Oxiane INSTITUT

make it **clever**

Using TDD with XSLT to produce high quality code

What is high quality XSLT ?

Is this a high quality XSL code ?

```
<xsl:apply-templates/>

<xsl:if test="./ano:Anomalie">
  <xsl:call-template name="display-ano">
    <xsl:with-param name="ano" select="./ano:Anomalie"/>
    <xsl:with-param name="display-link" select="0"/>
  </xsl:call-template>
</xsl:if>

<xsl:if test="./n:DonneesIndiv/ano:Anomalie">
  <xsl:call-template name="display-ano">
    <xsl:with-param name="ano" select="./n:DonneesIndiv/ano:Anomalie"/>
    <xsl:with-param name="display-link" select="1"/>
    <xsl:with-param name="coll">
      <xsl:value-of select="./n:Employeur/n:Siret/@V"/>
    </xsl:with-param>
    <xsl:with-param name="budg">
      <xsl:value-of select="./n:Budget/n:Code/@V"/>
    </xsl:with-param>
    <xsl:with-param name="pk">
      <xsl:value-of select="./@added:primary-key"/>
    </xsl:with-param>
  </xsl:call-template>
</xsl:if>
```

Is this a high quality XSL code ?

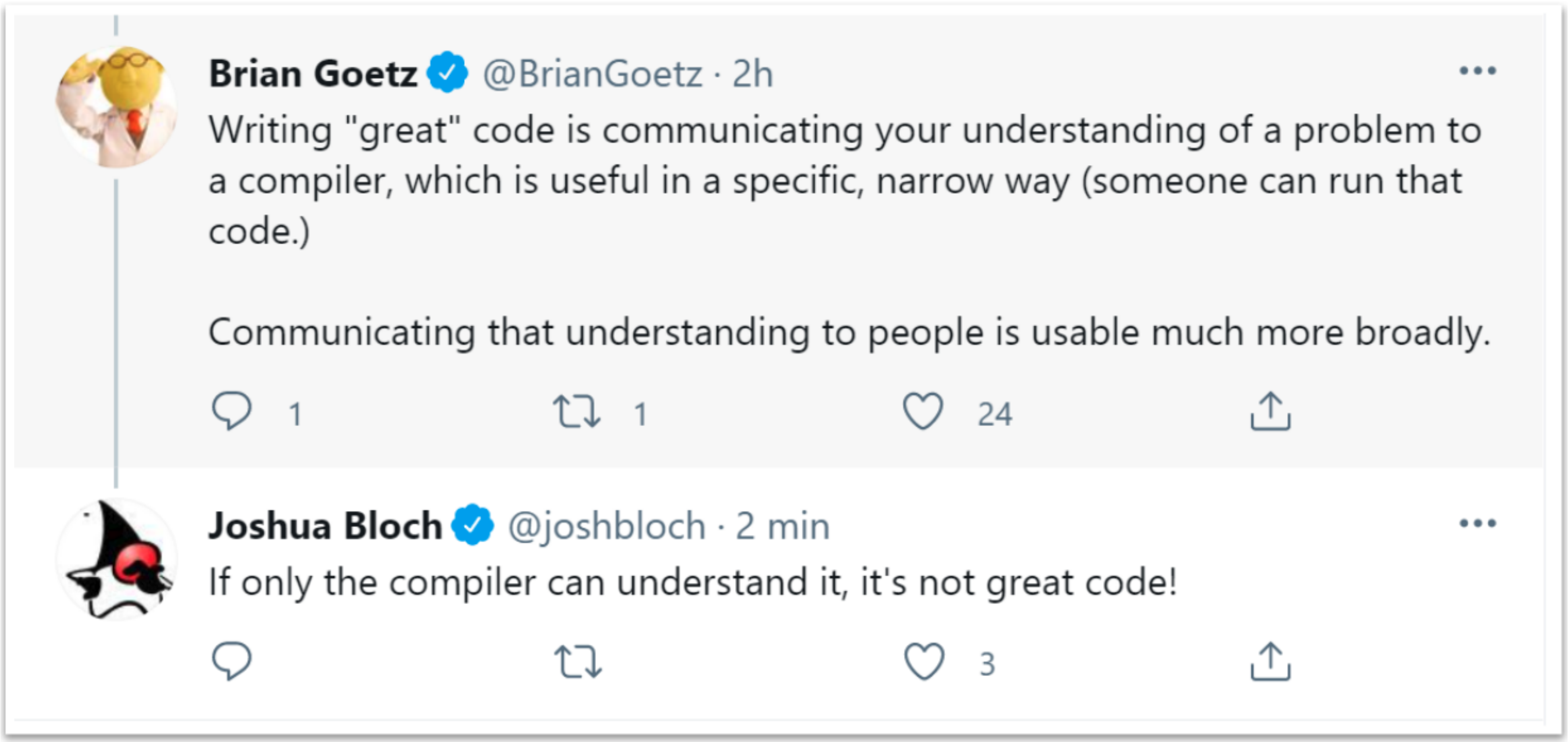
```xml
<xsl:template match="xsl:stylesheet | xsl:transform" as="element()+" mode="coverage-report">
  <xsl:variable name="ss-uri" as="xs:anyURI" select="base-uri()" />
  <xsl:variable name="ss-string" as="xs:string" select="unparsed-text($ss-uri)" />
  <xsl:variable name="lines" as="xs:string+" select="local:split-lines($ss-string)" />
  <xsl:variable name="number-of-lines" as="xs:integer" select="count($lines)" />
  <xsl:variable name="number-width" as="xs:integer"
      select="string-length(xs:string($number-of-lines))" />
  <xsl:variable name="number-format" as="xs:string"
      select="string-join(for $i in 1 to $number-width return '0')" />
  <xsl:variable name="module" as="xs:string?">
  <xsl:variable name="uri" as="xs:string"
      select="if (starts-with($ss-uri, '/')) then ('file:' || $ss-uri) else $ss-uri" />
    <xsl:sequence select="key('modules', $uri, $trace)/@id" />
  </xsl:variable>
  <h2>
    <xsl:text>module: {fmt:format-uri($ss-uri)}; {$number-of-lines} lines</xsl:text>
  </h2>
  <pre>
    <xsl:value-of select="format-number(1, $number-format)" />
    <xsl:text>: </xsl:text>
    <xsl:call-template name="output-lines">
      <xsl:with-param name="stylesheet-string" select="$ss-string" />
      <xsl:with-param name="number-format" select="$number-format" />
      <xsl:with-param name="module" select="$module" />
    </xsl:call-template>
  </pre>
```

