





Test-Driving Guided by ZOMBIES

James W Grenning
wingman-sw.com
aatc2016@gwingman-sw.com
 @jwgrenning





Copyright © 2017 James W. Grenning
 All Rights Reserved. @jwgrenning

Agile alliance Technical Conference 2017, Boston MA
 TDD Guided by ZOMBIES

www.wingman-sw.com
james@wingman-sw.com


Write a Test.

2

What Test?


3

Way Back at XPImmersion I



DTSTTCPW
 Add one behavior
 at a time

You're done when
 you run out of tests.



Copyright © 2017 James W. Grenning
 All Rights Reserved. @jwgrenning

Agile alliance Technical Conference 2017, Boston MA
 TDD Guided by ZOMBIES

www.wingman-sw.com
james@wingman-sw.com

4

Do the Simplest
thing that could
possibly work!

5

Add One Behavior
at a Time

6

You're done when
you run out of
tests!

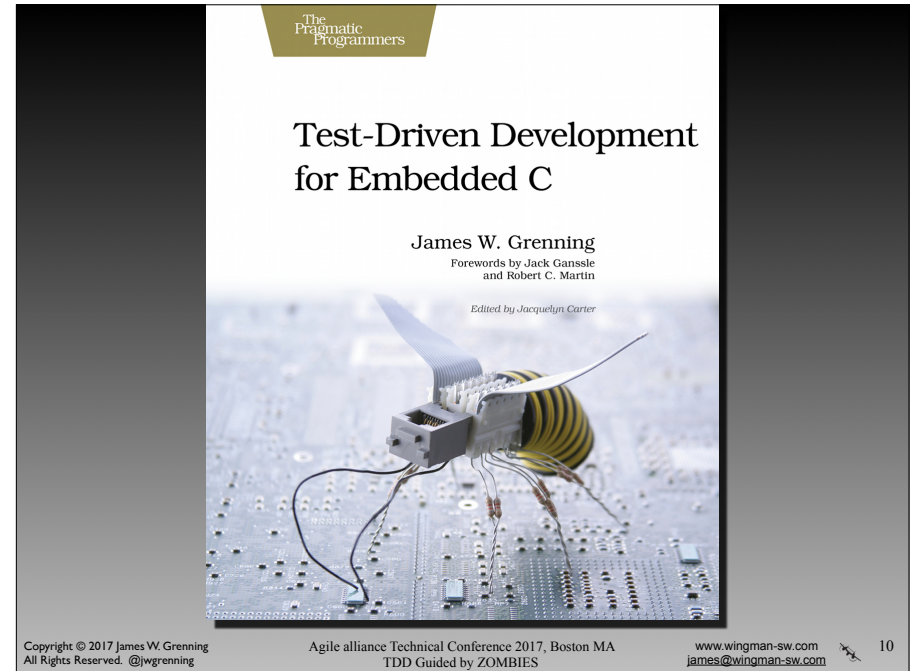
7

Incomplete!!
until you run out
of tests!?!?

8

There was a pattern

0, 1, Many



Martin Fowler @martinfowler · 1h
"@unclebobmartin wrote thousands of pages on clean code. @KentBeck wrote four lines" --@chethendrickson #aatc2017

bliki: BeckDesignRules
When developing Extreme Programming in the 90's, Kent Beck developed four rules of simple design: Passes the tests, reveals intention, no duplic...
martinfowler.com

Martin Fowler Retweeted
James Grenning @jwrenning · 51m
"@unclebobmartin wrote thousands of pages on clean code. @KentBeck wrote 4 lines" Why? steps 2,3,4 of @KentBeck's rules are hard. #aatc2017

Uncle Bob's Three Laws

- I. DO NOT WRITE ANY PRODUCTION CODE UNLESS IT IS TO MAKE A FAILING UNIT TEST PASS.
- II. DO NOT WRITE ANY MORE OF A UNIT TEST THAN IS SUFFICIENT TO FAIL; AND COMPILATION FAILURES ARE FAILURES.
- III. DO NOT WRITE ANY MORE PRODUCTION CODE THAN IS SUFFICIENT TO PASS THE ONE FAILING UNIT TEST.

Yeah, but its hard
to not finish
writing the code.

13

Incomplete
until you run out
of tests!

14

After Training Attendee's First TDD

Please provide your first impression of Test-Driven Development

What did you **like** about TDD?

What **concerns** you about TDD?

What **surprised** you about TDD?

