Good coding practice in real life

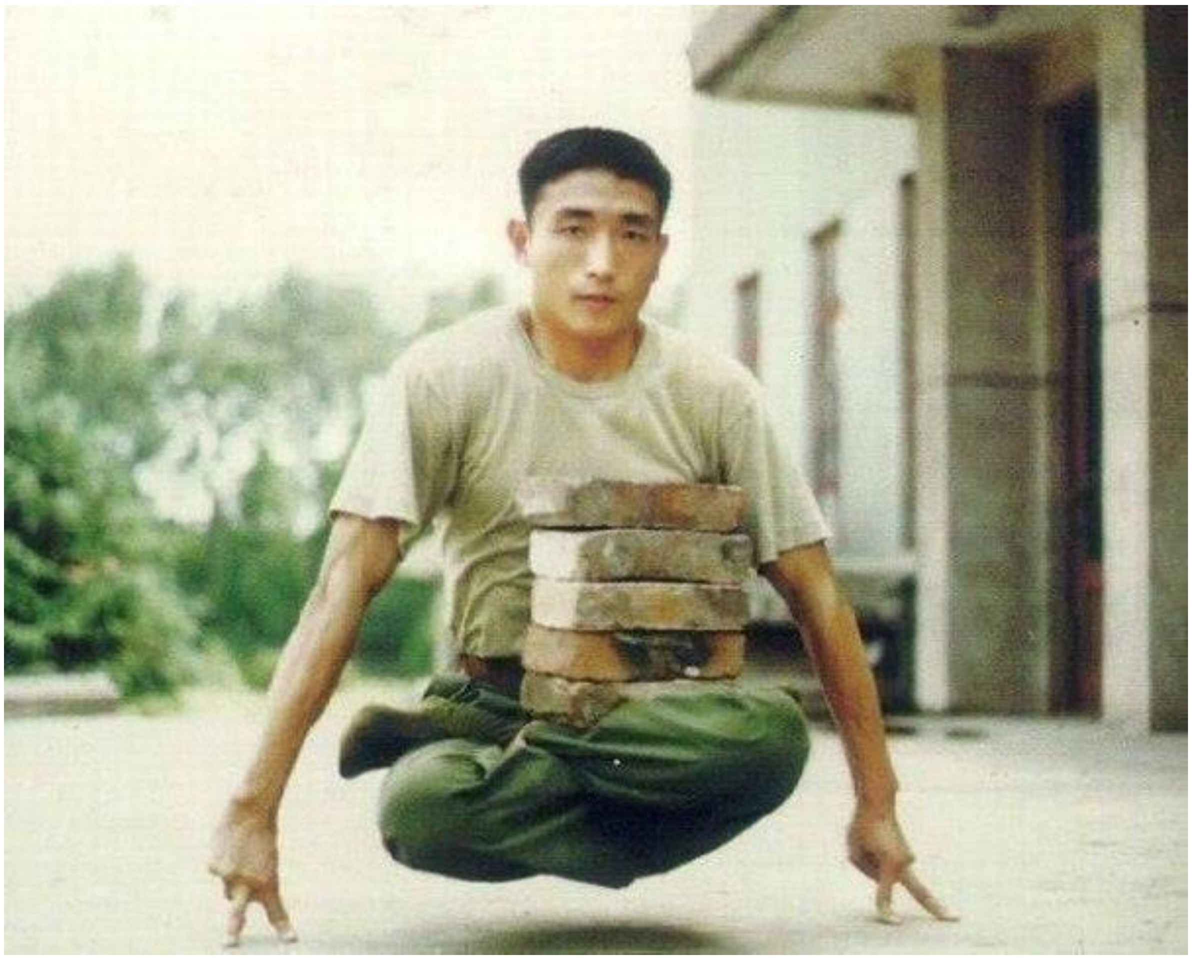
with a focus on

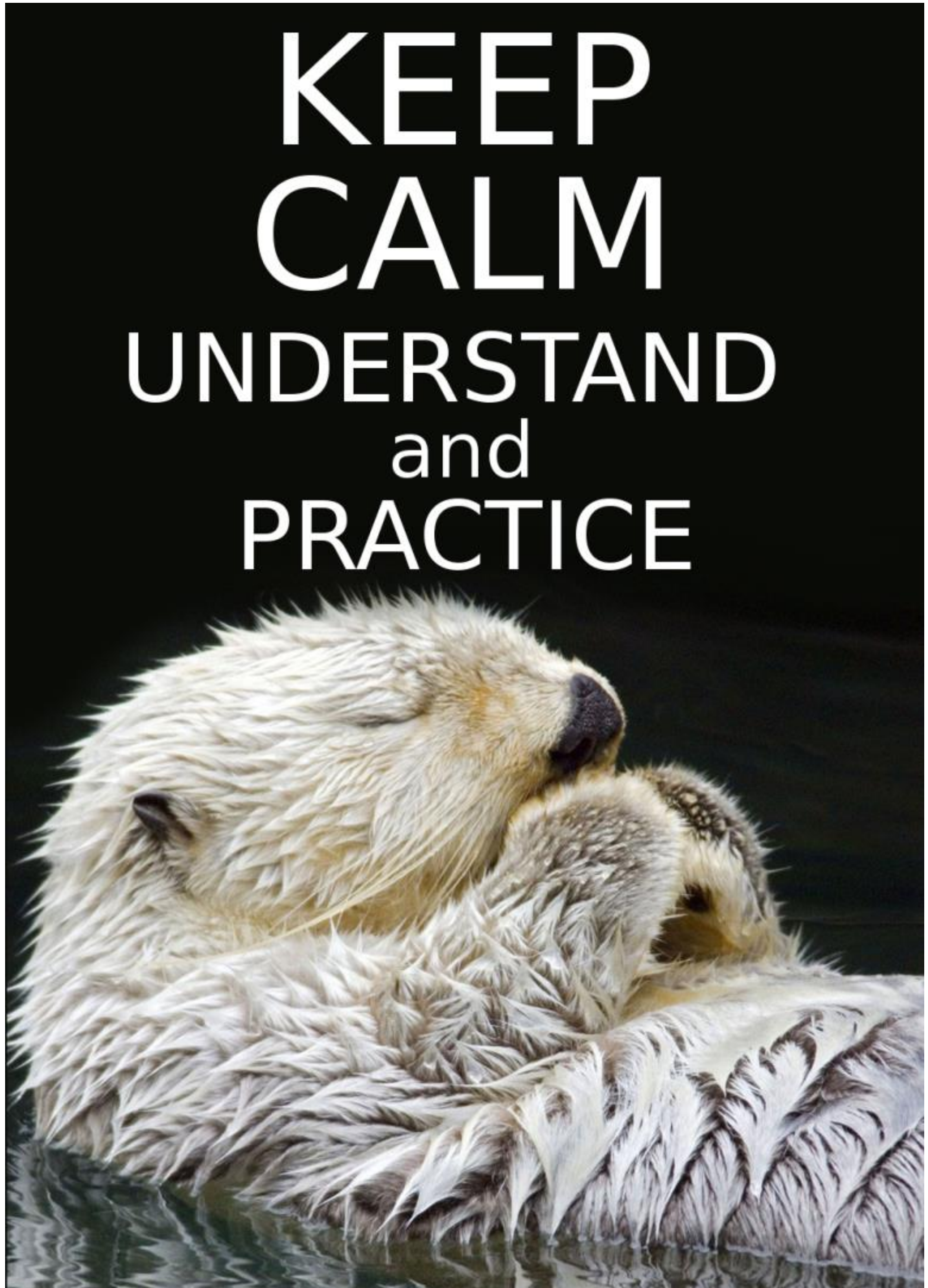
design by contract – part 3

Vladimir Alekseichenko

Good practices make life easier

Good practices are not easy




KEEP
CALM
UNDERSTAND
and
PRACTICE

Agenda

- Liskov substitution principle
- Inheritance and assertions
- Assertion redeclaration rule
- Parents’ Invariant rule
- Code Contracts in .Net
- A few examples

Liskov substitution principle

