

Abstract geometric lines in the top-left corner of the slide, consisting of several overlapping, irregular polygons and lines that create a complex, layered effect.

TEST-DRIVEN DEVELOPMENT FOR ACADEMIC SIMULATION

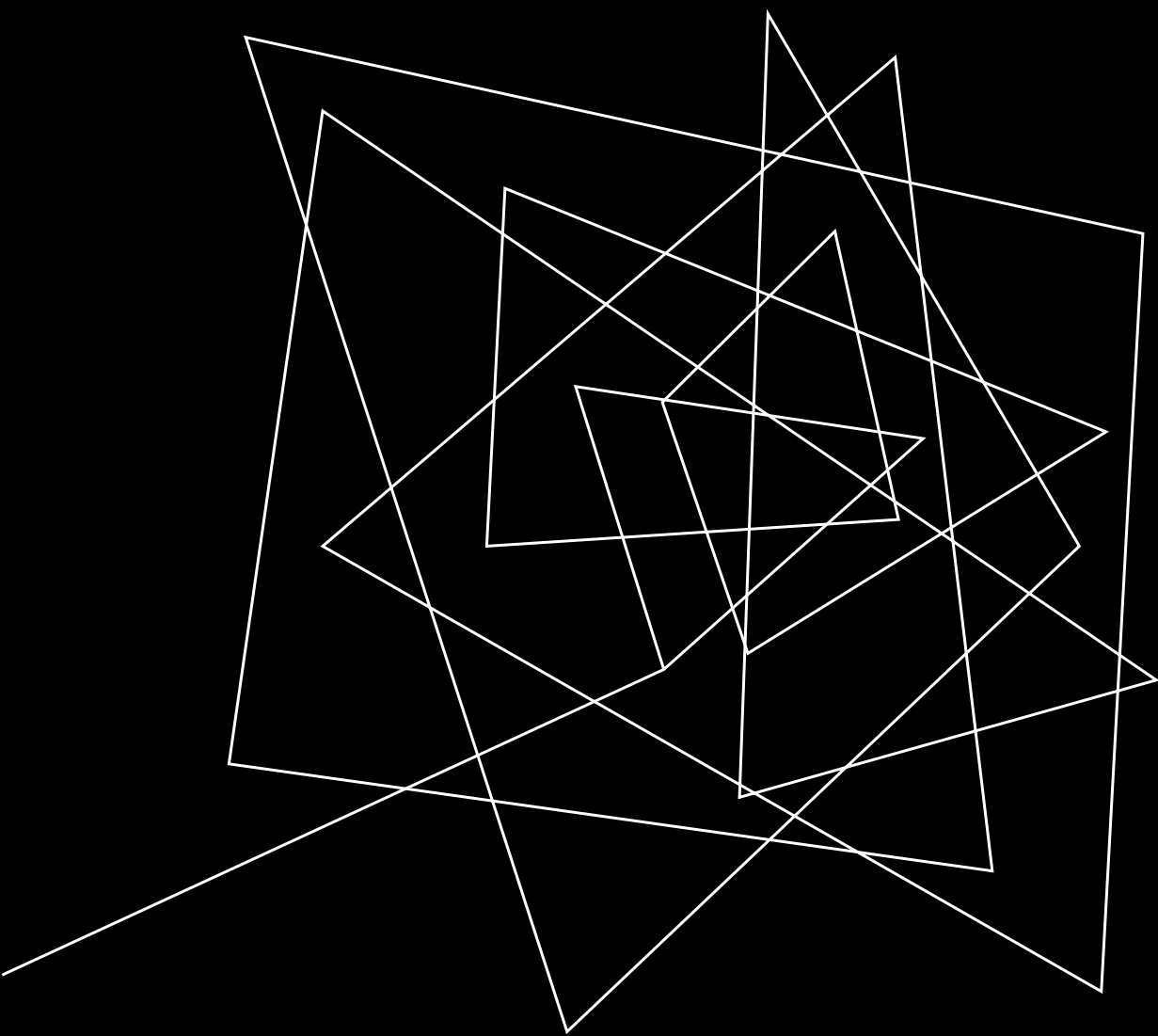
Paul A. Monderkamp

AGENDA

brief presentation

- Introduction
 1. why/what is test in general
 2. what are unit-tests
 3. what is test-driven development
- What is test-driven development
 1. workflow
 2. example
- 4 reasons for test-driven development

workshop



Introduction

1. why/what is test in general
2. what are unit-tests
3. what is test-driven development

INTRODUCTION—WHY TEST IN GENERAL?