See the replies

<https://wingman-sw.com/impressions/first/concerns>

<https://wingman-sw.com/impressions/first/likes>

Getting the right tests written is not trivial

Difficult to write tests that incrementally grows the solution

..how I can be sure I have a complete set of tests for a piece of code.

Developer may not be able to think of the complete set of tests ...

How can I be sure that my tests cover every corner cases?

Felt like I was careening from test to test

Don't know what order to write test cases

Don't know which test cases to write

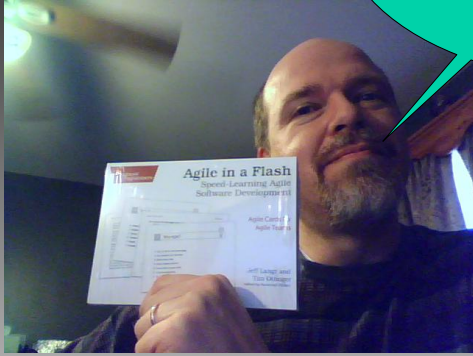
The biggest concern is over-testing.

Knowing when to stop is important.

Let's get back to

0, 1, Many

17



A photograph of a man with a beard holding a book titled "Agile in a Flash: Speed Learning, Agile Software Development". A red speech bubble above him contains the text "0, 1, Many?! Oh, that's ZOM." The book cover also lists "Agile Case Studies" and "Jeff Langr and Eric Kallgren".

Copyright © 2017 James W. Grenning
All Rights Reserved. @jwrenning

Agile alliance Technical Conference 2017, Boston MA
TDD Guided by ZOMBIES

www.wingman-sw.com
james@wingman-sw.com

18

Zero

Early tests focus on

interface

EXERCISING BEHAVIORS

EXERCISING BOUNDARIES

One

Early tests focus on

interface

EXERCISING BEHAVIORS

EXERCISING BOUNDARIES

MANY

EXERCISING BEHAVIORS

EXERCISING BOUNDARIES

When we
get close to
DONE

Don't forget about

ERROR
SCENARIOS

Don't forget about

EXCEPTIONAL
SCENARIOS

Simple Scenarios

Simple
Zero
Interfaces
Many
Exceptional
One
Scenarios



TDD Guided by ZOMBIES

Zero	Simple Scenarios Simple Solutions		
One			
Many			
	B	I	E
	o	n	x
	u	t	c
	n	e	e
	d	r	s
	a	a	
	r	n	
	i	c	
	e	e	
	s	s	

Not these zombies!

TDD Guided by ZOMBIES

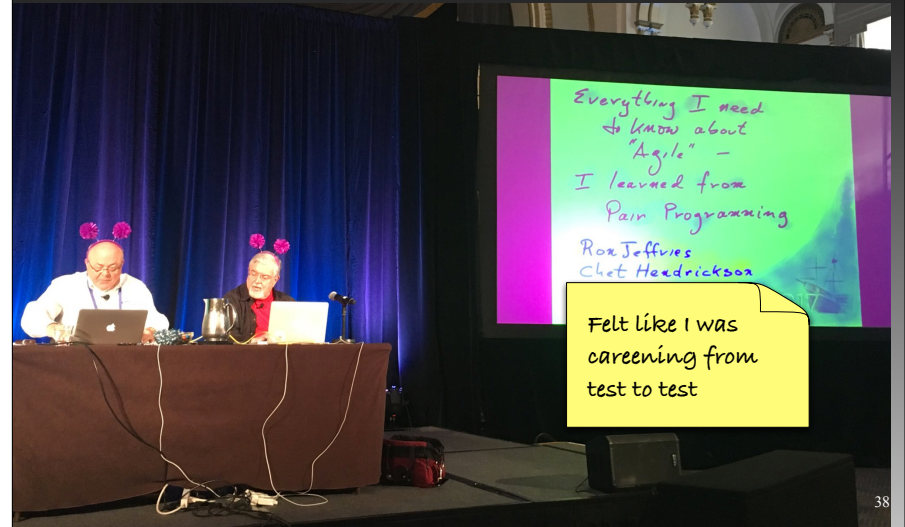
- The antithesis of mindless Zombies
- It is a very deliberate and systematic approach.