High Quality code
Definition

- **A High Quality code is**
  - Reliable
    - the code behaves as expected

High Quality code Definition

- **A High Quality code is**
  - Reliable
    - the code behaves as expected
  - Readable and understandable
    - by a human, not only by a XSLT Processor



Brian Goetz @BrianGoetz · 2h
Writing "great" code is communicating your understanding of a problem to a compiler, which is useful in a specific, narrow way (someone can run that code.)

Communicating that understanding to people is usable much more broadly.

1 1 24

Joshua Bloch @joshbloch · 2 min
If only the compiler can understand it, it's not great code!

3

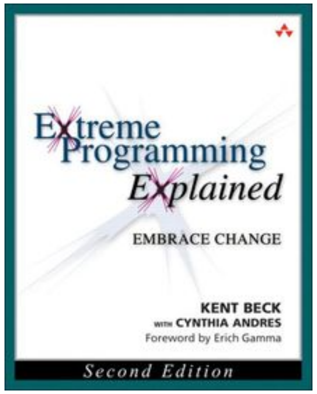
High Quality code Definition

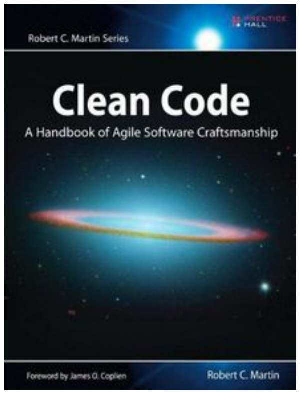
- **A High Quality code is**
  - Reliable
    - the code behaves as expected
  - Readable and understandable
    - by a human, not only by a XSLT Processor
  - Maintainable
    - with a constant cost

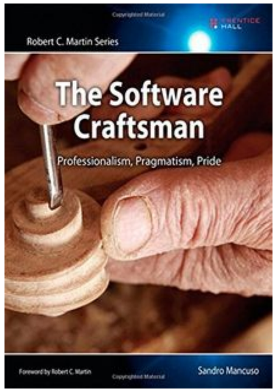
How to produce high quality code ?