**Errors may
lead to
catastrophe**

INTRODUCTION—WHY TEST IN GENERAL?

The New York Times



Airline Blames Bad Software in San Francisco Crash

Cause	Airworthiness revoked after recurring flight control failure
Budget	<ul style="list-style-type: none">• direct costs: US\$20 billion^[2]• indirect costs: US\$60 billion^[2]
Deaths	346 total: <ul style="list-style-type: none">• 189 on Lion Air Flight 610• 157 on Ethiopian Airlines Flight 302

“The MCAS software didn’t have any basic sanity checks to confirm the data was bad,” Travis said. What’s even more astounding is that Boeing is still trying to fix MCAS while all of its 737 MAX planes are grounded around the globe, he added. That includes 737 MAX planes used by American and Southwest in the U.S. “I don’t understand why Boeing is hell bent on fixing MCAS as opposed to retreating and taking another tack.”

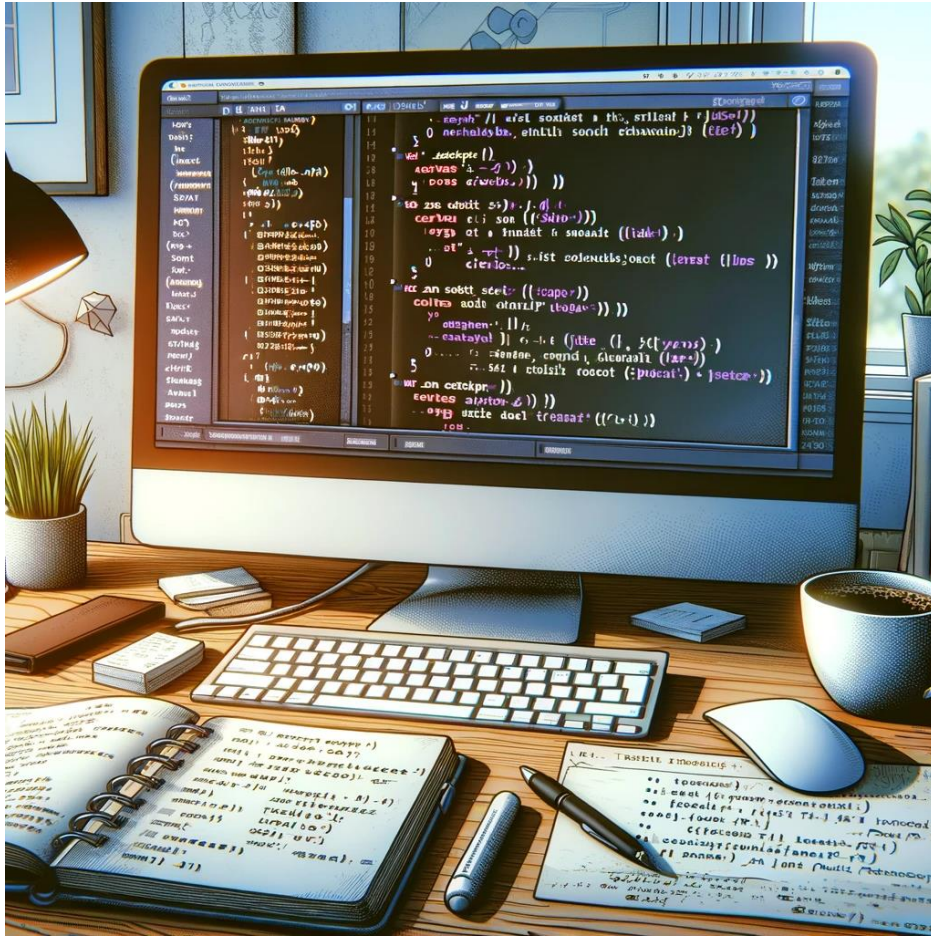
**Errors do
lead to
catastrophe**

INTRODUCTION—WHAT IS TEST IN GENERAL?



**Test is
quality
control**

INTRODUCTION—WHAT ARE UNIT-TESTS?



