

The logo for 'instil' is centered on a black background with a repeating pattern of small, light gray circles. The word 'instil' is written in a lowercase, white, sans-serif font. The dots above the 'i' and 'l' are replaced by small, solid yellow circles.

SOFTWARE | TRAINING | CONSULTING

Good Behaviour in Test Driven Development

Me?

- Tara Simpson
- Director of Instil Software Ltd
- Delivering training and services throughout Ireland and beyond
 - 80% coding – Application frameworks, rich-client and web apps, compilers, etc
 - 20% training – heavy focus on developer testing and technique
- 18+ years broad industry experience

Recent and Ongoing Work

- Design and development of AS/400 emulation
 - C/C++
 - 100+ man-year project
 - 2000+ integration tests
- Design and development of Telecom Web Services application framework
 - 40+ man-year project
 - Telco-scaling
 - 6000+ unit tests

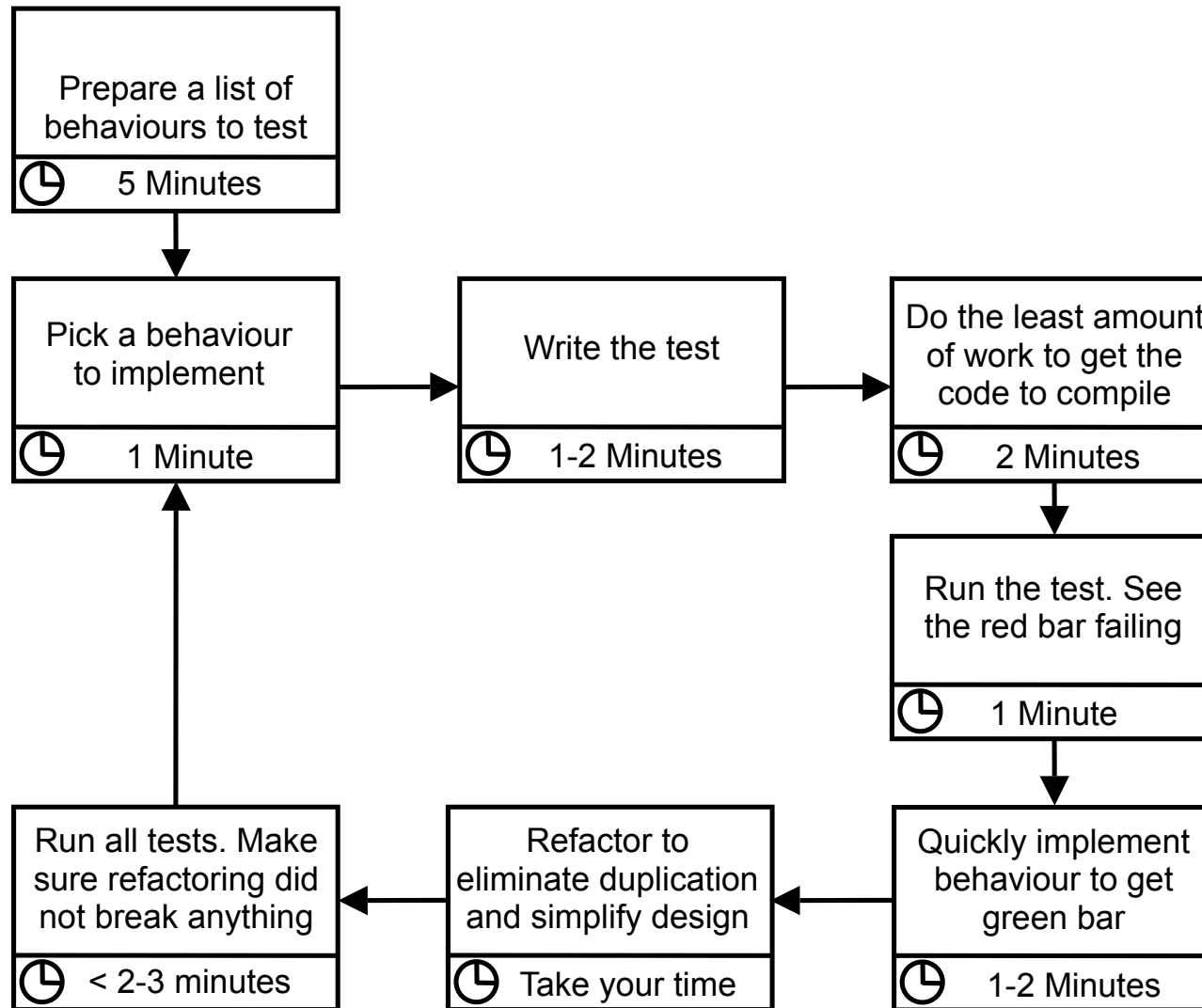
Intent

- Review Test-Driven Development
 - Focus on specific steps
- Understand importance of behaviour
 - In specifying tests
 - In driving forward design
 - In shaping the developer mindset