It was hard to stop thinking ahead

Its weird you're not supposed to think



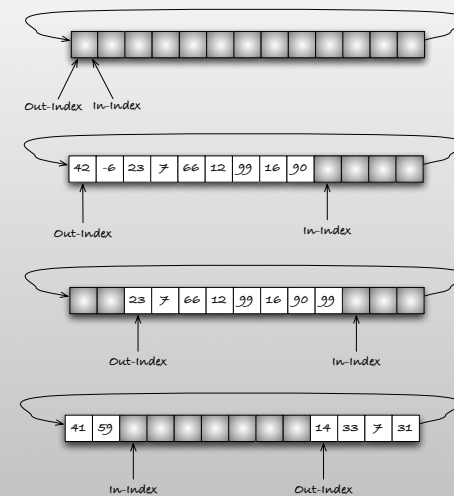
To the Person New to TDD it's Odd!



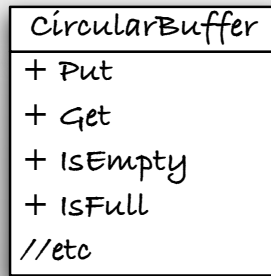
An Example

- The Circular Buffer holds a series of integers.
- Create a Module that manages the usage and integrity of a Circular Buffer.
- A full Circular Buffer rejects new values
- An empty buffer returns a default value

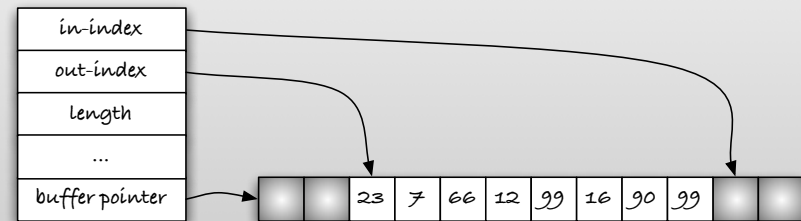
Create a Circular Buffer Module



The Anticipated Interface



Anticipated Internal Structure



Also Guided by a Test List

Circular Buffer Test List

Wrap around
Overflow
Underflow
Empty
Full
Happy path - FIFO

What are some Zero Scenarios?

- Creation
- Destruction
- Post condition to create

Boundaries Exercised: Create and Destroy

```
TEST_GROUP(CircularBuffer)
{
    CircularBuffer * buffer;

    void setup()
    {
        buffer = CircularBuffer_Create();
    }

    void teardown()
    {
        CircularBuffer_Destroy(buffer);
    }
};

TEST(CircularBuffer, can_be_created_and_destroyed)
{
}
```

Zero Scenarios Use Interfaces and Exercise Boundaries

```
TEST(CircularBuffer, is_empty_after_creation)
{
    CHECK_TRUE(CircularBuffer_IsEmpty(buffer));
}
```

Choose some Zero Scenarios

```
TEST(CircularBuffer, is_empty_after_creation)
{
    CHECK_TRUE(CircularBuffer_IsEmpty(buffer));
}

TEST(CircularBuffer, is_not_full_after_creation)
{
    CHECK_FALSE(CircularBuffer_IsFull(buffer));
}
```

Interfaces Defined and Used Once

```
#include <stdbool.h>

typedef struct CircularBufferStruct CircularBuffer;

CircularBuffer * CircularBuffer_Create(void);
void CircularBuffer_Destroy(CircularBuffer *);
bool CircularBuffer_IsEmpty(CircularBuffer *);
bool CircularBuffer_IsFull(CircularBuffer *);
```

Implementation is Incomplete

```
#include "CircularBuffer.h"

struct CircularBufferStruct
{
    int place_holder_delete_me_soon;
};

CircularBuffer * CircularBuffer_Create(void)
{
    CircularBuffer * self = (CircularBuffer *)calloc(1,
        sizeof(CircularBuffer));
    return self;
}

void CircularBuffer_Destroy(CircularBuffer * self)
{
    free(self);
}
```

Implementation is Incomplete

```
bool CircularBuffer_IsEmpty(CircularBuffer * self)
{
    return true;
}

bool CircularBuffer_IsFull(CircularBuffer * self)
{
    return false;
}
```

Creating a walking skeleton

- Growing the code,
behavior by behavior



Define Put () Interface and Exercise another Boundary

```
TEST(CircularBuffer, is_not_empty_after_put)
{
    CircularBuffer_Put(buffer, 42);
    CHECK_FALSE(CircularBuffer_IsEmpty(buffer));
}
```

