DisposableAttribute

```
#region license
// Copyright (c) 2005, Sorin Ionescu
// All rights reserved.
//
// Redistribution and use in source and binary forms, with or without
modification,
// are permitted provided that the following conditions are met:
//
//     * Redistributions of source code must retain the above copyright
notice,
//     this list of conditions and the following disclaimer.
//     * Redistributions in binary form must reproduce the above copyright
notice,
//     this list of conditions and the following disclaimer in the
documentation
//     and/or other materials provided with the distribution.
//     * Neither the name of Sorin Ionescu nor the names of its
//     contributors may be used to endorse or promote products derived from
this
//     software without specific prior written permission.
//
// THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS
IS" AND
// ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
IMPLIED
// WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE
// DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE
LIABLE
// FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
CONSEQUENTIAL
// DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS
OR
// SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
HOWEVER
// CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
LIABILITY,
// OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF
THE USE OF
// THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
#endregion

namespace Boo.Lang.Useful.Attributes

import System
import System.Threading
import Boo.Lang.Compiler
import Boo.Lang.Compiler.Ast
import Boo.Lang.Compiler.Steps
import Boo.Lang.Compiler.TypeSystem
```

```
class DisposableAttribute(AbstractAstAttribute):
"""
Implements the IDisposable pattern for a type.
"""
 _codeBuilder as BooCodeBuilder
 _typeSystemServices as TypeSystemServices
 _context as CompilerContext
 _disposableType as TypeDefinition
 _disposeMethodName as string
 _unmanagedDisposeMethod as ReferenceExpression
 _disposed as Field
 _disposeLock as Field
 _protectedDispose as Method
 _overrideBase as bool
 _specialMethodNames as List

 def constructor():
  self(null, null)

 def constructor(disposeMethodName as StringLiteralExpression):
  self(disposeMethodName, null)

 def constructor(unmanagedDisposeMethod as ReferenceExpression):
  self(null, unmanagedDisposeMethod)

 def constructor(
  disposeMethodName as StringLiteralExpression,
  unmanagedDisposeMethod as ReferenceExpression):

  _unmanagedDisposeMethod = unmanagedDisposeMethod

  if disposeMethodName is not null:
   _disposeMethodName = disposeMethodName.Value

 override def Apply(node as Node):
  assert node isa TypeDefinition

  _disposableType = node as TypeDefinition
  _codeBuilder = CodeBuilder
  _typeSystemServices = TypeSystemServices
  _context = Context
  _specialMethodNames = [
   "constructor",
   "destructor",
   "Dispose",
   "IDisposable.Dispose"]

  if _disposeMethodName is not null:
   _specialMethodNames.Add(_disposeMethodName)

  if _unmanagedDisposeMethod is not null:
   _specialMethodNames.Add(_unmanagedDisposeMethod.Name)
```

```
 CreateIDisposableDerivation()
 CreateDisposedField()
 CreateDisposedObjectCheck()
 CreateDisposeLockField()
 CreateProtectedDisposeMethod()
 CreateDestructor()
 CreateDisposeMethod()

 Context.Parameters.Pipeline.AfterStep += def (
  sender as object,
  e as CompilerStepEventArgs):

  if e.Step isa BindTypeDefinitions:
   RemoveIDisposableDerivation()

   _overrideBase = IsDisposable()

   if _overrideBase:
    _specialMethodNames.Remove("constructor")

    if _unmanagedDisposeMethod is not null:
     _specialMethodNames.Remove(_unmanagedDisposeMethod.Name)

    RemoveMethods(_specialMethodNames)
    CreateProtectedDisposeMethod()

   else:
    CreateIDisposableDerivation()

  if e.Step isa ProcessMethodBodies:
   CreateProtectedDisposeMethodBody()

private def IsDisposable():
 if _typeSystemServices.Map(IDisposable).IsAssignableFrom(
  _disposableType.Entity):

  return true

 else:
  return false

private def CreateIDisposableDerivation():
 _disposableType.BaseTypes.Add(
  _codeBuilder.CreateTypeReference(IDisposable))

private def CreateDisposedField():
