Using TDD

Making code maintainable, reusable and readable by writing tests

BUILD.ONE

Intro

- Julian Lyndon-Smith
  - Chief Enterprise Architect, Build.One
- Using progress since v3
- Several open source projects
  - Stomp : messaging for ActiveMQ
  - Loki : generate openedge classes from OpenApi spec
  - Maia: generate openedge classes for database access
  - UIB utilities
  - v8Stuff.com and v9Stuff.com

Introduction to TDD

- Test Driven Development
  - Test first
  - Code second
- Different from traditional development methods
  - Takes buy in from all involved
  - Makes you think differently
  - Hard to make the switch

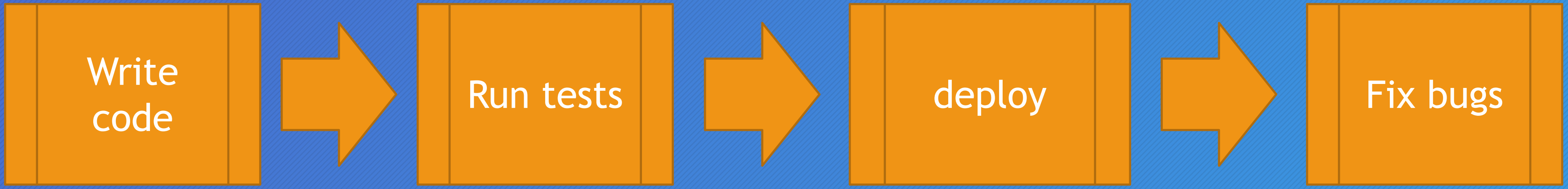
History of TDD

- 1989 : “fit”, one of the first testing frameworks written
- 1994 : Extreme Programming (XP) starts appearing
- 1999 : a number of books on TDD start appearing
- 2000 : jUnit launched
- 2004 : proUnit (http://prounit.sourceforge.net/userguide.html)
- 2010 : OEUnit (https://github.com/CameronWills/OEUnit)
- 2014 : initial release of ABLUnit
- 2018 : ABLUnit first class citizen in PDS

