



Odisee  
DE CO-HOGESCHOOL

# Pilaren van een goede test



Jens Baetens

## 4 Pilaren van een goede test

- ▣ Protectie tegen regressie → *uitbreiden*
- ▣ Resistent voor refactoring → *aanpassing*
- ▣ Snelle feedback
- ▣ Onderhoudbaar (*leesbaar*)

## Pilaar 1: Protectie tegen regressie

- ▣ “Code is not an asset, it’s a liability”
- ▣ Grotere code bases hebben meer risico’s op bugs
  - Belangrijk om je hiertegen te beschermen
- ▣ Hoe goed scoren we op deze pilaar?
  - Hoeveelheid code is uitgevoerd tijdens testing
  - De complexiteit van de applicatie
  - Domein Significantie


*code coverage*

## Pilaar 2: Resistent voor refactoring

- ▣ Hoe goed een test wijzigingen in de onderliggende code kan verdragen
- ▣ Wanneer een test faalt zonder reden => false positive
  - ▣ Belangrijk om dit te vermijden
    - Verliezen van vertrouwen in de tests
    - Verliezen van motivatie om tests/code na te kijken, true positives worden hierdoor niet gedetecteerd
- ▣ False positives zijn een indicatie van hoe goed we scoren op deze pilaar
  - ▣ Vaak een resultaat van te gedetailleerde testen op implementatie details

## False positives example

```
1 public class Message
2 {
3     public string Header { get; set; }
4     public string Body { get; set; }
5     public string Footer { get; set; }
6 }
7
8 public abstract class Renderer
9 {
10     string Render(Message message);
11 }
12
13 public class MessageRenderer : Renderer
14 {
15     public IReadOnlyList<Renderer> SubRenderers { get; }
16
17     public MessageRenderer()
18     {
19         SubRenderers = new List<Renderer>
20         {
21             new HeaderRenderer(),
22             new BodyRenderer(),
23             new FooterRenderer()
24         };
25     }
26
27     public string Render(Message message)
28     {
29         return SubRenderers
30             .Select(x => x.Render(message))
31             .Aggregate("", (str1, str2) => str1 + str2);
32     }
33 }
```



```
1 public void MessageRenderer_uses_correct_sub_renderers()
2 {
3     var sut = new MessageRenderer();
4
5     IReadOnlyList<Renderer> renderers = sut.SubRenderers;
6
7     Assert.Equal(3, renderers.Count);
8     Assert.IsAssignableFrom<HeaderRenderer>(renderers[0]);
9     Assert.IsAssignableFrom<BodyRenderer>(renderers[1]);
10    Assert.IsAssignableFrom<FooterRenderer>(renderers[2]);
11 }
```

Wat is hier verkeerd?

## False Positives example

```
1 public class Message
2 {
3     public string Header { get; set; }
4     public string Body { get; set; }
5     public string Footer { get; set; }
6 }
7
8 public abstract class Renderer
9 {
10     string Render(Message message);
11 }
12
13 public class MessageRenderer : Renderer
14 {
15     public IReadOnlyList<Renderer> SubRenderers { get; }
16
17     public MessageRenderer()
18     {
19         SubRenderers = new List<Renderer>
20         {
21             new HeaderRenderer(),
22             new BodyRenderer(),
23             new FooterRenderer()
24         };
25     }
26
27     public string Render(Message message)
28     {
29         return SubRenderers
30             .Select(x => x.Render(message))
31             .Aggregate("", (str1, str2) => str1 + str2);
32     }
33 }
```


Wat is hier verkeerd?

```
1 public void MessageRenderer_is_implemented_correctly()
2 {
3     string sourceCode = File.ReadAllText(@"
4     <project path>\MessageRenderer.cs");
5
6     Assert.Equal(
7         @"
8     public class MessageRenderer : IRenderer
9     {
10         public IReadOnlyList<IRenderer> SubRenderers { get; }
11
12         public MessageRenderer()
13         {
14             SubRenderers = new List<IRenderer>
15             {
16                 new HeaderRenderer(),
17                 new BodyRenderer(),
18                 new FooterRenderer()
19             };
20
21         public string Render(Message message)
22         {
23             return SubRenderers
24                 .Select(x => x.Render(message))
25                 .Aggregate("", (str1, str2) => str1 + str2);
26         }
27     }", sourceCode);
28 }
```

