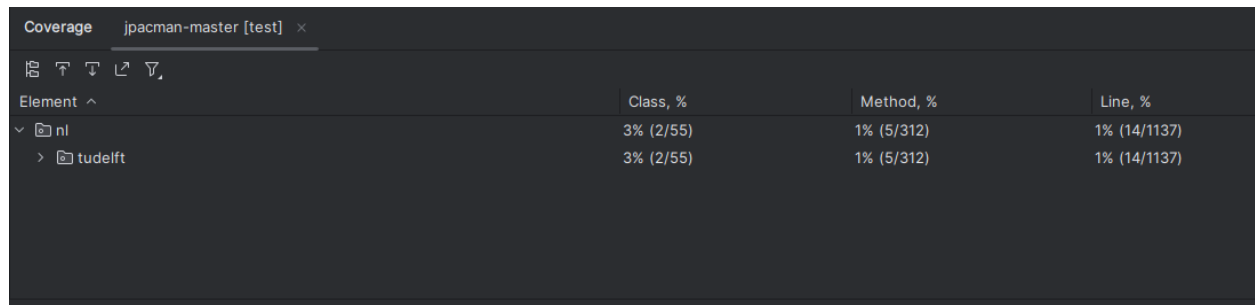


Task 1:

Is this coverage good enough?



The screenshot shows a code coverage tool interface. At the top, it says 'Coverage' and 'jpacman-master [test]'. Below this is a table with columns: 'Element', 'Class, %', 'Method, %', and 'Line, %'. The table shows two elements: 'nl' and 'tudelft'. Both have 3% class coverage and 1% method and line coverage.

Element	Class, %	Method, %	Line, %
nl	3% (2/55)	1% (5/312)	1% (14/1137)
> tudelft	3% (2/55)	1% (5/312)	1% (14/1137)

This coverage is way too low at 3% class coverage and 1% method and line coverage. The Senior Design Project has a required coverage of at least 70% so with coverage this low it would not be considered adequate.
(Task 2 on next page)

Task 2:

Coverage jpacman-master [test] ×			
⌵ ⌴ ⌶ ⌷ ⌸			
Element ^	Class, %	Method, %	Line, %
▼ 📁 nl	16% (9/55)	9% (31/312)	8% (101/1153)
> 📁 tudelft	16% (9/55)	9% (31/312)	8% (101/1153)

```
//declare it is a test
@Test
void testIsAlive() {
    // there is no PacMan sprite constructor so it inherents form SpriteStore
    // SpriteStore has a constructor with no parameters so initialization is as
follows public SpriteStore()
    PacManSprites newSprite = new PacManSprites();
    // PlayerFactory constructor is as follows public
PlayerFactory(PacManSprites spriteStore), hence
    PlayerFactory newPlayerFactory = new PlayerFactory(newSprite);

    //now to create the player we want to use the function in player factory
[public Player createPacMan()]
    Player newPlayer = newPlayerFactory.createPacMan();

    //now with a new Player we can test the function isAlive() by setting alive
to true and testing it
    newPlayer.setAlive(true);
    assertThat(newPlayer.isAlive()).isEqualTo(true);
}
```

After adding in the first unit test, which was `isAlive()` we see an increase to 16% class coverage and 9% method coverage. My method for execution is also added above and everything is functionally the same as the professor's example code. The main difference is the professor has declared the placeholder sprite, factory, and new player as private variables for the class so they can be used in further unit tests. This can be referred to later on as the refactoring phase in TDD but when working on the senior design project it's best practice to follow the refactoring method for better scalability.

Task 2.1

Now we begin to implement our own methods and see their effect on coverage.

Method 1: GameFactory.java#GameFactory()

Fully Qualified Name: src/main/java/nl/tudelft/jpacman/game/GameFactory.java

```
@Test
void testGameFactory()
{
    //similar to the example for class we need to instantiate a PacManSprites
    and a PlayerFactory to
    //test the constructor as it takes in a Player factory as a parameter
    PacManSprites spriteHolder = new PacManSprites();
    PlayerFactory playerFactoryHolder = new PlayerFactory(spriteHolder);
    //now we can test the constructor
    //implementation: create a GameFactory with playerFactoryHolder and test to
    make sure that GameFactory.PlayerFactory
    //is equal to factoryHolder, if it is then the constructor works as
    intended
    GameFactory gameFactoryHolder = new GameFactory(playerFactoryHolder);
    //expected value being that they are equal
    assertThat(gameFactoryHolder.getPlayerFactory()).isEqualTo(playerFactory
    Holder);
}
```

