ""
        if self.__apikey == "":
            raise ValueError()

        self.__url_servicio = url_servicio.strip() if url_servicio is not None
else ""
        if self.__url_servicio == "":
            raise ValueError()

        self.__id = id
        self.__ciudad = ciudad
        self.__pais = pais
        self.__codigo_postal = cp
        self.__coordenadas = coordenadas
        self.__idioma = idioma
        self.__grados = grados
```

```
(unit-testing) usuario@desktop:~/src/python/clima$ python3 climabuilder_test.py
.........................................
----------------------------------------------------------------------
Ran 42 tests in 0.002s

OK
(unit-testing) usuario@desktop:~/src/python/clima$ python3 clima_test.py
................................
----------------------------------------------------------------------
```

```
Ran 34 tests in 0.001s

OK
```

Todo listo para implementar la conexión con el servidor de *OpenWeather*. Sin embargo las pruebas unitarias no deben acceder a servicios externos ya que deben ser tan atómicos como sea posible, repetibles y con la menor latencia posible, por consiguiente se deberá modelar sin realizar la conexión efectiva contra el servidor.

Conexión y procesamiento de la información

Hasta este momento existe una clase *ClimaBuilder* que permite crear una instancia de *Clima* la cual, internamente, prepara la *url* a la cual se conectará para obtener la información del clima. Solo queda implementar un método *obtener* que así lo haga. Sabiendo que el servidor retornará un *Json*, es posible retornar directamente la estructura obtenida para delegar su procesamiento, o procesarla internamente y retornar los datos en una nueva estructura propia.

Ya que la segunda opción permite independizar al programa del formato del archivo *Json* que retorna el servidor se selecciona dicha alternativa: el método *obtener* va a devolver una nueva clase *Pronostico* que contará con un constructor que recibirá la respuesta y propiedades para acceder a los valores parseados (título, descripción, etc). También se va a sobrecargar el método mágico *__str__* para obtener los datos del clima en forma de oración.

Es posible verificar que la ciudad es la pedida, sin embargo el clima cambia diariamente haciendo imposible crear una prueba eficaz. Ese es uno de los motivos por los que se utilizan ya sean datos fijos (por ejemplo, una *url* de prueba de donde siempre vengan los mismos datos) o *mocking* o simulaciones de respuestas. En el caso particular de esta aplicación se agregará a *ClimaBuilder* un nuevo método que permita definir la *url* base del sitio al cual conectarse. Esa base se transferirá a la instancia de *Clima* a través del último parámetro de su constructor. De esta manera será posible crear una prueba unitaria con la cual siempre se recibirán los mismos datos.

Para evitar los factores externos se obtiene una copia del *Json* del servidor de prueba de prueba y se lo utiliza para inicializar la instancia de *Pronostico* a probar. Por brevedad se muestra

nuevamente el resultado final de la prueba creada:

pronostico_test.py

```
import unittest
import json
from pronostico import Pronostico

class PronosticoUnitTests(unittest.TestCase):
    def test_constructor_con_json_valido_deberia_incorporarlo(self):
        # Arrange
        json_data = json.loads('{"coord":{"lon":-0.13, "lat":51.51}, "weather":
[{"id":300, "main":"Drizzle", "description":"light intensity drizzle",
"icon":"09d"}], "base":"stations", "main":{"temp":280.32, "pressure":1012,
"humidity":81, "temp_min":279.15, "temp_max":281.15}, "visibility":10000,
"wind":{"speed":4.1, "deg":80}, "clouds":{"all":90}, "dt":1485789600, "sys":
{"type":1, "id":5091, "message":0.0103, "country":"GB", "sunrise":1485762037,
"sunset":1485794875}, "id":2643743, "name":"London", "cod":200}')

        # Act
        pronostico = Pronostico(json_data)

        # Assert
        self.assertIsNotNone(pronostico)
        self.assertEqual("London", pronostico.ciudad)
        self.assertEqual((51.51, -0.13), pronostico.coordenadas)
        self.assertEqual("Drizzle", pronostico.titulo)
        self.assertEqual("light intensity drizzle", pronostico.descripcion)
        self.assertEqual(280.32, pronostico.temperatura)
        self.assertEqual(279.15, pronostico.temperatura_minima)
        self.assertEqual(281.15, pronostico.temperatura_maxima)
        self.assertEqual("GB", pronostico.pais)

if __name__ == "__main__":
          unittest.main()



(unit-testing) usuario@desktop:~/src/python/clima$ python3 pronostico_test.py
Traceback (most recent call last):
  File "pronostico_test.py", line 3, in <module>
    from pronostico import Pronostico
ModuleNotFoundError: No module named 'pronostico'
```

Con esa prueba unitaria se acaba de definir la interface pública de la clase *Pronostico*, la cual se plasma en su correspondiente archivo fuente:

pronostico.py

```
from typing import Any

class Pronostico:
    def __init__(self, json: Any):
        self.__json_data = json
        self.__titulo = self.__json_data["weather"][0]["main"]
        self.__descripcion = self.__json_data["weather"][0]["description"]
        self.__temperatura = self.__json_data["main"]["temp"]
        self.__tempmin = self.__json_data["main"]["temp_min"]
```

```
        self.__tempmax = self.__json_data["main"]["temp_max"]
        self.__ciudad = self.__json_data["name"]
        self.__coordenadas = (self.__json_data["coord"]["lat"],
self.__json_data["coord"]["lon"])
        self.__pais = self.__json_data["sys"]["country"]