## False positives example

```
1 public class Message
2 {
3     public string Header { get; set; }
4     public string Body { get; set; }
5     public string Footer { get; set; }
6 }
7
8 public abstract class Renderer
9 {
10     string Render(Message message);
11 }
12
13 public class MessageRenderer : Renderer
14 {
15     public IReadOnlyList<Renderer> SubRenderers { get; }
16
17     public MessageRenderer()
18     {
19         SubRenderers = new List<Renderer>
20         {
21             new HeaderRenderer(),
22             new BodyRenderer(),
23             new FooterRenderer()
24         };
25     }
26
27     public string Render(Message message)
28     {
29         return SubRenderers
30             .Select(x => x.Render(message))
31             .Aggregate("", (str1, str2) => str1 + str2);
32     }
33 }
```

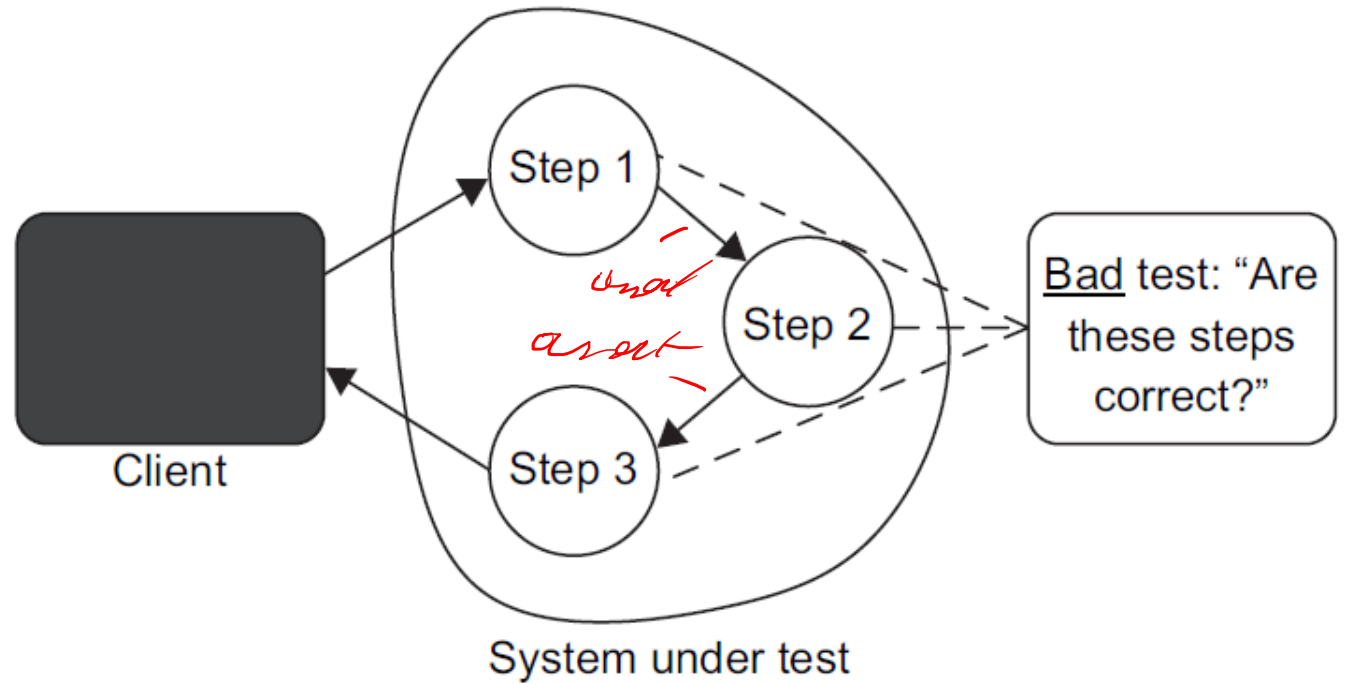
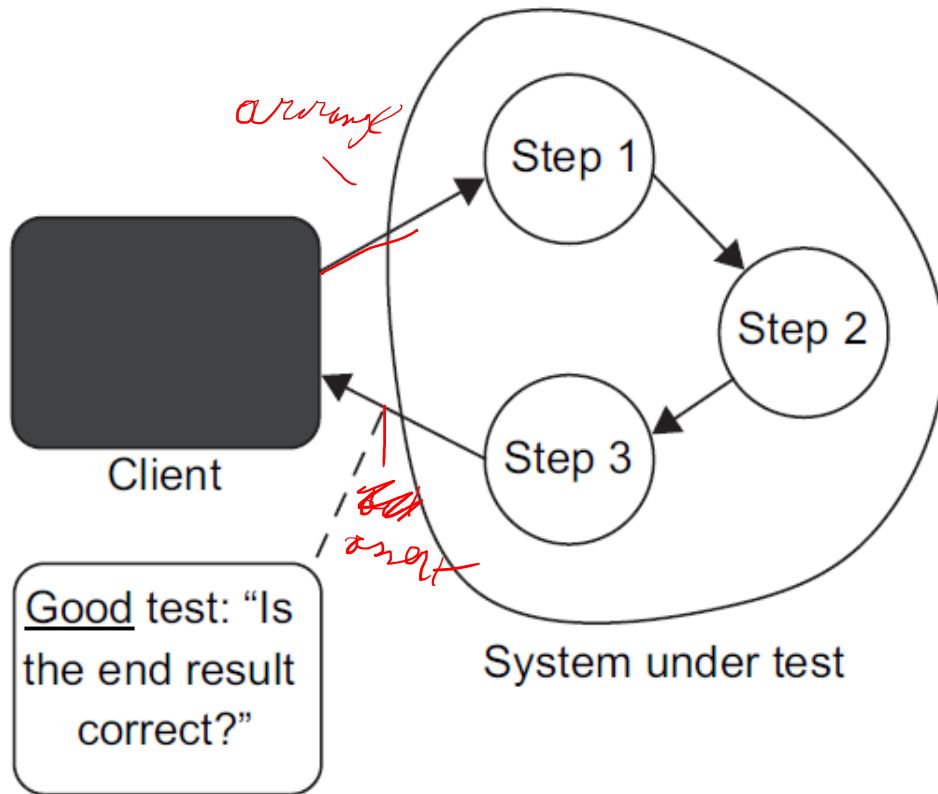
Wat is hier goed ?



```
1 public void Rendering_a_message()
2 {
3     var sut = new MessageRenderer();
4     var message = new Message
5     {
6         Header = "h",
7         Body = "b",
8         Footer = "f"
9     };
10
11     string html = sut.Render(message);
12
13     Assert.Equal("<h1>h</h1><b>b</b><i>f</i>", html);
14 }
```



