AsyncMethodAttribute

See the Async attribute in the examples folder included with boo.

Below is an updated version of this attribute that could be included with Boo.Lang.Useful:

```
#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.Runtime.Remoting.Messaging
import Boo.Lang.Compiler
import Boo.Lang.Compiler.Ast
import Boo.Lang.Compiler.Steps
```

```
class
AsyncMethodAttribute(AbstractAstAttribute):
"""
Adds the asynchronous helpers Begin<Method>
and
End<Method> for a method.
"""
 _method as Method
 _accessModifiers as TypeMemberModifiers
 _disposed as ReferenceExpression

 def constructor():
  pass

 def constructor(disposed as
ReferenceExpression):
  _disposed = disposed

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

  _method = node

  # Set the wrapper methods to the same
accessibility
  # as the wrappedmethod.
  if _method.IsProtected and
_method.IsInternal:
   _accessModifiers =
TypeMemberModifiers.Protected
```

```
   _accessModifiers = _accessModifiers |
TypeMemberModifiers.Internal

  elif _method.IsPublic:
   _accessModifiers =
TypeMemberModifiers.Public

  elif _method.IsProtected:
   _accessModifiers =
TypeMemberModifiers.Protected

  elif _method.IsInternal:
   _accessModifiers =
TypeMemberModifiers.Internal

  elif _method.IsPrivate:
   _accessModifiers =
TypeMemberModifiers.Private

  CreateBeginMethod()
  CreateEndMethod()

 private def CreateBeginMethod():
  beginMethod = Method(
   LexicalInfo: LexicalInfo,
   Name: "Begin${_method.Name}",
   Modifiers: _accessModifiers,
   ReturnType:
SimpleTypeReference(LexicalInfo,
"System.IAsyncResult"))
```

```
  # Copy the BeginInvoke parameters:

beginMethod.Parameters.ExtendWithClones(_met
hod.Parameters)

beginMethod.Parameters.Add(ParameterDeclarat
ion(
   LexicalInfo: LexicalInfo,
   "callback",
   SimpleTypeReference(LexicalInfo,
"System.AsyncCallback")))

  beginMethod.Parameters.Add(
   ParameterDeclaration(
    LexicalInfo: LexicalInfo,
    "state",
    SimpleTypeReference(LexicalInfo,
"object")))

  # Creates the delegate that will call the
wrapped method asynchronously.
  asyncInvocation =
MethodInvocationExpression(
   LexicalInfo: LexicalInfo,
   Target:
AstUtil.CreateReferenceExpression(
    "${_method.Name}.BeginInvoke"))

  for parameter in beginMethod.Parameters:
   asyncInvocation.Arguments.Add(
    ReferenceExpression(LexicalInfo,
```

```
parameter.Name))

  if _disposed is not null:
   CreateDisposedObjectCheck(beginMethod)

  beginMethod.Body.Add(
   ReturnStatement(
    LexicalInfo: LexicalInfo,
    Expression: asyncInvocation))


_method.DeclaringType.Members.Add(beginMetho
d)

 private def CreateEndMethod():
  endMethod = Method(
   LexicalInfo: LexicalInfo,
   Name: "End${_method.Name}",
   Modifiers: _accessModifiers)

  # Copy the EndInvoke parameters:
EndMethod(result as IAsyncResult).
  endMethod.Parameters.Add(
   ParameterDeclaration(
    LexicalInfo: LexicalInfo,
    "result",
    SimpleTypeReference(LexicalInfo,
"System.IAsyncResult")))

  # Creates the delegate that will call the
wrapped method asynchronously.
```

```
  asyncInvocation =
MethodInvocationExpression(
   LexicalInfo: LexicalInfo,
   Target:
AstUtil.CreateReferenceExpression(
    "${_method.Name}.EndInvoke"))

  asyncInvocation.Arguments.Add(
   ReferenceExpression(LexicalInfo,
"result"))

  if _disposed is not null:
   CreateDisposedObjectCheck(endMethod)

  endMethod.Body.Add(
   ReturnStatement(
    LexicalInfo: LexicalInfo,
    Expression: asyncInvocation))


_method.DeclaringType.Members.Add(endMethod)

  # Cache the voidType reference because the
CompilerContext
  # will be lost after this method returns.
  # See AbstractCompilerComponent.Dispose().
  voidType =
Context.TypeSystemServices.VoidType

  Context.Parameters.Pipeline.AfterStep +=
def (
```

```
   sender as object,
   e as CompilerStepEventArgs):

   if e.Step isa ProcessMethodBodies:
    if _method.ReturnType.Entity is
voidType:
     returnStatement =
endMethod.Body.Statements[-1] as
ReturnStatement

     endMethod.Body.Statements.Replace(
      returnStatement,
      ExpressionStatement(
       LexicalInfo: LexicalInfo,
       Expression:
returnStatement.Expression))

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

  exceptionCreation.Arguments.Add(
   StringLiteralExpression(LexicalInfo,
_method.DeclaringType.Name))

  trueBlock = Block()
```

```
trueBlock.Add(
 RaiseStatement(
  LexicalInfo: LexicalInfo,
  Exception: exceptionCreation))

method.Body.Add(
 IfStatement(LexicalInfo,
```

```
_disposed.CloneNode(), trueBlock, null))
```