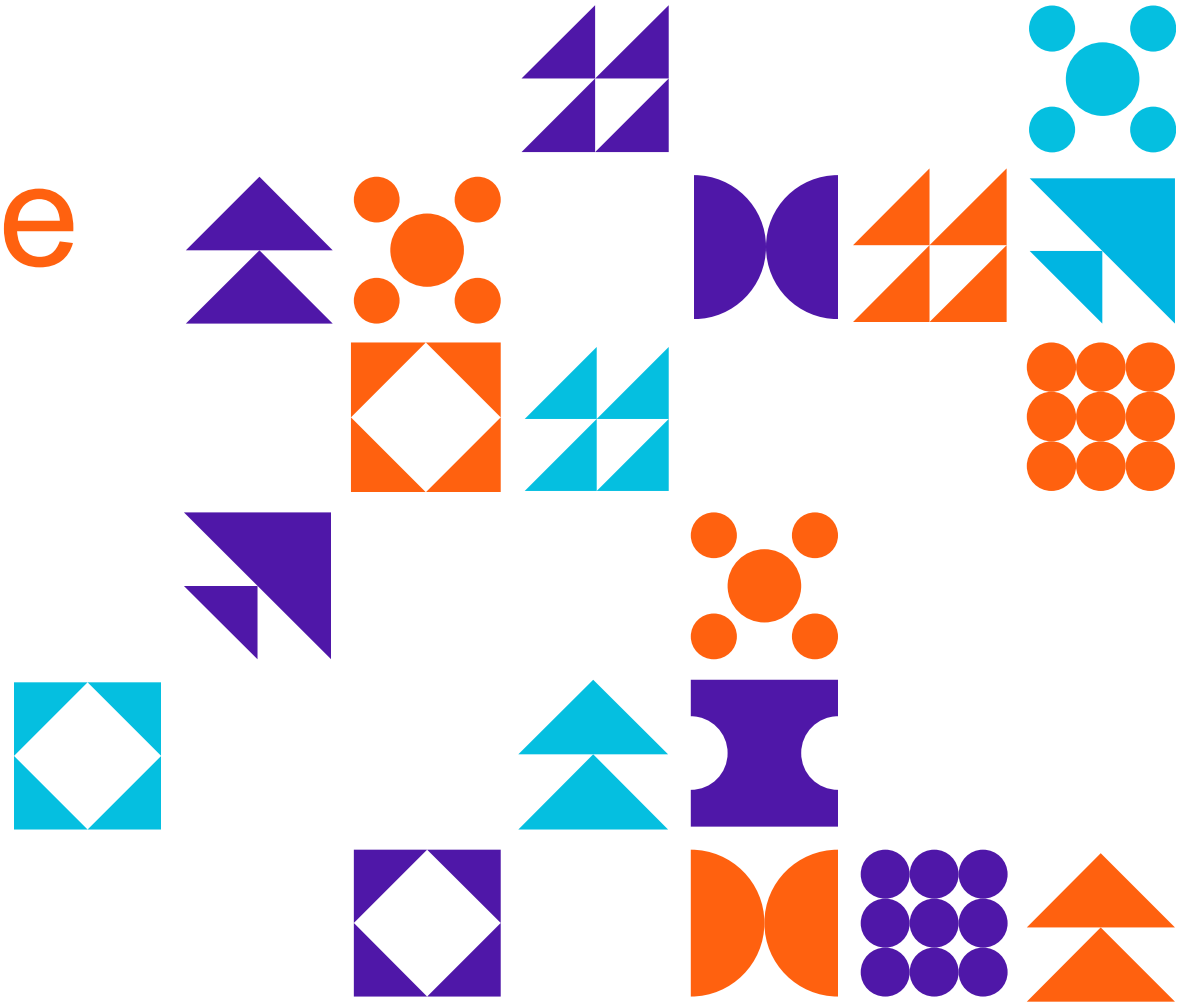
Specifying the Negative in TDD



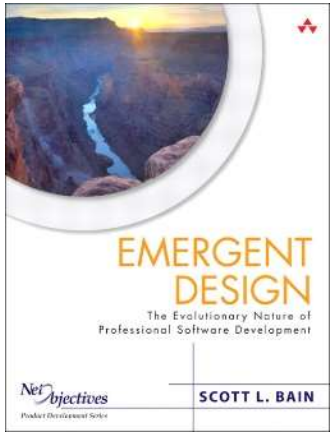
Disciplined Agile

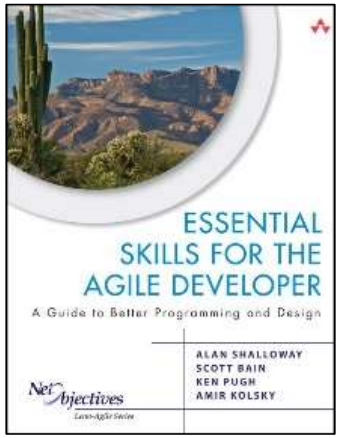
Scott Bain

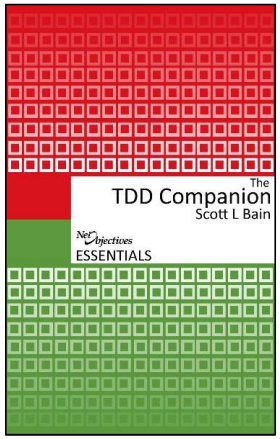
Senior Technical Trainer

Scott.Bain@PMI.org

Agile Development
Emergent Design, Design Patterns, OO Analysis and Modeling,
Test-Driven Development, Acceptance Test-Driven Development







A Joke: Tacos are Imaginary

tan = sin / cos (definition of tangent)

ta = i / co

x x

co co

taco = i

Falsifiability

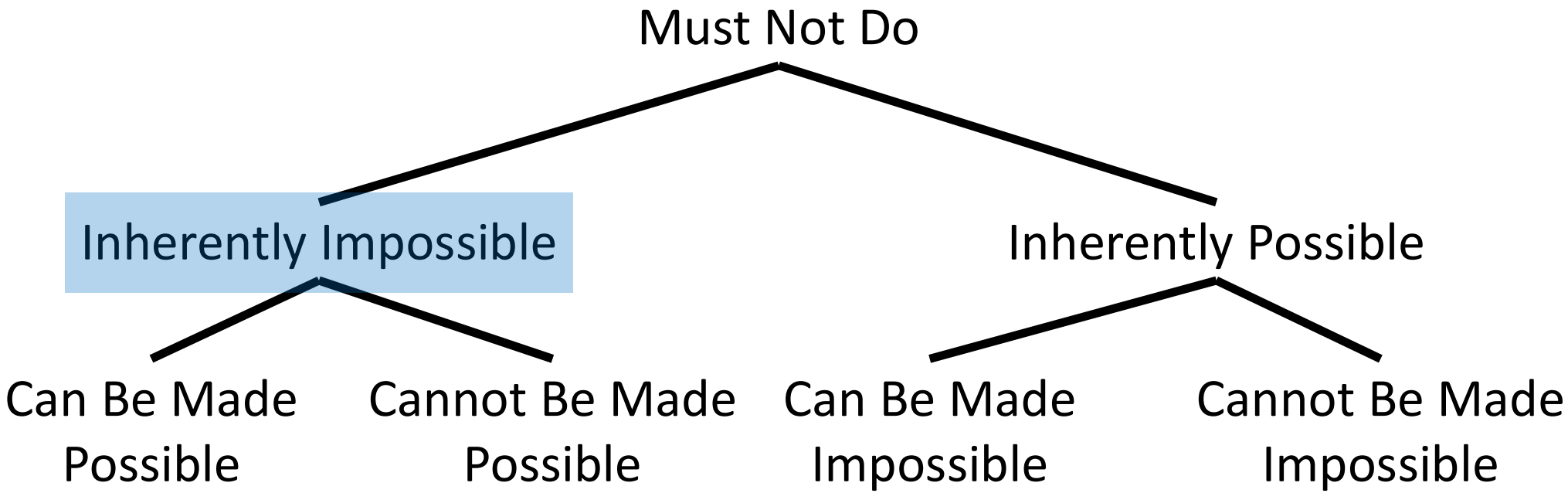
Falsifiability is a standard of evaluation of scientific theories and hypotheses that was introduced by the philosopher of science Karl Popper in his book The Logic of Scientific Discovery (1934). He proposed it as the cornerstone of a solution to both the problem of induction and the problem of demarcation. A theory or hypothesis is falsifiable (or refutable) if it can be logically contradicted by an empirical test that can potentially be executed with existing technologies.

https://en.wikipedia.org/wiki/Falsifiability

Behavior

- What TDD specifies is behavior
- Most behaviors are about what business value the stakeholders want from the system, and are willing to pay for
- Sometimes, however, they are about things the stakeholders want to guard against
- These are the “don’t do” rules
- How do we specify that a system **must not** do something?