If for each object **o1** of type **S** there is an object **o2** of type **T** such that for all programs **P** defined in terms of **T**, the behavior of **P** is unchanged when **o1** is substituted for **o2** then **S** is a subtype of **T**.

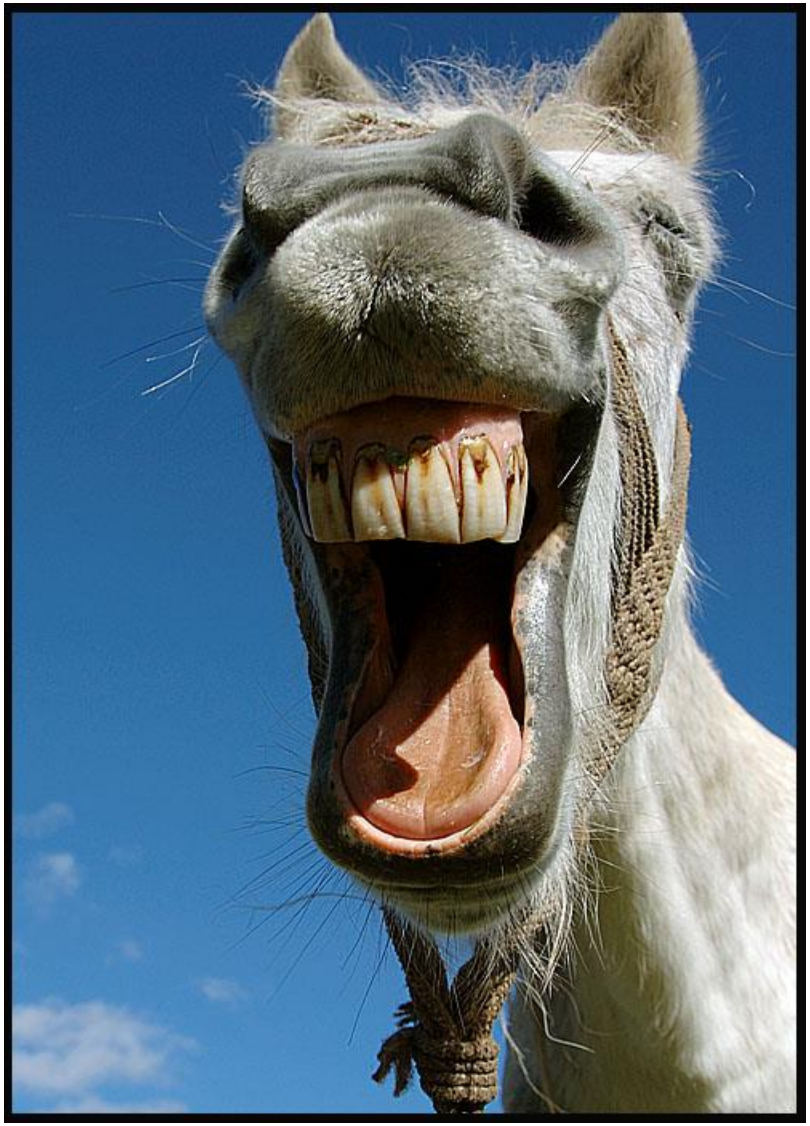
Is it a violation of the principle of LSP?

```
public class DoubleList<T> : IList<T>
{
    private readonly IList<T> _elements = new List<T>();

    public void Add(T item)
    {
        _elements.Add(item);
        _elements.Add(item);
    }
```

2!

Noooo!