 _disposed = Field(
  LexicalInfo: LexicalInfo,
  Name: "___disposed",
  Modifiers: TypeMemberModifiers.Private,
  Initializer: BoolLiteralExpression(LexicalInfo, false))

 _disposableType.Members.Add(_disposed)
```

```
private def CreateDisposeLockField():
 _disposeLock = Field(
  LexicalInfo: LexicalInfo,
  Name: "___disposeLock",
  Modifiers: TypeMemberModifiers.Private,
  Initializer: MethodInvocationExpression(
   LexicalInfo: LexicalInfo,
   Target: ReferenceExpression(
    LexicalInfo: LexicalInfo,
    Name: "object")))

 _disposableType.Members.Add(_disposeLock)

private def CreateDestructor():
 finalizer = Destructor(LexicalInfo: LexicalInfo, Name: "destructor")
 protectedDisposeInvocation = AstUtil.CreateMethodInvocationExpression(
  LexicalInfo,
  ReferenceExpression(
   LexicalInfo: LexicalInfo,
   Name: "Dispose"),
  BoolLiteralExpression(LexicalInfo, false))

 finalizer.Body.Add(protectedDisposeInvocation)

 _disposableType.Members.Add(finalizer)

private def CreateDisposeMethod():
 if _disposeMethodName is not null:
  CreateDisposeMethod(_disposeMethodName)
  CreateExplicitInterfaceDispose()

 else:
  CreateDisposeMethod("Dispose")

private def CreateDisposeMethod(disposeMethodName):
 dispose = Method(
  LexicalInfo: LexicalInfo,
  Name: disposeMethodName,
  Modifiers: TypeMemberModifiers.Public)

 protectedDisposeInvocation = AstUtil.CreateMethodInvocationExpression(
  LexicalInfo,
  ReferenceExpression(
   LexicalInfo: LexicalInfo,
   Name: "Dispose"),
  BoolLiteralExpression(LexicalInfo, true))

 gcInvocation = AstUtil.CreateMethodInvocationExpression(
  LexicalInfo,
  AstUtil.CreateReferenceExpression("System.GC.SuppressFinalize"),
  SelfLiteralExpression(LexicalInfo))
```

```
 dispose.Body.Add(protectedDisposeInvocation)
 dispose.Body.Add(gcInvocation)

 _disposableType.Members.Add(dispose)

private def CreateExplicitInterfaceDispose():
 iDisposableInfo = ExplicitMemberInfo(
  LexicalInfo,
  InterfaceType: SimpleTypeReference(LexicalInfo, "IDisposable"))

 iDisposableDispose = Method(
  LexicalInfo: LexicalInfo,
  ExplicitInfo: iDisposableInfo,
  Name: "Dispose",
  Modifiers: TypeMemberModifiers.Private)

 disposeInvocation = MethodInvocationExpression(
  LexicalInfo: LexicalInfo,
  Target: ReferenceExpression(_disposeMethodName))

 iDisposableDispose.Body.Add(disposeInvocation)

 _disposableType.Members.Add(iDisposableDispose)

private def CreateProtectedDisposeMethod():
 _protectedDispose = Method(LexicalInfo, Name: "Dispose")
 _protectedDispose.Parameters.Add(
  ParameterDeclaration(
   LexicalInfo: LexicalInfo,
   Name: "disposing",
   Type: SimpleTypeReference(LexicalInfo, "bool")))

 if _overrideBase:
  _protectedDispose.Modifiers |= \
   TypeMemberModifiers.Protected | TypeMemberModifiers.Override

 else:
  _protectedDispose.Modifiers |= \
   TypeMemberModifiers.Protected | TypeMemberModifiers.Virtual

 _disposableType.Members.Add(_protectedDispose)

private def CreateProtectedDisposeMethodBody():
 disposingReference = _codeBuilder.CreateReference(
  _protectedDispose.Parameters[0].Entity)

 ifNotDisposedCondition = _codeBuilder.CreateBoundBinaryExpression(
  _disposed.Type.Entity,
  BinaryOperatorType.Inequality,
  _codeBuilder.CreateReference(_disposed.Entity),
  _codeBuilder.CreateBoolLiteral(true))

 ifDisposingCondition = _codeBuilder.CreateBoundBinaryExpression(
```

```
 _disposed.Type.Entity,
 BinaryOperatorType.Equality,
 disposingReference,
 _codeBuilder.CreateBoolLiteral(true))

ifNotDisposed = IfStatement(
 LexicalInfo,
 ifNotDisposedCondition,
 Block(LexicalInfo),
 null)

ifDisposing = IfStatement(
 LexicalInfo,
 ifDisposingCondition,
 Block(LexicalInfo),
 Block(LexicalInfo))

ifNotDisposed.TrueBlock.Add(ifDisposing)

for typeMember in _disposableType.Members:
 if typeMember isa Field:
  field = typeMember as Field

  if _typeSystemServices.Map(IDisposable).IsAssignableFrom(
   field.Type.Entity):

    ifFieldNotNullCondition = \
     _codeBuilder.CreateBoundBinaryExpression(
      field.Type.Entity,
      BinaryOperatorType.Inequality,
      _codeBuilder.CreateReference(field.Entity),
      _codeBuilder.CreateNullLiteral())

    ifFieldNotNull = IfStatement(
     LexicalInfo,