How to produce high quality code ?

- **Extreme Programming, Clean Code, Software Craftsmanship**
  - Methods, technics and attitudes that produce High Quality code

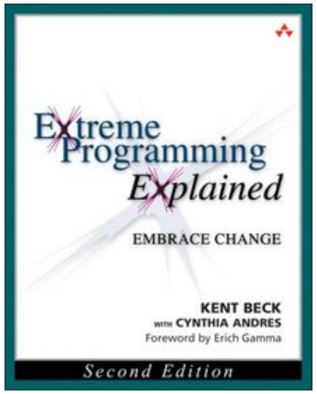


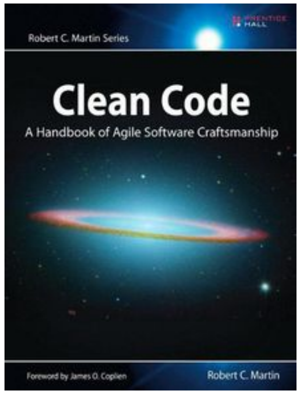

Robert C. Martin Series
Clean Code
A Handbook of Agile Software Craftsmanship
Foreword by James O. Coplien
Robert C. Martin

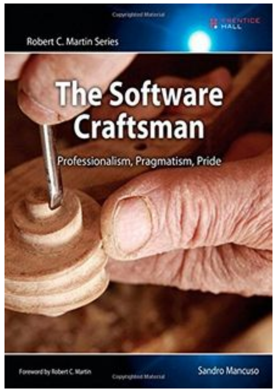


How to produce high quality code ?

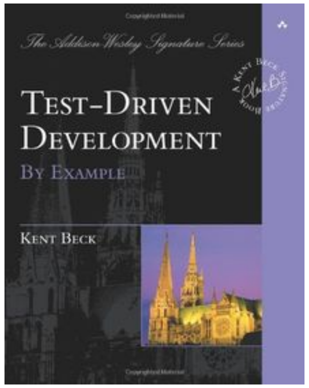
- **Extreme Programming, Clean Code, Software Craftsmanship**
  - Methods, technics and attitudes that produce High Quality code




Robert C. Martin Series
PRENTICE HALL
Clean Code
A Handbook of Agile Software Craftsmanship
Foreword by James O. Coplien
Robert C. Martin