## False positives vermijden



## Pilaar 3: Snelle feedback

- ▣ Hoe sneller tests zijn, hoe meer we er kunnen hebben en hoe vaker we ze kunnen uitvoeren
- ▣ Snelle tests => snelle feedback
  - => Hoe gemakkelijker/goedkoper om ze te verbeteren
- ▣ Trage tests moedigen niet aan om ze regelmatig te laten lopen

## Pilaar 4: Onderhoudbaar

### ■ Hoe moeilijk is het om de test te begrijpen

*Lijna  
coole* ■ Lengte van een test is belangrijk

■ Hoe korter de test, hoe begrijpbaarder

■ Maar niet artificieel kort maken, leesbaarheid is nog steeds belangrijk

### ■ Hoe moeilijk is het om de test uit te voeren?

■ Afhankelijkheden moeten onderhouden worden

■ Dependency Injection

■ Cleanup nodig na het uitvoeren van de test?

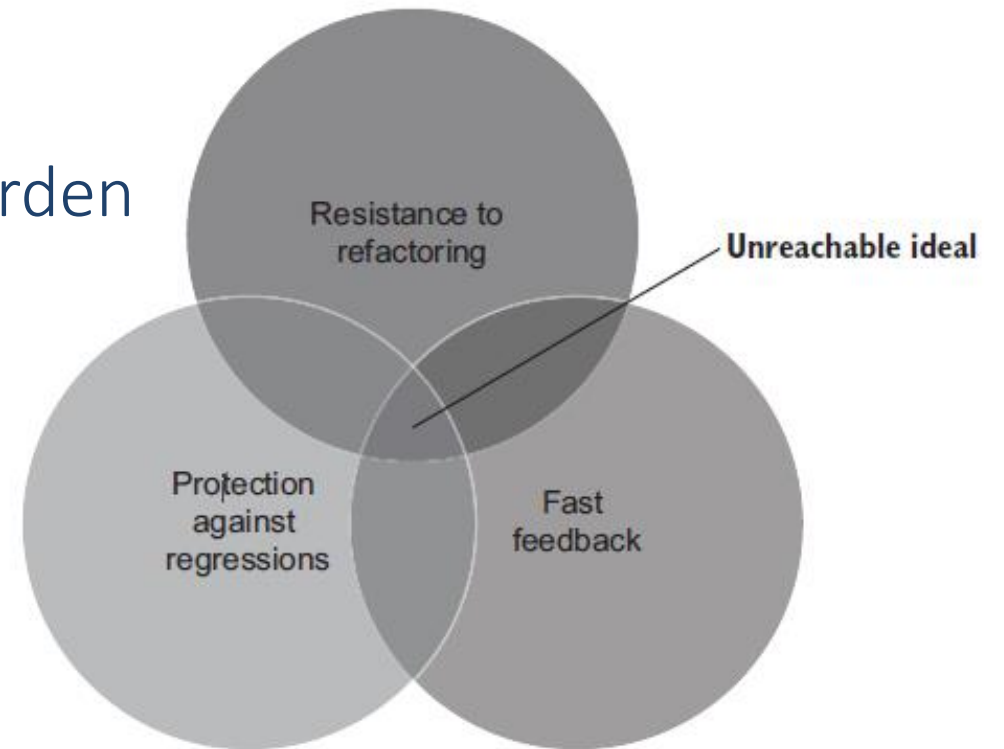
*↳ geen substitute → terug leggen database na testen*