     ifFieldNotNullCondition,
     Block(LexicalInfo),
     null)

    fieldReference = _codeBuilder.CreateReference(
     field.Entity)

    castExpression = _codeBuilder.CreateCast(
     _typeSystemServices.Map(
      IDisposable),
     fieldReference)

    disposeReference = _typeSystemServices.Map(
     typeof(IDisposable).GetMethod("Dispose"))

    disposeInvocation = _codeBuilder.CreateMethodInvocation(
     castExpression, disposeReference)
```

```
    fieldNullAssignment = _codeBuilder.CreateAssignment(
     fieldReference.CloneNode(),
     _codeBuilder.CreateNullLiteral())

    ifFieldNotNull.TrueBlock.Add(disposeInvocation)
    ifFieldNotNull.TrueBlock.Add(fieldNullAssignment)
    ifDisposing.TrueBlock.Add(ifFieldNotNull)

if _unmanagedDisposeMethod is not null:
 unmanagedDisposeMethodInvocation = \
  _codeBuilder.CreateMethodInvocation(
   _codeBuilder.CreateSelfReference(_disposableType.Entity),
   _disposableType.Members[_unmanagedDisposeMethod.Name].Entity)

 ifDisposing.FalseBlock.Add(unmanagedDisposeMethodInvocation)

if _overrideBase:
 superDisposeReference = cast(
  InternalMethod,
  _protectedDispose.Entity).Overriden

 superDisposeInvocation = _codeBuilder.CreateMethodInvocation(
  _codeBuilder.CreateSuperReference(
   cast(IType, _disposableType.Entity).BaseType),
   superDisposeReference,
   disposingReference.CloneNode())

 ifNotDisposed.TrueBlock.Add(superDisposeInvocation)

disposedTrueAssignment = _codeBuilder.CreateAssignment(
 _codeBuilder.CreateReference(_disposed.Entity),
 _codeBuilder.CreateBoolLiteral(true))

ifNotDisposed.TrueBlock.Add(disposedTrueAssignment)

monitorEnterInvocation = _codeBuilder.CreateMethodInvocation(
 _typeSystemServices.Map(typeof(Monitor).GetMethod("Enter")),
 _codeBuilder.CreateReference(_disposeLock.Entity))

monitorExitInvocation = _codeBuilder.CreateMethodInvocation(
 _typeSystemServices.Map(typeof(Monitor).GetMethod("Exit")),
 _codeBuilder.CreateReference(_disposeLock.Entity))

protectedBlock = Block(LexicalInfo)
protectedBlock.Add(ifNotDisposed)

ensureBlock = Block(LexicalInfo)
ensureBlock.Add(monitorExitInvocation)

_protectedDispose.Body.Add(monitorEnterInvocation)
_protectedDispose.Body.Add(
 TryStatement(
  LexicalInfo: LexicalInfo,
```

```
    ProtectedBlock: protectedBlock,
    EnsureBlock: ensureBlock))

  private def CreateDisposedObjectCheck():
   for typeMember in _disposableType.Members:
    if not typeMember.IsPrivate:
     if typeMember isa Method:
      method = typeMember as Method

      if not method.Name in _specialMethodNames:
       CreateDisposedObjectCheck(method)

     elif typeMember isa Property:
      property = typeMember as Property

      if property.Getter is not null:
       CreateDisposedObjectCheck(property.Getter)

      if property.Setter is not null:
       CreateDisposedObjectCheck(property.Setter)

  private def CreateDisposedObjectCheck(method as Method):
   exceptionCreation = MethodInvocationExpression(
    LexicalInfo: LexicalInfo,
    Target: AstUtil.CreateReferenceExpression(
     "System.ObjectDisposedException"))

   exceptionCreation.Arguments.Add(
    StringLiteralExpression(_disposableType.Name))

   trueBlock = Block(LexicalInfo)
   trueBlock.Add(
    RaiseStatement(
     LexicalInfo: LexicalInfo,
     Exception: exceptionCreation))

   body = method.Body
   method.Body = Block(LexicalInfo)
   method.Body.Add(
    IfStatement(
     LexicalInfo,
     ReferenceExpression(
      LexicalInfo: LexicalInfo,
      Name: _disposed.Name),
     trueBlock, null))

   method.Body.Add(body)

  private def RemoveIDisposableDerivation():
   baseTypes = _disposableType.BaseTypes.ToArray()
   _disposableType.BaseTypes = TypeReferenceCollection()

   for type in baseTypes:
```

```
   if  type.ToString() != "System.IDisposable":
    _disposableType.BaseTypes.Add(type)

  private def RemoveMethods(methodNames as List):
   typeMembers = _disposableType.Members.ToArray()
   _disposableType.Members = TypeMemberCollection()

   for member in typeMembers:
    if member isa Method:
     method = member as Method

    if not method.Name in methodNames:
     _disposableType.Members.Add(member)

    else:
```

```
    _disposableType.Members.Add(member)
```