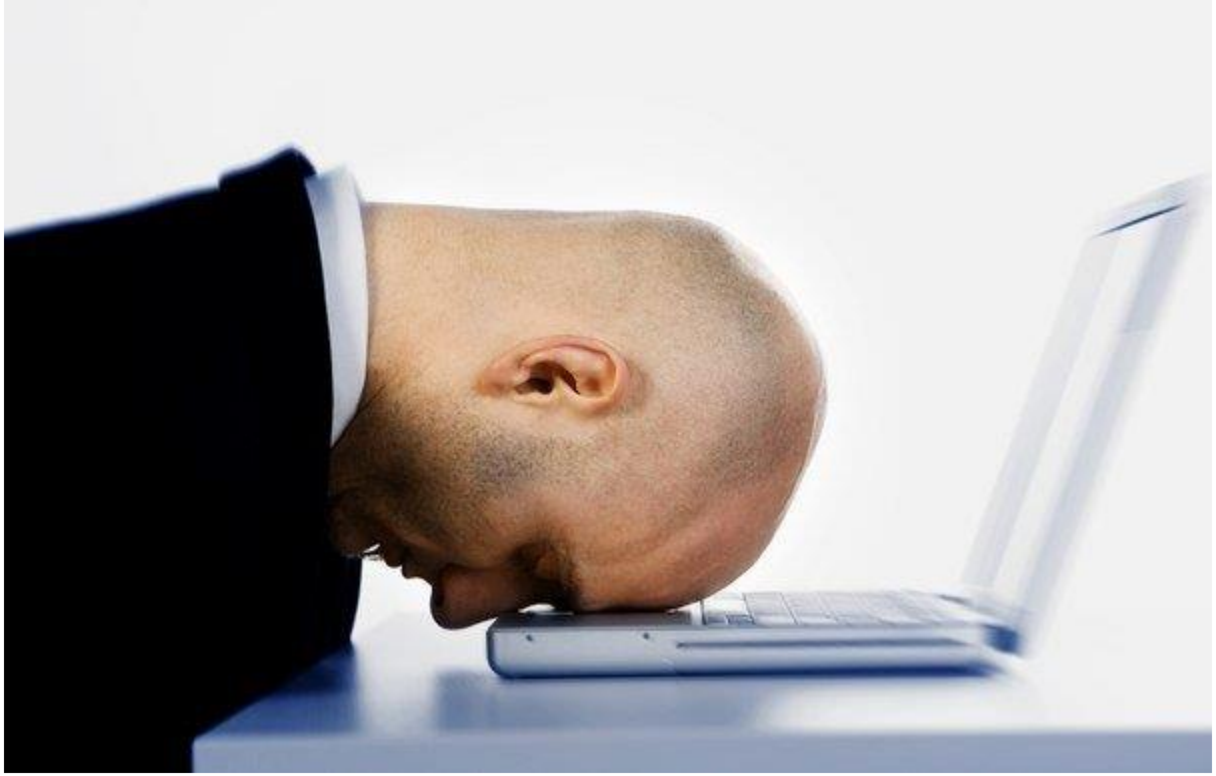


Why?

The answer

Because it isn’t specified, the behavior of the method **Add()**.

How to deal with it?



Just use a code contracts

Bertrand Meyer

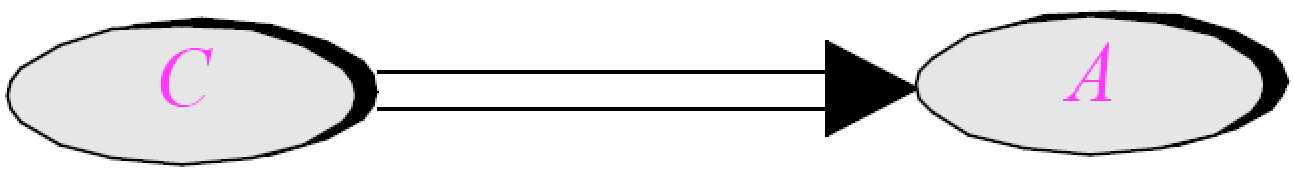
http://en.wikipedia.org/wiki/Bertrand_Meyer

Inheritance could be dangerous

Preconditions and postconditions

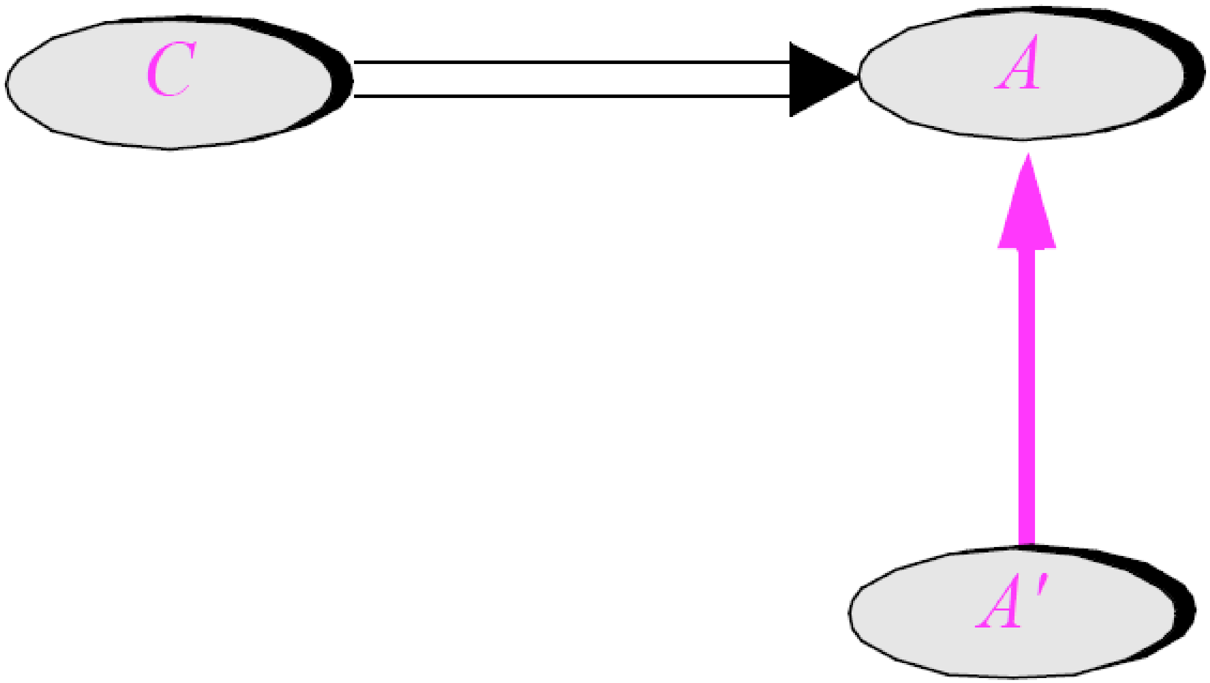
The general idea, is that any redeclaration must satisfy the assertions on the original routine.

