# Code Contracts

DinoEsposito

2009

# Content

- *The Code Contracts ecosystem*
  - *Library and helper tools*
- *Design by contract*
  - *Preconditions, postconditions, invariants, and more*
- *Implementation*

# A brief introduction

# What’s Testability?

- Refers to the capability of a system to be tested
  - Especially in an automated way

- **Design-for-Testability** methodology
  - Originally developed to improve building of low-level circuits within boards and chips
  - Automated testing through ad hoc programs
  - Adapted to software engineering

# Software Testability

- Software testability results from characteristics of the code that the development team should ideally guarantee
- **Visibility**
  - Ability to observe the current state of the software under test and any output it can produce
- **Control**
  - Degree at which the code allows testers to apply fixed input data to the software under test
- **Simplicity**
  - Simplest the code to test, most reliable your results will be

# Software DFT

- Maximize visibility, control, and simplicity in classes
- Side benefits
  - Unit tests much easier to write and more effective
  - Overall better design; maximum of maintainability
  - Help with code regression
  - Code easier to read
- DFT in practice?
  - Software contracts
- In .NET 4?
  - Code Contracts API

# The Ecosystem at a glance

- Code Contracts Library
- Rewriter
- Static checker
- Contract reference generator

# Code Contracts API

## *At a glance*

- 3 basic types of code contracts
  - Preconditions, postconditions, invariants
  - External to .NET 3.5, embedded in .NET 4
- Exposed via the **Contract** class
  - Static methods
  - System.Diagnostics.Contracts
- API not baked in any managed language (yet?)
  - No syntactic sugar
  - Same API regardless of the language

# Code Contracts Library

# Steps of a Method

1. Check the validity of parameters and current state
2. Proceed to do the real work
3. Check to see if all worked correctly
4. Update the object’s state accordingly

# Using the Code Contracts API

## *Set up*

- Reference to mscorlib in .NET 4 (default)
  - Reference to external assembly in .NET 3.5
  - Additional tab in Solution|Properties to configure code contracts
- Contract must be specified in the body of methods
  - Inserted via plain code
  - Playing a declarative role (like attributes, but more expressive)
- Contracts are **not** mandatory—you can do without …
  - Just a rich API to add **testability** to classes
  - Design-by-contract pattern is **cross-language**

# Using the Code Contracts API

## *Release mode*

- Help find bugs in code
- Typical usage
  - Included in debug builds
  - Removed from retail builds
- Conditionally compiled upon CONTRACTS_FULL
  - More flexible UI available in Solution|Properties

# Software Contracts

- Preconditions
- Postconditions
- Invariants
- Assert
- Assume
- Quantifiers
  - ForAll
  - Exists

# Preconditions

*Available API*

- Method **`Requires`**`(bool condition)`
  - A particular condition must be true upon entry to the method
- Method **`Requires<TException>`**`(bool condition)`
  - If required condition is not met, throw the given exception
  - This precondition won't be stripped off the retail code
- **If-then-throw** statements as an alternative
  - **EndContractBlock** following *if-then-throw*s transforms legacy preconditions into Code Contracts-compatible preconditions
- Method **`RequiresAlways`**`(bool condition)`
  - The check is not automatically stripped off in retail builds

# Preconditions

## *Usage*

- Preconditions should go at the beginning of the method
- Regular code can't precede preconditions
  - Other contracts can precede preconditions (i.e. postconditions)

```
public class Customer
{
    private int _ID;
    public int ID
    {
        get { return _ID; }
        set {
            Contract.Requires(value >0);
            _ID = value;
        }
    }
}
```

```
public class Customer
{
  private int _ID;
  public int ID
  {
    get { return _ID; }
    set {
      if (value >0)
         throw new InvalidArgumentException();
      Contract.EndCodeBlock();
      _ID = value;
    }
  }
}
```