    @property
    def titulo(self) -> str:
        return self.__titulo

    @property
    def descripcion(self) -> str:
        return self.__descripcion

    @property
    def temperatura(self) -> str:
        return self.__temperatura

    @property
    def temperatura_minima(self) -> str:
        return self.__tempmin

    @property
    def temperatura_maxima(self) -> str:
        return self.__tempmax

    @property
    def ciudad(self) -> str:
        return self.__ciudad

    @property
    def pais(self) -> str:
        return self.__pais

    @property
    def coordenadas(self) -> (float, float):
        return self.__coordenadas
```

```
(unit-testing) usuario@desktop:~/src/python/clima$ python3 pronostico_test.py
.
----------------------------------------------------------------------
Ran 1 test in 0.000s

OK
```

Inmediatamente se puede notar que si el *Json* es nulo el programa lanzará una excepción por intentar acceder a campos que no existen. Es necesario crear una prueba para validar el caso:

pronostico_test.py

```
    def test_constructor_con_json_nulo_deberia_lanzar_valueerror(self):
        with self.assertRaises(ValueError) as ve:
```

```
(unit-testing) usuario@desktop:~/src/python/clima$ python3 pronostico_test.py
```

```
E.
======================================================================
ERROR: test_constructor_con_json_nulo_deberia_lanzar_valueerror
(__main__.PronosticoUnitTests)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "pronostico_test.py", line 27, in
test_constructor_con_json_nulo_deberia_lanzar_valueerror
    sut = Pronostico(None)
  File "/home/usuario/src/python/clima/pronostico.py", line 6, in __init__
    self.__titulo = self.__json_data["weather"][0]["main"]
TypeError: 'NoneType' object is not subscriptable

----------------------------------------------------------------------
Ran 2 tests in 0.000s

FAILED (errors=1)
```

pronostico.py

```
    def __init__(self, json: Any):
        if json is None:
            raise ValueError()

        self.__json_data = json
        self.__titulo = self.__json_data["weather"][0]["main"]
        self.__descripcion = self.__json_data["weather"][0]["description"]
        self.__temperatura = self.__json_data["main"]["temp"]
        self.__tempmin = self.__json_data["main"]["temp_min"]
        self.__tempmax = self.__json_data["main"]["temp_max"]
        self.__ciudad = self.__json_data["name"]
        self.__coordenadas = (self.__json_data["coord"]["lat"],
self.__json_data["coord"]["lon"])
```

```
(unit-testing) usuario@desktop:~/src/python/clima$ python3 pronostico_test.py
..
----------------------------------------------------------------------
Ran 2 tests in 0.000s

OK
```

Solo queda reescribir el método mágico *__str__* para que imprima de manera prolija la información del clima. Se agrega una nueva prueba en *pronostico_test.py* y luego se implementa la solución en *pronostico.py*.

pronostico_test.py

```
    def  test___str__con_json_valido_deberia_generar_oracion(self):
        # Arrange
        json_data = json.loads('{"coord":{"lon":-0.13, "lat":51.51}, "weather":
[{"id":300,"main":"Drizzle", "description":"light intensity drizzle",
"icon":"09d"}], "base":"stations", "main":{"temp":280.32, "pressure":1012,
"humidity":81, "temp_min":279.15, "temp_max":281.15}, "visibility":10000,
"wind":{"speed":4.1, "deg":80}, "clouds":{"all":90}, "dt":1485789600, "sys":
```

```
"sunset":1485794875}, "id":2643743, "name":"London", "cod":200}')
        pronostico = Pronostico(json_data)

        # Act
        resultado = str(pronostico)

        # Assert
        self.assertEqual("London (GB) currently has light intensity drizzle with
a temperature of 280.32F.", resultado);
```

```
(unit-testing) usuario@desktop:~/src/python/clima$ python3 pronostico_test.py
F..
======================================================================
FAIL: test___str__con_json_valido_deberia_generar_oracion
(__main__.PronosticoUnitTests)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "pronostico_test.py", line 37, in
test___str__con_json_valido_deberia_generar_oracion
    self.assertEqual("London currently has light intensity drizzle, a
temperature of 280.32F.", resultado);
AssertionError: 'London currently has light intensity drizzle, a temperature of
280.32F.' != '<pronostico.Pronostico object at 0x7fc72ce4ded0>'
- London currently has light intensity drizzle, a temperature of 280.32F.
+ <pronostico.Pronostico object at 0x7fc72ce4ded0>


----------------------------------------------------------------------
Ran 3 tests in 0.000s

FAILED (failures=1)
```

Por defecto se está imprimiendo la dirección en memoria de la instancia de *Pronostico*. Mediante la siguiente implementación se modifica dicho comportamiento.

pronostico.py

```
    def __str__(self):
        return f"{self.ciudad} ({self.pais}) currently has {self.descripcion}
```

```
(unit-testing) usuario@desktop:~/src/python/clima$ python3 pronostico_test.py
...
----------------------------------------------------------------------
Ran 3 tests in 0.000s
```

En este momento la "F" de Fahrenheit está fija ya que ese dato no viaja en el *Json* de la respuesta, por consiguiente hay que pasársela a la clase al momento de la creación, siendo necesario modificar las pruebas unitarias de *Pronostico* para quedar como sigue:

pronostico_test.py

