Mutatest:

Texto

Descripción generada automáticamenteInterfaz de usuario gráfica, Texto

Descripción generada automáticamente

Explicaciónes de sobrevivientes en siguiente plana (enunciado indica que se excluyen de la plana actual)

Test Smells:

Los nombres de los test no son descriptivos por lo que no se identifica rápidamente ni claramente que esta testeando cada test. Además el código de los test tampoco ayuda a entenderlo por lo que los tests no cumplen los criterios de calidad FIRST.

Además se cumple el test smell Assertion Roullette donde hay multiples assertions en un solo test lo que dificulta entender que fallo en caso de no pasar un test. Un ejemplo de esto se puede ver en test\_case\_1 de display\_numbers:

-assert var\_0 == "513.06382"

-var\_1 = number\_display\_1.reset()

-assert number\_display\_1.value == 0

-var\_2 = number\_display\_1.reset()

-var\_3 = number\_display\_1.reset()

-var\_4 = number\_display\_1.clone()

-assert var\_4.value == 0

-var\_5 = number\_display\_1.invariant()

-assert var\_5 is True

Se puede ver claramente multiples assert y no se entiende que se esta testeando de verdad.

También existen casos de test donde no se puede apreciar ningún assertion por lo que no retorna un booleano para poder ver si se paso o no el test. Esto ocurre por ejemplo en test\_case\_0 de clock\_display.

Por otro lado también existe algunos test smells que califican como Eager Tests al hacer operaciones innecesarias dentro del test haciendo que se demoren más en ejecutar y haciendo más difícil el entender el código.

Por ejemplo se puede ver esto en los test de display number en el test\_case\_1:  
-dict\_0 = {int\_0: int\_0, int\_0: int\_0, bool\_0: int\_0}

-number\_display\_0 = module\_0.NumberDisplay(dict\_0, dict\_0)

Después nunca se utiliza number\_display\_0 por lo que estas líneas solo complejizan la lectura del código.

Por otro lado también hay un test smell que es código duplicado, por ejemplo, se repite la siguiente línea:

-clock\_display\_0 = module\_0.ClockDisplay(list\_0)

Esto se repite en la línea 18, 23, 60 y 78.

Además esta el test smell For Testers Only el cual se refiere a que algunos tests están hechos con detalles muy técnicos en vez de enfocarse en el comportamiento. Por ejemplo, en la línea 9 de los test de clock display se utilizan bytes cuando se podrían utilizar una estructura de datos más fácil de entender su funcionamiento como una lista.

Por ultimo esta el test smell indirect testing donde se testean funcionalidades con un setup muy complejo y sin tanta relación a la funcionalidad de verdad. Por ejemplo, el test case 4 de number display instancia el number display con tuplas vacias para después solo verificar si el reset se hizo correctamente. Es decir, fue totalmente innecesario instanciarlo con tuplas vacias, el setup fue complejo, no aporta en nada y dificulta la lectura.

Explicaciones Sobrevivientes:   
1) y 2) : Los primeros dos mutantes que sobrevivieron son aquellos que cambian el return de \_\_init\_\_ de None a True o False. En cualquier caso, el return de esta función no aporta información para el flujo del código, por lo que no significa un mutante que pueda afectar el correcto funcionamiento del mismo.

3) : El tercer mutante cambia el método increase de:

def increase(self):

        self.value = (self.value + self.limit + 1) % self.limit

        return self.value == 0

a

def increase(self):

        self.value = (self.value % self.limit + 1) % self.limit

        return self.value == 0

Matematicamente, el resultado sin importar los valores será siempre el mismo. Esto debido a que cualquier valor entre 0 y self.limit – 1 va a entregar el mismo valor cuando se aplique el % en el mutante, por lo que sumarle 1 y aplicar modulo nuevamente mantiene la magnitud y entrega siempre el mismo valor.

4) : El cuarto mutante cambia

def increase(self):

        self.value = (self.value + self.limit + 1) % self.limit

        return self.value == 0

a

def increase(self):

        self.value = (self.value + self.limit + 1) % self.limit

        return self.value <= 0

Por definición, el operador módulo (%) nunca entrega valores negativos. Por lo anterior, la condición entre “==” y “<=” se aplica de la misma forma, ya que al no entregar nunca valores negativos, por mas que se tenga un “<=”, solamente se aplicará la comparación “=”, manteniendo el flujo deseado del código.