The routine, the client and the contract




```
r is
    require
        α
    ...
    ensure
        β
    end
```

The routine, the client, the contract and the descendant



r **is**
  require
    α
  ...
  ensure
    β
  end

r^{++} **is**
  require
    γ
  ...
  ensure
    δ
  end

How to cheat clients?

http://strand-cafe-lang.de/ferienappartements/business-honesty

How to cheat clients?

- We could *require more* than the original **precondition** α.
- We could *ensure less* than the original **postcondition** β.

How to be honest?

http://mamuas.livejournal.com/11155.html

How to be honest?

- Replace the **precondition** by a ***weaker*** one.
- Replace the **postcondition** by a ***stronger*** one.

Assertion redeclaration rule

A routine redeclaration may only replace the original precondition by one equal or weaker, and the original postcondition by one equal or stronger.

Parents' Invariant rule

The **invariants** of all the parents of a class apply to the class itself.

Parents’ Invariant rule

The parents’ invariants are added to the class’s own, „addition” being here a logical **and then**.

(If no invariant is given in a class, it is considered to have True as invariant.)

The first example

NAME_LIST

```
class NAME_LIST

    feature
        has(a_name: STRING): BOOLEAN
                --Is 'a_name' in list?

        put(a_name: STRING)
                -- Add 'a_name' to list
            require
                not_already_in_the_list: not has(a_name)
            ensure
                on_the_list: has(a_name)
                number_of_names_increased: count = old count + 1
end
```

RELAXED_NAME_LIST

```
class
    RELAXED_NAME_LIST
inherit
    NAME_LIST
redefine
    put
end
```

RELAXED_NAME_LIST
preconditions

```
feature
    put(a_name: STRING)
            -- Add name to list


        require -- from NAME_LIST
            name_not_in_list: not has(a_name)
        require else
            -- not has(a_name) or else has(a_name) it's the same as True
            already_in_the_list: has(a_name)
```

RELAXED_NAME_LIST

postconditions

if

the name was not in the list

then

the count increase by one

else

the count doesn't change

RELAXED_NAME_LIST
postconditions

```
ensure -- from NAME_LIST
    count_increased:
        (old not has(a_name)) implies count = old count + 1
    name_in_list:
        (old not has(a_name)) implies has(a_name)

ensure then
    count_unchanged_if_name_was_already_in_list:
        (old has(a_name)) implies count = old count
```

Code Contracts in .Net

http://research.microsoft.com/en-us/projects/contracts/userdoc.pdf

Assertions

- Precondition
- Postcondition
- Class invariant
- Assertion instruction
- Loop invariant *