INTRODUCTION—WHAT ARE UNIT-TESTS?

```
def mySqrt(x):  
    ... #implementation
```

```
#test  
if (mySqrt(2.) - 1.414213) > 1E-6: print(„x = 2.: test failed“)  
elif (mySqrt(1.) != 1.)           : print(„x = 1.: test failed“)  
elif !(np.isnan(mySqrt(-1)))      : print(„x = -1: test failed“)  
else:                             : print(„all tests passed“)
```



Test a piece of
code with
examples

INTRODUCTION—WHAT ARE UNIT-TESTS?

```
def mySqrt(x):  
    ... #implementation
```

```
ASSERT_DOUBLE_EQ(mySqrt(2.), 1.414213562373095048801688 , „x=2“)  
ASSERT_DOUBLE_EQ(mySqrt(1.), 1. , „x=1“)  
ASSERT_FALSE(mySqrt(-1.) , „x=-1“)
```

Normally use a
software-toolkit

INTRODUCTION—WHAT ARE UNIT-TESTS?

```
class sphere:
    def __init__(self, X, Y, R):
        self.x = X
        self.y = Y
        self.r = R
```

Usually more
complicated
examples

INTRODUCTION—WHAT ARE UNIT-TESTS?

```
class sphere:
    def __init__(self, X, Y, R):
        self.x = X
        ...

    def overlap(self, anotherSphere):
        #implementation
```



Have some
member-
function

INTRODUCTION—WHAT ARE UNIT-TESTS?

```
class sphere:
    def __init__(self, X, Y, R):
        self.x = X
        ...

    def overlap(self, anotherSphere):
        #implementation
```

```
def testSpheresDoOverlap():
    sphere1 = sphere(0., 0., 1.)
    sphere2 = sphere(1., 1., 1.)
    assertTrue(sphere1.overlap(sphere2))
    assertTrue(sphere2.overlap(sphere1))
```

```
def testSpheresDontOverlap():
    sphere1 = sphere(0., 0., 1.)
    sphere2 = sphere(2., 2., 1.)
    assertFalse(sphere1.overlap(sphere2))
    assertFalse(sphere2.overlap(sphere1))
```



Think of some
test cases

INTRODUCTION—WHAT ARE UNIT-TESTS?

```
class sphere:
    def __init__(self, X, Y, R):
        self.x = X
        ...

    def overlap(self, anotherSphere):
        #implementation
```

```
def testSpheresDoOverlap():
    sphere1 = sphere(0., 0., 1.)
    sphere2 = sphere(1., 1., 1.)
    assertTrue(sphere1.overlap(sphere2))
    assertTrue(sphere2.overlap(sphere1))
```

```
def testSpheresDontOverlap():
    sphere1 = sphere(0., 0., 1.)
    sphere2 = sphere(2., 2., 1.)
    assertFalse(sphere1.overlap(sphere2))
    assertFalse(sphere2.overlap(sphere1))
```

```
def testSpheresDoOverlap():
    sphere1 = sphere(0., 0., 1.)
    sphere2 = sphere(sqrt(2), sqrt(2), 1.)
    assertTrue(sphere1.overlap(sphere2))
    assertTrue(sphere2.overlap(sphere1))
```

Think of critical cases

INTRODUCTION—WHAT ARE UNIT-TESTS?

```
class sphere:
    def __init__(self, X, Y, R):
        self.x = X
        ...

    def overlap(self, anotherSphere):
        ...
```

```
def testSpheresDoOverlap():
    sphere1 = sphere(0., 0., 1.)
    sphere2 = sphere(1., 1., 1.)
    assertTrue(sphere1.overlap(sphere2))
    assertTrue(sphere2.overlap(sphere1))
```

```
def testSpheresDontOverlap():
    sphere1 = sphere(0., 0., 1.)
    sphere2 = sphere(2., 2., 1.)
    assertFalse(sphere1.overlap(sphere2))
    assertFalse(sphere2.overlap(sphere1))
```

```
def testSpheresDoOverlap():
    sphere1 = sphere(0., 0., 1.)
    sphere2 = sphere(sqrt(2), sqrt(2), 1.)
    assertTrue(sphere1.overlap(sphere2))
    assertTrue(sphere2.overlap(sphere1))
```

```
if __name__ == "__main__":
    #runAllTests
```