The Graph of Negatives



Inherently Impossible

1. Technology-dependent

2. Impossible, and cannot be made possible:
   - Writing to a non-writable CD-ROM, or read-only memory
   - Directly reading/writing a field that has been made “private”*

3. Impossible, and can be made possible:
   - An immutable object

* Assuming the technology has the right idiom

Simple Example

- Your customer wants to capture an online “Sale Amount” for some kind of e-tail site
- They wants this amount to be accurately retrievable

Customer Requirement

"The value can be retrieved" is a **positive requirement**

Given:

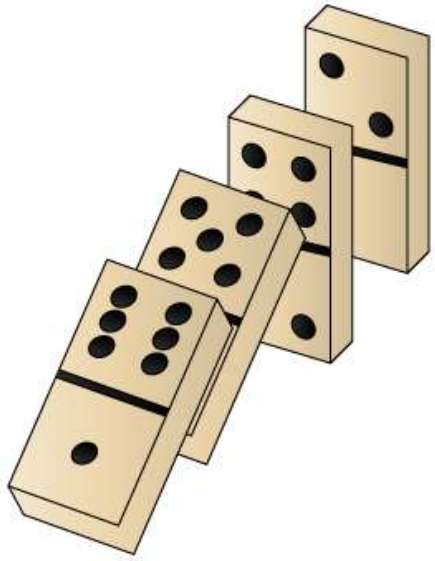
- SaleAmount S exists with value V

When:

- The value of S is requested

Then:

- V is returned



https://dannorth.net/introducing-bdd/

The Executable Specification

```
public class SaleAmountTest {
      @Test public void testSaleAmountCanBeRetieved() {
          double someAmount = 10.00;

          SaleAmount testSaleAmount =
                    new SaleAmount(someAmount);

          double retrievedAmount  = testSaleAmount.getValue();


          assertEquals(someAmount, retrievedAmount, .01);
      }
}
```

Won't Compile…

Drives This Stub

```
public class SaleAmount {
    public SaleAmount(double aValue) {}

    public double getValue() {
        return 0.00;
    }
}
```

Observed Failure

```
public class SaleAmountTest {
      @Test public void testSaleAmountCanBeRetieved(){
         double someAmount = 10.00;

         SaleAmount testSaleAmount = new
                        SaleAmount(someAmount);

         double retrievedAmount  = testSaleAmount.getValue();


         assertEquals(someAmount, retrievedAmount, 01);
      }
}
```

Drives This Code

```
public class SaleAmount {
    private double theValue;
    public SaleAmount(double aValue) {
        theValue = aValue;
    }

    public double getValue() {
        return theValue;
    }
}
```

Implementation Decisions

Not all implementation decisions are specified

But some come from the customer, and those must be or your specification is incomplete

Given:

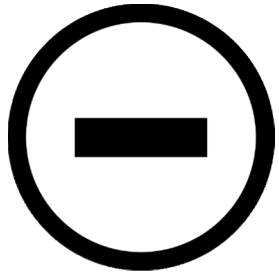
- A SaleAmount S with value V exists in the system

Then:

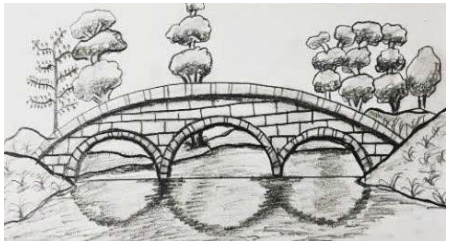
- You cannot change V

A Conundrum for Testing

- Private members are inherently unchangeable
- But they can be made changeable
  - For example, by adding a “set” method
- How can we ensure, in the future, that this has not happened?
- That’s a testing perspective
- Scientifically, how do you prove a negative?



A Conundrum for Specification



- Software is a verb, it does something
- Otherwise it is worthless
- The “something” must have a traceable path to business value. The specification shows this
- Here, the value to the customer is that the software is absent a behavior, lacks a verb
- How can we specify this, if our spec is a suite of tests?
- The spec, in this case, must be able to compile, and execute

Two Common Ideas

1. Add the `setValue()` method, but make it throw an exception if anyone ever calls it. Write a test that calls this method and fails if the exception is not thrown. Sometimes other actions are suggested if the method gets called, but an exception is quite common

2. Use reflection in the test to examine the object and, if `setValue()` is found, fail the test

Problem With Option #1

- ⃠ “Throw an exception” is not what the customer wanted
- ⃠ If we do this, we are creating our own specification
- ⃠ If this is object is used in other parts of the system, the exception will not be expected

Problems With Option #2

- Not all technologies provide refection mechanisms
- Reflection, generally, impedes performance. In TDD tests should run **fast** so they can be run **frequently**
- What are you going to look for in your reflective test?
  - `setValue()`
  - `changeValue()`
  - `putValue()`
  - `alterValue()`
  - `makeValue()`

You see the problem

The TDD Solution

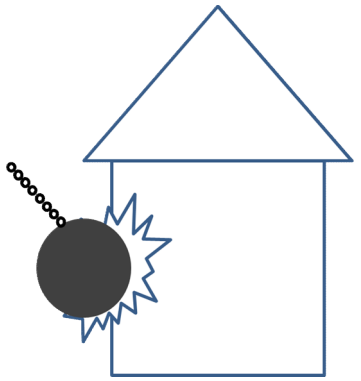
If we think of TDD as creating a specification (that later can also be used as a test) then…