* *There is no equivalent in the .Net.*

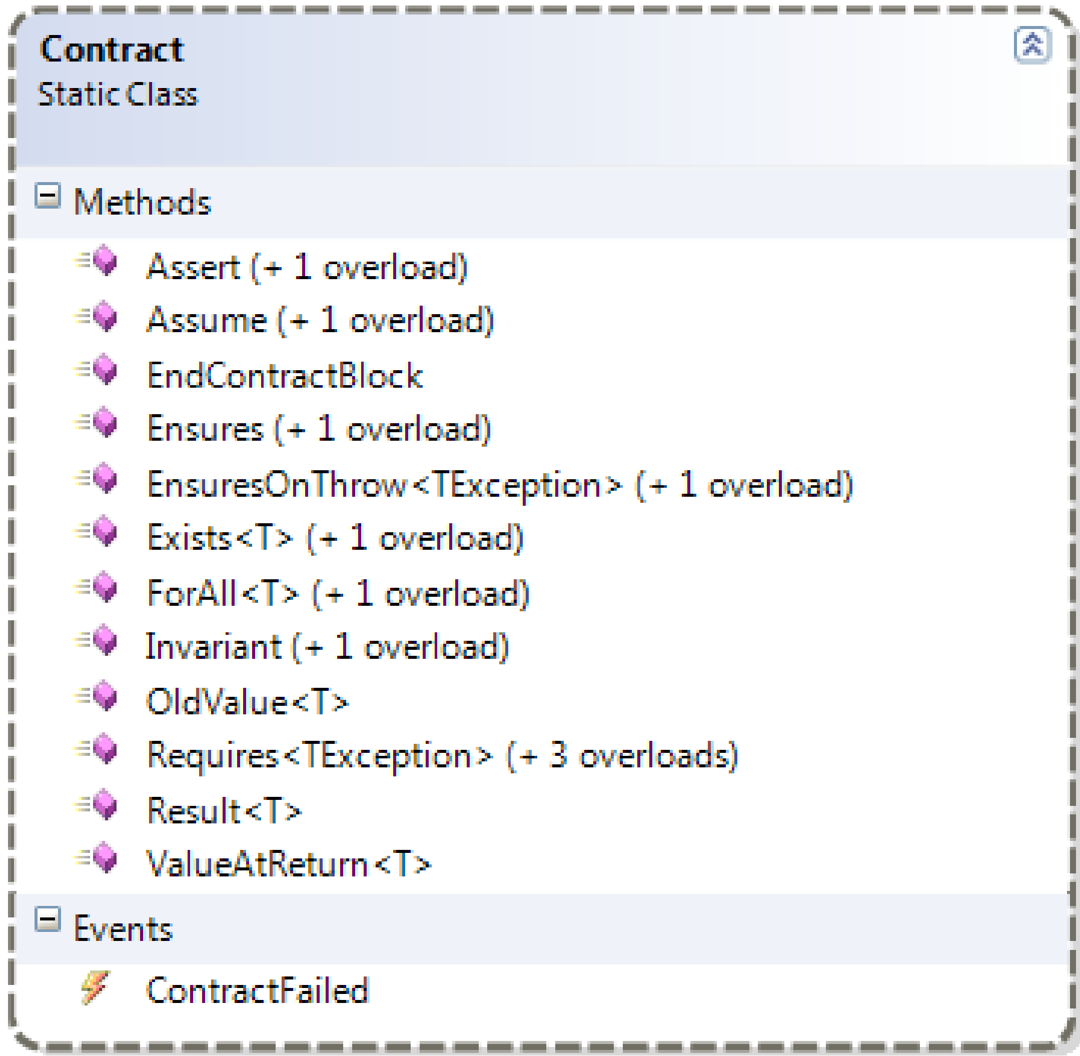
The namespace

All of the contract methods are **static** methods defined in the **Contract class** which appears in the:

```
using System.Diagnostics.Contracts;
```

http://research.microsoft.com/en-us/projects/contracts/userdoc.pdf

Contract


Contract
Static Class
Methods
Assert (+ 1 overload)
Assume (+ 1 overload)
EndContractBlock
Ensures (+ 1 overload)
EnsuresOnThrow<TException> (+ 1 overload)
Exists<T> (+ 1 overload)
ForAll<T> (+ 1 overload)
Invariant (+ 1 overload)
OldValue<T>
Requires<TException> (+ 3 overloads)
Result<T>
ValueAtReturn<T>
Events
ContractFailed

Preconditions

```
Contract.Requires(x != 0);
```

```
Contract.Requires<ArgumentException>(x != 0);
Contract.EndContractBlock();
```

http://research.microsoft.com/en-us/projects/contracts/userdoc.pdf

Preconditions

Legacy code

```
if (x == 0) throw new ArgumentException("x");
Contract.EndContractBlock();
```

```
Contract.Requires<ArgumentException>(x != 0, "x");
```

http://research.microsoft.com/en-us/projects/contracts/userdoc.pdf

Postconditions

```
Contract.Ensures(this.balance > 0);
```

```
Contract.EnsuresOnThrow<Exception>(this.balance > 0);
```

http://research.microsoft.com/en-us/projects/contracts/userdoc.pdf

Postconditions

Special methods within **postconditions**

- Method return values (Result)
- Prestate values (OldValue)
- Out parameters (ValueAndReturn)

http://research.microsoft.com/en-us/projects/contracts/userdoc.pdf

Postconditions

Method return values

```
Contract.Result<T>()
```

```
Contract.Ensures(0 < Contract.Result<int>());
```

http://research.microsoft.com/en-us/projects/contracts/userdoc.pdf

Postconditions
Prestate values

```
Contract.OldValue<T>(e)
```

where T is the type of e.

```
Contract.Ensures(this.State == Contract.OldValue(this.State));
```

http://research.microsoft.com/en-us/projects/contracts/userdoc.pdf

Postconditions

Out parameters

```
Contract.ValueAtReturn<T>(out T t)
```

```
public void OutParam(out int x)
{
    Contract.Ensures(Contract.ValueAtReturn (out x) == 3);
    x = 3;
}
```

http://research.microsoft.com/en-us/projects/contracts/userdoc.pdf

Object invariants

[ContractInvariantMethod]

```
[ContractInvariantMethod]
private void ObjectInvariant () {
    Contract.Invariant( this.y >= 0 );
    Contract.Invariant( this.x > this.y );
    //...
}
```

http://research.microsoft.com/en-us/projects/contracts/userdoc.pdf

Object **invariants**

Invariants on automatic properties

- A **pre**condition for the setter
- A **post**condition for the getter
- An **invariant** for the underlying backing field

http://research.microsoft.com/en-us/projects/contracts/userdoc.pdf

Object **invariants**

Invariants on automatic properties

```
public int MyProperty { get; private set; }

[ContractInvariantMethod]
private void ObjectInvariant()
{
   Contract.Invariant(this.MyProperty >= 0);
   //...
}
```

is equivalent to the following code

http://research.microsoft.com/en-us/projects/contracts/userdoc.pdf

```
private int _backingFieldForMyProperty;

public int MyProperty
{
   get
   {
      Contract.Ensures(Contract.Result<int>() >= 0);
      return this._backingFieldForMyProperty;
   }
   private set
   {
      Contract.Requires(value >= 0);
      this._backingFieldForMyProperty = value;
   }
}

[ContractInvariantMethod]
private void ObjectInvariant()
{
   Contract.Invariant(this._backingFieldForMyProperty >= 0);
   //...
}
```

http://research.microsoft.com/en-us/projects/contracts/userdoc.pdf

Assertion instruction

- Assert
- Assume
- Quantiers

http://research.microsoft.com/en-us/projects/contracts/userdoc.pdf

Assertion instruction

Assert

```
Contract.Assert(this.privateField > 0);
```

```
Contract.Assert(this.x == 3, "Why isn't the value of x 3?");
```

http://research.microsoft.com/en-us/projects/contracts/userdoc.pdf

Assertion instruction

Assume

```
Contract.Assume(this.privateField > 0);
```

```
Contract.Assume(this.x == 3, "Static checker assumed this");
```

http://research.microsoft.com/en-us/projects/contracts/userdoc.pdf

Assertion instruction
Quantiers

- **ForAll**
- Exists

http://research.microsoft.com/en-us/projects/contracts/userdoc.pdf

Assertion instruction

Quantiers

```
Contract.ForAll(xs, x => x != null)
```

```
public int Foo<T>(IEnumerable<T> xs) {
   Contract.Requires(Contract.ForAll(xs , x => x != null));
   //...
```

http://research.microsoft.com/en-us/projects/contracts/userdoc.pdf

Other features

- **Interface contracts**
- Contracts on abstract methods
- Overloads on contract methods
- Contract argument validator methods
- Contract abbreviator methods
- AssumeInvariant helper

http://research.microsoft.com/en-us/projects/contracts/userdoc.pdf

Interface contracts

```
[ContractClass(typeof (IFooContract))]
```

```
[ContractClassFor(typeof (IFoo))]
```