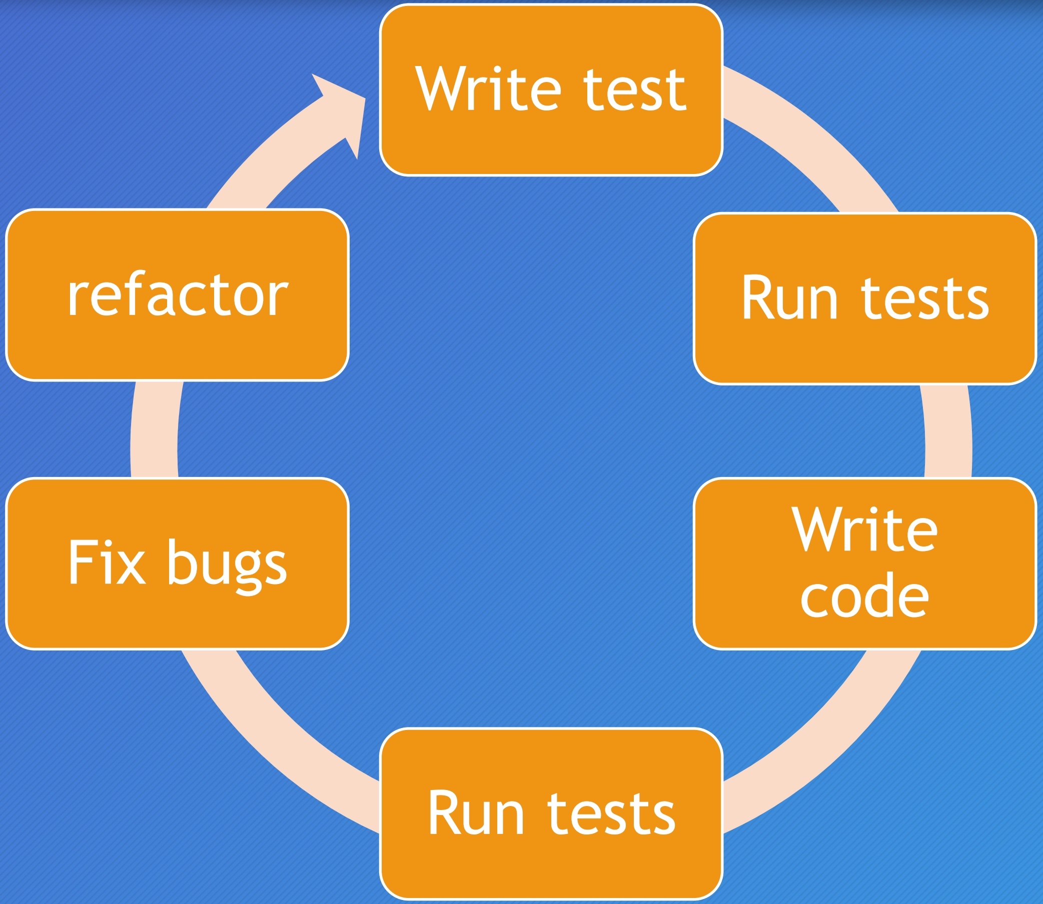
Framework Comparisons

https://community.progress.com/community_groups/openedge_development/f/19/t/11095

Traditional development cycle



TDD cycle



BUILD.ONE

TDD cycle : Red, green, refactor

- Red
  - Tests fail
- Green
  - Tests pass
- Refactor
  - Make code better to maintain and test

TDD cycle : Red

- Write a test for a class or method
  - That does not exist
  - A new requirement
- Test will (should) fail
- Why should we do this ?
  - Makes you think of what functionality you are testing
  - Makes you write the code required to pass the test only

BUILD.ONE

TDD cycle : Green

- Now, write just enough code to make the test pass
  - This is the difficult part !
- Take this business requirement:
  - A new method must, given an integer input value of 42, return false
  - What is enough code to make the test pass ?

When enough is too much

```
/** A new method must, given an integer input value
of 42, return false */


method public logical myMethod(a as int):
   if a eq 42 then return false.
   else return true.
end method.
```

WRONG…

- The requirement only stated what the method should do for an input value of 42.
- All other values are undetermined
- No requirement .. No test
    - Otherwise you are writing code that may never be used

When enough is enough

```
/** A new method must, given an integer input value
of 42, return false */

method public logical myMethod(a as int):
   return false.
end method.
```

Requirements are tests

- The previous example shows that there is at least one more requirement needed
  - Other numbers apart from 42
- The developer should liaise with other stakeholders
  - “are you needing different results for other numbers?”
- Write new tests for new requirements

TDD cycle : Refactor

- Refactoring code is done to make the code
  - Maintainable
  - Readable
  - Good code quality
- Your unit tests will help to check that you don’t break functionality
- DRY