```
import unittest
import json
from pronostico import Pronostico

class PronosticoUnitTests(unittest.TestCase):
    def test_constructor_con_json_valido_deberia_incorporarlo(self):
        # Arrange
        json_data = json.loads('{"coord":{"lon":-0.13, "lat":51.51}, "weather":
[{"id":300, "main":"Drizzle", "description":"light intensity drizzle",
"icon":"09d"}], "base":"stations", "main":{"temp":280.32, "pressure":1012,
"humidity":81, "temp_min":279.15, "temp_max":281.15}, "visibility":10000,
"wind":{"speed":4.1, "deg":80}, "clouds":{"all":90}, "dt":1485789600, "sys":
{"type":1, "id":5091, "message":0.0103, "country":"GB", "sunrise":1485762037,
"sunset":1485794875}, "id":2643743, "name":"London", "cod":200}')

        # Act
        pronostico = Pronostico(json_data, "fahrenheit")

        # Assert
        self.assertIsNotNone(pronostico)
        self.assertEqual("London", pronostico.ciudad)
        self.assertEqual((51.51, -0.13), pronostico.coordenadas)
        self.assertEqual("Drizzle", pronostico.titulo)
        self.assertEqual("light intensity drizzle", pronostico.descripcion)
        self.assertEqual(280.32, pronostico.temperatura)
        self.assertEqual(279.15, pronostico.temperatura_minima)
        self.assertEqual(281.15, pronostico.temperatura_maxima)
        self.assertEqual("GB", pronostico.pais)


    def test_constructor_con_json_nulo_deberia_lanzar_valueerror(self):
        with self.assertRaises(ValueError) as ve:
            sut = Pronostico(None, "fahrenheit")

    def  test___str__con_json_valido_deberia_generar_oracion(self):
        # Arrange
        json_data = json.loads('{"coord":{"lon":-0.13, "lat":51.51}, "weather":
[{"id":300, "main":"Drizzle", "description":"light intensity drizzle",
"icon":"09d"}], "base":"stations", "main":{"temp":280.32, "pressure":1012,
"humidity":81, "temp_min":279.15, "temp_max":281.15}, "visibility":10000,
"wind":{"speed":4.1, "deg":80}, "clouds":{"all":90}, "dt":1485789600, "sys":
{"type":1, "id":5091, "message":0.0103, "country":"GB", "sunrise":1485762037,
"sunset":1485794875}, "id":2643743, "name":"London", "cod":200}')
        pronostico = Pronostico(json_data, "fahrenheit")

        # Act
        resultado = str(pronostico)

        # Result
        self.assertEqual("London (GB) currently has light intensity drizzle with
a temperature of 280.32F.", resultado);

if __name__ == "__main__":
          unittest.main()


(unit-testing) usuario@desktop:~/src/python/clima$ python3 pronostico_test.py
EEE
======================================================================
ERROR: test___str__con_json_valido_deberia_generar_oracion
```

```
(__main__.PronosticoUnitTests)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "pronostico_test.py", line 32, in
test___str__con_json_valido_deberia_generar_oracion
    pronostico = Pronostico(json_data, "imperial")
TypeError: __init__() takes 2 positional arguments but 3 were given

======================================================================
ERROR: test_constructor_con_json_nulo_deberia_lanzar_valueerror
(__main__.PronosticoUnitTests)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "pronostico_test.py", line 27, in
test_constructor_con_json_nulo_deberia_lanzar_valueerror
    sut = Pronostico(None, "imperial")
TypeError: __init__() takes 2 positional arguments but 3 were given

======================================================================
ERROR: test_constructor_con_json_valido_deberia_incorporarlo
(__main__.PronosticoUnitTests)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "pronostico_test.py", line 11, in
test_constructor_con_json_valido_deberia_incorporarlo
    pronostico = Pronostico(json_data, "imperial")
TypeError: __init__() takes 2 positional arguments but 3 were given

----------------------------------------------------------------------
Ran 3 tests in 0.000s

FAILED (errors=3)
```

pronostico.py

```
    def __init__(self, json: Any, grados: str):
        if json is None:
            raise ValueError()

        self.__json_data = json
        self.__titulo = self.__json_data["weather"][0]["main"]
        self.__descripcion = self.__json_data["weather"][0]["description"]
        self.__temperatura = self.__json_data["main"]["temp"]
        self.__tempmin = self.__json_data["main"]["temp_min"]
        self.__tempmax = self.__json_data["main"]["temp_max"]
        self.__ciudad = self.__json_data["name"]
        self.__coordenadas = (self.__json_data["coord"]["lat"],
self.__json_data["coord"]["lon"])
        self.__pais = self.__json_data["sys"]["country"]
```

```
(unit-testing) usuario@desktop:~/src/python/clima$ python3 pronostico_test.py
...
----------------------------------------------------------------------
Ran 3 tests in 0.000s

OK
```

Las pruebas pasaron, sin embargo la "F" de la temperatura aún

está fija por lo que es necesario modificar la prueba *test___str__con_json_valido_deberia_generar_oracion* para agregarle una lista de datos y cambiar dinámicamente el texto. Primero hay que agregar una referencia al módulo *ddt* para luego modificar el resto.

pronostico_test.py

```
import unittest
import json
from ddt import ddt, data
from pronostico import Pronostico