http://research.microsoft.com/en-us/projects/contracts/userdoc.pdf

Interface contracts

```
[ContractClass(typeof (IFooContract))]
internal interface IFoo
{
   int Count { get; }
   void Put(int value);
}

[ContractClassFor(typeof (IFoo))]
internal abstract class IFooContract : IFoo
{
   int IFoo.Count
   {
      get
      {
         Contract.Ensures(0 <= Contract.Result<int>());
         return default(int); // dummy return
      }
   }

   void IFoo.Put(int value) { Contract.Requires(0 <= value); }
}
```

http://research.microsoft.com/en-us/projects/contracts/userdoc.pdf

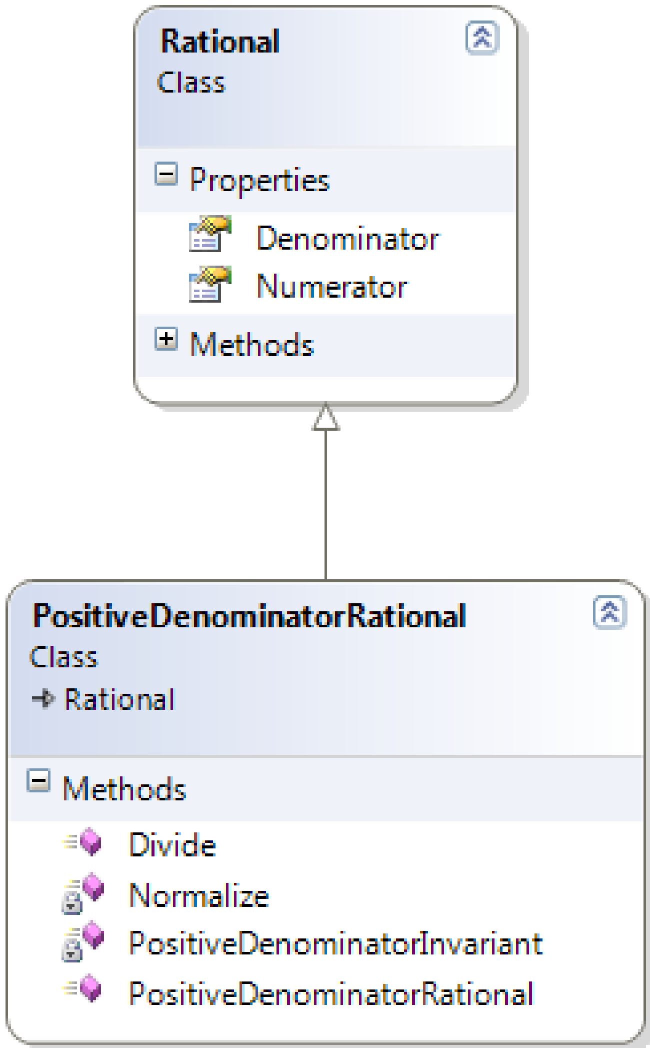
The second example

Main

```
var rational = new PositiveDenominatorRational(5, 3);
rational.Divide(-5);
var nextRational = new PositiveDenominatorRational(5, 0);
```

! -5

PositiveDenominatorRational *is* Rational



Rational

```
public class Rational {
  public int Numerator { get; protected set; }
  public int Denominator { get; protected set; }

  public Rational(int n, int d) {
    Contract.Requires(d != 0);

    this.Numerator = n;
    this.Denominator = d;
  }

  [ContractInvariantMethod]
  private void RationalInvariant() {
    Contract.Invariant(Denominator != 0);
  }
```