# Preconditions

*Code Contracts vs. Handwritten code*

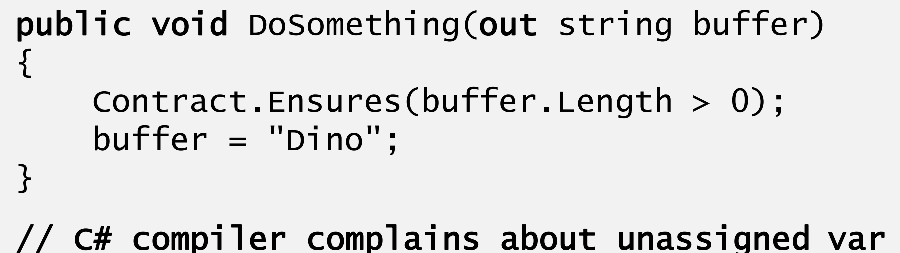
- **If-then-throw** statements are always compiled
- Contracts preconditions can be optionally stripped off
  - Preconditions should be left in the code
  - Exceptions make sense sometimes
- **EndContractBlock** allows to offer a common interface to Code Contracts tools while using legacy code

- Best practice
  - Always check all possible preconditions for a method
  - Up to you to decide what preconditions remain in released code

# Postconditions

## *Available API*

- Method **Ensures**(bool condition)
  - A particular condition must be true at the end of the method
  - Postconditions checked at every method exit
  - Postconditions can appear everywhere in the body
- Method **EnsuresOnThrow<TException>**(bool condition)
  - Specified condition must be true if the given exception  is thrown
- Helper methods
  - Result<T>()
  - OldValue<T>(T variable)
  - ValueAtReturn<T>(out T variable)



# Postconditions

## *Exceptional Returns*

- Exceptional returns are commonly due to:
  - OutOfMemory exceptions
  - StackOverflow exceptions
  - Network or I/O exceptions
- Exceptions raised by dependencies
  - Including exceptions raised by the method on incorrect results got from dependencies
- Use EnsuresOnThrow to check for **specific** exceptions
  - Make a verifiable statement about the state of the object after an anomalous return

# Postconditions

## *Usage*

```
Disassembly | Program.cs | ConsoleApplication1
Address: ConsoleApplication1.Program.Customer.ID.set(int)
                    set
                    {
00000000  push         ebp
00000001  mov          ebp,esp
00000003  push         edi
00000004  push         esi
00000005  push         ebx
00000006  sub          esp,38h
00000009  xor          eax,eax
0000000b  mov          dword ptr [ebp-10h],eax
0000000e  xor          eax,eax
00000010  mov          dword ptr [ebp-1Ch],eax
00000013  mov          dword ptr [ebp-3Ch],ecx
00000016  mov          dword ptr [ebp-40h],edx
00000019  cmp          dword ptr ds:[00359134h],0
00000020  je           00000027
00000022  call         71349119
00000027  nop
                        Contract.Requires(value > 0);
00000028  push          dword ptr ds:[03122088h]
0000002e  cmp           dword ptr [ebp-40h],0
00000032  setg          cl
00000035  movzx         ecx,cl
00000038  xor           edx,edx
0000003a  call          FF8BAFC0
                        _ID = value;
0000003f  mov           eax,dword ptr [ebp-3Ch]
00000042  mov           edx,dword ptr [ebp-40h]
00000045  mov           dword ptr [eax+8],edx
                    }
00000048  nop
00000049  jmp           0000004B
                        Contract.Ensures(_ID == value + 1);
0000004b  mov           eax,dword ptr [ebp-3Ch]
0000004e  mov           eax,dword ptr [eax+8]
00000051  mov           edx,dword ptr [ebp-40h]
00000054  inc           edx
```

```
public class Customer
{
   private int _ID;
   public int ID
   {
      get { return _ID; }
      set {
         Contract.Ensures(_ID == value);
         Contract.Requires(value >0);
         _ID = value;
      }
   }
}
```

```
public bool AddItem(OrderItem item)
{
   Contract.Ensures(
     TotalCost >= Contract.OldValue(TotalCost));

   Contract.Ensures(
     Items.Contains(item) ||
          (Contract.Result<Boolean>() == false));

   Contract.EnsuresOnThrow<IOException>(
      TotalCost == Contract.OldValue(TotalCost));

   :
}
```