@ddt
```

pronostico_test.py

```
    @data(("kelvin", "K"), ("imperial", "F"), ("metric", "C"))
    def test___str__con_json_valido_deberia_generar_oracion(self, grados: (str,
str)):
        # Arrange
        json_data = json.loads('{"coord":{"lon":-0.13, "lat":51.51}, "weather":
[{"id":300, "main":"Drizzle", "description":"light intensity drizzle",
"icon":"09d"}], "base":"stations", "main":{"temp":280.32, "pressure":1012,
"humidity":81, "temp_min":279.15, "temp_max":281.15}, "visibility":10000,
"wind":{"speed":4.1, "deg":80}, "clouds":{"all":90}, "dt":1485789600, "sys":
{"type":1, "id":5091, "message":0.0103, "country":"GB", "sunrise":1485762037,
"sunset":1485794875}, "id":2643743, "name":"London", "cod":200}')
        pronostico = Pronostico(json_data, grados[0])

        # Act
        resultado = str(pronostico)

        # Result
        self.assertEqual(f"London (GB) currently has light intensity drizzle
```

```
(unit-testing) usuario@desktop:~/src/python/clima$ python3 pronostico_test.py
F.F..
======================================================================
FAIL: test___str__con_json_valido_deberia_generar_oracion_1___kelvin____K__
(__main__.PronosticoUnitTests)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "/home/usuario/src/python/unit-testing/lib/python3.7/site-packages
/ddt.py", line 145, in wrapper
    return func(self, *args, **kwargs)
  File "pronostico_test.py", line 41, in
test___str__con_json_valido_deberia_generar_oracion
    self.assertEqual(f"London (GB) currently has light intensity drizzle with a
temperature of 280.32{grados[1]}.", resultado);
AssertionError: 'Lond[14 chars]tly has light intensity drizzle with a
temperature of 280.32K.' != 'Lond[14 chars]tly has light intensity drizzle with
a temperature of 280.32F.'
- London (GB) currently has light intensity drizzle with a temperature of
280.32K.
```

```
?
^
+ London (GB) currently has light intensity drizzle with a temperature of
280.32F.
?
^


======================================================================
FAIL: test___str__con_json_valido_deberia_generar_oracion_3___metric____C__
(__main__.PronosticoUnitTests)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "/home/usuario/src/python/unit-testing/lib/python3.7/site-packages
/ddt.py", line 145, in wrapper
    return func(self, *args, **kwargs)
  File "pronostico_test.py", line 41, in
test___str__con_json_valido_deberia_generar_oracion
    self.assertEqual(f"London (GB) currently has light intensity drizzle with a
temperature of 280.32{grados[1]}.", resultado);
AssertionError: 'Lond[14 chars]tly has light intensity drizzle with a
temperature of 280.32C.' != 'Lond[14 chars]tly has light intensity drizzle with
a temperature of 280.32F.'
- London (GB) currently has light intensity drizzle with a temperature of
280.32C.
?
^
+ London (GB) currently has light intensity drizzle with a temperature of
280.32F.
?
^


----------------------------------------------------------------------
Ran 5 tests in 0.001s

FAILED (failures=2)
```

Una vez verificado el error, se procede a solucionarlo:

pronostico.py

```
    def _visualizar_grados(self):
        if self.__grados == "celsius":
            return "C"
        elif self.__grados == "fahrenheit":
            return "F"
        else:
            return "K"

    def __str__(self):
        return f"{self.ciudad} ({self.pais}) currently has {self.descripcion}
```

```
(unit-testing) usuario@desktop:~/src/python/clima$ python3 pronostico_test.py
.....
----------------------------------------------------------------------
Ran 5 tests in 0.000s
```

```
OK
(unit-testing) usuario@desktop:~/src/python/clima$ python3 climabuilder_test.py
....................................
----------------------------------------------------------------------
Ran 42 tests in 0.001s

OK
(unit-testing) usuario@desktop:~/src/python/clima$ python3 clima_test.py
.............................
----------------------------------------------------------------------
Ran 34 tests in 0.001s

OK
```

Prueba de integración

La < href="https://es.wikipedia.org/wiki/Prueba_de_integraci %C3%B3n">prueba de integración se realiza para comprobar que los diferentes módulos del programa se interrelacionen correctamente entre ellos. Por ser este proyecto simple, la única prueba de integración que podría existir es el proceso completo del método *obtener* ya que necesita de dos de las tres clases (*Clima* y *Pronostico*), aunque desde la prueba se necesita el *builder* también.

Para realizar dicha prueba, por ejemplo, se debe utilizar a *ClimaBuilder* para crear el objeto *Clima*, luego se invoca al método *obtener* para que retorne una instancia de la clase *Pronostico* a la cual se le va a inyectar la respuesta del servidor.

Para la implementación del método que realizará la llamada al servidor se crea una nueva prueba:

clima_test.py

```
    def test_obtener_con_el_objeto_creado_deberia_obtener_json(self):
        # Arrange
        clima = (
            ClimaBuilder().con_apikey("abcdefg")
                          .con_ciudad("London")
                          .en_pais("uk")
                          .con_servicio("http://api.openweathermap.org/data/2.5
/weather")
                          .build()
        )
        # Act
        pronostico = clima.obtener()

        # Assert
        self.assertIsNotNone(pronostico)
```

```
(unit-testing) usuario@desktop:~/src/python/clima$ python3 clima_test.py
.........................E.....
======================================================================
ERROR: test_obtener_con_el_objeto_creado_deberia_obtener_json
(__main__.ClimaUnitTests)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "clima_test.py", line 152, in
test_obtener_con_el_objeto_creado_deberia_obtener_json
    pronostico = clima.obtener()
AttributeError: 'Clima' object has no attribute 'obtener'

----------------------------------------------------------------------
Ran 35 tests in 0.001s

