Präsentiert von

Framework Design Guidelines

Wiederverwendbare Frameworks in C#

Auch Klassenbibliotheken haben eine Benutzerschnittstelle!

```
namespace itConsulting.stuttgart.customDevTeam
{
    public class TomsUtility
    {
        public TomsUtility(object pConfigurationSetting1,
            object pConfigurationSetting2,
            bool bPerformInitialization)
        {
            […]
        }

        public DbConnection get_Connection()
        {
            […]
        }

        public bool statement_Executor(string szStatement, int nMode)
        {
            […]
        }

        public void terminate_Verbindung(bool interrupt)
        {
            […]
        }
    }
}
```

Unsere heutigen Ziele

- Wie entwickelt man Komponenten, in denen andere Entwickler sich zu Hause fühlen?
- Wie sieht modernes Frameworkdesign aus?
- Welche Standards und Werkzeuge gibt es zur Unterstützung?

Erstens:
Make-or-Buy Entscheidungen bewusst treffen

Fragestellungen

- Ist es meine Aufgabe, ein wiederverwendbares Framework zu erstellen?
- Welche Abhängigkeiten erzeuge ich durch die Make-or-buy Entscheidung?
- Gibt es diese Komponente bereits am Markt in einer für mich passenden Qualität?
- Bestehende Komponenten passen nicht 100%ig – ist es Zeit, die Anforderungen zu überdenken?
- U.v.m.

Tipps zur Komponentenauswahl

- Fertige Komponenten sind oft eine gute Wahl, wenn es sich nicht um eine differenzierenden Kernbereich handelt
  - Probleme, die viele Entwickler haben, sind in der Regel bereits gelöst
- Plattformen gegenüber Produkten vorziehen
- Marktposition des Anbieters berücksichtigen
- Aufwand zur Auswahl minimieren
  - KO Kriterien formulieren
  - Vergleichen statt endlosem Pflichtenheft

Zweitens:
Gute Namensgebung

Generelle Namensregeln

- Diskussionen darüber im Team? Warum nicht einfach die Regeln von Microsoft übernehmen?
- `PascalCasing` für alle Identifier mit Ausnahme von Parameternamen
  - Ausnahme: Zweistellige, gängige Abkürzungen (z.B. `IOStream`, aber `HtmlTag` statt `HTMLTag`)
- `camelCasing` für Parameternamen
  - Ausnahme: Zweistellige, gängige Abkürzungen (z.B. `ioStream`)
- Fields?
  - Nicht relevant, da nie `public` oder `protected` ;-)
- Dont‘s
  - Underscores
  - Hungarian notation (d.h. kein Präfix)
  - Keywords als Identifier
  - Abkürzungen (z.B. `GetWin`; sollte `GetWindow` heißen)

Namen für Assemblies und DLLs

- Keine Multifileassemblies
- Oft empfehlenswert, den Namespacenamen zu folgen
- Namensschema
  - `<Company>.<Component>.dll`
  - `<Produkt>.<Technology>.dll`
  - `<Project>.<Layer>.dll`
- Beispiele
  - `System.Data.dll`
  - `Microsoft.ServiceBus.dll`
  - `TimeCockpit.Data.dll`
  - **`Transporters.TubeNetwork`**

Namen für Namespaces

- Eindeutige Namen verwenden (z.B. Präfix mit Firmen- oder Projektname)
- Namensschema
  - `<Company>.<Product|Technology>[.<Feature>][.<Subnamespace>]`
- Versionsabhängigkeiten soweit möglich vermeiden (speziell bei Produktnamen)
- Organisatorische Strukturen beeinflussen Namespaces nicht
- Typen nicht gleich wie Namespaces benennen
- Nicht zu viele Namespaces
- Typische Szenarien von seltenen Szenarien durch Namespaces trennen (z.B. `System.Mail` und `System.Mail.Advanced`)
- `PascalCasingWithDotsRecommended`

Klassen-, Struktur- und Interfacenamen

- `PascalCasingRecommended`
- Keine üblichen oder bekannten Typnamen verwenden (z.B. keine eigene Klasse `File` anlegen)
- Klassen- und Strukturnamen
  - Meist Hauptwörter (z.B. `Window`, `File`, `Connection`, `XmlWriter` etc.)
- Interfacenamen
  - Wenn sie eine Kategorie repräsentieren, Hauptwörter (z.B. `IList`, etc.)
  - Wenn sie eine Fähigkeit ausdrücken, Adjektiv (z.B. `IEnumerable`, etc.)
- Kein Präfix (z.B. „C…“)
  - „I“ für Interfaces ist historisch gewachsen und deshalb sehr bekannt