All tests are
called somehow
(very often)

INTRODUCTION—WHAT ARE UNIT-TESTS?

```
class Date:
    def __init__(self, YEAR, MONTH, DAY):
        self.YEAR = YEAR
        ...
    def shiftOneMonth():
        ...
```

```
def testShiftOctoberToChristmas():
    myDate = Date(2018, 10, 25)
    myDate.shiftOneMonth()
    myDate.shiftOneMonth()
    ASSERT_EQ(myDate, Date(2018, 12, 27))
```

Can involve
several steps

INTRODUCTION—WHAT ARE UNIT-TESTS?

```
class ActiveBrownianParticle:
    def __init__(self, v0, Dr):
        self.pos = (0., 0.)
        self.phi = rand()
        ...
    def applyNoise(...):
        ...
    def propellForward(...):
        ...
    def trajectory(...):
        ...
```


INTRODUCTION—WHAT ARE UNIT-TESTS?

```
class ActiveBrownianParticle:
    def __init__(self, v0, Dr):
        self.pos = (0., 0.)
        self.phi = rand()
        ...
```

```
def testFitLinear():
    allTrajectories = []
    for i in range(oneBillionBillion):
        oneParticle = ActiveBrownianParticle(...)
        allTrajectories.append(oneParticle.trajectory(...))
    linearFit = ...
    assertTrue(linearFit.fitError < 0.0001)
```

```
def testDifflong():
    allTrajectories = []
    for i in range(oneBillionBillion):
        oneParticle = ActiveBrownianParticle(...)
        allTrajectories.append(oneParticle.trajectory(...))
    Difflong = ...
    assertTrue(Difflong*2*Dr/v0^2 -1 < 0.0001)
```



How do deal
with
randomness

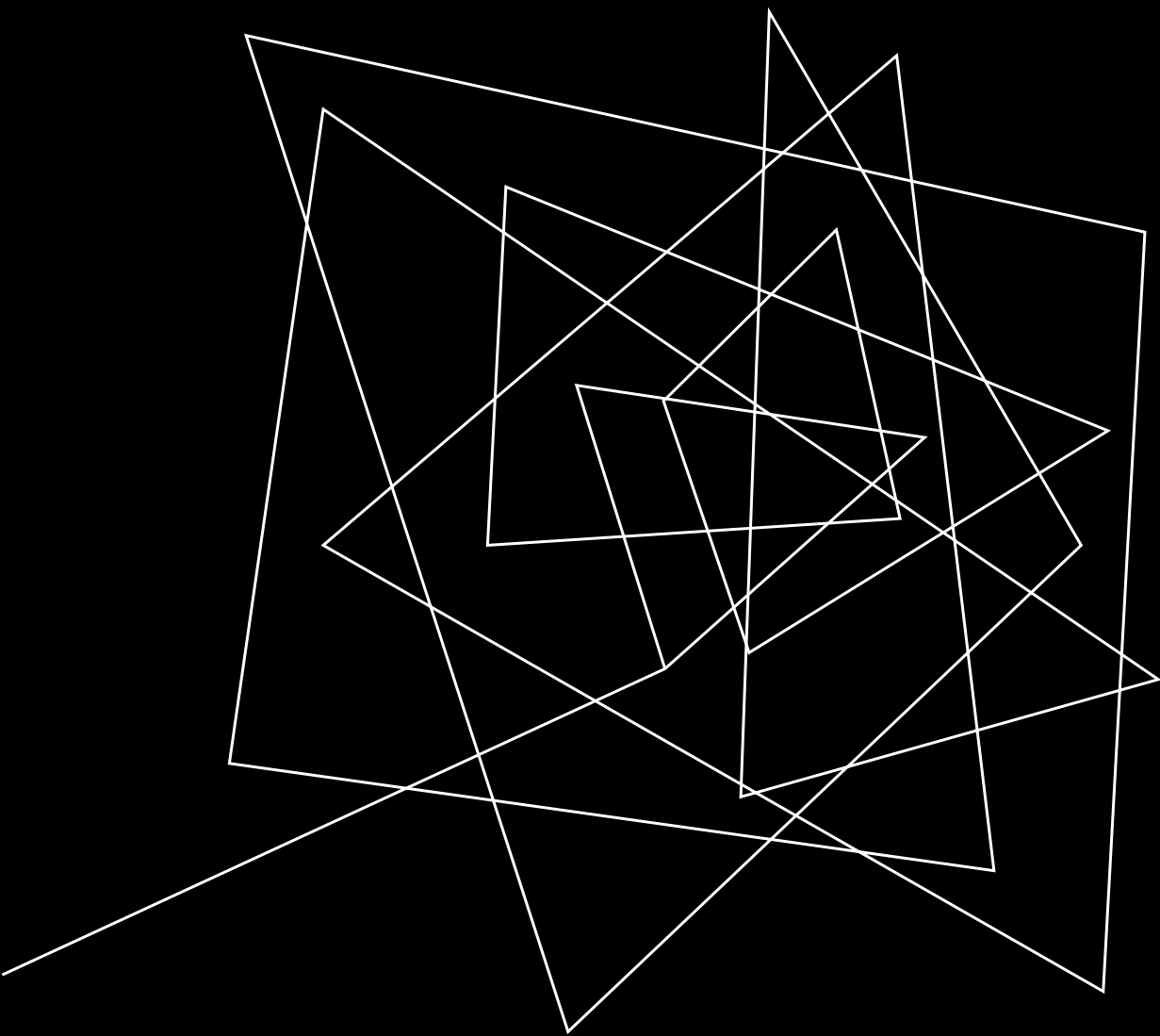
INTRODUCTION—WHAT ARE UNIT-TESTS?

```
class ActiveBrownianParticle:
    def __init__(self, v0, Dr):
        self.pos = (0., 0.)
        self.phi = rand()
        ...
```

```
def testFitLinear():
    allTrajectories = []
    for i in range(oneBillionBillion):
        oneParticle = ActiveBrownianParticle(...)
        allTrajectories.append(oneParticle.trajectory(...))
    linearFit = ...
    assertTrue(linearFit.fitError < 0.0001)
```

```
def testDifflong():
    allTrajectories = []
    for i in range(oneBillionBillion):
        oneParticle = ActiveBrownianParticle(...)
        allTrajectories.append(oneParticle.trajectory(...))
    Difflong = ...
    assertTrue(Difflong*2*Dr/v0^2 -1 < 0.0001)
```

