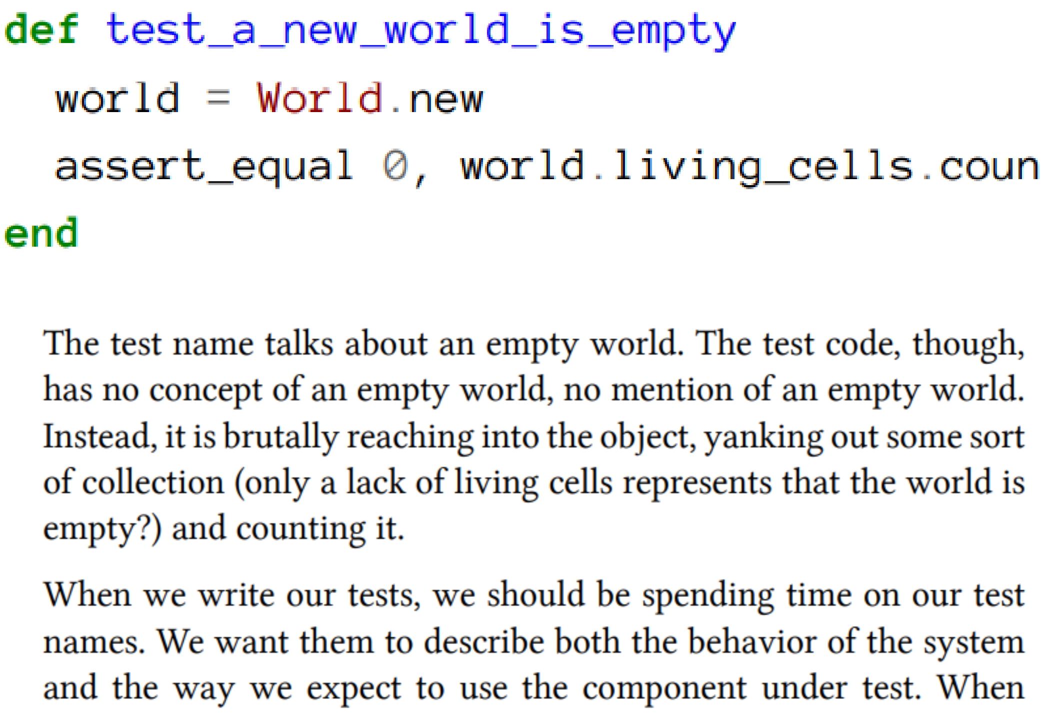
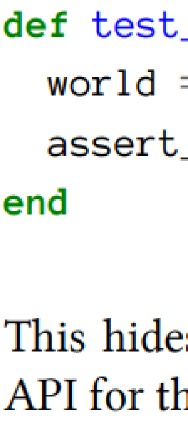


```
def test_a_new_world_is_empty
  world = World.new
  assert_equal 0, world.living_cells.count
end
```

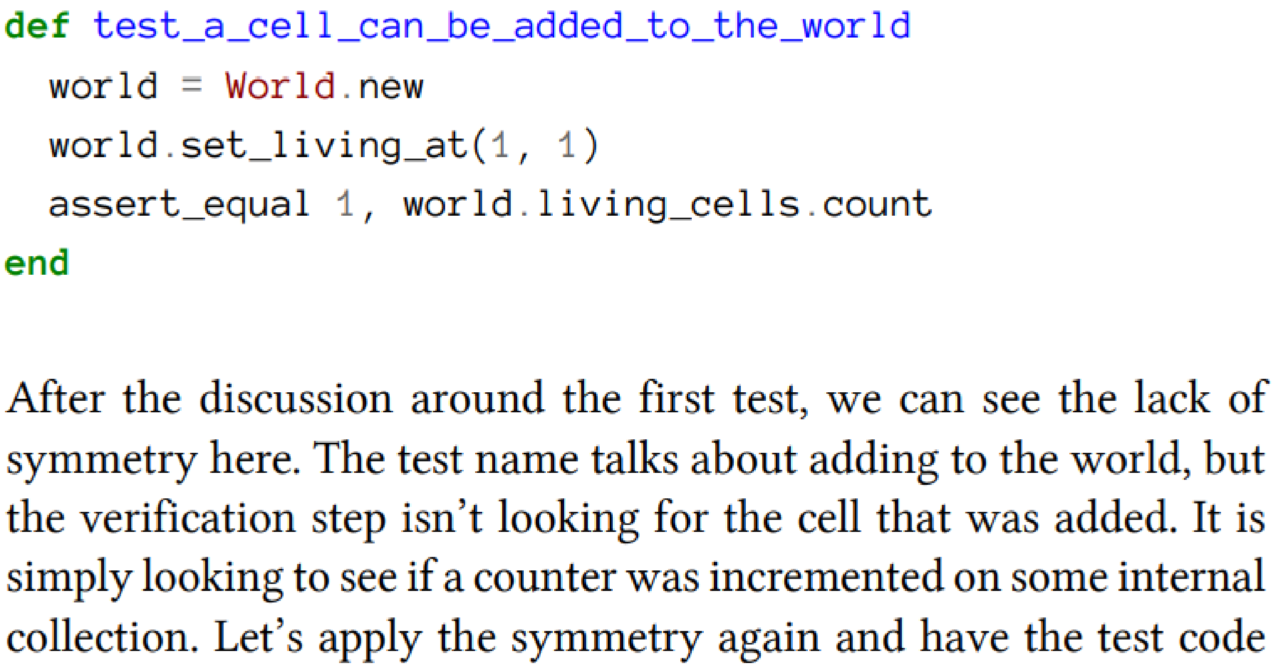
The test name talks about an empty world. The test code, though, has no concept of an empty world, no mention of an empty world. Instead, it is brutally reaching into the object, yanking out some sort of collection (only a lack of living cells represents that the world is empty?) and counting it.