Coverage jpacman-master [test] ×			
🔍 ⬆ ⬇ ↗ 🔍			
Element ^	Class, %	Method, %	Line, %
▼ 📁 nl	18% (10/55)	10% (33/312)	9% (105/1161)
> 📁 tudelft	18% (10/55)	10% (33/312)	9% (105/1161)

For the first method, I decided to go easy mode and pick to test the constructor for the gameFactory class. This problem was easy because we only needed to generate a gameFactory variable then we could test the constructor with a placeholder to see if a valid gameFactory was generated. Ideally, each constructor for a given class could be done in one unit test by first using the default constructor and moving on to the copy constructor and so on but because gameFactory only had one we could test it as such. The code coverage for Class coverage increased by 2% which seems insignificant but when we consider that we only increased the class coverage from 9 classes to 10 it is understandable that the jump was small. The jump in coverage does verify that the unit test works and is configured correctly with IntelliJ.

Method 2: DefaultPointCalculator.java#consumedAPellet()

Fully Qualified Name: src/main/java/nl/tudelft/jpacman/points/DefaultPointCalculatorTest.java

```
//we will test consumedAPellet(Player player, Pellet pellet) which adds the
pellet
//value to the player value. The Idea is to create a Player and pellet with
known values so
//we can then run the function consumedAPellet and check to see if the
function works as anticipated
@Test
void testConsumedPellet()
{
    //similar to the example we must create a player by first creating the
PacManSprites
    PacManSprites spritePlaceholder = new PacManSprites();
    //now generate player factory to create player
    PlayerFactory playerFactoryHolder = new PlayerFactory(spritePlaceholder);
    Player newPlayer = playerFactoryHolder.createPacMan();
    //when a player is instantiated they start at 0 points and now we have a
valid starting point

    //now we generate the pellet with a ImageSprite that uses a null
placeholder image
    Image imgPlaceholder = null;
    ImageSprite imageSpritePlaceholder = new ImageSprite(imgPlaceholder);
    Pellet pelletPlaceholder = new Pellet(5, imageSpritePlaceholder);

    //now we have a player with 0 points a pellet worth 5 points so if we use
the
    //consumeAPellet function within a DefaultPointCalculator then the player
points should
    //increment and be equal to 5
    DefaultPointCalculator pointCalculator = new DefaultPointCalculator();
    pointCalculator.consumedAPellet(newPlayer, pelletPlaceholder);
    assertThat(newPlayer.getScore()).isEqualTo(pelletPlaceholder.getValue());
}
```

With an easy method knocked out I felt like it was only fair to pick my next method with at least one parameter. This leads me to consumedAPellot which takes in pellot and a player and updates the player's score accordingly. The main struggle in this example was figuring out valid parameters for the Pellet in the function because the requirement was an Image sprite which in return also needed an Image to be made. Thankfully I was able to leave the image null but this still resulted in at least 7 variable placeholders before I could finally call in the function that I wanted to test. This was great practice at calling in maneuvering through the bigger code base in an attempt to find the constructors I needed. Below we see an increase of code coverage by 3% with 2 new classes being covered by this test.

Coverage jpacman-master [test] x			
Element ^	Class, %	Method, %	Line, %
nl	21% (12/55)	12% (38/312)	9% (116/1164)
> tudelft	21% (12/55)	12% (38/312)	9% (116/1164)

Method 3: SinglePlayerGame.java#getLevel()

SinglePlayerGame.java#getLevel()