BUILD.ONE

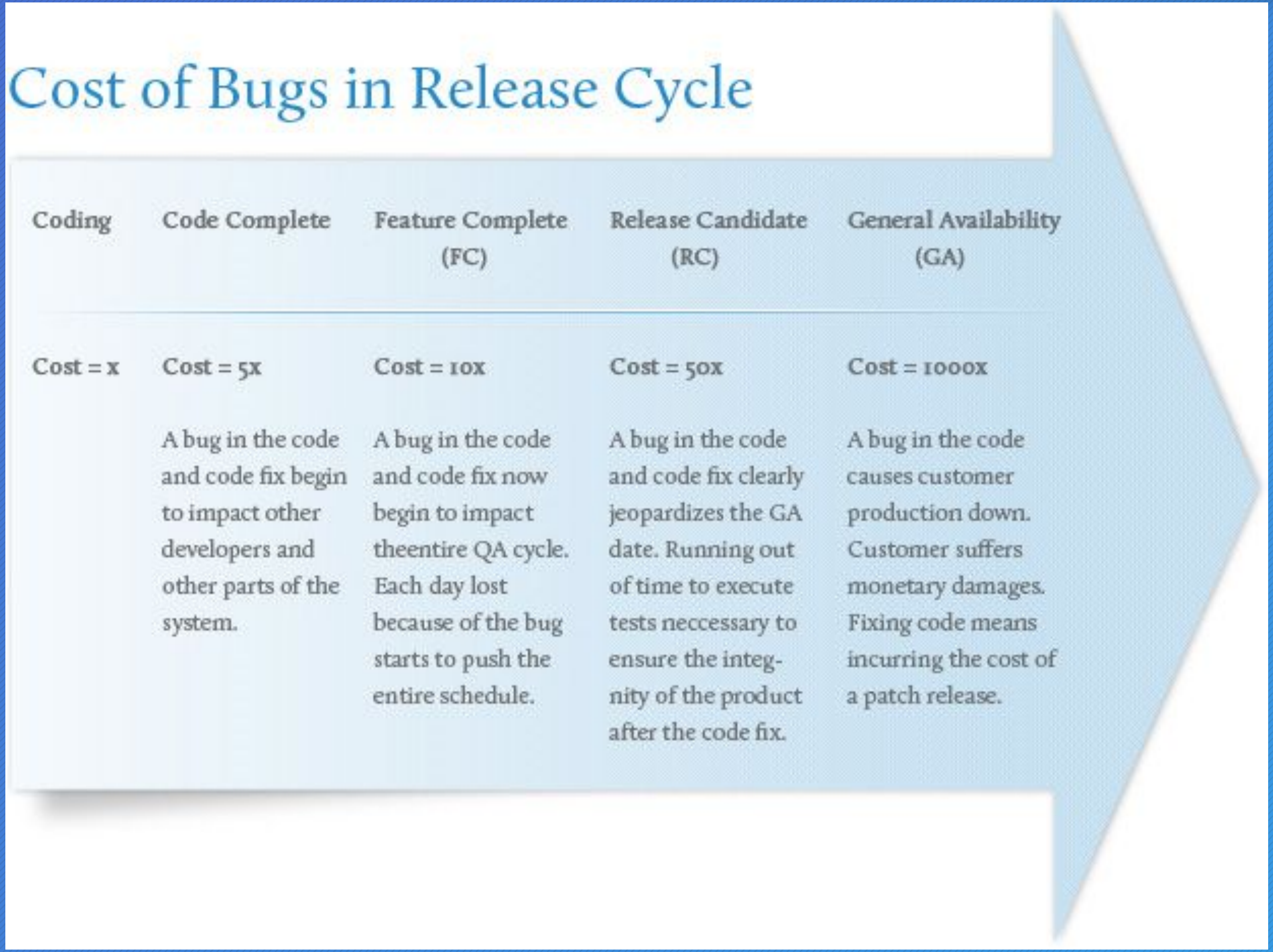
The benefits of TDD

- Ensures quality code from the very beginning
- Promotes loosely-coupled code
- Can provide specifications by tests
- Give you confidence that your code works

The benefits of TDD

- Keeps unused code out of your systems
- Makes you think hard about application design
- Finds bugs quickly