precondition

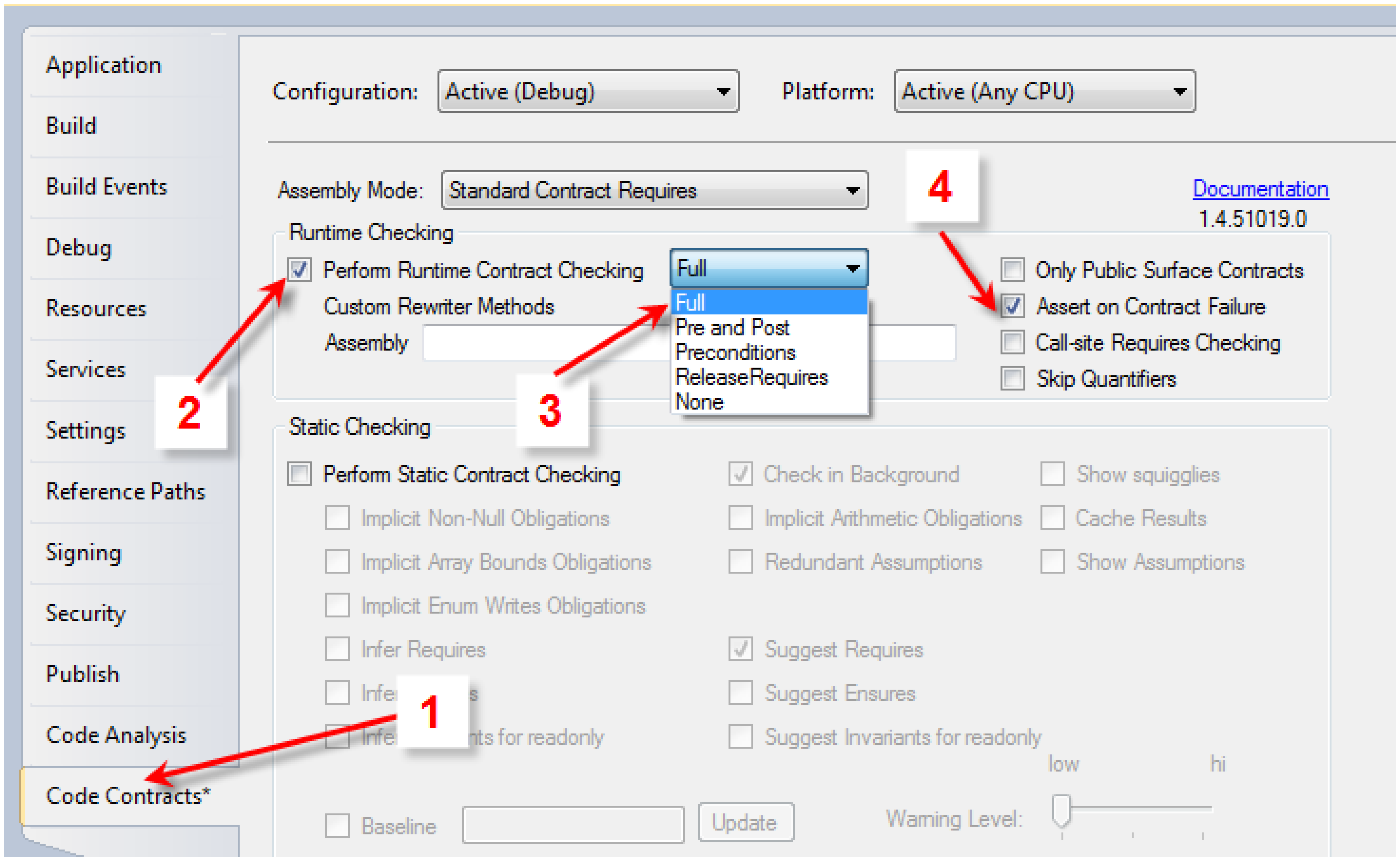
class invariant

PositiveDenominatorRational

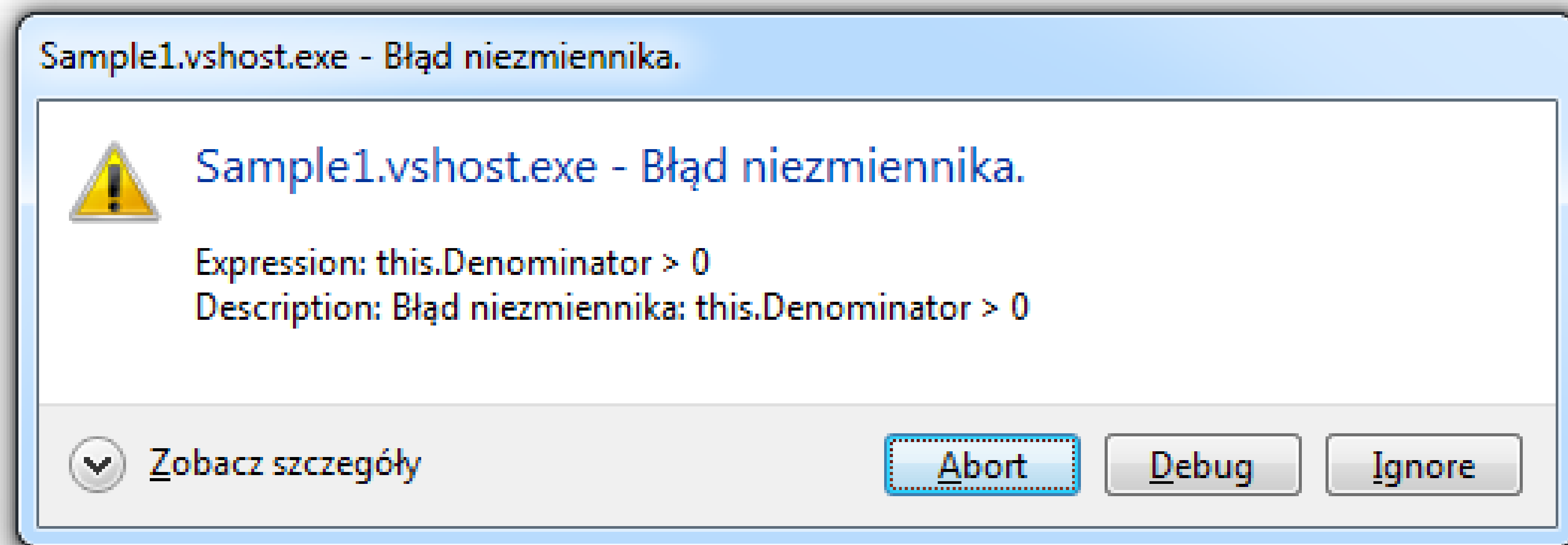
```
public class PositiveDenominatorRational : Rational
{
  public PositiveDenominatorRational(int n, int d) :
    base(n,d)
  {
    Contract.Requires(d != 0);
    Normalize();
  }

  [ContractInvariantMethod]
  private void PositiveDenominatorInvariant()
  {
     Contract.Invariant(this.Denominator > 0);
  }
```

Properties


Application
Build
Build Events
Debug
Resources
Services
Settings
Reference Paths
Signing
Security
Publish
Code Analysis
Code Contracts*
Configuration: Active (Debug)
Platform: Active (Any CPU)
Assembly Mode: Standard Contract Requires
Documentation
1.4.51019.0
Runtime Checking
Perform Runtime Contract Checking
Full
Full
Pre and Post
Preconditions
ReleaseRequires
None
Custom Rewriter Methods
Assembly
Only Public Surface Contracts
Assert on Contract Failure
Call-site Requires Checking
Skip Quantifiers
Static Checking
Perform Static Contract Checking
Check in Background
Show squigglies
Implicit Non-Null Obligations
Implicit Arithmetic Obligations
Cache Results
Implicit Array Bounds Obligations
Redundant Assumptions
Show Assumptions
Implicit Enum Writes Obligations
Infer Requires
Suggest Requires
Suggest Ensures
Suggest Invariants for readonly
low
hi
Baseline
Update
Warning Level:
1
2
3
4

Invariant failed



```
[ContractInvariantMethod]
private void PositiveDenominatorInvariant()
{
    Contract.Invariant(this.Denominator > 0);
}
}
```

Main

```
var rational = new PositiveDenominatorRational(5, 3);
rational.Divide(-5);
var nextRational = new PositiveDenominatorRational(5, 0);
```