Fully Qualified Name: src/main/java/nl/tudelft/jpacman/game/SinglePlayerGameTest.java

```
//SinglePlayer Game test is a class created to test the Single Player Game
class
public class SinglePlayerGameTest {
    //we create a test that checks to make sure that getLevel() function is
    working as intended
    @Test
    void testGetLevel()
    {
        //the GetLevel function is located within SinglePlayerGame so first we
        generate a variable with the constructor
        //SinglePlayerGame(Player player, Level level, PointCalculator
        pointCalculator)
        //first we generate a player place holder, again starting off like the
        example
        PacManSprites spriteHolder = new PacManSprites();
        PlayerFactory playerFactoryHolder = new PlayerFactory(spriteHolder);
        Player newPlayer = playerFactoryHolder.createPacMan();

        //now generate a level with the level constructor
        // Level(Board board, List<Ghost> ghosts, List<Square> startPositions,
        CollisionMap collisionMap)
        //step 1: declare a valid board, ghosts, and startPositions for the
        constructor to work
        //Board(Square[][] grid) will require a Square[][] grid with no tiles
        being null
        //to generate this we will use a boardFactory
        BoardFactory boardFactoryHolder = new BoardFactory(spriteHolder);
        //we can generate a valid square with board factory and use that square
        to generate a valid grid
        Square squareHolder = boardFactoryHolder.createGround();
        Square[][] gridHolder = new Square[][]{{squareHolder}};
        //Now using the gridHolder grid we can generate a valid board with
        board factory
        Board boardHolder = boardFactoryHolder.createBoard(gridHolder);
```

```

        //now create a valid list of ghost by first making a valid ghost for
the list
        //because ghost is an abstract class we need to fill the ghost list
with one type of ghost
        //Blinky(Map<Direction, Sprite> spriteMap), we leave the spriteMap as
null

        Blinky emptyBlinky = new Blinky(null);
        List<Ghost> spookyList = new ArrayList<Ghost>();
        spookyList.add(emptyBlinky);

        //next we create a List<Square> that is not null
        //we can use the squareHolder before to ensure the list is not null
        List<Square> squareListHolder = new ArrayList<>();
        squareListHolder.add(squareHolder);

        //for the final variable we can use null since it is not checked so now
we can create a valid level variable
        Level newLevel = new Level(boardHolder, spookyList, squareListHolder,
null);

        //finally we can move onto the PointCaclulator which is the final
variable for the SinglePlayerGame constructor
        DefaultPointCalculator newCalculatorHolder = new
DefaultPointCalculator();

        //now we can generate the SinglePlayerGame with the initialized
variables
        SinglePlayerGame newSinglePlayerGame = new SinglePlayerGame(newPlayer,
newLevel, newCalculatorHolder);
        //now test if the get level function is working properly
        assertThat(newSinglePlayerGame.getLevel()).isEqualTo(newLevel);
    }

```

Now we get to the horror story that is `getLevel()`. Initially, I assumed this function was returning an integer and as such entered this challenge with full confidence that I could handle it. 15 declarations and 2 days later I finally became victorious after realizing some class constructors check if the variable is passed a null parameter and as such, just like with the image sprite above, I had to go another step back and make a temporary variable for my temporary variable. All in all this example helped me solidify my confidence with running unit tests on a larger code base. While this example was most frustrating I will admit it again helped me practice digging through constructor after constructor to get down to the fundamental elements needed to run the unit test and I believe this will not only be useful but required when it comes time to write unit tests for the project. Below we see a notable code coverage increase of 17% which comes from covering 9 more class tests making our final code coverage right around 38%. Although it is not our goal for the project, seeing the starting point at 3% I can understand why its easier to maintain a code base with high coverage as appose to start from scratch.

Coverage jpacman-master [test] ×			
<div> <div></div> <div></div> <div></div> <div></div> <div></div> </div>			
Element ^	Class, %	Method, %	Line, %
▼ nl	38% (21/55)	21% (68/312)	19% (224/1168)
> tudelft	38% (21/55)	21% (68/312)	19% (224/1168)

Task 3: Jpacman.html

jpacman

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
nl.tudelft.jpacman.level	<div><div></div></div>	68%	<div><div></div></div>	58%	72	155	102	344	21	69	4	12
nl.tudelft.jpacman.npc.ghost	<div><div></div></div>	71%	<div><div></div></div>	55%	56	105	43	181	5	34	0	8
nl.tudelft.jpacman.ui	<div><div></div></div>	77%	<div><div></div></div>	47%	54	86	21	144	7	31	0	6
default	<div><div></div></div>	0%	<div><div></div></div>	0%	12	12	21	21	5	5	1	1
nl.tudelft.jpacman.board	<div><div></div></div>	86%	<div><div></div></div>	58%	44	93	2	110	0	40	0	7
nl.tudelft.jpacman.sprite	<div><div></div></div>	86%	<div><div></div></div>	59%	30	70	11	113	5	38	0	5
nl.tudelft.jpacman	<div><div></div></div>	69%	<div><div></div></div>	25%	12	30	18	52	6	24	1	2
nl.tudelft.jpacman.points	<div><div></div></div>	60%	<div><div></div></div>	75%	1	11	5	21	0	9	0	2
nl.tudelft.jpacman.game	<div><div></div></div>	89%	<div><div></div></div>	60%	9	24	3	45	1	14	0	3
nl.tudelft.jpacman.npc	<div><div></div></div>	100%	<div><div></div></div>	n/a	0	4	0	8	0	4	0	1
Total	1,203 of 4,694	74%	291 of 637	54%	290	590	226	1,039	50	268	6	47