FAILED (errors=1)
```

Y se implementa la solución más simple, importando *Pronostico*:

clima.py

```
import urllib.request, json
from pronostico import Pronostico
```

clima.py

```
    def obtener(self) -> Pronostico:
        try:
            with urllib.request.urlopen(self.__url) as url:
                data = url.read().decode()

                try:
                    json_data = json.loads(data)
                    return Pronostico(json_data, self.__grados)
                except:
                    return None
        except:
```

```
(unit-testing) usuario@desktop:~/src/python/clima$ python3 clima_test.py
.........................F.....
======================================================================
FAIL: test_obtener_con_el_objeto_creado_deberia_obtener_json
(__main__.ClimaUnitTests)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "clima_test.py", line 155, in
test_obtener_con_el_objeto_creado_deberia_obtener_json
    self.assertIsNotNone(pronostico)
AssertionError: unexpectedly None

----------------------------------------------------------------------
Ran 35 tests in 0.502s

FAILED (failures=1)
```

El problema que sucedió es que se intentó conectar al servidor de producción con una *API key* incorrecta, por consiguiente no es posible conocer si el método en sí funcionó o no. La mejor solución es crear nuevamente un *mock* u objeto simulado para tener una respuesta fija que podamos inyectar. Sin embargo, antes se debe dividir al método *obtener* en dos o más secciones para poder inyectar únicamente la respuesta del servidor.

clima.py

```
    def obtener(self) -> Pronostico:
        data = self._get_server_data(self.__url)

        if data:
            json = self._deserialize_json(data)

            if json:
                return Pronostico(json, self.__grados)

    def _deserialize_json(self, data: str) -> Any:
        try:
            return json.loads(data)
        except:
            return None

    def _get_server_data(self, link: str) -> str:
        try:
            with urllib.request.urlopen(link) as url:
                data = url.read().decode()
        except:
            data = None
```

Ahora que se ha dividido a la obtención de datos del procesamiento, es posible crear una nueva prueba que inyecte el *Json* de ejemplo. Primero se importan los módulos correspondientes:

clima_test.py

```
import unittest
from unittest.mock import patch
from ddt import ddt, data, unpack
from climabuilder import ClimaBuilder
```

Y se reescribe la prueba para que utilice simulación:

clima_test.py

```
    def test_obtener_con_el_objeto_creado_deberia_obtener_json(self):
        # Arrange
        respuesta = '{"coord":{"lon":-0.13, "lat":51.51}, "weather":[{"id":300,
"main":"Drizzle", "description":"light intensity drizzle", "icon":"09d"}],
```

```
"base":"stations", "main":{"temp":280.32, "pressure":1012, "humidity":81,
"temp_min":279.15, "temp_max":281.15}, "visibility":10000, "wind":{"speed":4.1,
"deg":80}, "clouds":{"all":90}, "dt":1485789600, "sys":{"type":1, "id":5091,
"message":0.0103, "country":"GB", "sunrise":1485762037, "sunset":1485794875},
"id":2643743, "name":"London", "cod":200}'

        with patch.object(Clima, "_get_server_data", return_value=respuesta) as
mock_method:
            clima = (
                ClimaBuilder().con_apikey("abcdefg")
                              .con_ciudad("London")
                              .en_pais("uk")
                              .con_servicio("http://api.openweathermap.org
/data/2.5/weather")
                              .build()
            )

            # Act
            pronostico = clima.obtener()

            # Assert
            self.assertIsNotNone(pronostico)
            self.assertEqual(clima.ciudad, pronostico.ciudad)



(unit-testing) usuario@desktop:~/src/python/clima$ python3 clima_test.py
.................................
----------------------------------------------------------------------
Ran 35 tests in 0.001s

OK
```

Nuevamente, por ser un programa simple la prueba de integración no costará más recursos (tiempo o procesamiento) y puede ser parte de las pruebas unitarias. Pero en un sistema profesional debería permanecer separado del resto de las pruebas ya que se suponen son pruebas mucho más pesadas y sólo deberían ejecutarse al combinar módulos durante el proceso de integración.

Prueba de sistema

Probar ya no un método específico sino el programa entero y su interacción con el resto de los recursos externos (en nuestro caso, que pueda comunicarse con un servidor externo para extraer la información) se denomina *prueba de sistema*. La prueba de sistema es la última oportunidad que tienen los desarrolladores de capturar errores en el programa antes de que los clientes puedan encontrarlas en las *pruebas de validación*.

En este programa, la prueba de sistema es igual a la prueba de integración excepto que no tendrá ninguna simulación, se conectará a los recursos externos (en este caso, a la *API* externa).

Al igual que la prueba de integración debe estar separada del resto. Por consiguiente, se creará un nuevo archivo, *sistema_test.py*. Como al llegar a este paso el sistema ya está completo, no debería ser necesario modificar el código fuente.

sistema_test.py

```
import unittest
from climabuilder import ClimaBuilder
from clima import Clima
from pronostico import Pronostico

class SistemaTests(unittest.TestCase):
    def test_con_el_sistema_completo_deberia_funcionar(self):
        # Arrange
        clima = (
            ClimaBuilder().con_apikey("b6907d289e10d714a6e88b30761fae2")
                          .con_ciudad("London")
                          .en_pais("uk")
                          .con_servicio("https://samples.openweathermap.org
/data/2.5/weather")
                          .build()
        )