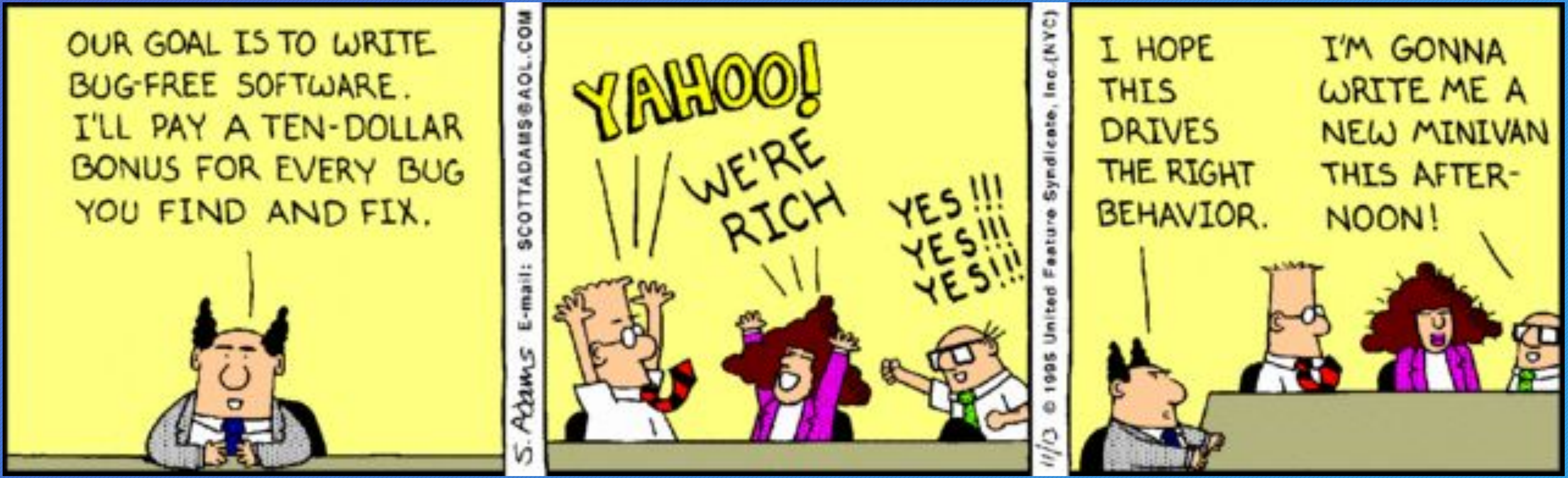
The cost of bugs - augmentum.com



Cost of Bugs in Release Cycle

Coding	Code Complete	Feature Complete (FC)	Release Candidate (RC)	General Availability (GA)
Cost = x	Cost = 5x	Cost = 10x	Cost = 50x	Cost = 1000x
	A bug in the code and code fix begin to impact other developers and other parts of the system.	A bug in the code and code fix now begin to impact theentire QA cycle. Each day lost because of the bug starts to push the entire schedule.	A bug in the code and code fix clearly jeopardizes the GA date. Running out of time to execute tests neccessary to ensure the integ-nity of the product after the code fix.	A bug in the code causes customer production down. Customer suffers monetary damages. Fixing code means incurring the cost of a patch release.

The cost of bugs – Dilbert


OUR GOAL IS TO WRITE BUG-FREE SOFTWARE. I'LL PAY A TEN-DOLLAR BONUS FOR EVERY BUG YOU FIND AND FIX.
S. Adams E-mail: SCOTTADAMS@AOL.COM
YAHOO!
WE'RE RICH
YES!!! YES!!! YES!!!
11/13 © 1995 United Feature Syndicate, Inc.(NYC)
I HOPE THIS DRIVES THE RIGHT BEHAVIOR.
I'M GONNA WRITE ME A NEW MINIVAN THIS AFTER-NOON!

BUILD.ONE

The benefits of TDD

- Code coverage
- Regression testing for free
- Stops recurring bugs
- Clean API design
- Reduced debugging
- Reduced development costs

The benefits of TDD (real world)

- Development of Maia4
  - Complete rewrite of Maia
- Took 3 weeks
  - Including writing unit tests
- Found 4 bugs in progress …

The benefits of TDD (real world)

- Return in finally
  - Doesn’t return longchars
- Recursive delete of a directory fails
  - If path contains a folder starting with “.”
- Json parser hangs if data contains comments
  - /* */
- Static properties and method calls as part of a parameter cause gpf

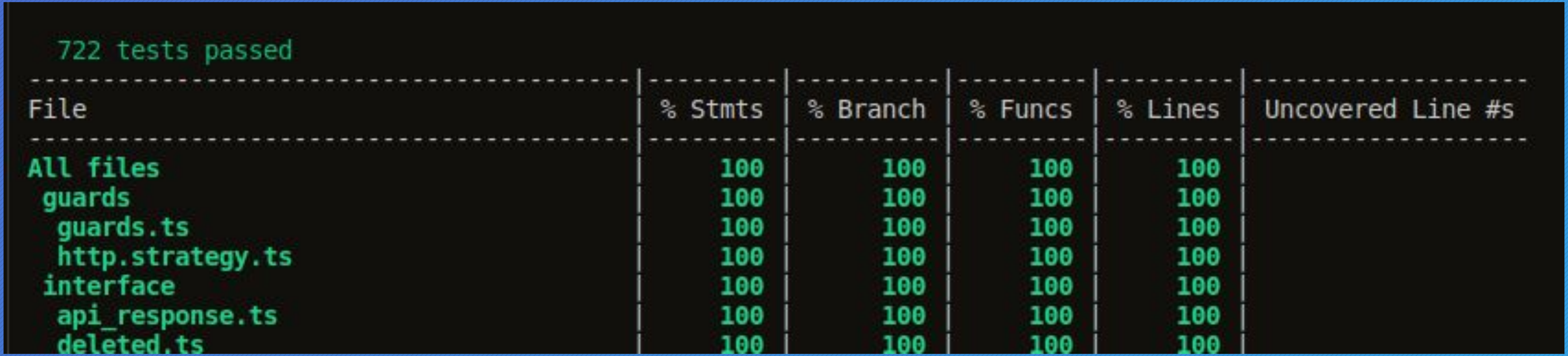
The benefits of TDD (real world)

- Only 2 bugs in maia4 found after initial alpha release
  - Extents not generated at all (missing code)
  - Custom properties assigned to db