Task Questions:

- Are the coverage results from JaCoCo similar to the ones you got from IntelliJ in the last task? Why so or why not?
 - The results in the JaCoCo are similar to IntelliJ but also there are discrepancies in the report. The first to note is the line count in IntelliJ is 1168 and in JaCoCo there are 1039 leaving around 129 line discrepancy while the line coverage for both is only off by 2 lines since in IntelliJ it is 224 and in JaCoCo it is 226. This discrepancy continues throughout the other values like in methods the total number counted is off by 54 methods but the total methods tested is only off by 18.
- Did you find helpful the source code visualization from JaCoCo on uncovered branches?
 - Yes, while IntelliJ breaks it down to the java files and tells the user how many methods are in each file, JaCoCo goes one step further and list each method name making it easier to find out which method is missing coverage. Below is an example with the SinglePlayerGame file broken down by method.

jpacman > nl.tudelft.jpacman.game > SinglePlayerGame

SinglePlayerGame

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods
SinglePlayerGame(Player, Level, PointCalculator)	<div><div></div></div>	73%	<div><div></div></div>	50%	4	5	0	7	0	1
static {...}	<div><div></div></div>	75%	<div><div></div></div>	50%	1	2	0	1	0	1
getPlayers()	<div><div></div></div>	100%	<div><div></div></div>	n/a	0	1	0	1	0	1
getLevel()	<div><div></div></div>	100%	<div><div></div></div>	n/a	0	1	0	1	0	1
Total	10 of 45	77%	5 of 10	50%	5	9	0	10	0	4

3. Which visualization did you prefer and why? IntelliJ's coverage window or JaCoCo's report?
- a. While IntelliJ's coverage window offered ease of use and a fairly detailed breakdown I would have to say that I prefer JaCoCo's report style. This is again for the reason stated above, the break down is simply easier to read and by extension makes it easier to maintain a higher coverage. Notice in the IntelliJ breakdown we are missing 10% line coverage in singleplayergame.java and finding that would be tedious. We see the full break down in JaCoCo which pinpoints the missing branches and instructions we aren't duplicating test accidentally.

Coverage jpacman-master [test] x			
Element	Class, %	Method, %	Line, %
> board	60% (6/10)	45% (24/53)	49% (71/144)
> fuzzer	0% (0/1)	0% (0/6)	0% (0/32)
> game	100% (3/3)	42% (6/14)	40% (18/45)
Game	100% (1/1)	14% (1/7)	16% (5/30)
GameFactory	100% (1/1)	66% (2/3)	80% (4/5)
SinglePlayerGame	100% (1/1)	75% (3/4)	90% (9/10)
> integration	0% (0/1)	0% (0/4)	0% (0/6)

Task 4: NoseTest

The following code snippets will show the test for the lines of code that still need coverage. The lines are as follows 35-36, 46-49, 53-55, 75-76.

Lines 35-36

```
#test for 35-36
    #the code is for the from_dict function
def test_from_dict(self):
    #generate a valid dict using the method in the last test
    data = ACCOUNT_DATA[self.rand] #generate a random entry
    account = Account(**data)      # cast that entry into an account type
    result = account.to_dict()      # generate a serialized dictionary
    # now adjust an attribute within that dictionary and test from_dict
    result["phone_number"] = "999-999-9999"
    account.from_dict(result)
    #check to ensure value is updated within the account
    self.assertEqual(account.phone_number, result["phone_number"])

    #once created coverage has increased to the following | 46    9    80%|
```

The test_from_dict is testing the from_dict function located on lines 35-36. For this test we will simply update a value in the dictionary and compare the result from_dict to the dictionary to ensure the update was copied over.