# Invariants

## *Available API*

- Method `Invariant(bool condition)`
  - Object-wide condition that is guaranteed to always hold
- All invariants for a class defined in one place
  - No multiple invariant methods in a class
  - Usually a void parameterless method named **ObjectInvariant**
  - Marked **protected** to avoid accidental calls from top-level code
  - Decorated with **ContractInvariantMethod** attribute
- Best practice
  - Decide upon invariants before implementing the class
  - Express the conditions under which the object is in a good state
  - Can be hard to add them at a later time

# Invariants

## *Usage*

- An invariant method won't have any other code in it
  - Invariants checked at the end of all public members on the class

```
[ContractInvariantMethod]
protected void ObjectInvariant()
{
   Contract.Invariant(this.TotalCost >= 0);
   Contract.Invariant(this.GetNextID() > 0);
}
```

# Assert and Assume

## *Available API*

- At run time, fully equivalent to Debug.Assert
  - Not for static analysis run by the static checker tool
  - Used in any place where you can use Debug.Assert
- Static checker attempts to prove any **Assert**
  - Emitting a warning if it fails
- Static checker treats any Assume as definitely true
  - Add the assumption to its collection of facts
  - Use that information for producing its output
- Assumptions and assertions are conditionally defined
  - Exist in the build only when the full-contract symbol or the DEBUG symbol is defined

# Quantifiers

## *Available API*

- Used to iterate a check on all elements of a list
- Method **ForAll**
  - True if condition is met by all elements in the list
- Method **Exists**
  - True if condition is met by at least one elements in the list
  - Enumeration stops at first match

```
public int DoSomething<T>(IEnumerable<T> list)
{
   Contract.Requires(
      Contract.ForAll(list, (T item) => item != null)
   );
}
```

# Interface Contracts

- Can have a contract on an interface or abstract classes
  - Can't add code to interface members (no mixin in C#)
- Define a separate class for the contract
  - Link it to the interface via attribute

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

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

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

# Contracts and Inheritance

- Optionally compiled, contracts are part of the class code
- Contracts inherited by derived classes
  - Regardless of whether you actually invoke base methods in overrides
  - Preconditions and postconditions sum up

```
public class Customer
{
  public virtual void DoWork(int data)
  {
    Contract.Requires(data >10);
    Contract.Ensures(
      data > Contract.OldValue<int>(data)
    );
        :
   }
}
```

```
public class PreferredCustomer : Customer
{
   public override void DoWork(int data)
   {
      // Existing preconditions and
      // postconditions apply no matter
      // you invoke base methods

      base.DoSomeWork(data);
   }
}
```

# Tools

# Rewriter

***At a glance***

- Enabled explicitly to translate contracts into code
  - Full, Pre/Post, Pre-only
- Modifies MSIL instructions to place contract checks where they logically belong
  - You “declare” contracts, the rewriter enforces contracts
  - Grabs implicit contracts from base classes
  - No runtime help in debug sessions without a rewriter
- Rewriter transforms “documentation” in executable code
- Executable is *ccrewrite.exe*