## Op zoek naar de ideale test

- ▣ De ideale test scoort op alle 4 de pijlers goed
- ▣ Value estimate =  $\text{score}(\text{RegressionProtection}) * \text{score}(\text{RefactoringResistance}) * \text{score}(\text{FastFeedback}) * \text{score}(\text{Maintainability})$
- ▣ Value estimate =  $[0..1] * [0..1] * [0..1] * [0..1]$
- ▣ Wanneer 1 van de scores 0 is, is de test waardeloos

## Maar deze bestaat niet

- ▣ De eerste 3 pijlers conflicteren met elkaar
  - Kunnen niet tegelijkertijd optimaal zijn
- ▣ De 4e pijler kan steeds geoptimaliseerd worden

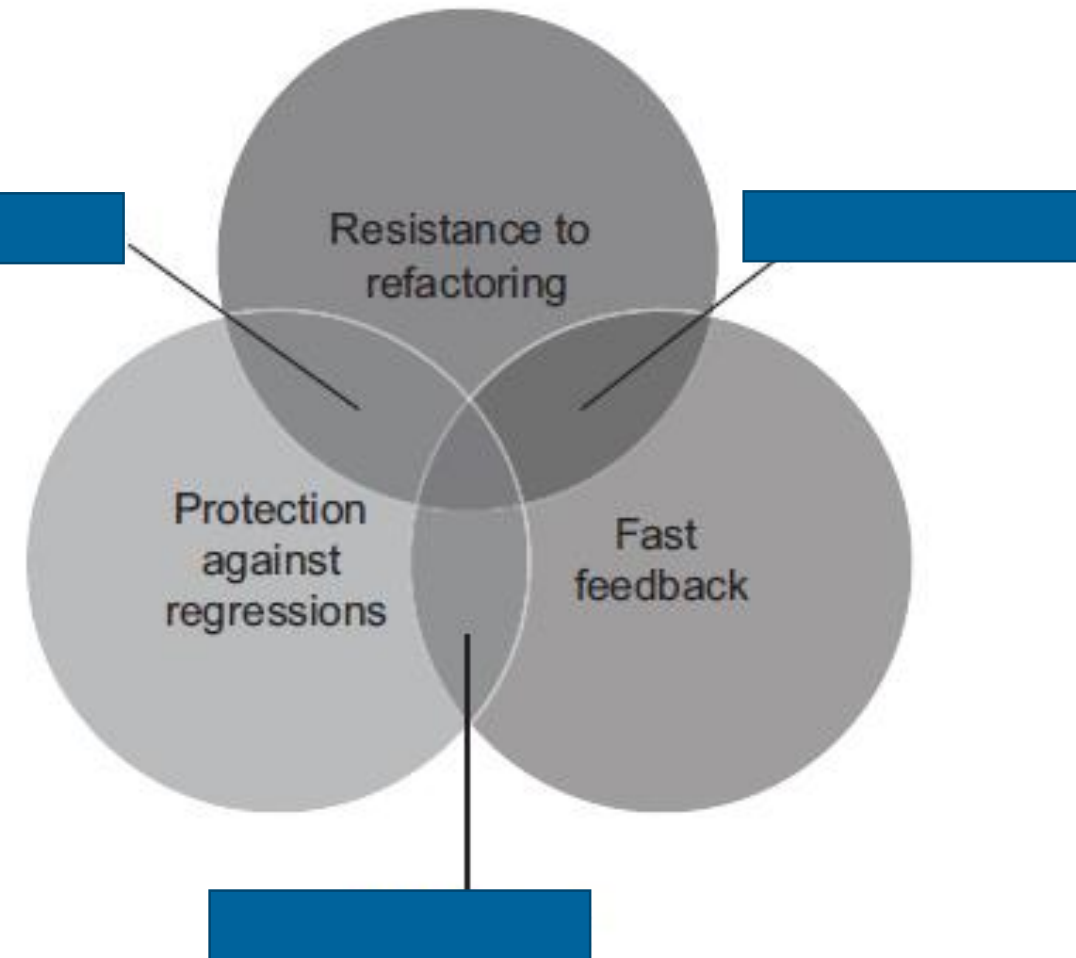


## End-to-End tests

- ▣ Zijn van nature traag
- ▣ Groot code pad doorlopen
- ▣ Kijken enkel naar input/output