Define Get () Interface and Exercise another Boundary

```
TEST(CircularBuffer, is_empty_after_put_then_get)
{
    CircularBuffer_Put(buffer, 42);
    CircularBuffer_Get(buffer);
    CHECK_TRUE(CircularBuffer_IsEmpty(buffer));
}

TEST(CircularBuffer, get_equals_put_for_one_item)
{
    CircularBuffer_Put(buffer, 42);
    LONGS_EQUAL(42, CircularBuffer_Get(buffer));
}
```

Interfaces Defined and Used

```
#include <stdbool.h>

typedef struct CircularBufferStruct CircularBuffer;

CircularBuffer * CircularBuffer_Create(void);
void CircularBuffer_Destroy(CircularBuffer *);
bool CircularBuffer_IsEmpty(CircularBuffer *);
bool CircularBuffer_IsFull(CircularBuffer *);
void CircularBuffer_Put(CircularBuffer *, int);
int CircularBuffer_Get(CircularBuffer *);
```



What has been
accomplished?

Common Answer:

You didn't test
anything!

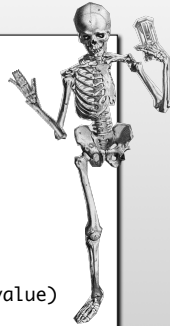
Skeletal Implementation

```
bool CircularBuffer_IsEmpty(CircularBuffer * self)
{
    return self->index == self->outdex;
}

bool CircularBuffer_IsFull(CircularBuffer * self)
{
    return false;
}

void CircularBuffer_Put(CircularBuffer * self, int value)
{
    self->index++;
}

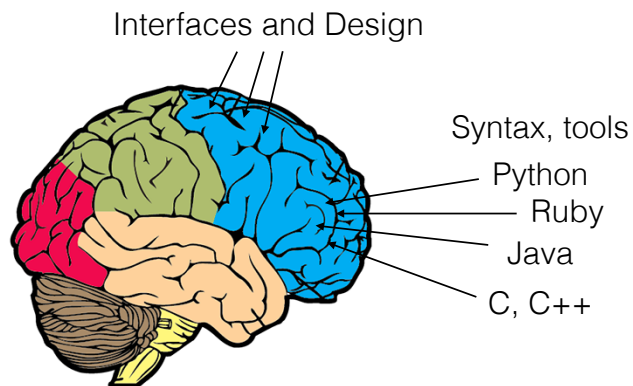
int CircularBuffer_Get(CircularBuffer * self)
{
    self->outdex++;
}
```



You didn't test anything!

- True enough. The thing we wrote tests for surely was not a CircularBuffer.
- But we had a nearly complete API, and several important boundary test cases defined.
- We also do understand quite well what shared variables are responsible for differentiating empty from not empty.

Exercising Different parts of your brain



Now for the
First of Many

The First of Many

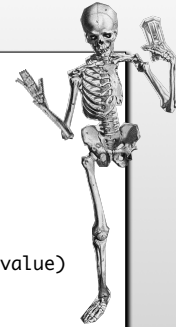
```
TEST(CircularBuffer, put_get_is_fifo)
{
    CircularBuffer_Put(buffer, 41);
    CircularBuffer_Put(buffer, 42);
    CircularBuffer_Put(buffer, 43);
    LONGS_EQUAL(41, CircularBuffer_Get(buffer));
    LONGS_EQUAL(42, CircularBuffer_Get(buffer));
    LONGS_EQUAL(43, CircularBuffer_Get(buffer));
}
```

The First of Many

```
struct CircularBufferStruct
{
    int index;
    int outdex;
    int values[10];
};
//snip

void CircularBuffer_Put(CircularBuffer * self, int value)
{
    self->values[self->index] = value;
    self->index++;
}

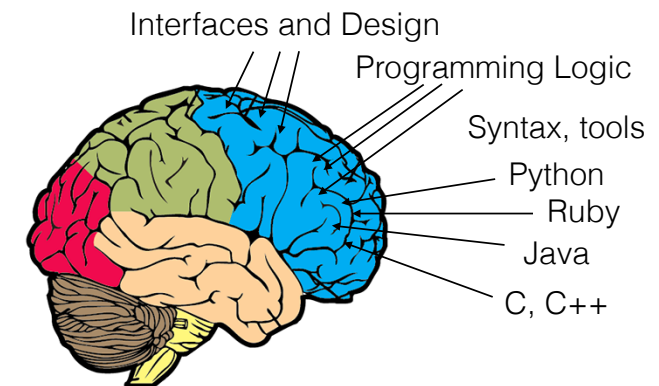
int CircularBuffer_Get(CircularBuffer * self)
{
    int value = self->values[self->outdex];
    self->outdex++;
    return value;
}
```