TDD in a Nutshell

- A core discipline of agile development
 - Applicable within any macro-process model
- Its goal is 'clean code that works'
 - Working Code.. delivered through tests
 - Clean Code.. enabled through refactoring
- Design viewed as a process of discovery rather than speculation
 - Foremost a design methodology

The Test-Driven Process



Step 1: Create a List of Tests

- Draw up a list of *behavioural* specifications that need to be supported
 - The things the object under test should do
- Each test specifies a single behavioural requirement
- Each test defines an achievable goal; a unit of work
 - The complete list is your TODO list, to be added to at any time

What is Behaviour?

- The observable *result* of invoking one or more methods on an object
 - Methods will behave differently depending on the input stimulus
 - Implies that there is not always a one-to-one mapping between methods and behaviour
- The observable *interactions* between an object and its collaborators

Test Behaviour NOT Methods

Understanding Behaviour

- Consider a utility method to validate that a String can be parsed into an integer

```
public final class NumberUtils {  
  
    public static boolean isInteger(String text) {  
        try {  
            Integer.parseInt(text);  
        } catch (NumberFormatException e) {  
            return false;  
        }  
        return true;  
    }  
}
```

- What behaviour must we verify?
 - Should return `true` for integers
 - Should return `false` for non-integers

Understanding Behaviour

- Consider a sequencer that returns the values in the sequence 1, 2, 3, ...

```
public class SimpleSequencer implements ISequencer {  
    private int count = 0;  
  
    public int next() {  
        return count++; // several issues here!!  
    }  
}
```

- What behaviour must we verify?
 - Should return integer values, in sequence, starting at 1
 - How should it behave in a multi-threaded or even multi-node context?

100% Code Coverage
≠
100% Behavioral Coverage

Interaction-Based Behaviour

- Consider a method that returns void
 - How do we verify its behaviour if it has no observable result or internal state?

```
public class Registry {  
    public void destroy() {  
        dao.destroy(id);  
    }  
}
```

- Methods collaborate with other objects
 - Therefore verify behaviour by validating an object's interactions its neighbours
 - Mock-driven development

Naming Tests

- Test names should describe the behaviour being tested, not the method being tested or how it is implemented
 - *shouldUpdateAccountWhenDebited* ✓
 - *shouldWithdrawWhenSufficientFunds* ✓
 - *withdraw300Pounds()* ✗
- Change of emphasis: try starting names with the word '*should*' rather than '*test*'
 - *shouldPrintHelpWhenSuppliedInvalidOptions*
 - *shouldThrowExceptionWhenNoFreePorts*
 - *shouldNotRestartContainerAfterMaximumRetries*
 - *shouldRegisterActiveContainerOnStartup*

Behaviour-Bounded Design Increments

- Each test defines scope and complexity of a design step
 - You control the size and scope of each step - dictated by what you are confident you can implement
- The Ratchet Effect
 - Each new test that passes moves you closer to being done
 - Experiments are inexpensive because you only fall down so far

Step 2: Pick a Test

- Ask yourself:
 - What is the most important behaviour not yet implemented?
 - What is the simplest behaviour not yet implemented?
- Some behaviour will be dependent on other behaviour being in place
 - Natural ordering

Step 3: Write the Test

- Writing a test forces you to think about the design of the code from its intended use, *not* its implementation
 - ‘Design by Intention’
 - Tests focus on the interface of a unit
- Provides immediate feedback on a design’s usability and testability
 - Public API is based on needs not assumptions
 - Avoiding pollution of the API with unnecessary methods

Writing Tests - Example

- As ever, focus is on equivalence sets and boundary value analysis

```
@Test
public void shouldConfirmIsInteger() throws Exception {
    assertTrue(NumberUtils.isInteger(String.valueOf(Integer.MAX_VALUE)));
    assertTrue(NumberUtils.isInteger("0"));
    assertTrue(NumberUtils.isInteger(String.valueOf(Integer.MIN_VALUE)));
}
```

```
@Test
public void shouldConfirmNotInteger() throws Exception {
    assertFalse(NumberUtils.isInteger("123456H"));
    assertFalse(NumberUtils.isInteger("1234569879879879"));
    assertFalse(NumberUtils.isInteger("0.89"));
    assertFalse(NumberUtils.isInteger(String.valueOf(2L ^ 31)));
    assertFalse(NumberUtils.isInteger(String.valueOf((-2L ^ 31) - 1L)));
}
```

Tests Difficult to Write?