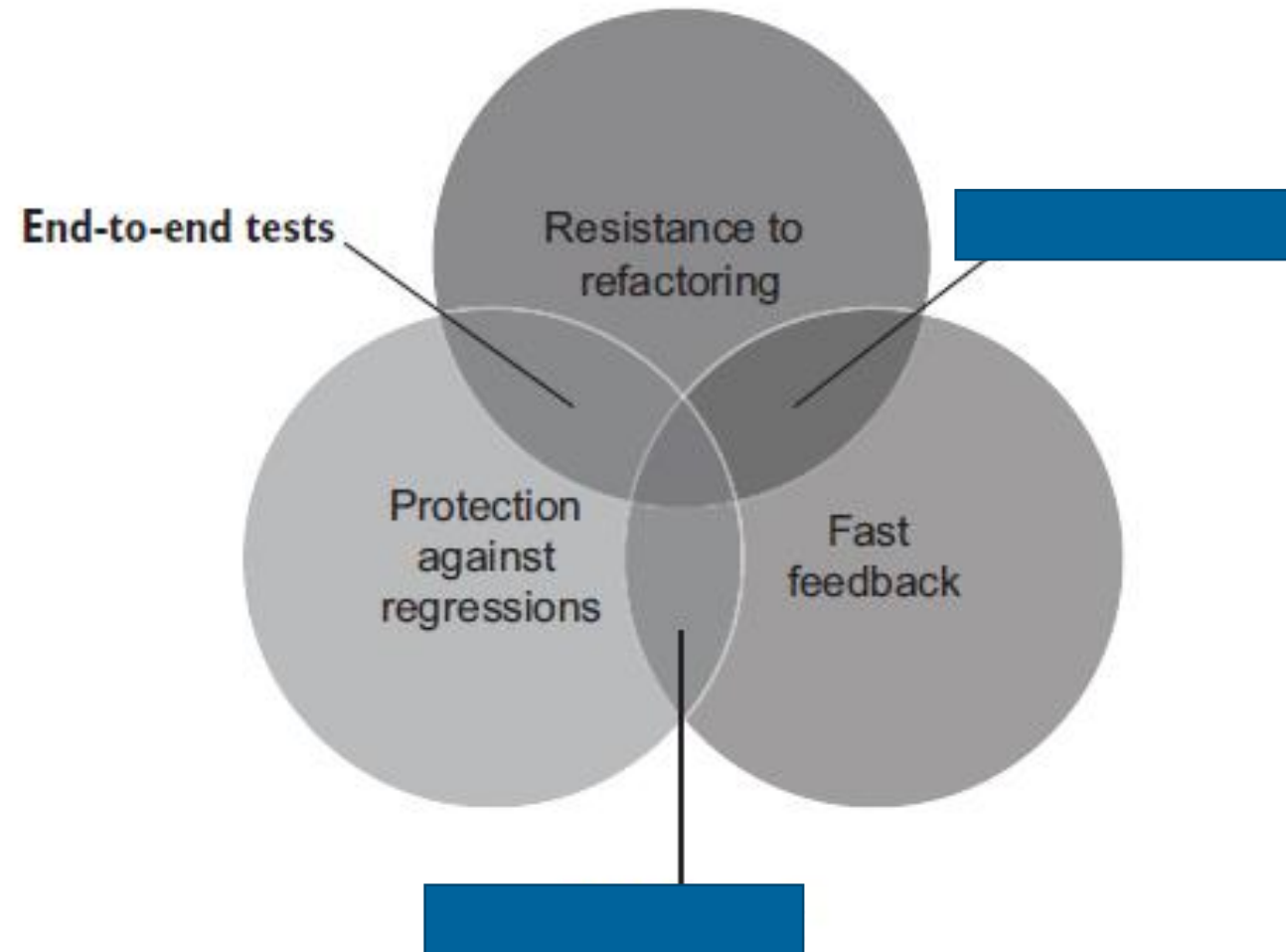
↳ goed voor refactoring

goed voor regressies!



