Michael Stal
Michael.Stal@siemens.com

präsentiert

Effective C#

ASP konferenz VS 2005 VB moves SQL konferenz

Werbeseite 😊

(c) 2005, Michael Stal
Patterns and Platforms

- Most patterns are independent of specific platforms, languages, paradigms, ...
- Reason: Patterns are only blueprints.
- However, the platform used has an impact on pattern implementations.
- Object-Orientation as example:
  - On one hand, patterns are not restricted to OO environments.
  - On the other hand, OO features are really helpful for implementing patterns:
    - encapsulation, polymorphism, inheritance, interfaces
.NET/ C# Features

- Runtime System: Garbage Collection is really useful. Mind its non-determinism!
- Language Interoperability: E.g. Strategy: you may provide different strategies in different languages.
- Uniform Type System: Helps to build containers and ease implementations.
- Multithreading Support: Important for patterns such as Leaders/Followers.

.NET/ C# Features (cont’d)

- Reflection is important for dynamic reconfiguration. Example: Component Configurator, (Dynamic) Proxy.
- Delegates and Events: Many patterns use Observer as sub-pattern.
- v2 Generics: E.g., parametrization with strategies.
- v2 Partial Types: Separate concerns on the class level.
Pattern Variations

- Depending on architectural granularity and context we can differentiate the following styles:
  - **Idioms**
  - Design Patterns
  - Architectural Patterns
  - Best Practice Patterns
- There are even more styles but we won’t cover other flavors in this talk.

(c) 2005, Michael Stal

Idioms

- Idioms represent patterns applicable to a specific paradigm, language, or system architecture.
- Thus, idioms are less focused on application domains.
- Often, idioms are only useful in one concrete context.
- Examples: Explicit Termination Method, Object Resurrection in .NET.
Explicit Termination Method

- Problem: Assure that resources are freed.
- Forces:
  - .NET Garbage Collection is non-deterministic with respect to finalization.
  - Resources denote limited entities that should be only acquired for a minimum time.
- Solution idea: Make resource release explicit.

ETM Idiom

```csharp
class ResourceHolder: System.IDisposable {
    ResourceHandle h; // a limited resource
    // ... further parts
    public void Dispose() {
        Dispose(true);
        GC.SuppressFinalize(this);
    } 
    protected void Dispose(bool isDisposing) {
        // free resource
        // depending on isDisposing
    } 
    ~ResourceHolder () { 
        Dispose(false);
    }
}
```

(c) 2005, Michael Stal
Client Code

```csharp
try {
    r = new ResourceHolder();
    // use the resource here …
}
finally {
    r.Dispose();
}
```

// Optimization:
```csharp
using (r = new ResourceHolder()) {
    // using resource
}
```

Object Resurrection Idiom

- Problem: Resurrection of large objects.
- Forces:
  - Large structures are expensive to create and to keep in memory.
  - Reallocation and deletion of these structures is non-deterministic.
- Solution: Use weak references to free objects and recreate them when necessary.
Object Resurrection Idiom

```java
// Resource Allocation:
LargeObject large = new LargeObject(/* params */);

// Introduce weak reference:
WeakReference weak = new WeakReference(large);

// When object is not used anymore deallocate it:
large = null;

// Later on object is re-used. Try if it still exists:
large = wr.Target;
if (null == large) large = new LargeObject(/* params */);
// ... otherwise object is resurrected
```

(c) 2005, Michael Stal

Static Factory Methods

```java
class Wrapper {
    private int val;
    public int Value { get { return val; } }
    private Wrapper(int i) { val = i; }
    public static Wrapper valueOf(int arg) {
        return new Wrapper(arg);
    }
}
```

- Static factory methods have several advantages:
  - They have names
  - For optimization reasons they are not required to create objects each time (but can use caching and other means)
  - They can also return sub-types
- Disadvantage: Classes with inaccessible constructors cannot be subclassed

(c) 2005, Michael Stal
Immutable Classes

- Immutable classes are classes
  - where instances cannot be modified
  - are easier to design, implement, and use
- Example:

```csharp
class Point { // example class
    private readonly double a;
    private readonly double b;
    public Point(double op1, double op2)
    { a = op1; b = op2; }
    public Point add(Point rhs) {
        return new Point(a + rhs.a, b + rhs.b);
    }
    public double A { get { return a; } }
    public double B { get { return b; } }
}
```

Immutable Classes - Rules (1)