The rules of **specification** apply:

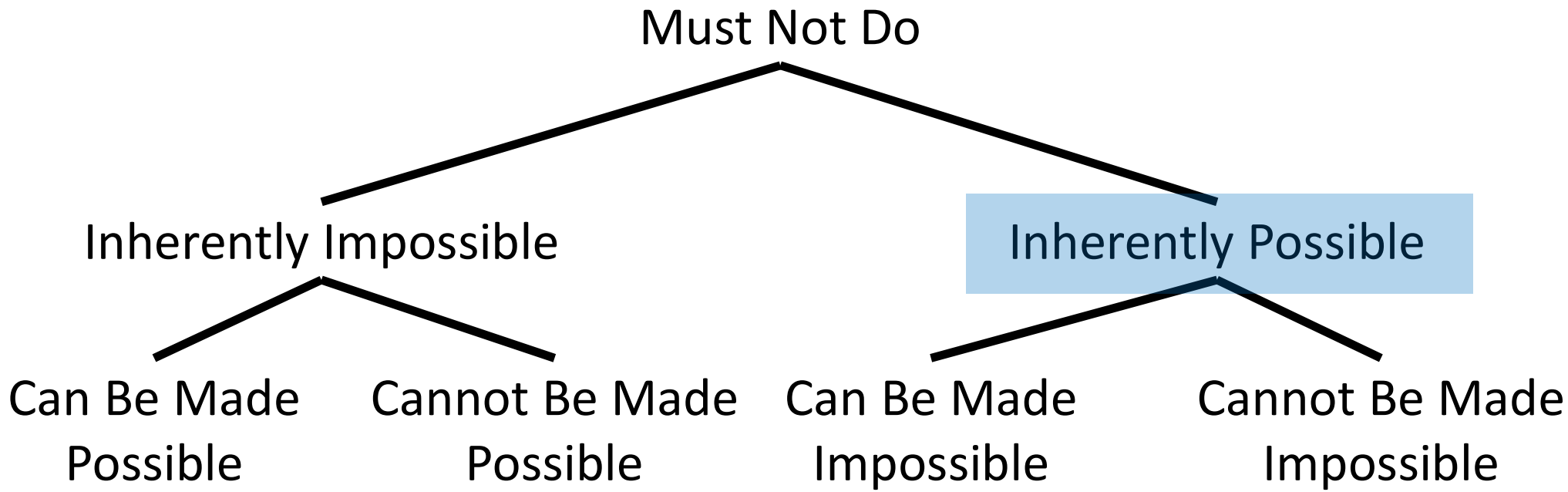
1. The specification must be complete
2. Anything not specified is something the system does not do

TDD as a Process

- In TDD, tests are **always** written first
- In TDD, production code is **never** written without a failing test
- In TDD, the production code is **only** what is needed to make the test pass
- Any production code written without a failing test is an **attack on the system**
- Only the process can prevent this
- No process works if you don't follow it



The Graph of Negatives



Inherently Possible

1. Inherently impossible, we said, can be technology dependent
   - EG: What if we're working in a language without "private"?
2. Even languages that have good encapsulation still cannot prevent all bad behavior
   - But maybe we can guard against it in the executable specification
3. We need to examine the difference between defect prevention and defect detection

Technology Problems

Maybe we're using the wrong technology

You should never assume the customer doesn't care

Some things can only be caught in static analysis

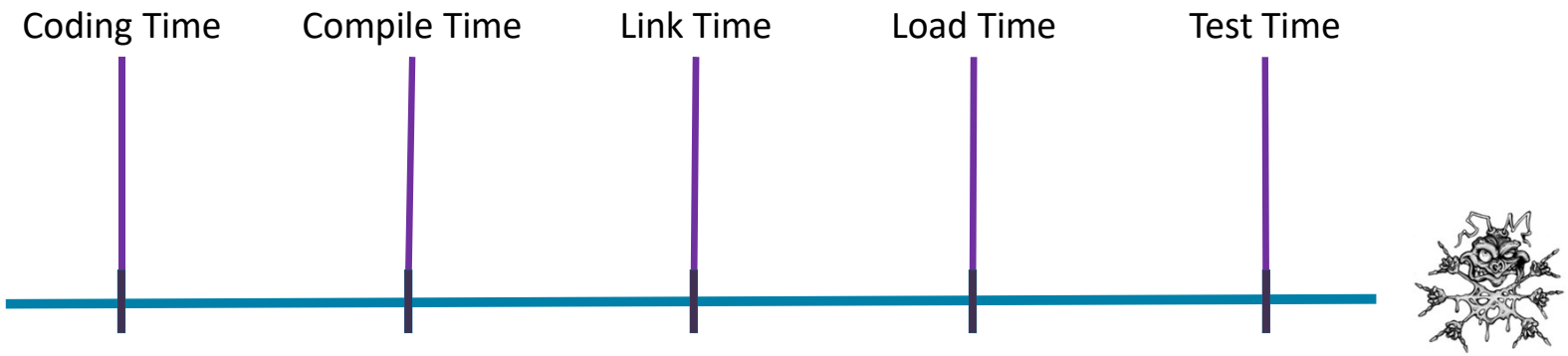
- A code/design review, for example
- …or a QA pass on the code