When we write our tests, we should be spending time on our test names. We want them to describe both the behavior of the system and the way we expect to use the component under test. When



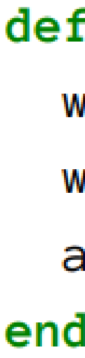
```
def test_
  world =
  assert_
end
```

This hide
API for th



```
def test_a_cell_can_be_added_to_the_world
  world = World.new
  world.set_living_at(1, 1)
  assert_equal 1, world.living_cells.count
end
```

After the discussion around the first test, we can see the lack of symmetry here. The test name talks about adding to the world, but the verification step isn't looking for the cell that was added. It is simply looking to see if a counter was incremented on some internal collection. Let's apply the symmetry again and have the test code



```
def
  w
  w
  a
end
```

Test will fail if coodinate system is changed.

```
_a_new_world_is_empty
= World.new
_true world.empty?
```

s the internals of the object, while building up a usable
he rest of the system to consume.

```
 test_a_cell_can_be_added_to_the_world
orld = World.new
orld.set_living_at(1, 1)
ssert_true world.alive_at?(1, 1)
```

Focusing on the s
under tests is a su
design influence
an important one
cycle, take a mom
you say you are t

```
def test_after_a
  world = World
  world.set_livi
  assert_false w
end
```

We also could ad
living_at.

ymmetry between a good test name and the code
ubtle design technique. It is definitely not the only
that our tests can have on our code, but it can be
. So, next time you are flying through your TDD
ent to make sure that you are actually testing what
esting.

```
adding_a_cell_the_world_is_not_empty
.new
ing_at(1, 1)
world.empty?
```

d a test around the `empty?` method using `set_-`

```
def test_world_is_not_empty_after_adding_a_cell
  world = World.empty
  world.set_living_at(Location.new(1,1))
  assert_false world.empty?
end
```

```
def te
  worl
  worl
  asse
end
```

```
st_world_is_not_empty_after_adding_a_cell
d = World.empty
d.set_living_at(double(:location_of_cell))
rt_false world.empty?
```

```
def test_world_
  world = World
  world.set_liv
  assert_false 
end
```

```
is_not_empty_after_adding_a_cell
.empty
ing_at(Object.new)
world.empty?
```

```
class Cell
  # ...
  def alive_in
    if state =
      stable_n
    elsif stat
      genetica
    end
  end
end
```

```
n_next_generation?
== ALIVE
neighborhood?
te == DEAD
ally_fertile_neighborhood?
```

```
class LivingCell
  def alive_in_next_generation?
    # neighbor_count == 2 || neig
    stable_neighborhood?
  end
end
class DeadCell
  def alive_in_next_generation?
    # neighbor_count == 3
    genetically_fertile_neighborh
  end
end
```

```
class ZombieCell
  def alive_in_next_generat
    # new, possibly more co
  end
end
```

```
hbor_count == 3

ood?

ion?
mplex rules
```

```ruby
class LivingCell
  def stays_alive?
    neighbor_count == 2 || neighbor_count ==
  end
end
class DeadCell
  def comes_to_life?
    neighbor_count == 3
  end
end
```

3