The First of Many

- We are into a test that is not about defining a new interface.
- Most the prior tests have been.
- Making this test pass is purely about functionality.
- *Many* brings the logical challenges moving past the syntax and tool challenges

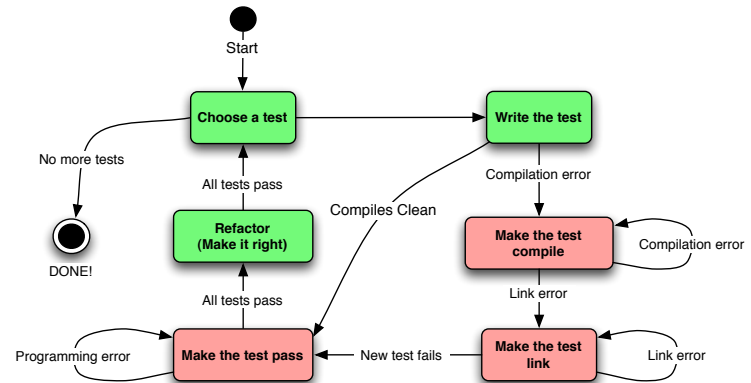
Exercising Different parts of your brain



How did you learn to program?

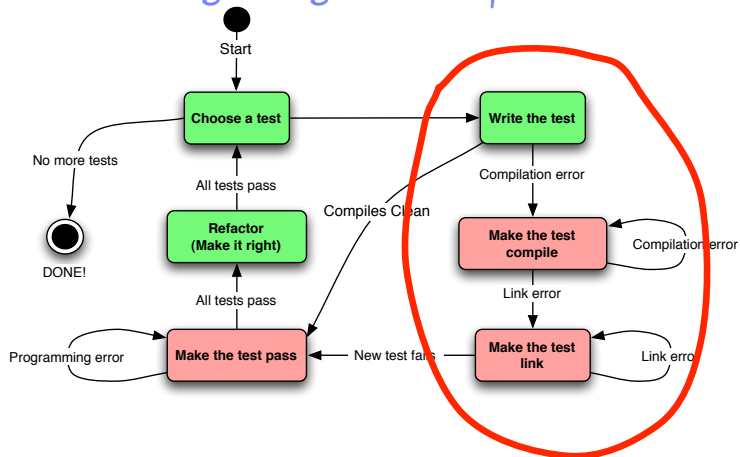
In your own unique and personal way.

TDD State Machine in C/C++ (solving one problem at a time)

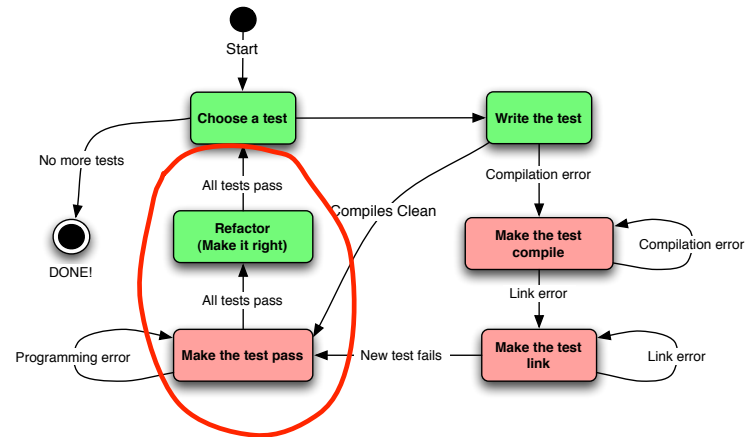


Add new tests to the test list as needed

Zero and One Mostly Defining Interfaces and Fighting the compiler



TDD State Machine in C/C++ (solving one problem at a time)



How Many?

```
TEST(CircularBuffer, report_capacity)
{
    LONGS_EQUAL(CAPACITY, CircularBuffer_Capacity(buffer));
}
```

How Many?

```
TEST(CircularBuffer, report_capacity)
{
    LONGS_EQUAL(CAPACITY, CircularBuffer_Capacity(buffer));
}

TEST(CircularBuffer, capacity_is_adjustable)
{
    CircularBuffer * buffer = CircularBuffer_Create(CAPACITY+2);
    LONGS_EQUAL(CAPACITY+2, CircularBuffer_Capacity(buffer));
    CircularBuffer_Destroy(buffer);
}
```