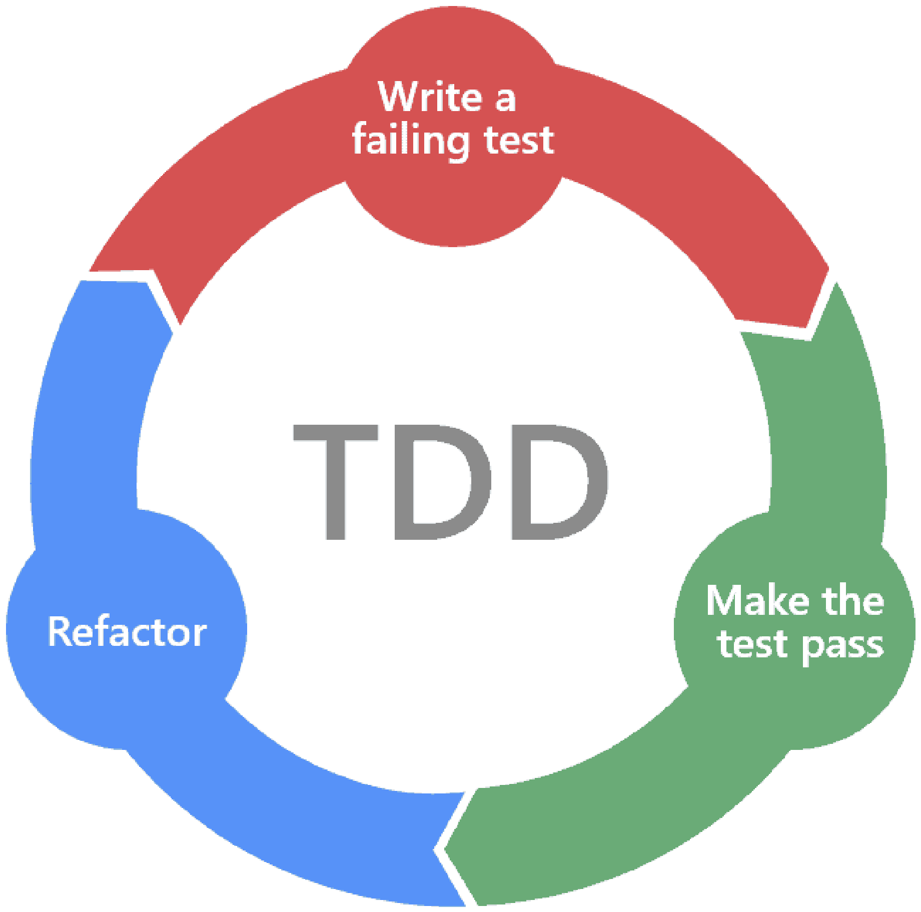


- **Test Driven Development**
  - Introduced by Kent Beck in Extreme Programming Explained


The Addison-Wesley Signature Series
TEST-DRIVEN DEVELOPMENT
BY EXAMPLE
KENT BECK
A Kent Beck Signature Book

Test Driven Development

- **A loop method**

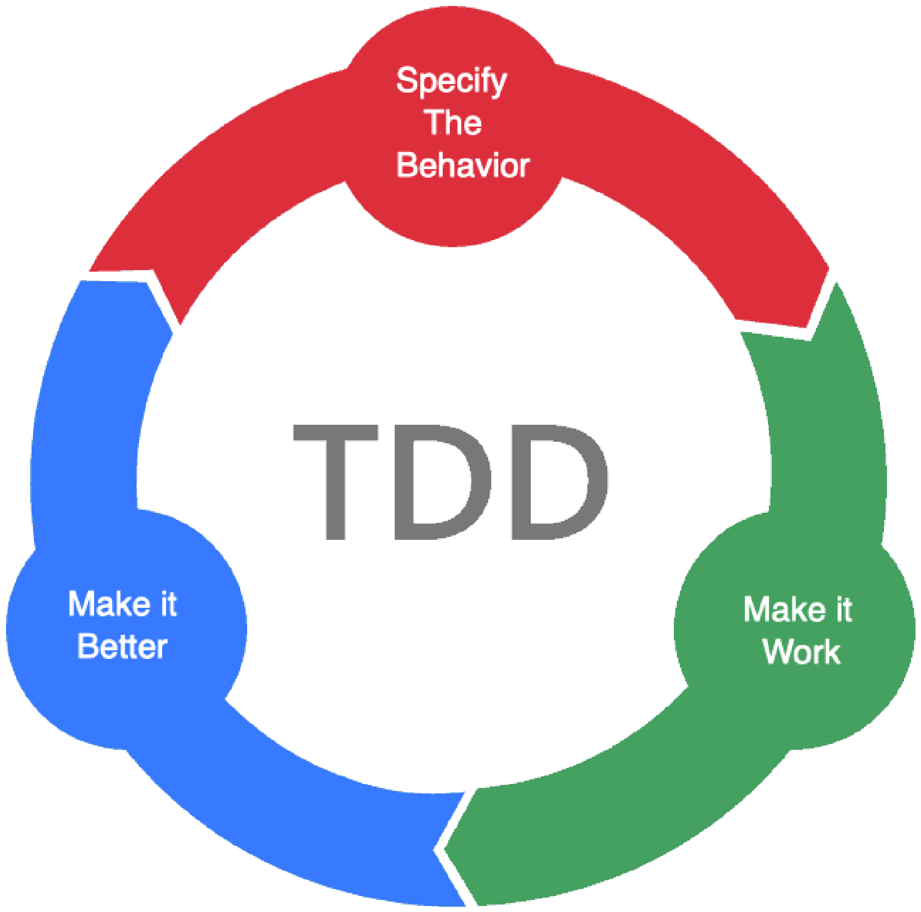


Test Driven Development

- **Write a failing test**
  - That describes behavior and expectations
  - It must either not compile, or fail
- **Make the test pass**
  - Write minimal code to make the test succeed
  - Really the minimum
  - The test - and all existing tests - must succeed
- **Refactor**
  - Eliminate duplication
  - Add readability
  - Introduce design
  - Check that all tests succeed