## End-to-End tests

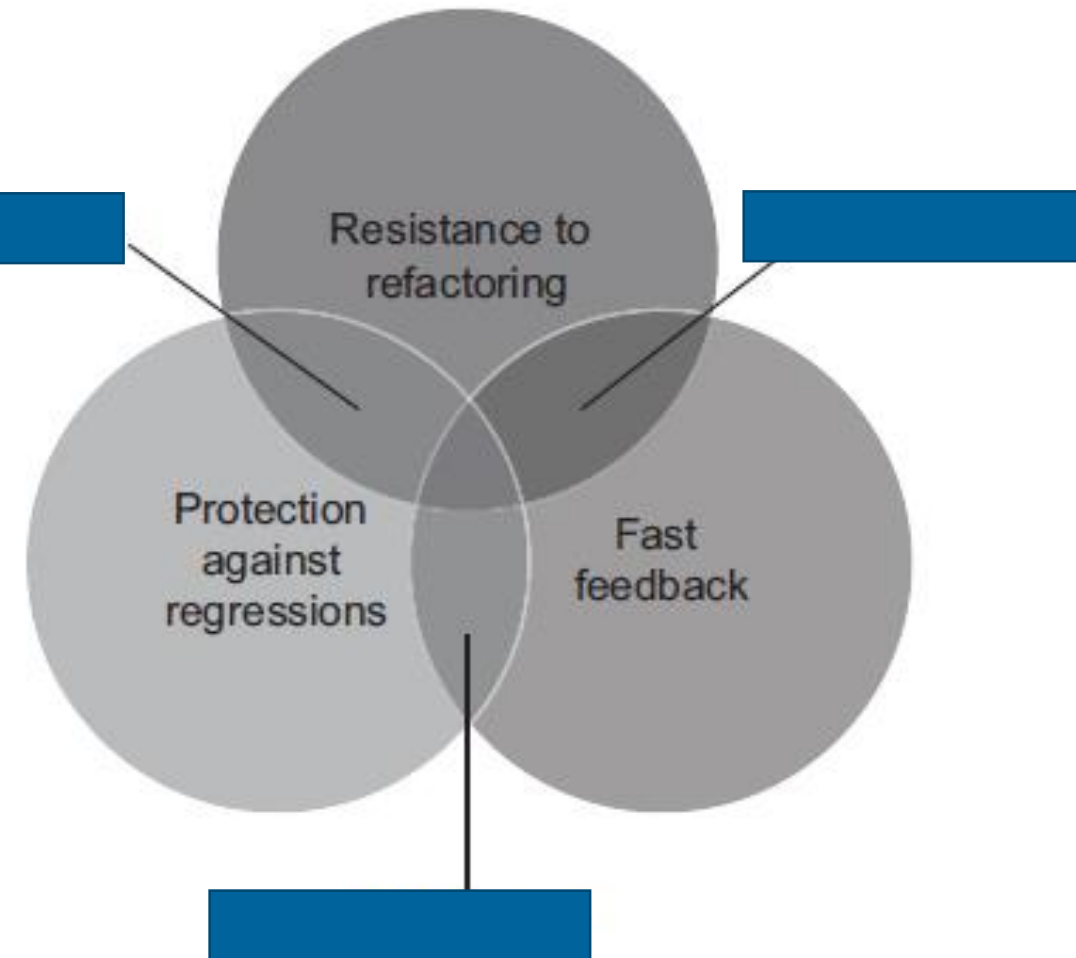
- ▣ Zijn van nature traag
- ▣ Groot code pad doorlopen
- ▣ Kijken enkel naar input/output



## Trivial tests

- ▣ Simpele testen
- ▣ Zeer snel uit te voeren
- ▣ Kleine kans op false positive
- ▣ Weinig complexiteit
  - ▬ Bijna boilerplate

```
1
2 public class User
3 {
4     public string Name { get; set; }
5 }
6
7 [Test]
8 public void Test()
9 {
10     var sut = new User();
11     sut.Name = "John Smith";
12     Assert.Equal("John Smith", sut.Name);
13 }
```

