C# Best Practices

Tom Becker

Silicon Valley Code Camp 2019

Online Material

- https://github.com/TomBecker-BD/csharp-essentials
- Long-form articles
- Utility library
- Sample code
- Unit tests

C# Best Practices

- Code-level design patterns
- Simple and easy to use
- Will help you see problems in code more quickly
- Important for scalable and reliable code
- Not commonly taught in books or schools

Topics

- Memory and resource management
- Properties
- Error handling
- Exception safety

Memory and Resource Management

- Garbage collection works great
  - Except for event handlers
  - Except for non-memory resources
- Solution: Use the Disposable pattern
  - Manage memory as if there isn’t a garbage collector
  - Then the garbage collector really does work great

Zombie Object Anti-pattern

- The amp's event has a reference to the zombie object
- The zombie object will not be garbage collected
- The zombie object will continue to receive events

```
public class ZombieObject
{
    IAmp _amp;

    public ZombieObject(IAmp amp)
    {
        _amp = amp;
        _amp.PropertyChanged += Amp_PropertyChanged;
    }
```

Preventing Zombies

- Implement IDisposable using the Disposable pattern
- Remove event handlers in the Dispose method

```
public class ZombieFree : IDisposable
{
    protected virtual void Dispose(bool disposing)
    {
        if (disposing)
        {
            if (_amp != null)
            {
                _amp.PropertyChanged -= Amp_PropertyChanged;
                _amp = null;
            }
        }
    }

    public void Dispose()
    {
        Dispose(true);
    }
```

Non-Memory Resources

- Non-memory resources need to be released in your application code
  - Files, network connections, threads, mutexes, OS handles, etc.
- Finalizers are called only when the application is low on memory
  - Causes unexpected slowdowns

```
public class LazyCleanup
{
    SafeMemoryMappedFileHandle _file;

    ~LazyCleanup()
    {
        if (_file != null)
        {
            _file.Dispose();
            _file = null;
        }
    }
```

Resource Cleanup

- Release non-memory resources in the Dispose method
  - Minimizes resource usage
  - Smoother performance
- Can release resources sooner if the owning object is done with them

```
public class BetterCleanup : IDisposable
{
   SafeMemoryMappedFileHandle _file;

    protected virtual void Dispose(bool disposing)
    {
        if (disposing)
        {
            if (_file != null)
            {
                _file.Dispose();
                _file = null;
             }
        }
    }
```

Properties

- When to use auto properties
- When to use an old-school property with a backing field
- Managing event handlers
- Avoiding race conditions

Immutable State

- Auto properties are great for making immutable (read-only) state objects
- Object contains only state, no logic
- Object can be replaced but not modified
- Thread-safe
- Highly scalable

```
public class AmpState
{
    public int Level { get; private set; }

    public AmpState(int level)
    {
        Level = level;
    }
```

Binding

- Set method raises the PropertyChanged event only if the value actually changed
- Prevents event storms
- Backing field is needed to detect non-changes

```
public class AmpViewModel : INotifyPropertyChanged
{
    int _level;
    public int Level
    {
        get { return _level; }
        set
        {
            if (_level != value)
            {
                _level = value;
                OnPropertyChanged(nameof(Level));
            }
        }
    }
```

Managing Event Handlers

- Modularize adding and removing event handlers
- Scales to multiple events
- Property can be private

```
public class Guitar : IDisposable
{
    IAmp _amp;
    IAmp Amp
    {
        get { return _amp; }
        set {
            if (_amp != value) {
                if (_amp != null) {
                    _amp.PropertyChanged -= Amp_PropertyChanged;
                }
                _amp = value;
                if (_amp != null) {
                    _amp.PropertyChanged += Amp_PropertyChanged;
                }
            }
        }
    }
```

Removing Event Handlers

- Easy

```
protected virtual void Dispose(bool disposing)
{
    if (disposing)
    {
        Amp = null;
    }
}
```

Avoiding Race Conditions

- The first example gets the Amp twice
- No guarantee it will be the same object
- Getting the Amp property could be expensive
- Better to use a local variable
  - Cost is very low
  - Behavior is more predictable

```
// Possible race
const int MaxLevel = 11;
if (guitar.Amp.Level < MaxLevel)
{
    guitar.Amp.Level = MaxLevel;
}

// Better
var amp = guitar.Amp;
if (amp.Level < MaxLevel)
{
    amp.Level = MaxLevel;
}
```