Keep separated
and light-weight



What is test-driven
developent?

1. workflow
2. example

WHAT IS TEST-DRIVEN DEVELOPMENT—WORKFLOW

1. Write a shell-implementation for in incrementational expansion of the existing code, which returns a wrong result
2. Write a set of appropriate unit-tests
3. Run all unit-tests and make sure they fail
4. Write the implementation of the code
5. Run all unit-tests and make sure they pass.
If not, go back to step 4.



**General
workflow**

WHAT IS TEST-DRIVEN DEVELOPMENT—EXAMPLE—

STEP 1: SHELL IMPLEMENTATION

Implementation code

```
class myMathFunctions:  
    def __init__(self):  
        ..  
    def mySqrt(self,x):  
        return -1
```

Test code

```
import myMathFunctions  
  
if __name__ == „__main__“:  
    # do nothing so far
```

Shell
implementation

WHAT IS TEST-DRIVEN DEVELOPMENT—EXAMPLE—

STEP 2: WRITE TESTS

Implementation code

```
class myMathFunctions:
    def __init__(self):
        ...
    def mySqrt(self,x):
        return -1
```

Test code

```
import myMathFunctions
def testSqrtOfOne():
    functionInstance = myMathFunctions()
    ASSERT_EQ(functionInstance.mySqrt(1.), 1., „x=1“)
def testSqrtOfTwo():
    functionInstance = myMathFunctions()
    ASSERT_EQ(functionInstance.mySqrt(2.), 1.414..., „x=2“)
def testSqrtOfZero():
    functionInstance = myMathFunctions()
    ASSERT_EQ(functionInstance.mySqrt(0), 0, „x=0“)
def testSqrtOfNegative():
    functionInstance = myMathFunctions()
    ASSERT_FALSE(functionInstance.mySqrt(-1.), „x=-1“)

if __name__ == „__main__“:
    testSqrtOfOne()
    testSqrtOfTwo()
    testSqrtOfZero()
    testSqrtOfNegative()
```