        # Act
        pronostico = clima.obtener()

        # Assert
        self.assertIsNotNone(pronostico)
        self.assertEqual(clima.ciudad, pronostico.ciudad)
        self.assertEqual((51.51, -0.13), pronostico.coordenadas)
        self.assertEqual("Drizzle", pronostico.titulo)
        self.assertEqual("light intensity drizzle", pronostico.descripcion)
        self.assertEqual(280.32, pronostico.temperatura)
        self.assertEqual(279.15, pronostico.temperatura_minima)
        self.assertEqual(281.15, pronostico.temperatura_maxima)
        self.assertEqual("GB", pronostico.pais)

if __name__ == "__main__":
    unittest.main()
```

```
(unit-testing) usuario@desktop:~/src/python/clima$ python3 sistema_test.py
.
----------------------------------------------------------------------
Ran 1 test in 1.012s

OK
```

Test suites

Hasta este momento las pruebas han sido agrupadas en casos de pruebas o *test cases*. Sin embargo, existe una agrupación mayor: *test suites*. Un *test suite* puede agrupar a varios *test*

cases, y un *test case* puede agrupar a varios *test methods* o métodos de prueba. Los *suites* pueden agrupar casos de distintos grupos de pruebas, por ejemplo, un *suite* podría contener todas las pruebas referentes a climas por ciudad, mientras que otro podría contener pruebas referentes con mapas de climas.

full_test.py

```
import unittest
import clima_test
import climabuilder_test
import pronostico_test
import sistema_test

if __name__ == "__main__":
    suite = unittest.TestSuite()

    suite.addTest(unittest.makeSuite(clima_test.ClimaUnitTests))
    suite.addTest(unittest.makeSuite(climabuilder_test.ClimaBuilderUnitTests))
    suite.addTest(unittest.makeSuite(pronostico_test.PronosticoUnitTests))
    suite.addTest(unittest.makeSuite(sistema_test.SistemaTests))

    unittest.TextTestRunner(verbosity=3).run(suite)
```

```
(unit-testing) usuario@desktop:~/src/python/clima$ python3 full_test.py
test_constructor_con_celsius_deberia_incorporarlo (clima_test.ClimaUnitTests)
... ok
test_constructor_con_ciudad_codigo_postal_coordenadas_id_nulas_deberia_lanzar_va
lueerror (clima_test.ClimaUnitTests) ... ok
test_constructor_con_ciudad_valida_deberia_incorporarla
(clima_test.ClimaUnitTests) ... ok
test_constructor_con_cualquier_codigo_postal_deberia_incorporarlo_1_None
(clima_test.ClimaUnitTests) ... ok
test_constructor_con_cualquier_codigo_postal_deberia_incorporarlo_2_
(clima_test.ClimaUnitTests) ... ok
test_constructor_con_cualquier_codigo_postal_deberia_incorporarlo_3_13233
(clima_test.ClimaUnitTests) ... ok
test_constructor_con_cualquier_coordenadas_deberia_incorporarlo
(clima_test.ClimaUnitTests) ... ok
test_constructor_con_cualquier_id_deberia_incorporarlo_1_None
(clima_test.ClimaUnitTests) ... ok
test_constructor_con_cualquier_id_deberia_incorporarlo_2__11
(clima_test.ClimaUnitTests) ... ok
test_constructor_con_cualquier_id_deberia_incorporarlo_3_0
(clima_test.ClimaUnitTests) ... ok
test_constructor_con_cualquier_id_deberia_incorporarlo_4_10
(clima_test.ClimaUnitTests) ... ok
test_constructor_con_cualquier_id_deberia_incorporarlo_5_2172797
(clima_test.ClimaUnitTests) ... ok
test_constructor_con_cualquier_idioma_deberia_incorporarlo_1_None
(clima_test.ClimaUnitTests) ... ok
test_constructor_con_cualquier_idioma_deberia_incorporarlo_2_
(clima_test.ClimaUnitTests) ... ok
test_constructor_con_cualquier_idioma_deberia_incorporarlo_3_sp
(clima_test.ClimaUnitTests) ... ok
test_constructor_con_cualquier_pais_deberia_incorporarlo_1_None
(clima_test.ClimaUnitTests) ... ok
test_constructor_con_cualquier_pais_deberia_incorporarlo_2_
```