Membernamen

- `PascalCasingRecommended`
- Methoden
  - Verb als Name verwenden (z.B. `Print`, `Write`, `Trim`, etc.)
- Properties
  - Hauptwörter oder Adjektive (z.B. `Length`, `Name`, etc.)
  - Mehrzahl für Collection Properties verwenden
  - Aktiv statt passiv (z.B. `CanSeek` statt `IsSeekable`, `Contains` statt `IsContained`, etc.)
- Events
  - Verb als Name verwenden (z.B. `Dropped`, `Painting`, `Clicked`, etc.)
  - Gegenwart und Vergangenheit bewusst einsetzen (z.B. `Closing` und `Closed`, etc.)
  - Bei Eventhandler typisches Pattern verwenden (`EventHandler`-Postfix, `sender` und `e` als Parameter, `EventArgs` als Postfix für Klassen
- Fields
  - Keine `public` oder `protected` Fields
  - Kein Präfix

time cockpit

StyleCop
Code Analysis (FxCop) in Visual Studio 2010

DEMO

Tools für besseren Code

- Code Analysis (FxCop)
  - „reports information about the assemblies, such as possible design, localization, performance, and security improvements”
  - Automatisierte Qualitätssicherung
  - Nicht nur für Profis, auch optimal zum Lernen
- StyleCop
  - Sorgt für schönen Code (d.h. garantiert die Einhaltung von Code Style Rules)

Drittens:
Typische Einsatzszenarien legen das Design fest

Designhilfsmittel

- Client-Code First
  - Test Driven Development
- Aufwand minimieren
  - Word & PowerPoint
  - Pseudocode
  - Scriptsprachen
- *„Simple things should be simple and complex things should be possible“* (Alan Kay, Turing-Preisträger)
- Feedback der späteren Anwender einholen

This solution would replace `Install.stg` and `CodeBatchCollection`. There would be a Xaml file compiled into time cockpit's resources. There has to be a function to apply all update batches in the Xaml file similar to today's `InstallBatchManager.Install`. Additionally there will be functions to

1. find out all update batches that are missing on a certain database.
2. find out if the application can work with a certain database.

The following code snippets show how the API to install update batches would work:

Get update batch from XAML file stored in the assembly's resources.

```
UpdateBatch updateBatch = this.ReadUpdateBatchFromResources();
```

Note that DbClient.Create *will* ***not*** *automatically install update batches in the future any more.*

```
using (var dbClient = new DbClient.Create(...))
{
```

Find out which update batches are not installed in the database that dbClient *is pointing to.*

```
	IEnumerable<UpdateBatch> missingBatches =
		dbClient.GetMissingUpdateBatches(updateBatch);
	foreach (var missingBatch in missingBatches)
	{
		Console.WriteLine("{0} is missing", missingBatch.Guid);
	}

	Find out if app can run without executing any batches (i.e. if mandatory batches are missing).
	switch (dbClient.GetUpdateBatchStatus(updateBatch))
	{
		case BatchStatus.Complete:
			Console.WriteLine("Update batch is completely installed.");
			break;
		case BatchStatus.Acceptable:
			Console.WriteLine("Some non-mandatory batches are missing.");
			break;
		case BatchStatus.Incomplete:
			Console.WriteLine("Mandatory batches are missing.");
			break;
	}

	Install all missing update batches.
	dbClient.InstallMissingUpdateBatches(updateBatch);
}
```

Feature Files

On model level time cockpit will be extended by "features". A feature is a part of the logical data

Beispiele

- Typische Szenarien von seltenen Szenarien durch Namespaces trennen (z.B. `System.Mail` und `System.Mail.Advanced`)
- Kurze Methodensignaturen für einfache Szenarien
  - Method overloading
  - Null als Defaultwert akzeptieren
  - C# 4: Named and Optional Arguments (siehe MSDN)
- Konstruktoren anbieten
  - Defaultkonstruktor wenn möglich/sinnvoll
  - Konstruktor mit wichtigsten Properties
- Einfache Szenarien sollten das Erstellen von wenigen Typen brauchen
- Keine langen Initialisierungen vor typischen Szenarien notwendig machen
- Sprechende Exceptions

Viertens:
Möglichkeiten von Sprache und Plattform kennen und nutzen

Beispiel: Aktuelle Themen in C#

- LINQ
  - Auch ohne DB
  - Expression Trees
- Funktionale Sprachelemente in C#
  - Grundprinzipien der funktionalen Programmierung
  - Func<>, Action<>, Lambdas & Co
- Dynamische Sprachelemente in C#
  - DLR
  - dynamic
- Auf CLS Compliance achten falls Sprachenunabhängigkeit wichtig ist