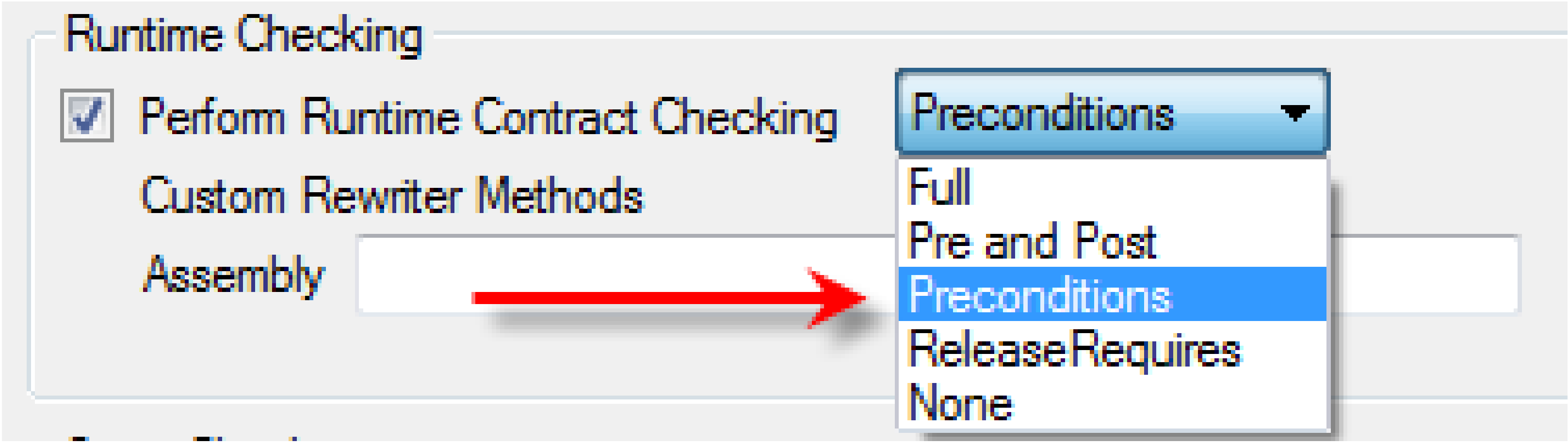
! -5

Divide

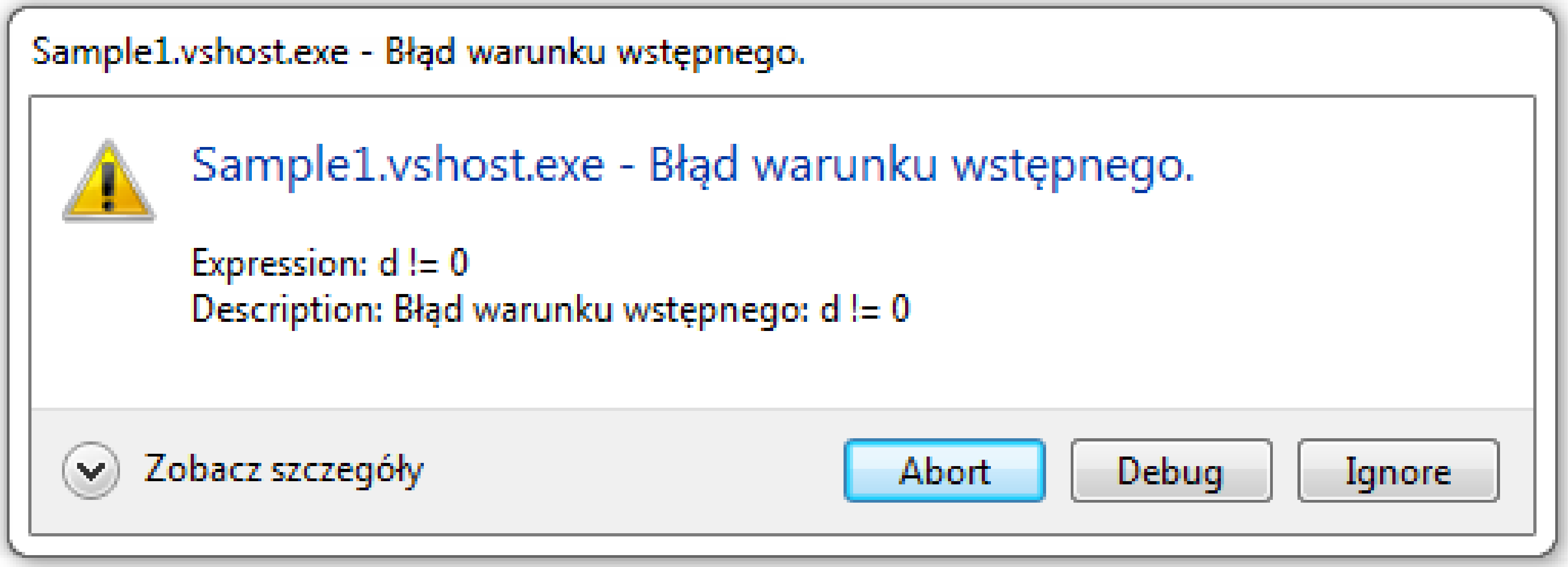
```
public virtual void Divide(int divisor)
{
  Contract.Requires<ArgumentOutOfRangeException>(divisor != 0);

  this.Denominator = this.Denominator * divisor;
}
```

Checking only preconditions



Precondition failed



Precondition

```
public PositiveDenominatorRational(int n, int d) :
  base(n,d)
{
  Contract.Requires(d != 0);
  Normalize();
}
```

In summary

- You can weaken the **pre**condition
- You can strengthen the **post**condition
- A subclass inherits its superclass's **invariant**
- You can strengthen the **invariant**

Resources

Books and papers

- **Object-Oriented Software Construction** by *Bertrand Meyer*
- **Design by Contract, by Example** by *Richard Mitchell, Jim McKim*
- **Code Contracts User Manual** (Microsoft Corporation March 17, 2013)