Lines 46, 47, 49

```
#test for 46, 47, 49
    #the code in this block is for the update() function that updates an
account int the data base
def test_update_success(self):
    #again start with generating a valid account to test the funtion on
    data = ACCOUNT_DATA[self.rand] #generate a random entry
    account = Account(**data)      # cast that entry into an account type
    #create an account in the data base
    account.create()
    #now grab a copy of the data base
    resultsinit = account.to_dict()
    result = account.to_dict()      # generate a serialized dictionary
    #now update the database and call the update function
    result["phone_number"] = "999-999-9999"
    account.from_dict(result)
    #update the account database and check to see that the updated value is
true
    account.update()
    self.assertEqual(account.phone_number, result["phone_number"])
```

```

        #also verify that the initial results are different from the updated
results
        self.assertNotEqual(resultsinit["phone_number"],result["phone_number"])
        #once created coverage has increased to the following | 46    6    87%|

```

To cover lines 46, 47, and 49 we perform a similar test but we will then use the update function to make sure that the database is updated and compare those results with known results to both test what the value was before the test (and ensure it updated) to the value it is at the end of the test.

Lines 53-55

```

#lines 53-59 correlate with the function delete() used to remove an account
from the database
def test_delete(self):
    #generate a valid dict using the method in the last test
    data = ACCOUNT_DATA[self.rand] #generate a random entry
    account = Account(**data)      # cast that entry into an account type000
    #now generate the database
    account.create()
    #now delete it all and check to make sure it is deleted
    account.delete()
    #to test this we compare data(the initial account) to the new account and
see if they are different
    self.assertNotEqual(account, Account(**data))

    #once created coverage has increased to the following | 46        3    93%|

```

To test the delete function on lines 53-55 we will simply create a database and delete it. To check if the database was deleted we can use a placeholder and ensure they no longer match

Lines 75-76

```

#lines 75-76 are located with the function find(cls, account_id: int)
def test_find(self):
    #generate a valid dict using the method in the last test
    data = ACCOUNT_DATA[self.rand] #generate a random entry
    account = Account(**data)      # cast that entry into an account type
    #generate the account
    account.create()
    #use find and the account id to pass the account to a temporary variable
    temp = account.find(account.id)
    #now test if the temporary variable and the original account match
    self.assertEqual(account, temp)

    #once created coverage has increased to the following | 46        1    98%|

```

Lines 75 and 76 cover the find function which takes in an account id and returns the account entry. We can simply use the account.id to test the function and ensure that the account returned is the same account as the original account.

Lines 48, Special Case

```
#48
#notably line 48 is a special if case that is set off by calling the
function with an empty id field which sets off an error
def test_update_failure(self):
    #rather than initialize any data we simply call the function with an
empty id field
    data = ACCOUNT_DATA[self.rand] #generate a random entry
    account = Account(**data)      # cast that entry into an account type
    #initialize account
    account.create()
    #now make id empty to set off the final branch
    account.id = ""
    account.update()
    self.assertEqual(account.id, "")

#once created coverage has increased to the following | 46    0 100%|
```

Now we run into the issue of line 48. The special case here is set off by having an empty ID field and attempting to update the account. This causes one branch of code to throw an exception and we need to trigger this to achieve 100% code coverage. To do this I chose to change the id field by hand and trigger the update function to ensure that the exception would be thrown which is what occurred.

Task 5: Test Driven Development

test_update_a_counter

Red Phase:

```
#now we will create out own test called test_update_a_counter(self)
def test_update_a_counter(self):
    #result should test the update method, first create a counter
    counter = app.test_client()
    basename = '/counters/foobar'
    result = counter.post(basename)
    #now test to make sure it was actually created
    self.assertEqual(result.status_code, status.HTTP_201_CREATED)
    #we already store a baseline for the counter
    base = result.get_json()['foobar']
    #update the counter
    result = counter.put(basename)
    #now first test that it updated and assert they arent equal
```

```

        self.assertEqual(result.status_code, status.HTTP_200_OK)
        #to check the counter value is one more than the baseline we need to
pull the new baseline and compare
        count = result.get_json()['foobar']
        self.assertEqual(base+1, count)