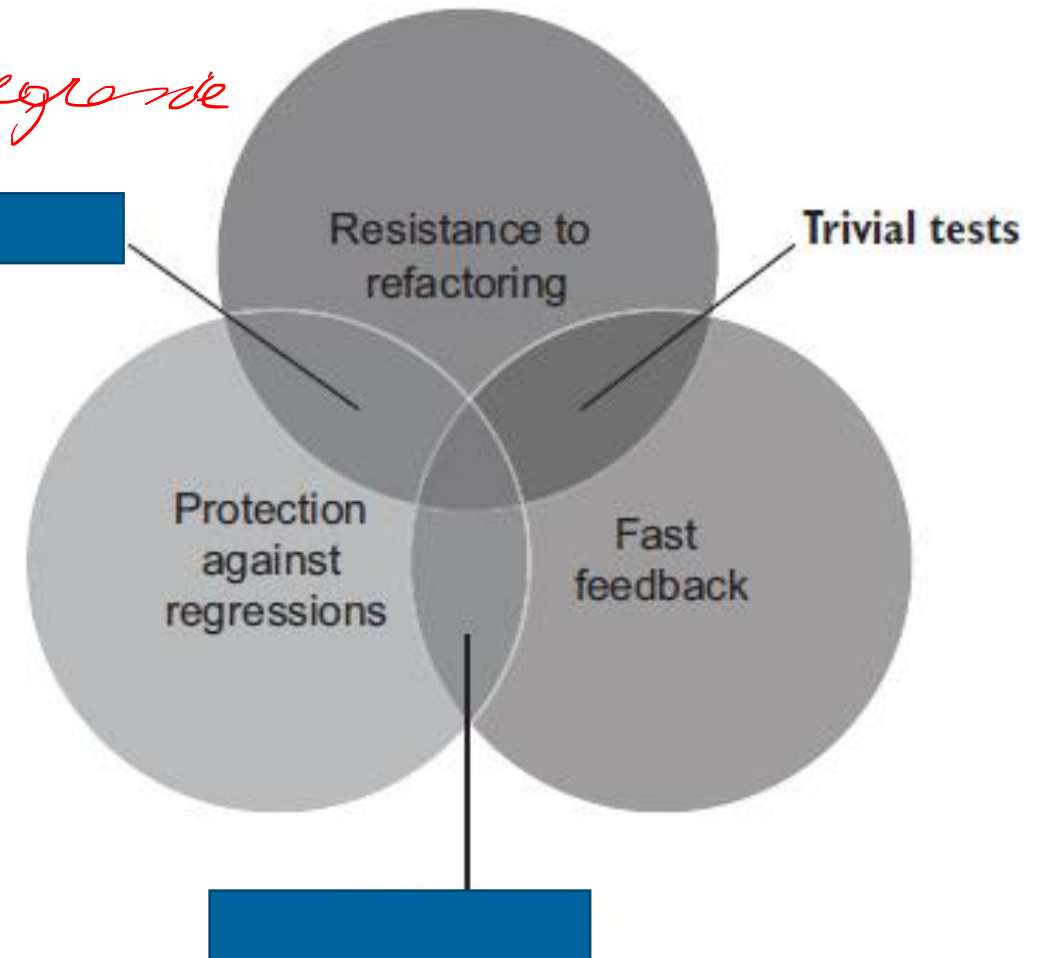




# Trivial tests

- ▣ Simpele testen
- ▣ Zeer snel uit te voeren
- ▣ Kleine kans op false positive
- ▣ Weinig complexiteit
  - ▣ Bijna boilerplate

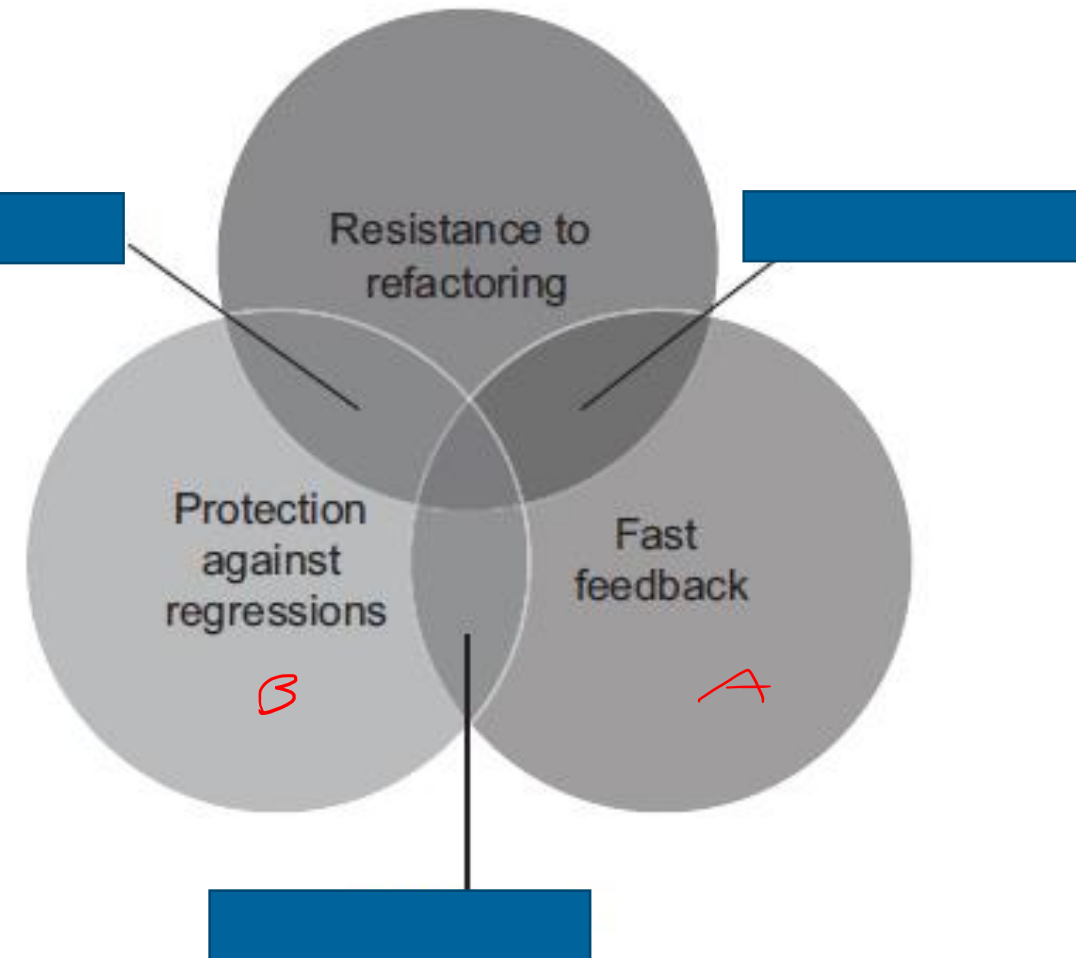
*weinig code*  
*→ niet goed voor regressie*



```
1 public class User
2 {
3     public string Name { get; set; }
4 }
5
6
7 [Test]
8 public void Test()
9 {
10     var sut = new User();
11     sut.Name = "John Smith";
12     Assert.Equal("John Smith", sut.Name);
13 }
```