- Several bugs found in UI …
  - No unit tests ☺️

The benefits of TDD (real world)

- Development of Security-Hub
  - NestJs/Angular app
  - Server side has 700+ tests
  - 100% coverage



```
 722 tests passed
-----------------------------------------|----------|----------|----------|----------|-------------------
File                                     | % Stmts  | % Branch | % Funcs  | % Lines  | Uncovered Line #s
-----------------------------------------|----------|----------|----------|----------|-------------------
All files                                |      100 |      100 |      100 |      100 |
 guards                                  |      100 |      100 |      100 |      100 |
  guards.ts                              |      100 |      100 |      100 |      100 |
  http.strategy.ts                       |      100 |      100 |      100 |      100 |
 interface                               |      100 |      100 |      100 |      100 |
  api_response.ts                        |      100 |      100 |      100 |      100 |
  deleted.ts                             |      100 |      100 |      100 |      100 |
```

BUILD.ONE

The benefits of TDD (real world)

- Added new SAML Auth module
- Added secure token login
- Added MySecrets module

Ran test suite - fixed any bugs found

Have deployed several versions now without any regressions or known bugs

The downsides of TDD

- Big time in investment
- Additional Complexity
- Harder than you think
- Selling to management
- Selling to developers ;)
- You lose the title of “Hacker” !

Unit tests

- What is a unit test ?
  - Test of one requirement for one method
- Isolation
  - other code / tests
  - Other developers
- Targeted
- Repeatable
- Predictable

Achieving good design

- Writing tests first means that you have to describe what you want to achieve before you write the code
- In order to keep tests understandable and maintainable, keep them as short as possible. Long tests imply that the unit under test is too large
- If a component requires too many dependencies, then it is too difficult to test

Code design to help with TDD

- SOLID
- Code “smell”
- Refactoring

SOLID

- Single responsibility
  - Each method and class should have only one responsibility
  - Open / Close principle
  - Open for extension, closed for modification
  - Inheritance
- Liskov subsititution principle
  - An object should be replaceable by the super class without breaking the application

SOLID

- Interface segregation principle
    - Must not rely on interfaces that a client does not need
- Dependency inversion
    - Code should depend on abstractions, not implementation

Code smell

- Mistaks: repeated mistaks ;)
- Duplicate code
- Big classes, huge methods
- Comments
    - controversial …
- Bad names
- Too many if .. Then or case statements

Code refactoring – rename members

```
method public decimal getValue(a as int,b as int):
   def var p as dec init 3.14159265359 no-undo.
   return (a * a) * b * p.
 end method.
```

BUILD.ONE

Code refactoring – rename members

```
method public decimal getCylinderVolume(radius as int, height as int):
   def var Pi as dec init 3.14159265359 no-undo.
   return (radius * radius) * height * Pi.
 end method.
```

Code refactoring

- Extract methods
- Extract interfaces
  - Multiple implementation
- Encapsulation of properties
  - Get / set
- Replace conditionals with polymorphism

Achieving good design

- Code which is complicated is
  - Bad design
  - Hard to maintain
  - Hard to test
  - Expensive to fix

TDD: Testing “smells” (1)

- Writing tests after writing code
- Not writing tests !
- Duplicate logic in tests
- Code apart from asserts / setup
  - logic in tests == bugs in tests (>90% likelyhood)

TDD: Testing “smells” (2)

- Remove tests
- Change tests
- Have test dependent on another test
- Have multiple asserts per test
  - unless checking multiple properties per object

TDD: Best Practices (1)

- Increase code coverage
- Test reviews
- Manually introduce a bug
  - if all tests pass, there’s a problem with the test
- Write tests first
- Make tests isolated

TDD: Best Practices (2)

- Ensure all unit tests pass. None should fail
- Integration tests should be in a separate project
- Test only publics (If possible)
- SOLID design
- Use Setup methods / refactor code into “helpers”

TDD: Best Practices (3)

- Make tests readable rather than maintainable
- Enforce test isolation
- Each test should set up and clean up it’s own state
- Any test should be repeatable
- Use variables instead of constants to make tests readable

TDD: Best Practices (bad naming)

```
@Test.
 method public void test#1():
   def var lv_data as char no-undo.
   assign lv_data = gecode:getGPS("maitland","southend").
   AssertString:IsNotNullOrEmpty(lv_data).
 end method.
```