```

In order to test this function we must first start like previous examples by creating a counter and posting a name for it. We first check to ensure that the counter is created then we move on to grabbing the value of that counter. Once we get it we update the counter and compare the base counter value with the new updated counter value. Ideally the Put function will handle any issues as far as running into exceptions so for now we consider this failed state a successful red Phase:

```

Counter tests
- It should create a counter
- It should return an error for duplicates
- update a counter (FAILED)

=====
FAIL: test_update_a_counter (test_counter.CounterTest.test_update_a_counter)
-----
Traceback (most recent call last):
  File "D:\Backups\2023Backups\FALL2023\CS472\tdd-main\tests\test_counter.py", line 57, in test_update_a_co
unter

```

Green Phase:

```

#create a route for method PUT
@app.route('/counters/<name>', methods=['PUT'])
def update_counter(name):
    #increment the counter by 1
    COUNTERS[name] = COUNTERS[name] + 1
    #return the new counter and a 200_ok return code
    return {name: COUNTERS[name]}, status.HTTP_200_OK

```

To achieve the update_counter method we simply increment the counter count by 1 and add the new counter name to the count. Once this is done we return the success code and find the test now passes.

```

Counter tests
- It should create a counter
- It should return an error for duplicates
- update a counter

Name           Stmts   Miss  Cover   Missing
-----
src\counter.py   15      0   100%
src\status.py     6      0   100%
-----
TOTAL             21      0   100%
-----
Ran 3 tests in 0.283s

```

Refactor Phase:

```
#refactored code from prof
def setUp(self):
    self.client = app.test_client()
```

For this phase, we can mostly replace the counter in exchange for the client in the refactored code from the professor and save ourselves a variable. Also Seeing how 'foobar' is spelled 3 times perhaps better code practice would see this value being replaced with a temporary shared variable that can be updated with each test.

test_read_a_counter

Red Phase:

```
#for the read test we will create a counter and test the read function
def test_read_a_counter(self):
    #create a test counter
    counter = app.test_client()
    #try to get counter name but no counter is there
    result = counter.get('/counters/supersecretsite')
    self.assertEqual(result.status_code, status.HTTP_404_NOT_FOUND)
    #now add the counter to the list of counters
    counter.post('/counters/supersecretsite')
    #now counter is made so test get success code
    result = counter.get('/counters/supersecretsite')
    self.assertEqual(result.status_code, status.HTTP_200_OK)
```

Thankfully read_a_counter function was a lot easier and only required the 2 scenarios to be considered. We try to read a counter and do not find one or we find it and we get a success code. The first scenario fails without first posting the counter and the second scenario returns the correct return code so we again assume this red phase is complete.

```
Counter tests
- It should create a counter
- It should return an error for duplicates
- read a counter (FAILED)
- update a counter

=====
FAIL: test_read_a_counter (test_counter.CounterTest.test_read_a_counter)
-----
Traceback (most recent call last):
  File "D:\Backups\2023Backups\FALL2023\CS472\tdd-main\tests\test_counter.py", line 65, in test_read_a_counter
    self.assertEqual(result.status_code, status.HTTP_404_NOT_FOUND)
AssertionError: 405 != 404
```

Green Phase:

```
#create a route for method GET
@app.route('/counters/<name>', methods=['GET'])
def get_counter(name):
```

```

#check to see if the name does not exist and return 404 if not
if name not in COUNTERS:
    return {"Message":f"Counter {name} does not exists"},
status.HTTP_404_NOT_FOUND
#if the name is in the counters list then return a status code okay
return {name: COUNTERS[name]}, status.HTTP_200_OK

```

For the function we must replace the PUT method with GET and the other parameter for app.route stays the same. Now this function is as simple as returning an error if the counter is not in the list or returning the success code if the counter is in the list.

```

Counter tests
- It should create a counter
- It should return an error for duplicates
- read a counter
- update a counter

Name                Stmts   Miss  Cover   Missing
-----
src\counter.py       20      0   100%
src\status.py        6      0   100%
-----
TOTAL                26      0   100%
-----

Ran 4 tests in 0.297s

```

Refactor Phase:

```

#refactored code from prof
def setUp(self):
    self.client = app.test_client()
    name = 'CounterName'

```

Although I did not use it in the original implementation, we could use self.client and name in this function as a replacement for the counter variable. Using Name we could also replace '/counters/supersecretsite' with the shared variable so that we have easier-to-follow code seeing as it is used 3 times within 5 lines of code.