Error Handling

- What if there is an exception in your application?
- In a desktop application, the application exits
  - User loses unsaved work
- In a web application, the client gets an HTTP status 400
- Error message is not user-friendly

Error Handling Strategy

- Define an IErrorHandler service and inject it
- Define an IAsyncCommand interface that extends ICommand
  - ICommand is useful but it predates async
- ExecuteAsync method catches exceptions
- Pass the command name and the exception to the error handler
- Web API catch errors in each API controller method
  - See the “Chromate” project (in resources) for examples

IErrorHandler Interface

```
public interface IErrorHandler
{
    void HandleError(string operation, Exception ex);
}
```

MessageBoxErrorHandler

```
public void HandleError(string operation, Exception ex)
{
    string message = string.Format("Could not {0} because {1}", operation, ex.Message);
    _logger.Error(ex, message);
    _messageBox.Show(message, "Error", MessageBoxButton.OK, MessageBoxImage.Error);
}
```

UnitTestErrorHandler

```
public void HandleError(string operation, Exception ex)
{
    Console.Error.WriteLine("Could not {0} because {1}", operation, ex.Message);
    throw new ApplicationException(string.Format("Could not {0}", operation), ex);
}
```

IAsyncCommand Interface

```
// For reference – from System.Windows.Input
public interface ICommand
{
    event EventHandler CanExecuteChanged;
    bool CanExecute(object parameter);
    void Execute(object parameter);
}

// Add async execution
public interface IAsyncCommand : ICommand
{
    Task ExecuteAsync(object parameter);
}
```

AsyncCommand Implementation

```
public Task ExecuteAsync(object parameter)
{
    Executing = true;
    return _execute(parameter)
        .ContinueWith(t =>
        {
            if (t.IsFaulted)
            {
                _errorHandler.HandleError(_operation, t.Exception);
            }
            Executing = false;
        });
}

public void Execute(object parameter)
{
    ExecuteAsync(parameter);
}
```

Using AsyncCommand

```
public class CartViewModel
{
    AsyncCommand _checkout;
    List<string> _cart = new List<string>();

    public IAsyncCommand CheckoutCommand
    {
        get { return _checkout; }
    }

    public List<string> Cart
    {
        get { return _cart; }
        set
        {
            _cart = value;
            _checkout.RaiseCanExecuteChanged();
        }
    }
```

```
public CartViewModel(IErrorHandler errorHandler)
{
    _checkout = new AsyncCommand("checkout",
        errorHandler, CanCheckout, Checkout);
}

bool CanCheckout(object parameter)
{
    return _cart.Count > 0;
}

Task Checkout(object parameter)
{
    return Task.Run(() =>
    {
        // TODO: Confirm order and payment
        Cart = new List<string>();
    });
}
```

Unit Testing Example

```
[Test()]
public void TestCheckoutEmptiesCart()
{
    var vm = new CartViewModel(new UnitTestErrorHandler());
    vm.Cart = new List<string> { "Wensleydale" };
    vm.CheckoutCommand.ExecuteAsync(null).Wait();
    Assert.That(vm.Cart, Is.Empty);
}
```

Unhandled Exception Handler

```
public static void Main(string[] args)
{
    AppDomain.CurrentDomain.UnhandledException += CurrentDomain_UnhandledException;
    try
    {
        // Run the application
    }
    catch (Exception ex)
    {
        LogError(ex, "Error starting the application");
        MessageBox.Show(string.Format("Error starting the application: {0}", ex.Message),
            “Error", MessageBoxButton.OK, MessageBoxImage.Error);
    }
}
static void CurrentDomain_UnhandledException(object sender, UnhandledExceptionEventArgs e)
{
    Exception ex = e.ExceptionObject as Exception;
    if (ex != null)
    {
        LogError(ex, "Unhandled exception");
    }
}
```

Exception Safety

- When an exception occurs
  - Objects can be left in a partially modified state
  - Obects can be leaked (and may become zombies)
  - Can cause serious errors and instability
- Cleanup code is needed
  - Where does it go?
  - How can you make sure it works?