```
(clima_test.ClimaUnitTests) ... ok
test_constructor_con_cualquier_pais_deberia_incorporarlo_3_ar
(clima_test.ClimaUnitTests) ... ok
test_constructor_con_cualquier_pais_deberia_incorporarlo_4_uy
(clima_test.ClimaUnitTests) ... ok
test_constructor_con_fahrenheit_deberia_incorporarlo (clima_test.ClimaUnitTests)
... ok
test_constructor_con_kelvin_deberia_incorporarlo (clima_test.ClimaUnitTests) ...
ok
test_constructor_con_latitud_o_longitud_nulas_y_ninguna_otra_locacion_deberia_la
nzar_valueerror (clima_test.ClimaUnitTests) ... ok
test_constructor_con_none_deberia_lanzar_valueerror (clima_test.ClimaUnitTests)
... ok
test_constructor_con_un_url_servicio_invalido_deberia_lanzar_valueerror_1_None
(clima_test.ClimaUnitTests) ... ok
test_constructor_con_un_url_servicio_invalido_deberia_lanzar_valueerror_2_
(clima_test.ClimaUnitTests) ... ok
test_constructor_con_un_url_servicio_invalido_deberia_lanzar_valueerror_3_____
(clima_test.ClimaUnitTests) ... ok
test_constructor_con_una_apikey_en_blanco_deberia_lanzar_valueerror
(clima_test.ClimaUnitTests) ... ok
test_constructor_con_una_apikey_vacia_deberia_lanzar_valueerror
(clima_test.ClimaUnitTests) ... ok
test_constructor_con_una_apikey_valida_deberia_incorporarla
(clima_test.ClimaUnitTests) ... ok
test_obtener_con_el_objeto_creado_deberia_obtener_json
(clima_test.ClimaUnitTests) ... ok
test_url_con_datos_deberia_crear_url_1__2172797__None__None__None___None__None__
______fahrenheit____http___api_openweathermap_org_data_2_5_weather_appid_abcdefg
_id_2172797_units_imperial__ (clima_test.ClimaUnitTests) ... ok
test_url_con_datos_deberia_crear_url_2__None___Sydney____au___None___None__None_
___fr____kelvin____http___api_openweathermap_org_data_2_5_weather_appid_abcdefg_
q_Sydney_au_lang_fr__ (clima_test.ClimaUnitTests) ... ok
test_url_con_datos_deberia_crear_url_3__None__None__None__None___34_6037__58_381
6____es____celsius____http___api_openweathermap_org_data_2_5_weather_appid_abcde
fg_lat_34_6037_lon_58_3816_lang_es_units_metric__ (clima_test.ClimaUnitTests)
... ok
test_url_con_datos_deberia_crear_url_4__None__None__None___94040____None__None__
__ja____fahrenheit____http___api_openweathermap_org_data_2_5_weather_appid_abcde
fg_zip_94040_lang_ja_units_imperial__ (clima_test.ClimaUnitTests) ... ok
test_url_con_datos_deberia_crear_url_5__None__None___ar____94040____None__None__
__ja____fahrenheit____http___api_openweathermap_org_data_2_5_weather_appid_abcde
fg_zip_94040_ar_lang_ja_units_imperial__ (clima_test.ClimaUnitTests) ... ok
test_build_con_varios_valores_deberia_crearlo_con_ellos_1__2172797__None__None__
None__None__None___english____fahrenheit____http___api_openweathermap_org_data_1
_0_weather__ (climabuilder_test.ClimaBuilderUnitTests) ... ok
test_build_con_varios_valores_deberia_crearlo_con_ellos_2__None___Sydney____au__
_None__None__None___australian_english____kelvin____http___api_openweathermap_or
g_data_1_5_weather__ (climabuilder_test.ClimaBuilderUnitTests) ... ok
test_build_con_varios_valores_deberia_crearlo_con_ellos_3__None__None__None__Non
e__34_6037__58_3816___sp____celsius____http___api_openweathermap_org_data_2_0_we
ather__ (climabuilder_test.ClimaBuilderUnitTests) ... ok
test_build_con_varios_valores_deberia_crearlo_con_ellos_4__None__None__None___92
730___None__None___en____fahrenheit____http___api_openweathermap_org_data_2_5_we
ather__ (climabuilder_test.ClimaBuilderUnitTests) ... ok
test_con_apikey_con_cualquier_apikey_deberia_incorporarla_1_None
(climabuilder_test.ClimaBuilderUnitTests) ... ok
test_con_apikey_con_cualquier_apikey_deberia_incorporarla_2_
(climabuilder_test.ClimaBuilderUnitTests) ... ok
test_con_apikey_con_cualquier_apikey_deberia_incorporarla_3___
(climabuilder_test.ClimaBuilderUnitTests) ... ok
test_con_apikey_con_cualquier_apikey_deberia_incorporarla_4_abcdefg
(climabuilder_test.ClimaBuilderUnitTests) ... ok
```