TDD does not replace other good practices

Inherently Possible

1. Inherently impossible, we said, can be technology dependent
   - EG: What if we're working in a language without "private"?
2. Even languages that have good encapsulation still cannot prevent all bad behavior
   - But maybe we can guard against it in the executable specification
3. We need to examine the difference between defect prevention and defect detection

Notable Times In Programming



Defect Detection vs. Defect Prevention

```
public void setDayOff(int day) {
  // detect <1 or >7 and
  // do something about it
}
```

- Throw an exception
- Perfect the value
- Interpret the value
- Etc…

Defect Detection vs. Defect Prevention

```
public void setDayOff(int day) {
  // detect <1 or >7 and
  // do something about it
}
```

- Throw an exception
- Perfect the value
- Interpret the value
- Etc…

```
public enum DOW {
  MON, TUE, WED, THU, FRI,
  SAT, SUN}
}

public void setDayOff(DOW day){
  // code assumes compiler
  // ensures valid value
}
```

New Customer Requirement

Customer says: “No Sale Amount over a given maximum makes any sense. Nobody is going to spend a million dollars at my online store. If that happens, either we’re being hacked, or something has gone seriously wrong.”

Let’s look at our code again:

The Code

```
public class SaleAmount {
    private double theValue;
    public SaleAmount(double aValue) {
        theValue = aValue;
    }

    public double getValue() {
        return theValue;
    }
}
```

What will the compiler allow?

A Very Big Sale!

$179,769,313,486,231,520,616,720,392,992,464,536,472,240,560,432,240,240,944,616,576,160,448,992,408,768,712,032,320,616,672,472,536,248,456,776,672,352,088,672,544,960,568,304,616,280,032,664,704,344,880,448,832,696,664,856,832,848,208,048,648,264,984,808,584,712,312,912,080,856,536,512,272,952,424,048,992,064,568,952,496,632,264,936,656,128,816,232,688,512,496,536,552,712,648,144,200,160,624,560,424,848,368

Is this the maximum the customer had in mind? I rather doubt it. What was?

A: You have to ask

Customer Q & A

Q: “Customer, you said any Sale Amount over a maximum is not credible, and represents some kind of problem to you. What is the maximum?”

A: “Anything over a thousand dollars is ridiculous. Probably a hacker or a really bad calculation somewhere.’

Q: “Um, how much over? A penny? A dollar? Ten dollars?”

A: “A penny.”

Q: “Okay. Unfortunately, our technology can’t prevent that, but we can detect it. What should we do if that happens?”

A: “Raise an alarm! Make sure we know about it so we can do something…”

Capturing the First Two Answers: Constants

Given:

- The system

Then:

- The Maximum value for a Sale Amount is $1000.00 within the required Tolerance

---

Given:

- The System

Then:

- The Tolerance for the comparison of SaleAmount values is 1 cent

The Executable Specification

```
@Test
public void specifyMaximumDollarValue() {
    assertEquals(1000d, SaleAmount.MAXIMUM,
                        SaleAmount.TOLERANCE );
}
```

```
@Test
public void specifySaleamountTolerance() {
    assertEquals(.01, SaleAmount.TOLERANCE);
}
```

Making the Test Compile/Fail

```
public class SaleAmount {
    public static final double MAXIMUM = -42;
    public static final double TOLERANCE = -42;
    private double theValue;

    public SaleAmount(double aValue) {
        theValue = aValue;
    }

    public double getValue() {
        return theValue;
    }
}
```

Drives These Changes to the Code

```
public class SaleAmount {
    public static final double MAXIMUM = 1000d;
    public static final double TOLERANCE = .01;
    private double theValue;

    public SaleAmount(double aValue) {
        theValue = aValue;
    }

    public double getValue() {
        return theValue;
    }
}
```

Capturing Answer 3: Raise the Alarm!