# Static Checker

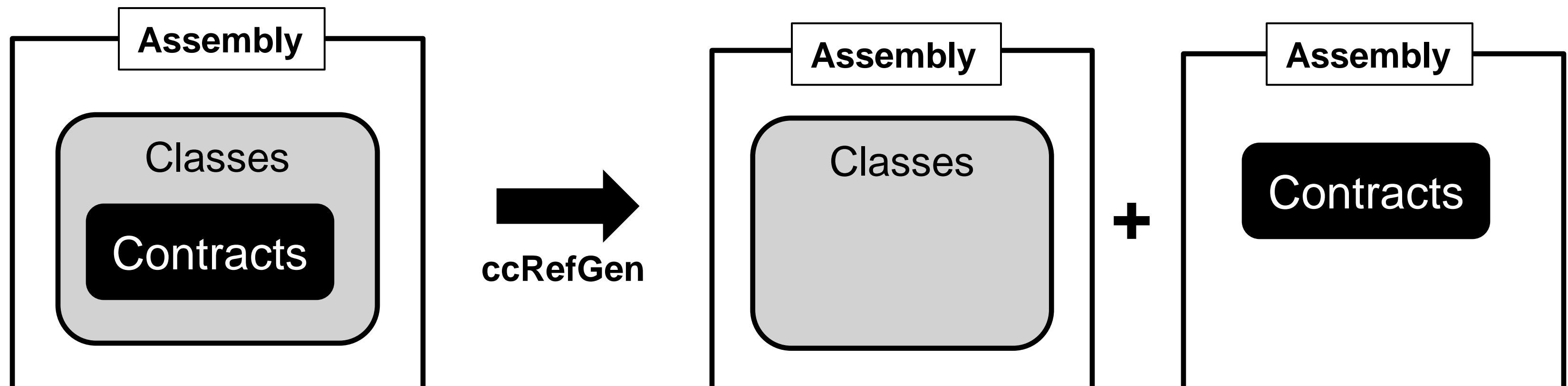
## *At a glance*

- Examine code **without executing** it and **tries** to prove that all of the contracts are satisfied
- List unproven contracts for you to fix
  - Without static checking you should go through all of your code in a debug session to see if the contract is verified
  - Or cover it with unit tests
  - Get to know whether there's a **possibility** of contract violation
- Needs a contracted API to recognize contracts
  - Key difference with **manual** implementation of contracts
  - Attributes let you specify which assemblies, types, or members should be statically checked
- Executable is *cccheck.exe*
  - Requires Visual Studio Team System