- Listen to what your tests are telling you
 - They provide valuable insight
- Tests that are 'accidentally complex' imply your code has
 - Too many dependencies
 - Too many responsibilities
 - Filled with duplication and redundancy
 - Scattered logic
- Fix through effective partitioning

Step 4: Run the Test

- See **RED!** – the test *should* fail
 - Since the behaviour has not yet been implemented
- A passing test may indicate a false positive
 - Something else is working but for the wrong reasons

Step 5: Implement the Behaviour

- Go GREEN...
- Do the least amount of work to get the test to pass
 - Don't worry about clean code just yet
 - The psychology of success – keep moving forward
- Focus only on implementing immediate requirements, not on future-proofing
 - Encourages simple, sufficient designs

Discovering New Behaviour

- Implementing a behaviour can lead to the discovery of new, lower level behaviours
 - Implemented internally or by an outgoing collaborator
- Collaborators and their behavioural contracts specified using interfaces
 - An object is said to behave correctly if
 - (a) it returns the correct result and
 - (b) it interacts with its neighbours as expected

Writing Tests Backwards

- Using TDD to 'discover interfaces' results in tests being written a different order:
 - Create test to verify behaviour, as standard
 - During implementation step, discover need for a new outgoing service
 - Define that service's interface and how it will be invoked by the object under test
 - Update test to mock out that service
 - Set and verify expectations against mock

Separating Behaviour From Construction

- Dependency injection plays vital role in enabling testability
- Deferring object wiring to an IoC framework (e.g. Spring) means
 - Creational activities separated from business logic
 - Developers focus on roles, responsibilities and behaviour
 - Removes layer of complexity from design
 - Framework manages lifecycle, wiring, etc

Step 6: Making it Clean

- Refactor only after you have the green bar passing
- Refactor both production and test code to..
 - Remove duplication
 - Remove redundancy
 - Clarify intent
 - Keep it simple
- No obvious defects, or obviously no defects?

Step 7: Run all Tests

- Ensure that your latest changes have not broken existing tests
- A failing test can mean one of 3 things
 - You broke the implementation. Go fix it.
 - The behaviour is still correct but has moved. Move the test.
 - The premise of the system has changed. Remove the test

Applying TDD

- Applying Test-Driven Development across large teams is a challenge
 - Requires discipline and skill
 - You cannot measure success through blameware alone (e.g. coverage reports)
- Help your cause through
 - Education and constant mentoring
 - Static code analysis
 - Benevolent dictatorship

Pragmatic TDD

- Once the discipline is engrained, feel free to tweak the process
 - Red/Green/Refactor or Red/Refactor/Green?
 - Be practicable, both work
 - Bouts of up-front design?
 - Don't feel guilty but limit the scope
 - Writing tests before, at same time or after implementation?
 - It's OK so long as you actually write the test

Bibliography

Kent Beck, *Test-Driven Development: By Example*, Addison-Wesley, 2003

Kent Beck, *Extreme Programming Explained: Embrace Change*, Addison-Wesley, 2005

Michael Feathers, *Working Effectively with Legacy Code*, Prentice Hall, 2005

Martin Fowler, *Refactoring: Improving the Design of Existing Code*, Addison-Wesley, 1999

Dan North, Introducing Behaviour Driven Development, <http://dannorth.net/introducing-bdd>

Freeman et al, *Mock Roles, not Objects*, <http://www.jmock.org/oopsla2004.pdf>

Gamma et al, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995

Kevlin Henney, *Test-Driven Development: Bare Bones & Basics*, JAOO presentation, 2004

Rod Johnson, *J2EE Development Without EJB*, Wiley, 2004

Jack Reeves, *What is Software Design*, http://www.developerdotstar.com/mag/articles/reeves_design.html