This Many

```
typedef struct CircularBufferStruct
{
    int index;
    int outdex;
    int capacity;
    int values[10];
} CircularBuffer;

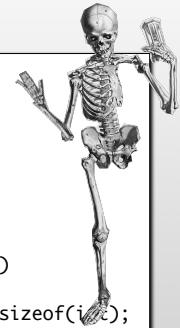
CircularBuffer * CircularBuffer_Create(int capacity)
{
    CircularBuffer * self = calloc(1, sizeof(CircularBuffer));
    self->capacity = capacity;
    return self;
}
//snip
unsigned int CircularBuffer_Capacity(CircularBuffer * self)
{
    return self->capacity;
}
```



Refactored to use the heap

```
struct CircularBufferStruct
{
    int index;
    int outdex;
    int capacity;
    int values[];
};

CircularBuffer * CircularBuffer_Create(int capacity)
{
    int size = sizeof(CircularBuffer) + capacity * sizeof(int);
    CircularBuffer * self = (CircularBuffer *)malloc(size);
    self->capacity = capacity;
    self->index = 0;
    self->outdex = 0;
    return self;
}
```



Boundary: Fill to the top

```
static void fillItUp(CircularBuffer * buffer)
{
    for (int i = 0; i < CircularBuffer_Capacity(buffer); i++)
        CircularBuffer_Put(buffer, i);
}

TEST(CircularBuffer, fill_to_capacity)
{
    fillItUp(buffer);
    CHECK_TRUE(CircularBuffer_IsFull(buffer));
}
```

Boundary: Full then not Full

```
TEST(CircularBuffer, is_not_full_after_get_from_full_buffer)
{
    fillItUp(CircularBuffer_Capacity(buffer));
    CircularBuffer_Get(buffer);
    CHECK_FALSE(CircularBuffer_IsFull(buffer));
}
```

Boundary: Wrap

```
TEST(CircularBuffer, is_not_full_after_get_from_full_buffer)
{
    fillItUp(CircularBuffer_Capacity(buffer));
    CircularBuffer_Get(buffer);
    CHECK_FALSE(CircularBuffer_IsFull(buffer));
}
```

Boundary: Wrap to Full

```
TEST(CircularBuffer, full_after_wrapping)
{
    CircularBuffer * buffer = CircularBuffer_Create(2);
    CircularBuffer_Put(buffer, 1);
    CircularBuffer_Put(buffer, 2);
    CircularBuffer_Get(buffer);
    CircularBuffer_Put(buffer, 3);
    CHECK_TRUE(CircularBuffer_IsFull(buffer));
    CircularBuffer_Destroy(buffer);
}
```

Exceptional: Put to Full

```
TEST(CircularBuffer, put_to_full_fails)
{
    CircularBuffer * buffer = CircularBuffer_Create(1);
    CHECK_TRUE(CircularBuffer_Put(buffer, 1));
    CHECK_FALSE(CircularBuffer_Put(buffer, 2));
    CircularBuffer_Destroy(buffer);
}
```

Exceptional: Get from Empty

```
TEST(CircularBuffer, put_to_full_fails)
{
    CircularBuffer * buffer = CircularBuffer_Create(1);
    CHECK_TRUE(CircularBuffer_Put(buffer, 1));
    CHECK_FALSE(CircularBuffer_Put(buffer, 2));
    CircularBuffer_Destroy(buffer);
}

TEST(CircularBuffer, get_from_empty_returns_default_value)
{
    LONGS_EQUAL(DEFAULT_VALUE, CircularBuffer_Get(buffer));
}
```

Unlike these Zombies

- Every step of ZOMBIES is deliberate.
- Programming is based on defining one behavior at a time.
- Relying on cause and effect.
- Starting on simpler concepts working your way to the challenge of the full fears



See the article on my blog <http://blog.wingman-sw.com/archives/677>

Could ZOMBIES have Prevented Boeing 787 Dreamliners' Potentially Catastrophic Software Bug?



<https://arstechnica.com/information-technology/2015/05/boeing-787-dreamliners-contain-a-potentially-catastrophic-software-bug/>

TDD Guided by ZOMBIES

Talk to me on Twitter
@jwgrenning

Find my book at
<http://wingman-sw.com/tddec>

Find me on linkedin.com
<http://www.linkedin.com/in/jwgrenning>
Please remind me how we met.

<http://facebook.com/wingman.sw>
<http://www.wingman-sw.com>
<http://www.wingman-sw.com/blog>