- Don't provide setters or mutator methods
- Make the class sealed to prevent malicious subclassing
- Make all fields readonly and private; use constructors or static factory methods
- Make defensive copies of mutable arguments passed by clients:

```csharp
class Argument { public int val; }
sealed class Wrapper {
    private readonly Argument arg;
    public Wrapper(Argument a) {
        arg = a; // wrong: this should have been copied
    }
    public Argument A { get { return arg; } }
    public double A { get { return a.val; } }
    public double B { get { return b; } }
}
```

Argument a = new Argument(); a.val = 4711;
Wrapper w = new Wrapper(a);
a.val = 42; // external client changes w
Console.WriteLine(w.A.val); // 42 => immutability violated!
Immutable Classes – Rules (2)

- Don’t let clients obtain references to internal mutable fields. Make defensive copies:

```csharp
class Argument { public int val; }
sealed class Wrapper {
    private readonly Argument arg;
    public Wrapper(Argument a) {
        arg = new Argument(); arg.val = a.val;
    }
    public Argument A {
        get {
            Argument tmp = new Argument();
            tmp.val = arg.val;
            return tmp;
        }
    }
}
```

(c) 2005, Michael Stal

Immutable Types – Companion Idiom

- Strings are immutable objects; every operation yields a new string instance.
- Using the string class is expensive:

```csharp
System.String s; // 4 instances are created
s = "Hello"; s += " "; s += "Universe"; s += "!";
```

- Prefer StringBuilder instead:

```csharp
StringBuilder sb = new StringBuilder(15);
sb.Append("Hello"); sb.Append(" "); sb.Append("Universe");
sb.Append("!");
String s = sb.ToString();
```

- Consider similar strategy for your own custom immutable types

(c) 2005, Michael Stal
Value Types versus Reference Types

- Use value types for transfer objects (lightweight data aggregates) that focus on data but not on behavior
- Value types are not polymorphic and don't have sub-types
- It is non-trivial to migrate between both types:

```csharp
/* class */ struct UserType
{
    private int s;
    public UserType(int secret) { s = secret; }
    public int Secret {
        set { s = value; }
        get { return s; } }
}
static void changer(UserType u) { u.Secret += 1; }
static void Main(string[] args)
{
    UserType u = new UserType(42);
    changer(u);
    Console.WriteLine(u.Secret); // result depends on u
}
```

Minimize Boxing/ Unboxing

- Boxing between value types and reference parts involves copy operations
  
  **Boxing**

<table>
<thead>
<tr>
<th>System.Int32 value = 14;</th>
</tr>
</thead>
<tbody>
<tr>
<td>System.Object o = value;</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Stack</th>
<th>Heap</th>
</tr>
</thead>
<tbody>
<tr>
<td>14</td>
<td>alloc 14</td>
</tr>
<tr>
<td>copy</td>
<td></td>
</tr>
</tbody>
</table>

  
  **Unboxing**

<table>
<thead>
<tr>
<th>System.Int32 value = 14</th>
</tr>
</thead>
<tbody>
<tr>
<td>System.Int32 0 = value;</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Stack</th>
<th>Heap</th>
</tr>
</thead>
<tbody>
<tr>
<td>14</td>
<td>alloc 14</td>
</tr>
<tr>
<td>copy</td>
<td></td>
</tr>
</tbody>
</table>
Different Means to Prevent Boxing/ Unboxing

- Use conversions:

```csharp
int i = 42;
Console.WriteLine(i); // i will be boxed
Console.WriteLine(i.ToString()); // no boxing
```

- Use generic collections:

```csharp
ArrayList al = new ArrayList(10);
al.Add(1); // implicit boxing;
al.Add(2); // implicit boxing
foreach (int el in al) { // implicit unboxing
    Console.WriteLine(el);
}
List<int> l = new List<int>();
l.Add(1); // no boxing
l.Add(2); // no boxing
foreach (int el in l) { // no unboxing
    Console.WriteLine(el);
}
```

Some Objects Are Equal

- It is important to understand the equality contract
- Never override:
  - `static bool Equals(object lhs, object rhs)`
    - returns true when objects have same identity
    - Otherwise, returns false if one of them is null
    - Otherwise, delegates to equals-method of left argument
  - `static bool ReferenceEquals(object a, object b)`
    - true iff objects have same identity
    - returns false when a and b are the same value object (boxing!)
- Overriding the two other methods is recommended:
  - `virtual bool Equals(object o)`
  - `public static bool operator==(MyType lhs, MyType rhs)`