Levels of Exception Safety

- **Basic**: If a method has an exception, affected objects can be disposed safely, and no resources are leaked.
- **Strong**: If a method has an exception, the state of the program is left the same as it was before the method was called.
- **No-throw**: The method always succeeds and never emits an exception.

Strategy

- All code must be at least **basic** exception safe
- Changes to existing objects must be at least **strong** exception safe
- Use **no-throw** exception safe swap methods for the **exception-neutral** coding pattern

Basic Exception Safety

- Minimum requirement
- Good enough for most methods
- Easy
- Many methods are already basic exception safe

```
public void ConnectAmp()
{
    _amp?.Dispose();
    _amp = new Amp();
    _amp.Level = 11;
}

public void Dispose()
{
    _amp?.Dispose();
    _amp = null;
    _effect?.Dispose();
    _effect = null;
}
```

Not Safe

- If there is an exception
  - New Amp is leaked
  - StreamReader is leaked
- Even without an exception
  - Old Amp is leaked
  - Old Effect is leaked

```
public void Config(string path) // BAD
{
    var reader = File.OpenText(path);
    _effect = new Effect(reader.ReadLine());
    _amp = new Amp()
    {
        Level = int.Parse(reader.ReadLine())
    };
    reader.Close();
}
```

Making it Basic Exception Safe

- Old Amp is always disposed
- Old Effect is always disposed
- StreamReader is always closed
- If there is an exception
  - New Amp will not be leaked

```
public void Config(string path) // BASIC
{
    _amp?.Dispose();
    _effect?.Dispose();
    using (var reader = File.OpenText(path))
    {
        _amp = new Amp();
        _amp.Level = int.Parse(reader.ReadLine());
        _effect = new Effect(reader.ReadLine());
    }
}
```

Special Case for Constructors

- If an exception occurs in the constuctor
  - Object is not created
  - Dispose cannot be called
- Constructor must handle exceptions

```
public Guitar(string effectName, int level) // BAD
{
    _effect = new Effect(effectName);
    _amp = new Amp() { Level = level };
}

public Guitar(int level, string effectName) // BASIC
{
    try
    {
        _amp = new Amp();
        _amp.Level = level;
        _effect = new Effect(effectName);
    }
    catch
    {
        _amp?.Dispose();
        _effect?.Dispose();
        throw;
    }
}
```

Strong Exception Safety

- Success:
  - tempAmp replaces the old amp
- Exception:
  - tempAmp is not leaked
  - old amp is still there
  - object state is unmodified

```
public void ConnectAmp() // STRONG
{
    Amp tempAmp = new Amp();
    try
    {
        tempAmp.Level = 11;
    }
    catch
    {
        tempAmp.Dispose();
        throw;
    }
    _amp?.Dispose();
    _amp = tempAmp;
}
```

No-Throw Exception Safety

- Dispose methods should never throw exceptions
- Swap methods never throw exceptions
  - Useful for the exception-neutral coding pattern

```
public static class Util
{
    public static void Swap<T>(ref T a, ref T b)
    {
        T temp = a;
        a = b;
        b = temp;
    }
```

Higher-Level Swap Methods

- Swap all the fields
- No memory allocation
- Never fails

```
public static void Swap(Guitar a, Guitar b) // NOTHROW
{
    Util.Swap(ref a._amp, ref b._amp);
    Util.Swap(ref a._effect, ref b._effect);
}
```

Exception-Neutral Coding Pattern

- Code is the same (neutral) for success and exception cases
  - There is no catch
- Single code path
- Simpler!
- Cleanup code is tested in all use cases

```
public void ConnectAmp() // STRONG
{
    Amp tempAmp = new Amp();
    try
    {
        tempAmp.Level = 11;
        Util.Swap(ref _amp, ref tempAmp);
    }
    finally
    {
        tempAmp.Dispose();
    }
}
```

Exception-Neutral Coding Pattern

```
try
{
    // 1. Create new state objects (or copy existing).
    // 2. Modify new state objects.
    // 3. Swap new and old state objects (no-throw).
}
finally
{
    // 4. Delete unused state objects.
}
```

Resources

- C# Essentials — sample code and extended documentation for this presentation
  - https://github.com/TomBecker-BD/csharp-essentials
- HTML5 UI including error handling
  - https://github.com/TomBecker-BD/Chromate
- Exception safety in C++
  - https://www.boost.org/community/exception_safety.html
  - http://exceptionsafecode.com