```
test_con_ciudad_con_cualquier_ciudad_deberia_incorporarla_1_None
(climabuilder_test.ClimaBuilderUnitTests) ... ok
test_con_ciudad_con_cualquier_ciudad_deberia_incorporarla_2_
(climabuilder_test.ClimaBuilderUnitTests) ... ok
test_con_ciudad_con_cualquier_ciudad_deberia_incorporarla_3___
(climabuilder_test.ClimaBuilderUnitTests) ... ok
test_con_ciudad_con_cualquier_ciudad_deberia_incorporarla_4_abcdefg
(climabuilder_test.ClimaBuilderUnitTests) ... ok
test_con_id_con_cualquier_id_deberia_incorporarlo_1__33
(climabuilder_test.ClimaBuilderUnitTests) ... ok
test_con_id_con_cualquier_id_deberia_incorporarlo_2_0
(climabuilder_test.ClimaBuilderUnitTests) ... ok
test_con_id_con_cualquier_id_deberia_incorporarlo_3_1
(climabuilder_test.ClimaBuilderUnitTests) ... ok
test_con_id_con_cualquier_id_deberia_incorporarlo_4_10
(climabuilder_test.ClimaBuilderUnitTests) ... ok
test_con_id_con_cualquier_id_deberia_incorporarlo_5_1000
(climabuilder_test.ClimaBuilderUnitTests) ... ok
test_con_servicio_con_cualquier_url_deberia_incorporarlo_1_None
(climabuilder_test.ClimaBuilderUnitTests) ... ok
test_con_servicio_con_cualquier_url_deberia_incorporarlo_2_
(climabuilder_test.ClimaBuilderUnitTests) ... ok
test_con_servicio_con_cualquier_url_deberia_incorporarlo_3_____
(climabuilder_test.ClimaBuilderUnitTests) ... ok
test_con_servicio_con_cualquier_url_deberia_incorporarlo_4_http___www_example_co
m (climabuilder_test.ClimaBuilderUnitTests) ... ok
test_con_servicio_con_cualquier_url_deberia_incorporarlo_5_http___api_openweathe
rmap_org_data_2_5_weather (climabuilder_test.ClimaBuilderUnitTests) ... ok
test_constructor_con_creacion_deberia_tener_idioma_ciudad_grados_pais_url
(climabuilder_test.ClimaBuilderUnitTests) ... ok
test_en_celsius_con_celsius_flag_deberia_incorporarlo
(climabuilder_test.ClimaBuilderUnitTests) ... ok
test_en_codigo_postal_con_cualquier_codigo_postal_deberia_aceptarlo_1_None
(climabuilder_test.ClimaBuilderUnitTests) ... ok
test_en_codigo_postal_con_cualquier_codigo_postal_deberia_aceptarlo_2_
(climabuilder_test.ClimaBuilderUnitTests) ... ok
test_en_codigo_postal_con_cualquier_codigo_postal_deberia_aceptarlo_3___
(climabuilder_test.ClimaBuilderUnitTests) ... ok
test_en_codigo_postal_con_cualquier_codigo_postal_deberia_aceptarlo_4_abcdefg
(climabuilder_test.ClimaBuilderUnitTests) ... ok
test_en_coordenadas_con_cualquier_coordenada_deberia_incorporarla_1__0__0_
(climabuilder_test.ClimaBuilderUnitTests) ... ok
test_en_coordenadas_con_cualquier_coordenada_deberia_incorporarla_2___1__10_
(climabuilder_test.ClimaBuilderUnitTests) ... ok
test_en_coordenadas_con_cualquier_coordenada_deberia_incorporarla_3__13___1_
(climabuilder_test.ClimaBuilderUnitTests) ... ok
test_en_coordenadas_con_cualquier_coordenada_deberia_incorporarla_4___8___6_
(climabuilder_test.ClimaBuilderUnitTests) ... ok
test_en_fahrenheit_con_fahrenheit_flag_deberia_incorporarlo
(climabuilder_test.ClimaBuilderUnitTests) ... ok
test_en_idioma_con_cualquier_idioma_deberia_incorporarlo_1_
(climabuilder_test.ClimaBuilderUnitTests) ... ok
test_en_idioma_con_cualquier_idioma_deberia_incorporarlo_2___
(climabuilder_test.ClimaBuilderUnitTests) ... ok
test_en_idioma_con_cualquier_idioma_deberia_incorporarlo_3_es
(climabuilder_test.ClimaBuilderUnitTests) ... ok
test_en_idioma_con_cualquier_idioma_deberia_incorporarlo_4_english
(climabuilder_test.ClimaBuilderUnitTests) ... ok
test_en_kelvin_con_kelvin_flag_deberia_incorporarlo
(climabuilder_test.ClimaBuilderUnitTests) ... ok
test_en_pais_con_cualquier_pais_deberia_incorporarlo_1_None
(climabuilder_test.ClimaBuilderUnitTests) ... ok
test_en_pais_con_cualquier_pais_deberia_incorporarlo_2_
```

```
(climabuilder_test.ClimaBuilderUnitTests) ... ok
test_en_pais_con_cualquier_pais_deberia_incorporarlo_3___
(climabuilder_test.ClimaBuilderUnitTests) ... ok
test_en_pais_con_cualquier_pais_deberia_incorporarlo_4_abcdefg
(climabuilder_test.ClimaBuilderUnitTests) ... ok
test___str__con_json_valido_deberia_generar_oracion_1___kelvin____K__
(pronostico_test.PronosticoUnitTests) ... ok
test___str__con_json_valido_deberia_generar_oracion_2___imperial____F__
(pronostico_test.PronosticoUnitTests) ... ok
test___str__con_json_valido_deberia_generar_oracion_3___metric____C__
(pronostico_test.PronosticoUnitTests) ... ok
test_constructor_con_json_nulo_deberia_lanzar_valueerror
(pronostico_test.PronosticoUnitTests) ... ok
test_constructor_con_json_valido_deberia_incorporarlo
(pronostico_test.PronosticoUnitTests) ... ok
test_con_el_sistema_completo_deberia_funcionar (sistema_test.SistemaTests) ...
ok

----------------------------------------------------------------------
Ran 83 tests in 1.079s

OK
```

Conclusiones

El costo de corrección de un error aumenta a medida que se avanza en el desarrollo del software, siendo mucho más barato resolver un problema en la etapa de análisis que en la de implementación, y las pruebas unitarias ayudan a descubrirlos de una manera rápida. El tiempo que se invierte implementando las pruebas debe verse como una inversión y no como una pérdida. Es mucho más fácil recrear el código fuente a partir de un juego completo de pruebas unitarias que recrear las pruebas a partir del código fuente.

La metodología *TDD* puede también aplicarse a proyectos ya maduros: Primero se deben generar pruebas de integración que validen el funcionamiento correcto de los módulos entre sí, y a partir de ahí generar las pruebas unitarias por cada módulo utilizando las pruebas de integración para verificar que no se ha modificado el funcionamiento.

Bibliografía

- Software Testing - A Self-Teaching Introduction, Rajiv Chopra
- Design Patterns - Elements of Reusable Object-Oriented Software, Gamma, Helm, Johnson, Vlissides
- Test Driven Development: By Example, Kent Beck)

- Extreme Programming Explained: Embrance Change, Kent Beck
- Diseño Ágil con TDD, Carlos Blé Jurado