(c) 2005, Michael Stal
## Equals

- **virtual bool Equals()** should be symmetric, reflective, transitive

```csharp
class MyClass {
    double d;
    public MyClass(double init) { d = init; }
    override public bool Equals(object rhs) {
        // check for identity:
        if (ReferenceEquals(this, rhs)) return true;
        // check null - this can't be null:
        if (rhs == null) return false;
        // check for same types:
        if (this.GetType() != rhs.GetType()) return false;
        // check for base class quality if not derived
        // from Object or ValueType directly
        // if (typeof(base).Equals(rhs)) return false;
        MyClass right = rhs as MyClass;
        // do same for fields:
        if (this.d.Equals(right.d)) return true;
        return false;
    }
}
```

## Operator==

- **Operator==** should be overridden for value types
- Otherwise, operator implementation is generated that relies on reflection

```csharp
struct MyStruct {
    // further details omitted
    public int i; // don't do this normally
    static public bool operator==(MyStruct lhs, MyStruct rhs) {
        return lhs.i == rhs.i;
    }
    static public bool operator !=(MyStruct lhs, MyStruct rhs) {
        return lhs.i != rhs.i;
    }
}
```

(c) 2005, Michael Stal
**GetHashCode**

- Implement `GetHashCode` whenever overriding Equals
- Otherwise, Hashtables/Dictionaries won't work as expected:

  ```csharp
  MyClass x = new MyClass(1.0);
  MyClass y = new MyClass(1.0);
  Console.WriteLine(x.Equals(y)); // => true
  Hashtable h = new Hashtable();
  h.Add(x, 42);
  Console.WriteLine(h.Contains(y)); // => false
  ```

- In the sample class include:

  ```csharp
  public override int GetHashCode() {
      return (int)d;
  }
  ```

**Rules for Hash Code Generation**

- Must be instance invariant
- If `a.Equals(b)` is true, then both objects should return the same hash code
- The generated hash codes should be evenly distributed
Always Override `ToString()`

- Overriding `ToString()`
  - Makes your class more pleasant to use
  - Helps to return all interesting information
  - Should be accompanied by documentation of `ToString()`
  - Example:

```csharp
class Person {
    int age;
    string name;
    public Person(string n, int a) { age = a; name = n; }
    public override string ToString() {
        return "Person " + name + " " + age;
    }
}
```

- Return useful text instead of just `ApplicationName.ClassName`

Comparisons

- Two interfaces
  - `IComparable` used to define natural ordering of a type
  - `IComparer` implements additional ordering: not shown in this talk
  - `IComparable` contains only one method:
    - `public int CompareTo(object rhs)`
    - `o1.CompareTo(o2)` yields
      - 0, if both objects are equal with respect to ordering
      - -1, if `o1 < o2`
      - +1, if `o1 > o2`
Example: `IComparable`

```csharp
class OrderedPair<S, T> : IComparable
    where S : IComparable<S>
    where T : IComparable<T> {
private S s; private T t;
public OrderedPair(S sArg, T tArg) {
    s = sArg; t = tArg;
}
public int CompareTo(object o) {
    if (!(o is OrderedPair<S,T>))
        throw new ArgumentException("bad type");
    OrderedPair<S, T> tmp = o as OrderedPair<S, T>;
    if (this.s.CompareTo(tmp.s) == 0)
        return (this.t.CompareTo(tmp.t));
    else
        return (this.s.CompareTo(tmp.s));
}
}
```

Exception Handling 101

- Instead of introducing your own exception types first try using existing exception types
- Makes your API simpler to learn and to read
- Use most derived exception that matches your need
- Return exceptions according to abstraction level:

```csharp
static object get() {
    try {
        // access file system
    }
    catch (LowerLevelException lle) {
        throw new HigherLevelException(lle);
    }
}
```

(c) 2005, Michael Stal
Defensive Event Publishing

- Problem: If a subscriber throws an exception during event handling the publisher does not care
- However:
  - Event publishing is interrupted
  - Manually iterating over subscriber list is tedious

```csharp
public delegate void SomeDelegate(int num, string str);

public class MySource
{
    public event SomeDelegate SomeEvent;
    public void FireEvent(int num, string str)
    {
        if (SomeEvent != null)
            SomeEvent(num, str); // interrupted on exception
    }
}
```