Write
Tests

WHAT IS TEST-DRIVEN DEVELOPMENT—EXAMPLE—

STEP 3: RUN ALL TESTS—MAKE SURE THEY FAIL

Implementation code

```
class myMathFunctions:
    def __init__(self):
        ...
    def mySqrt(self,x):
        return -1
```

Test code

```
import myMathFunctions
def testSqrtOfOne():
    functionInstance = myMathFunctions()
    ASSERT_EQ(functionInstance.mySqrt(1.), 1., „x=1“)
def testSqrtOfTwo():
    functionInstance = myMathFunctions()
    ASSERT_EQ(functionInstance.mySqrt(2.), 1.414..., „x=2“)
def testSqrtOfZero():
    functionInstance = myMathFunctions()
    ASSERT_EQ(functionInstance.mySqrt(0), 0, „x=0“)
def testSqrtOfNegative():
    functionInstance = myMathFunctions()
    ASSERT_FALSE(functionInstance.mySqrt(-1.), „x=-1“)

if __name__ == „__main__“:
    testSqrtOfOne()
    testSqrtOfTwo()
    testSqrtOfZero()
    testSqrtOfNegative()
```

Run all tests
Make sure they
fail

WHAT IS TEST-DRIVEN DEVELOPMENT—EXAMPLE— STEP 4&5: IMPLEMENT FUNCTION AND USE TESTS

Implementation code

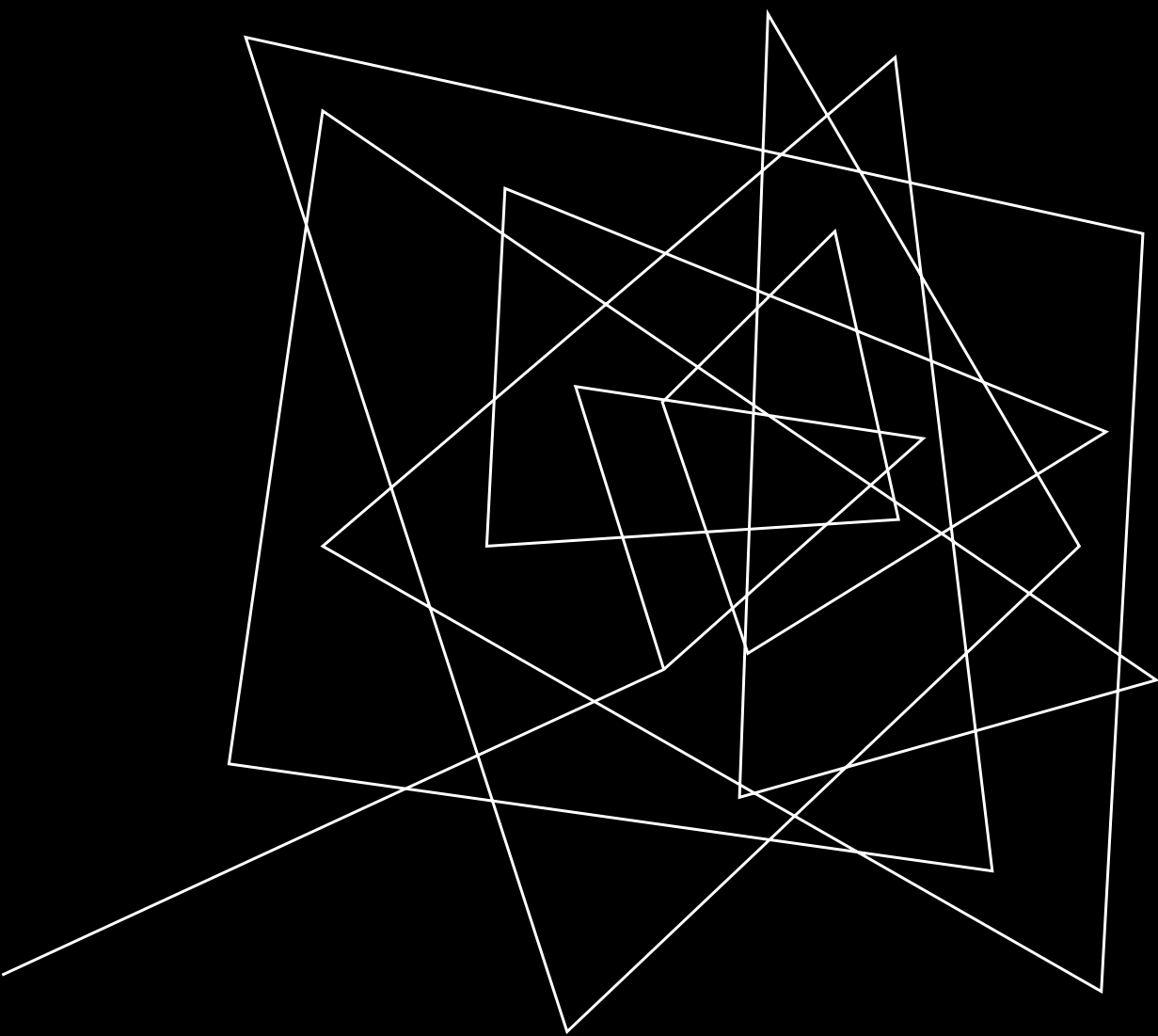
```
class myMathFunctions:
    def __init__(self):
        ...
    def mySqrt(self, x):
        if x < 1: return NaN
        else:
            for i in range(10):
                x = NewtonMethodStep(y**2 - x)
            return x
```