BUILD.ONE

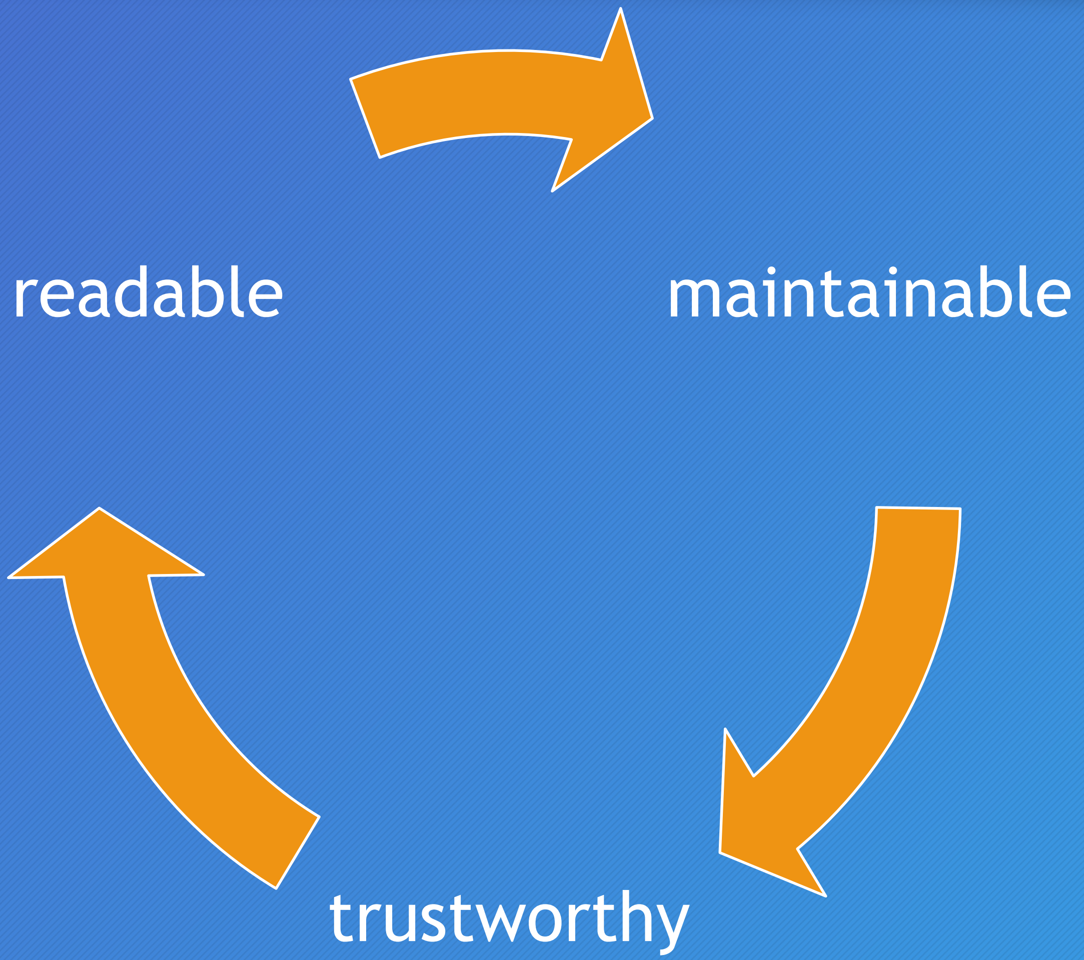
TDD: Best Practices (good naming)

```
@Test.
 method public void getGPSCoordinatesForBuildingInTown():
   def var lv_gpsCoord as char no-undo.
   def var lv_Town     as char init "southend" no-undo.
   def var lv_Building as char init "maitland" no-undo.

   assign lv_gpsCoord = gecode:getGPS(lv_Building,lv_Town).

   AssertString:IsNotNullOrEmpty(lv_gpsCoord ).
 end method.
```

TDD: Best Practices (4)

- Tests should run in any order
- Name tests appropriately (divideByZeroThrowsException)
- Name variables / use pre-processor
- Start using Interfaces to facilitate tests
  "mocks"

The three pillars of unit tests



BUILD.ONE

books

- The art of unit testing (Roy Osherove)
  - Second edition
- Clean Code: A Handbook of Agile Software Craftsmanship (Robert C. Martin)
- Dependency Injection (Steven van Deursen & Mark Seemann)

The obligatory My Little Pony

BUILD.ONE

Questions