- There are multiple ways to do this
  - One typical way: throw an exception
- So now we have our new requirement
- Let's do the Given, When, Then

Required Behavior

Given:

- Amount S greater than or equal to Maximum + Tolerance

When:

- An attempt is made to create a SaleAmount with value S

Then:

- An exception is thrown

Executable Specification

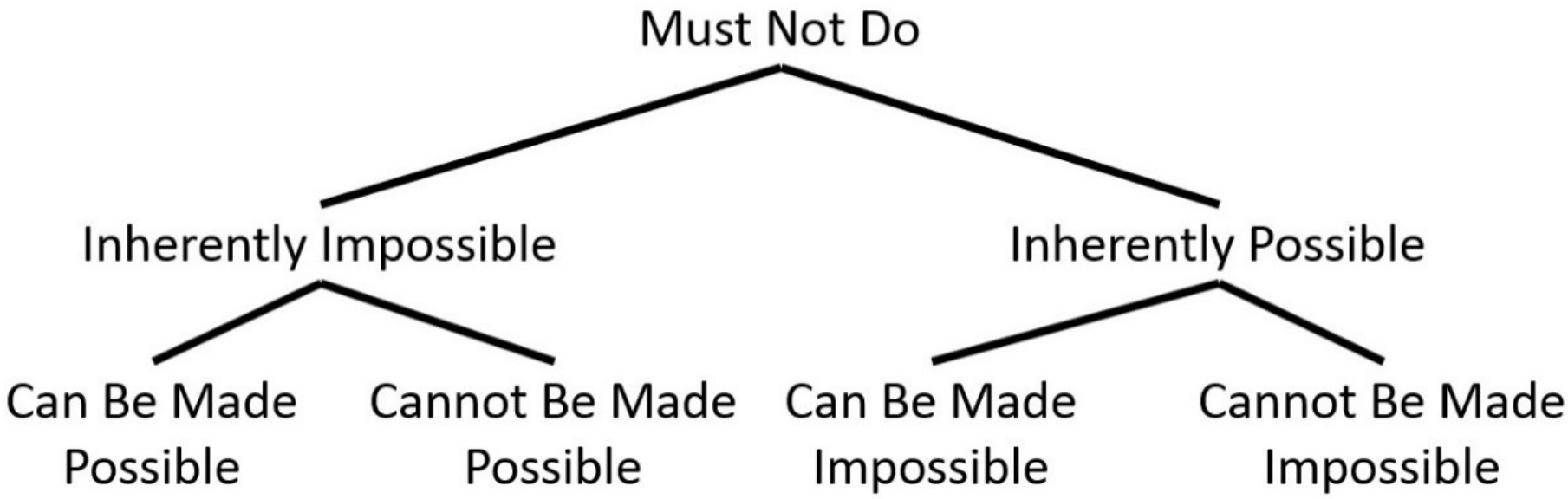
```
@Test
public void testExcessiveSaleAmountThrowsException() {
    double excessiveValue = SaleAmount.MAXIMUM + SaleAmount.TOLERANCE;

    try {
        new SaleAmount(excessiveValue);
        fail("SaleAmount created with excessive amount should have " +
              thrown an exception");
    }
    catch (ValueTooLargeException) { }
}
```

Drives These Changes to the Code

```
public class SaleAmount {
    public static final double MAXIMUM = 1000d;
    public static final double TOLERANCE = .01;
    private double theValue;

    public SaleAmount(double aValue) {
        validateValue(aValue);
        theValue = aValue;
    }

    private void validateValue(double value) {
        if (value >= MAXIMUM + TOLERANCE) {
            throw new ValueTooLargeException();
        }
    }

    public double getValue() {
        return theValue;
    }
```