Test Driven Development

- **But also a workflow process**



Test Driven Development

- **Specify the behavior**
  - Explain how to use the code
  - Explain which behavior is expected
- **Make it run**
  - Write minimal code that implements the behavior
  - Check the test passes, and all tests succeed
- **Make it Better**
  - Do not add code, refactor by using IDE's refactoring facilities
    - Or apply refactoring methods described in Refactoring book (Martin Fowler)
  - Remove duplication
  - Add readability
  - Add understandability
  - Add design and patterns

Test Driven Development

- **Process by baby steps**
  - Write a test that is just the next baby step after the previous one
- **When implementing a RPN calculator, tests should be**
  - Given expression **1**, when calculating result should be **1**
  - Given expression **2**, when calculating result should be **2**
  - Given expression **3**, when calculating result should be **3**
  - Given expression **1 1 +**, when calculating result should be **2**
  - Given expression **1 2 +**, when calculating result should be **3**

- **Use corks when definitive implementation can not be found**

XSLT 4.0

```
<xsl:function name="f:calculate" as="xs:double">
  <xsl:param name="expression" as="xs:string"/>
  <xsl:switch select="$expression">
    <xsl:when test="'1'" select="1.0"/>
    <xsl:when test="'2'" select="2.0"/>
    <xsl:when test="'3'" select="3.0"/>
    <xsl:otherwise select="error('Unsupported expression')"/>
  </xsl:switch>
</xsl:function>
```

Test Driven Development

- **Refactor to eliminate duplication**
  - Without changing code behavior

XSLT 4.0

```
<xsl:function name="f:calculate" as="xs:double">
  <xsl:param name="expression" as="xs:string"/>
  <xsl:if
      test="f:isNumber($expression)"
      then="number($expression)"
      else="error('Unsupported expression')"/>
  </xsl:if>
</xsl:function>
```

  - This removes corks
  - Use `extract function` refactoring operation to introduce business concepts
- **Apply Clean Code principles**
- **Respect SOLID principles**
  - Single Responsability Principle
  - Open Close Principle

OXiane use case

OXiane courses

- **At OXiane, we write courses**
  - With a slide-deck
  - With an exercise book
- **We want to write our courses in a text format**
  - Many people should be able to work together on the same course
- **Exercise book are written in MarkDown**

```
# XSLT 4.0

This exercise book contains all exercises for the XSLT4 course.
Solutions are located at the end of this book.

## XPath 4.0

### New array functions

- Construct a new array that contains many `xs:double`
  - try to use the new `array:of` function
  - use it with 1 argument
  - use it with 5 arguments
  - use it with 7 arguments
```

OXiane courses

- **We decided to write our own tool**
  - To control exactly the syntax used by writers
- **We need a tool that transforms from Markdown to HTML**
  - It must be reliable
  - It must be easy to maintain
  - It must be self documented
- **OXiane business is to sell courses**
  - Not to maintain tools
  - So we need a High Quality tool !

Live Coding !

Live Coding

- **Implement a Markdown to XML transformer**
  - With reduced syntax
- **Here, limited to**
- **Title level 1**
  - `# This is the exercise book title`
  - Expected : `<title>This is the exercise book title</title>`
- **Title level 2**
  - `## This is the chapter title`
  - Expected : `<chapter>This is the chapter title</chapter>`

Pros & Cons

Pros & Cons

- **Business intents are described by tests**
  - From a business point of view, not from a code point of view
- **All production code is required by business rules**
  - As we write the minimal code to make the test pass
- **Code can be read as a book**
  - From top to bottom
- **All code is covered by unit tests**
  - Even if, for Markup, it's not enough
- **Code can be delivered each time tests are green**
  - Which improves client's benefits
- **Developers can modify code with serinity**
  - Unit tests indicate if code behavior has been changed

Pros & Cons

- **Coding is longer**
  - We have to write tests
  - We have to refactor
- **But it costs less**
  - Code is much more robust
  - Adding new features is simpler, and cheaper
- **Tests have to be maintained**
  - When a new feature is defined, some of exiting tests have to be corrected

Thanks for you attention
Questions ?