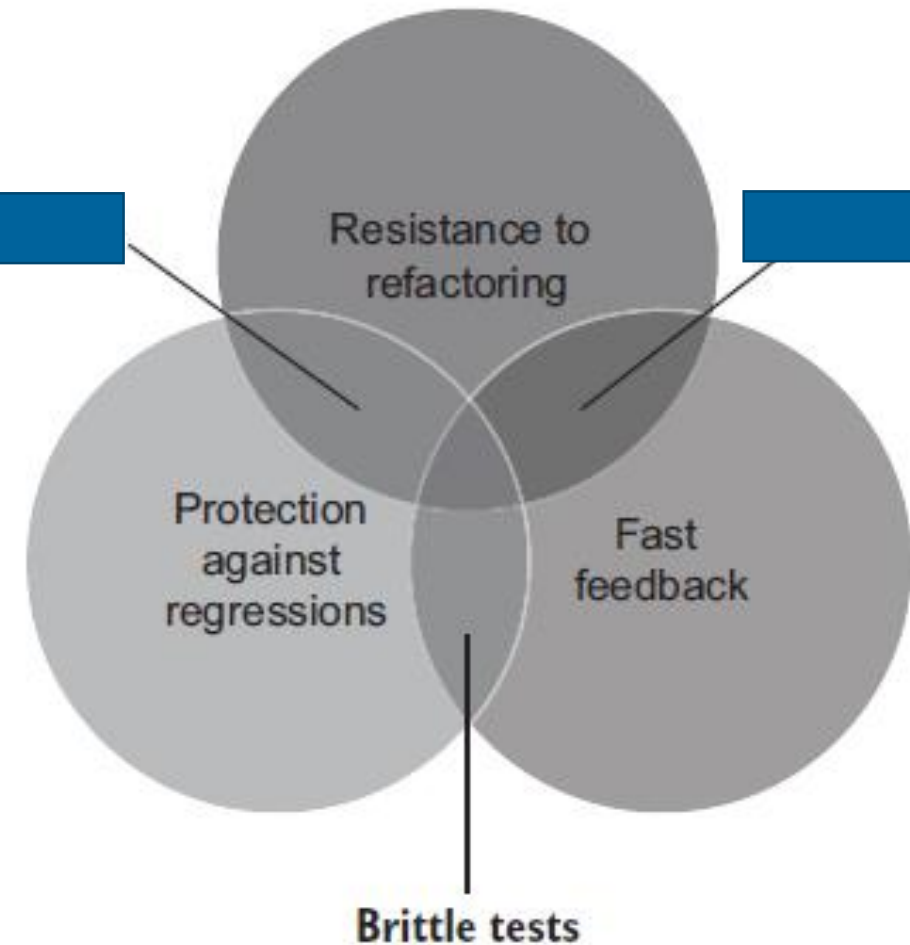
## Brittle tests / makkelijk breekbare testen

- Een <sup>A</sup>snelle test (zoals een trival test)
- Getest code voldoende <sup>B</sup>complex
- Vb: SQL-query kan op verschillende manieren geschreven worden



## Brittle tests / makkelijk breekbare testen

- ▣ Een snelle test (zoals een trival test)
- ▣ Getest code voldoende complex
- ▣ Vb: SQL-query kan op verschillende manieren geschreven worden



## Goede vs slechte unit tests

- ▣ Niet testen omdat het moet
- ▣ Slechte tests zijn op termijn even slecht als geen tests