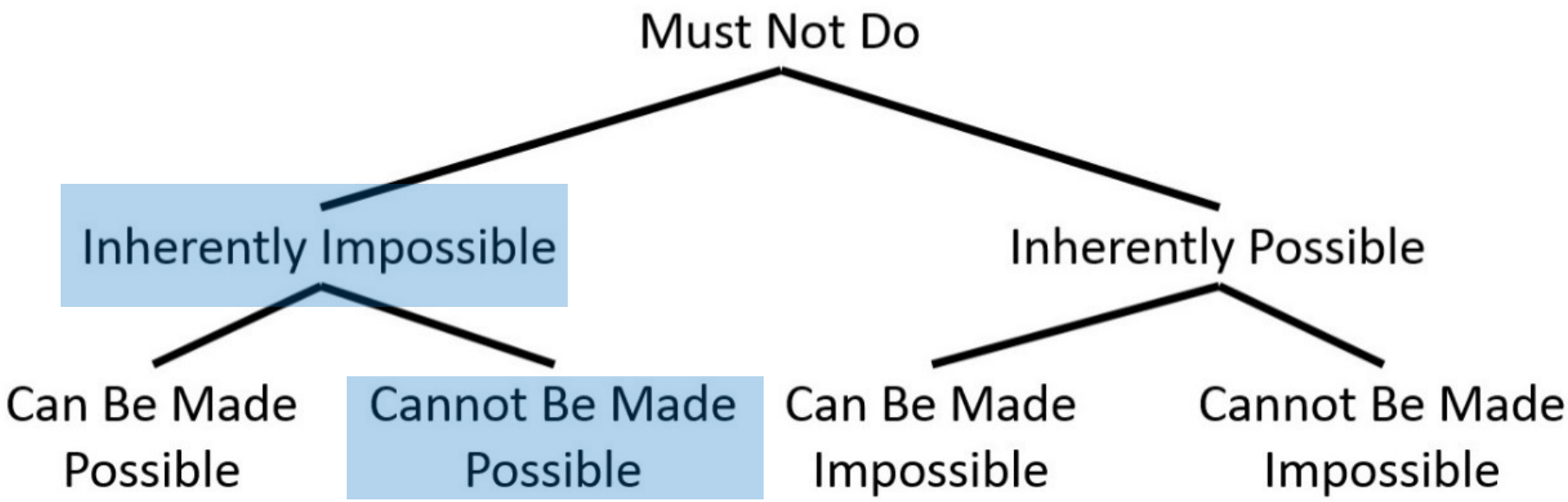
Conclusions

Specifying negative requirements begins with identifying where the requirement lives on the graph of negatives:



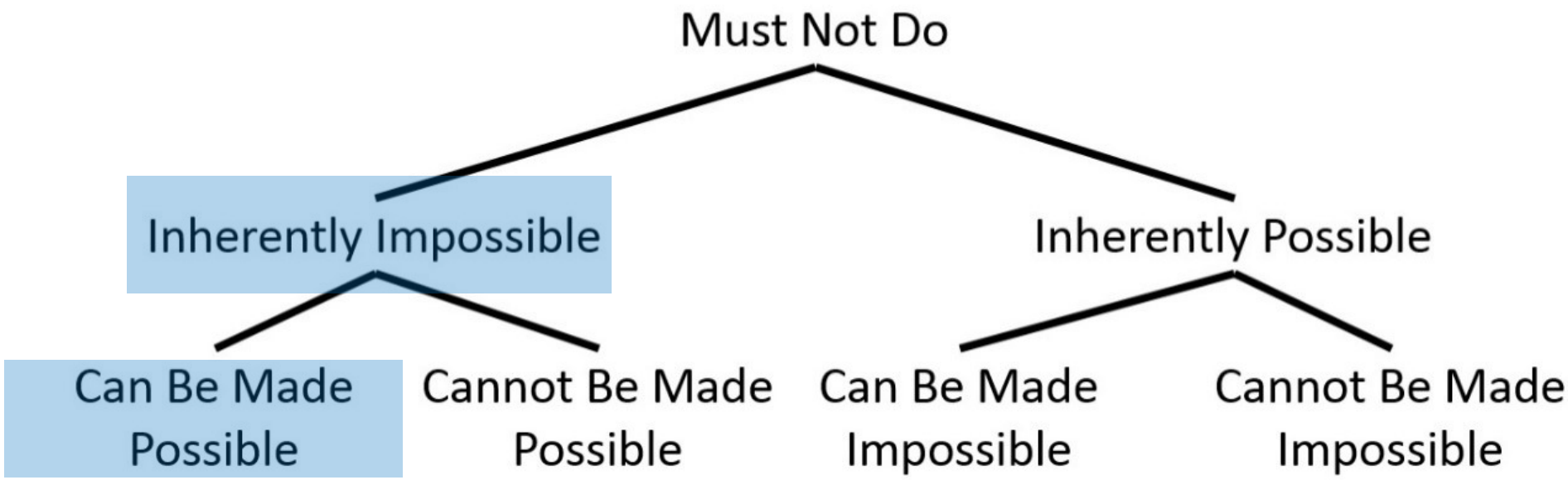
Inherently Impossible, Can't Be Made Possible