Fünftens:
Objektorientierte Prinzipien richtig einsetzen

Klasse oder Struktur

- Strukturen in Betracht ziehen, wenn…
  - …der Typ klein ist UND
  - …Instanzen typischerweise kurzlebig sind UND
  - …meist eingebettet in andere Typen vorkommt.
- Strukturen nicht, wenn…
  - …der Typ logisch mehr als einen Wert repräsentiert ODER
  - …Größe einer Instanz >= 16 Bytes ODER
  - …Instanzen nicht immutable sind ODER
- Generell: Strukturen sind in C# sehr selten

Tipps für abstrakte und statische Klassen

- Abstrakte Klassen
  - `protected` oder `internal` Konstruktor
  - Zu jeder abstrakten Klasse mind. eine konkrete Implementierung
- Statische Klassen
  - Sollten die Ausnahme sein
  - Nicht als Misthaufen verwenden

Virtuelle Members und Sealing

- Virtuelle Members
  - `virtual` nur, wo Erweiterbarkeit explizit gewünscht ist
- C# Schlüsselwort `sealed`
- Kann angewandt werden auf
  - Klasse
  - Members
- Kein `sealed` bei Klassen außer es gibt gute Gründe
  - Grund könnten z.B. sicherheitsrelevante Eigenschaften in `protected` Members sein
- `sealed` macht oft Sinn bei überschriebenen Members

Basisklasse oder Interface?

- Generell Klassen Interfaces vorziehen
  - Aber ist das Interface nicht ein gutes Mittel zum Abbilden eines „Contracts“ zwischen Komponenten?
- Abstrakte Basisklasse statt Interface, um Contract und Implementation zu trennen
  - Interface ist nur Syntax, Klasse kann auch Verhalten abbilden
  - Beispiel: `DependencyObject` in WPF
- Tipp (Quelle: Jeffrey Richter)
  - Vererbung: „is a“
  - Implementierung eines Interface: „“can-do“

Sechstens:
Collections so einsetzen, wie sie gedacht sind

Regeln für Collections

- Keine „weakly typed“ Collections in öffentlichen APIs
  - Verwenden Sie stattdessen Generics
- `List<T>`, `Hashtable`, `Dictionary<T>` sollten in öffentlichen APIs nicht verwendet werden
  - Warum? Beispiel `List<T>.BinarySort`
- Collection Properties…
  - …dürfen nicht schreibbar sein
  - Read/Write Collection Properties: `Collection<T>`
  - Read-Only Collection Properties: `ReadOnlyCollection<T>` oder `IEnumerable<T>`

Regeln für Collections

- *„Require the weakest thing you need, return the stronges thing you have“*
  (A. Moore, Development Lead, Base Class Libarary of the CLR 2001-2007)
- `KeyCollection<TKey, TItem>` nützlich für Collections mit primary Keys
- Collection oder Array?
  - Generell eher Collection statt Array (Ausnahme sind Dinge wie `byte[]`)
  - `Collection`- bzw. `Dictionary`-Postfix bei eigenen Collections

```
public IEnumerable<DateTime> GetCalendar(
DateTime fromDate, DateTime toDate)
{
  var result = new List<DateTime>();
  for (; fromDate <= toDate;
    fromDate = fromDate.AddDays(1))
  {
    result.Add(fromDate);
  }

  return result;
}
```

Dafür gibt es `yield` Blocks!

Siebtens:
Nehmen Sie sich ein Beispiel an LINQ

time cockpit

Funktionsparameter
Extension Methods

DEMO

Extension Methods

- Sparsam damit umgehen!
  - Können das API-Design zerstören
- Verwenden, wenn Methode relevant für alle Instanzen eines Typs
- Können verwendet werden, um Abhängigkeiten zu entfernen
- Können verwendet werden, um Methoden zu Interfaces hinzuzufügen
  - Immer die Frage stellen: Wäre eine Basisklasse besser?

Funktionsparameter

- Wiederverwendbarkeit durch Funktionsparameter erhöhen
- `Func<>` und `Action<>` sind kurz aber nur bedingt sprechend
  - `delegates` sind oft klarer

Achtens:
Exceptions statt Rückgabewerten

Exceptions

- Exceptions statt error codes!
- `System.Environment.FailFast` in Situationen, bei denen es unsicher wäre, weiter auszuführen
- Exceptions nicht zur normalen Ablaufsteuerung verwenden
- Eigene Exceptionklassen erstellen, wenn auf den Exceptiontyp auf besondere Weise reagiert werden soll
- `finally` für Cleanup, nicht `catch`!
- Standard Exceptiontypen richtig verwenden
- Möglicherweise `Try…` Pattern verwenden (z.B. `DateTime.TryParse`)

Neuntens:
Unit Tests

Zehntens:
Hirn einschalten und neugierig bleiben

time cockpit

Zeit, die bleibt

Rainer Stropek
http://www.timecockpit.com

rainer@timecockpit.com