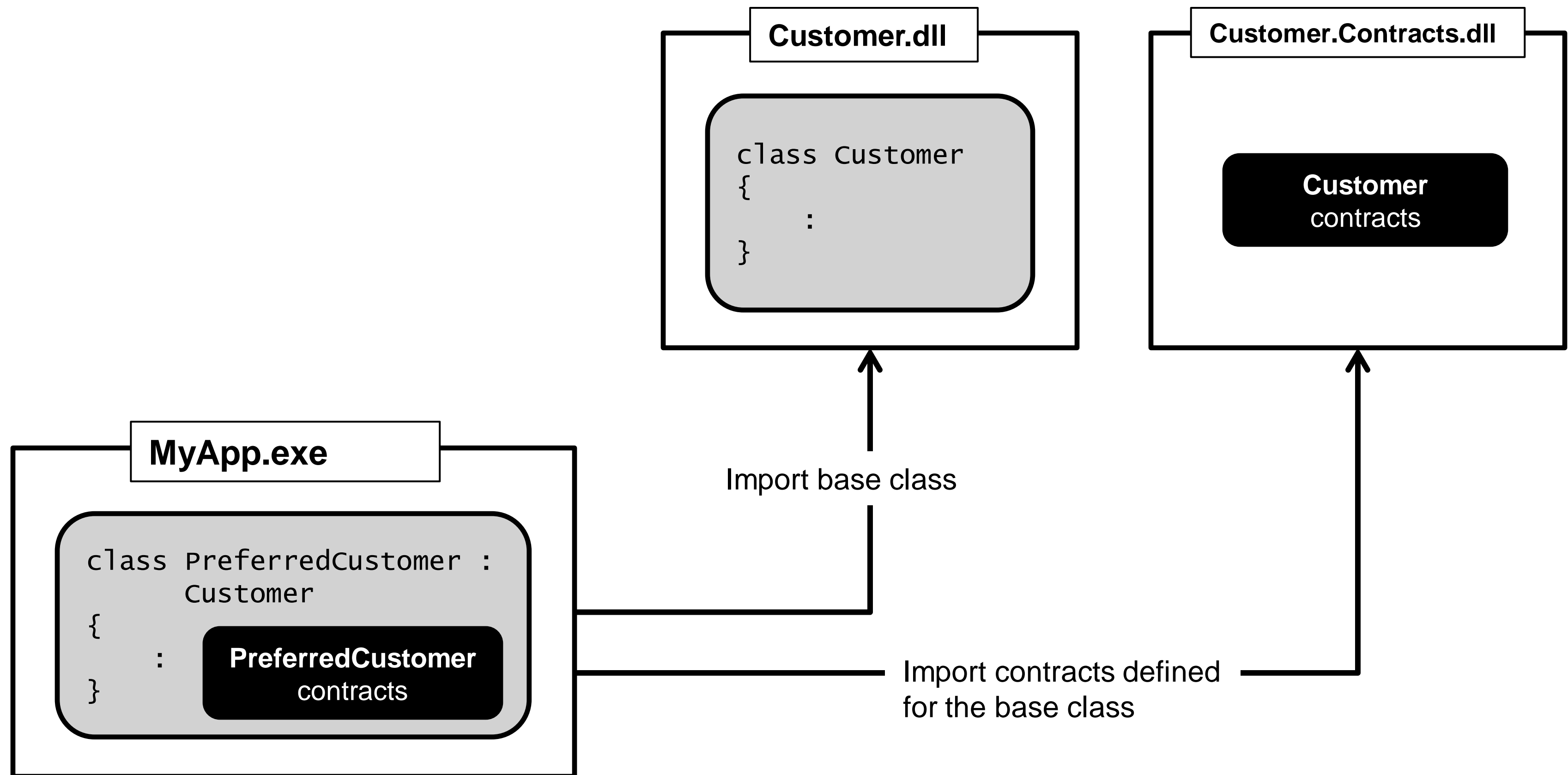
# Assembly Reference Generator

## *At a glance*

- Extract contract information from source code and return a new assembly with contract-only information
- Used by rewriter (and static checker)
  - Import contracts in assemblies that use derived classes
  - Keep contract and code separate
  - Generator must be enabled/configured explicitly
- Executable is *ccrefgen.exe*



# Ref-gen in action



# Code Contracts and Debugging

# Contract Failures

## *Available API*

- When a failure is detected, the contract raises an event
  - ContractFailedException
  - ContractFailedEventArgs
- No default handler (means it'll throw)

```
public sealed class ContractFailedEventArgs : EventArgs
{
   public string Message { get; }
   public string Condition { get; }
   public ContractFailureKind FailureKind { get; }
   public Exception OriginalException { get; }
   public bool Handled { get; }
   public bool Unwind { get; }
   public void SetHandled();
   public void SetUnwind();
   public ContractFailedEventArgs(ContractFailureKind failureKind,
                                 string message,
                                 string condition,
                                 Exception originalException);
}
```

# Contract Failures

## *Usage*

- Register handler to **ContractFailedException** at startup
  - Just before anything else you may have in the code
- May use your handler as a logging mechanism
- Use **SetHandled** to prevent further steps
  - No popup window
- Use **SetUnwind** to clear the stack

```
static void Main(string[] args)
{
   Contract.ContractFailed += new
        EventHandler<ContractFailedEventArgs>(Contract_ContractFailed);
}

static void Contract_ContractFailed(object sender, ContractFailedEventArgs e)
{
   // your code
}
```

# Contract Failures and Unit Tests

*Handling exceptions*

- You don’t want (contract) exceptions in unit tests
  - You rather want exceptions to be reported as test failures

**MSTest**

```
[AssemblyInitialize]
public static void AssemblyInitialize(TestContext testContext)
{
   Contract.ContractFailed += (sender, args) =>
   {
      args.SetHandled();
      args.SetUnwind();
      Assert.Fail(args.Message);       // report as test failure
   };
}
```

# Summary

- Code Contracts as part of the design effort
- Add testability to classes (and code)
- New API coming up in C# 4

# **The** Murphy**'s Corner**

- *All's well that ends.*
- *You can never tell which way the train went by looking at the track.*