There is nothing to do, it's done



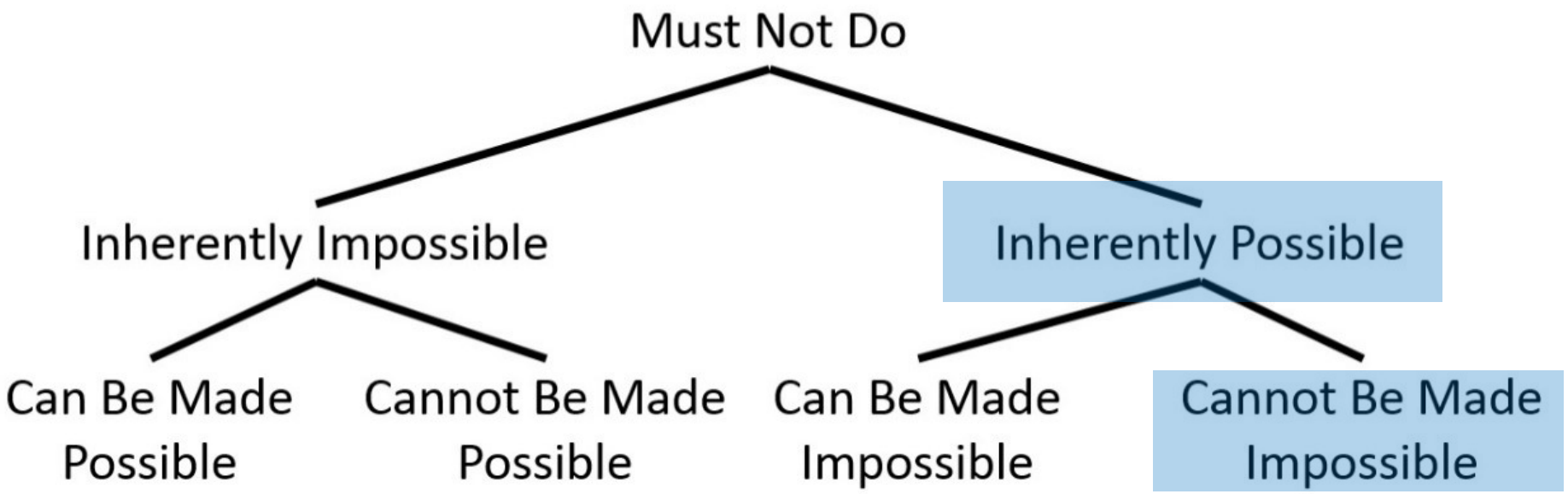
Inherently Impossible, Can Be Made Possible

Follow the TDD process diligently



Inherently Possible, Can't Be Made Impossible

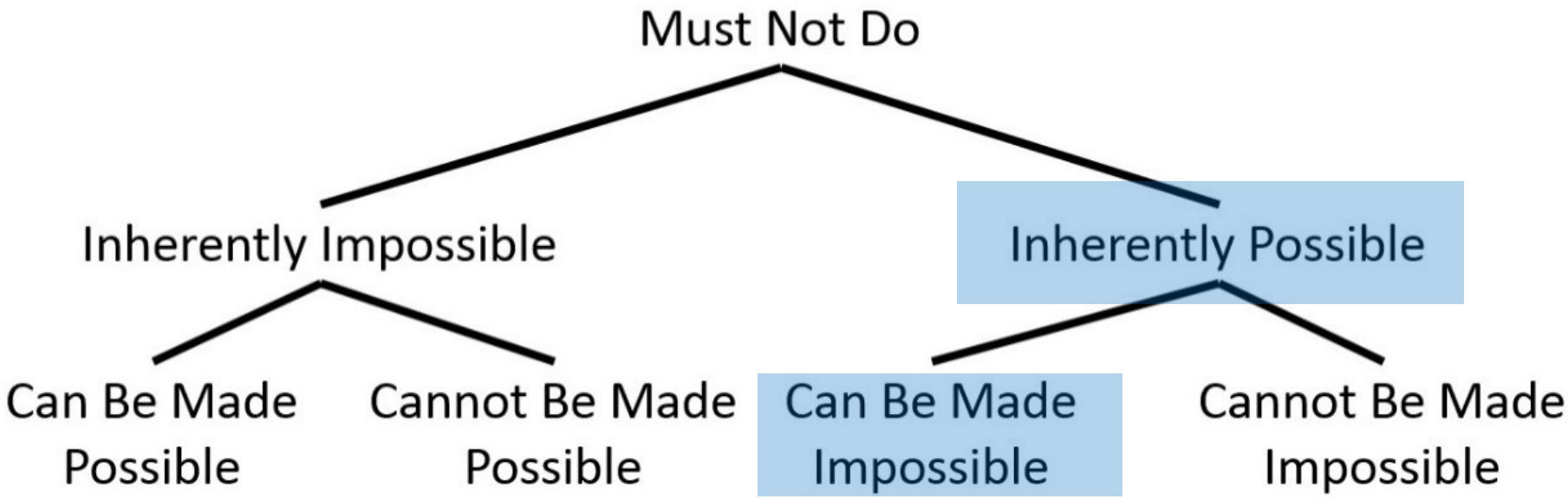
Switch technology, increase QA and reviews



Inherently Possible, Can Be Made Impossible

Defect **Detection** vs. Defect **Prevention**

Detection: Ask for the right behavior, test-drive it



Staying in Touch

- I have written about a lot of TDD topics
- I also post frequently on LinkedIn about this stuff
- If you want to connect with me, you can do so at:
  - ProjectManagement.com
  - LinkedIN.com

https://www.projectmanagement.com/blogs/654443/Sustainable-Test-Driven-Development

Training Courses at PMI

- Acceptance Test-Driven Development

- Design Patterns Thinking
- Advanced Software Design
- Sustainable Test-Driven Development

Scott.Bain@pmi.org

https://www.pmi.org/business-solutions/agile-training/technical-solutions

Thank You!

Feel free to email questions:
scott.bain@pmi.org