Test code

```
import myMathFunctions
def testSqrtOfOne():
    functionInstance = myMathFunctions()
    ASSERT_EQ(functionInstance.mySqrt(1.), 1., „x=1“)
def testSqrtOfTwo():
    functionInstance = myMathFunctions()
    ASSERT_EQ(functionInstance.mySqrt(2.), 1.414..., „x=2“)
def testSqrtOfZero():
    functionInstance = myMathFunctions()
    ASSERT_EQ(functionInstance.mySqrt(0), 0, „x=0“)
def testSqrtOfNegative():
    functionInstance = myMathFunctions()
    ASSERT_FALSE(functionInstance.mySqrt(-1.), „x=-1“)

if __name__ == „__main__“:
    testSqrtOfOne()
    testSqrtOfTwo()
    testSqrtOfZero()
    testSqrtOfNegative()
```

Implement the
desired
functionality



4 reasons for test-driven development:

- confidence and speed
- easier problem solving
- better code
- debugging

4 REASONS—REASON 1: CONFIDENCE AND SPEED

```
def pointsDistance(x,y):  
    return (x[0]-y[0])*(x[0]-y[0]) +  
           (x[1]-y[1])*(x[1]-y[1]) +  
           (x[2]-y[2])*(x[2]-y[2])
```

**More confident
not to commit
errors and
therefore faster**

4 REASONS—REASON 1: CONFIDENCE AND SPEED

```
def pointsDistance(x,y):  
    return (x[0]-y[0])*(x[0]-y[0]) +  
           (x[1]-y[1])*(x[1]-y[1]) +  
           (x[2]-y[2])*(x[2]-y[2])
```

```
typedef array<double,3> vec3d;  
vec3d operator*(const vec3d & myVec, const double scalar) {  
    const vec3d result = myVec;  
    for (auto &a : myVec) {  
        a *= scalar;  
    }  
    return result;
```

More confident
in new syntax
Learn more/
More efficient


4 REASONS—REASON 2: EASIER PROBLEM SOLVING

```
class Date:
    def __init__(self, YEAR, MONTH, DAY):
        self.YEAR = YEAR
        ...
    def shiftOneMonth():
        ...
```


```
def testShiftFebruaryToEaster():
    myDate = Date(2018, 2, 1)
    myDate.shiftOneMonth()
    myDate.shiftOneMonth()
    ASSERT_EQ(myDate, Date(2018, 4, 2))
```

Easier to think
concrete;
You can guess

4 REASONS—REASON 3: BETTER CODE




**Bad code is often difficult to test in hindsight due to little encapsulation, i.e., lot of dependencies
=> Cant isolate individual units**



Since code that is developed test-driven, is already individually tested, tends to have less of those problems



**Easier to understand and therefore debug
Because easier units can be understood in isolation**



**Better code;
Easier to
understand**

4 REASONS—REASON 4: DEBUGGING



Quality control: Confidence in functionality



automatic bug searching



Can actually use ChatGPT code