(c) 2005, Michael Stal

---

Defensive Event Publishing - EventsHelper

- Solution: Introduce Events Helper

```csharp
public class EventsHelper
{
    public static void Fire(Delegate del, params object[] args)
    {
        if (del == null)
        {
            return;
        }
        Delegate[] delegates = del.GetInvocationList();
        foreach (Delegate sink in delegates)
        {
            try
            {
                sink.DynamicInvoke(args);
            }
            catch{}
        }
    }
}
```

(c) 2005, Michael Stal
Defensive Event Publishing - Code

Here is the re-factored solution:

```csharp
public delegate void SomeDelegate(int num, string str);

public class MySource
{
    public event SomeDelegate SomeEvent;
    public void FireEvent(int num, string str)
    {
        EventsHelper.Fire(SomeEvent, num, str);
    }
}
```

(c) 2005, Michael Stal

Reflection: Prefer Custom Attributes

Custom Attributes help where otherwise configuration files or marker interfaces are used

```csharp
interface IReadable { void print(); }
class MyDocument : IReadable {
    public void print() { }
}

[AttributeUsage(AttributeTargets.Class)]
class PrintableAttribute : Attribute { /* ... */ };  
[AttributeUsage(AttributeTargets.Method)]
class Print : Attribute { /* ... */ };  
[Printable]
class MyDocumentAttr {
    [Print]
    void prettyPrint() { }
}
```
Performance Matters: String Interning

- Consider the following code:

```csharp
String x = "42";
String y = "42";
Console.WriteLine(Object.Equals(x, y)); // returns true
Console.WriteLine(Object.ReferenceEquals(x, y)); // returns true
```

- Guess, why this happens.
- Answer: CLR internally uses a hash map which is filled by JIT compiler: Reuse of string literals! Note: interned strings will not be freed by GC.
- You can leverage it yourself:

```csharp
String x = "Micha "; x += "Stal";
String y = String.Intern(x);
```

(c) 2005, Michael Stal

---

Performance Matters: foreach loops

- Foreach loops are optimized in that they don’t check array bounds multiple times

```csharp
static void Main(string[] args)
{
    int [] list = new int[] { 1,2,3,4,5,6,7,8,9,10};
    // slow:
    for (int index = 0; index < list.Length; index++)
    {
        Console.WriteLine(list[index]);
    }
    // fast:
    foreach (int el in list)
    {
        Console.WriteLine(el);
    }
}
```

(c) 2005, Michael Stal
Design Pattern Example: Master Slave

- Problem: Supporting fault-tolerance and parallel computation.
- Idea: *Divide et Impera* - partition tasks into subtasks and let components compute subtasks in parallel.

<table>
<thead>
<tr>
<th>Master</th>
<th>Slave</th>
</tr>
</thead>
<tbody>
<tr>
<td>mySlaves</td>
<td>Delegate computation of subtasks to slaves</td>
</tr>
<tr>
<td>splitWork</td>
<td>subTask</td>
</tr>
<tr>
<td>callSlaves</td>
<td></td>
</tr>
<tr>
<td>combineResults</td>
<td></td>
</tr>
<tr>
<td>service</td>
<td></td>
</tr>
</tbody>
</table>

**Slaves**

- Slaves calculate sub arrays. They are supposed to be executed within threads:

```java
class Slave {
    private double m_result;
    private double[] m_dList;
    private int m_start;
    private int m_end;
    public Slave(double[] dList, int start, int end) {
        m_start = start; m_end = end; m_dList = dList;
    }
    public double Result { get { return m_result; }}
    public void DoIt() {
        m_result = 0.0;
        for (int i = m_start; i <= m_end; i++)
            m_result += m_dList[i];
    }
}
```

(c) 2005, Michael Stal
The master uses slaves to calculate sub-arrays:

```csharp
class Master {
    public double CalculateSum(double[] dList, int start, int end) {
        if (start > end) throw new ArgumentException();
        if (start == end) return dList[start];
        int mid = (end - start) / 2;
        Slave s1 = new Slave(dList, start, mid);
        Slave s2 = new Slave(dList, mid + 1, end);
        Thread t1 = new Thread(new ThreadStart(s1.DoIt));
        Thread t2 = new Thread(new ThreadStart(s2.DoIt));
        t1.Start(); // start first slave
        t2.Start(); // start second slave
        t1.Join();  // wait for first slave
        t2.Join();  // wait for second slave
        return s1.Result + s2.Result; // combine results
    }
}
```

Putting it together

The manager class illustrates the configuration of the participants at runtime:

```csharp
class Manager {
    static void Main(string[] args) {
        double[] d = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
        Console.WriteLine(new Master().CalculateSum(d, 0, 9));
    }
}
```

(c) 2005, Michael Stal
Singleton

Intent
- ensure a class only ever has one instance, and provide a global point of access to it

Applicability
- when there must be exactly one instance of a class
- when sole instance should be extensible by subclassing

Structure

Consequences
- reduced name space pollution

Implementation
- C#: declare constructor as protected to guard against multiple singleton instances

Singleton

method()
method()
static instance()
static instance

Singleton - General Approach

First variant of Singleton using static initialization:

class MyClass1 /* statically initialized */ {
    private static MyClass1 m_Instance = new MyClass1();
    private MyClass1 () {}

    public static MyClass1 Instance
    {
        get { return m_Instance; }
    }
}

(c) 2005, Michael Stal
Static Singleton in .NET

■ Now, the solution is refined:

```csharp
sealed class MyClass1b /* statically initialized */ {
    public static readonly MyClass1b m_Instance = new MyClass1b();

    private MyClass1b () {}
}
```

Singleton with dynamic Initialization

■ Object is created on demand:

```csharp
class MyClass2 /* dynamically initialized on demand */ {
    private static MyClass2 m_Instance;
    private MyClass2() {}
    public static MyClass2 Instance
    {
        get {
            if (null == m_Instance) {
                m_Instance = new MyClass2();
            }
            return m_Instance;
        }
    }
}
```
**Singleton - Threadsafe**

- Multithreading - naive solution:

```csharp
class MyClass3a /* dynamically initialized on demand */ {
    private static MyClass3a m_Instance;
    private MyClass3a() {}
    public static MyClass3a Instance {
        get { lock(typeof(MyClass3a)) {
            if (null == m_Instance) {
                m_Instance = new MyClass3a();
            }
            return m_Instance;
        }
    }
}
```

(c) 2005, Michael Stal

---

- Efficient solution:

```csharp
class MyClass3b /* double checked locking */ {
    private static MyClass3b m_Instance;
    private MyClass3b() {}
    public static MyClass3 Instance {
        get {
            if (null == m_Instance) {
                lock(typeof(MyClass3b)) {
                    if (null == m_Instance)
                        m_Instance = new MyClass3b();
                }
            }
            return m_Instance;
        }
    }
}
```

(c) 2005, Michael Stal
Observer

Intent
Define a dependency between objects so that when one object changes state then all its dependents are notified

Applicability
- When a change to one object requires changing others
- Decouple notifier from other objects

Structure

```java
while (!e.hasMoreElements())
    Observer o = (Observer) e.nextElement();
    o.update(this);

state = newState;
notify();
```

Observer (cont’d)

Consequences
- abstract coupling of subject and observer
- object is responsible only for its own state Þ reusability
- unexpected updates, update overhead

Implementation
- push/pull model for notifications
- change interests

Known Uses
- Smalltalk MVC
- MFC
- MacApp, ET++
Observer in C#

Subject that emits events:

```csharp
public class Subject {
    public delegate void Notify();
    public event Notify OnNotify;
    public void DoSomething() {
        // now create an event
        if (null != OnNotify) {
            Console.WriteLine("Subject fires event");
            OnNotify();
        }
    }
}
```

Observer in C# (cont’d)

Observer that registers with Subject:

```csharp
class ObserverDemo {
    class Observer {
        public Observer(Subject s) {
            s.OnNotify += new Subject.Notify(TellMe);
        }
        public void TellMe() {}
    }
    static void Main(string[] args) {
        Subject s = new Subject();
        Observer o1 = new Observer(s);
        Observer o2 = new Observer(s); ...
    }
}
```

(c) 2005, Michael Stal
Summary

- C# idioms and styles useful to achieve C# mastership
- Idioms are patterns that leverage intrinsic knowledge of C# and .NET specialties
- This talk could just scratch the surface
- There are lots of more issues to discuss such as multithreading, resource management issues, distribution
- Unfortunately, wisdom is spread across many books
- Developer community should spend much more efforts on this
- Future evolution of C# (e.g., lambdas) will lead to new idioms and styles

Books

- Bill Wagner, Effective C#, Addison-Wesley, 2005
- Joshua Bloch, Effective Java, Addison-Wesley, 2001
Wir sehen uns wieder...

Juni 2006

November 2006

Februar 2007