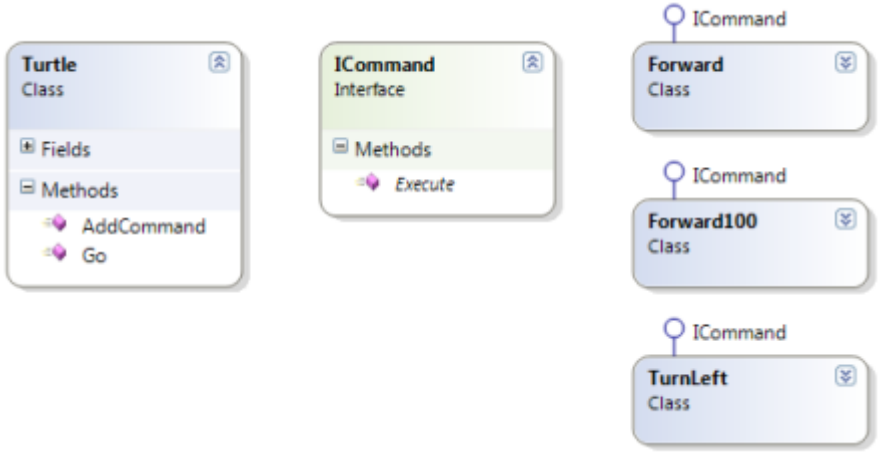
Pattern of the Week Collection

Pattern of the Week #7: Command Pattern

Diese Woche steht unter dem Stern des Command Pattern. Das Command Pattern ist ebenso wie das Adapter Pattern eines der häufiger verwendeten Pattern.

Klassische Implementierung

Das naheliegendste Beispiel für ein Command Pattern ist eine Schildkröte (Turtle), die über eine Methode (AddCommand) Kommandos erhält und diese anschließend ausführt (Go).




```
public class Turtle
{
private readonly IList<ICommand> _moveCommands = new List<ICommand>();

public void AddCommand(ICommand command)
{
this._moveCommands.Add(command);
}

public void Go()
{
foreach (var command in _moveCommands)
command.Execute();
}
}
```

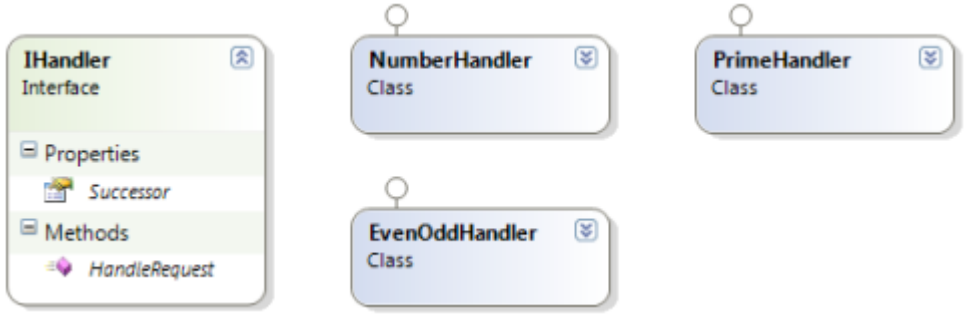
Die Klasse Turtle kennt nur ICommand. Die konkrete Implementierung und besonders die Initialisierung bleibt der Klasse Turtle verborgen.

```
var turtle = new Turtle();

ICommand cmd1 = new TurnLeft(90);
ICommand cmd2 = new Forward100();

turtle.AddCommand(cmd1);
turtle.AddCommand(cmd2);

turtle.Go();
```

Die Execute() Methoden der Commands werden nacheinander ausgeführt und sind sogar mehrfach ausführbar. Damit führt die Klasse Turtle die Commands wie eine Art Makro aus. Das Command Pattern kann also auch zum Scripten verwendet werden.

Undo Commands

Eine ebenso klassische Erweiterung ist ein Undo-Mechanismus. Dazu erhält die Schnittstelle ICommand eine weitere Undo() Methode. So wie der Script-Ablauf zum Go kann ein Undo die komplementären Funktionen umgekehrter Reihenfolge ausführen.




```
public class UndoTurtle
{
private readonly IList<IUndoCommand> _moveCommands
= new List<IUndoCommand>();

public void AddCommand(IUndoCommand command)
{
this._moveCommands.Add(command);
}

public void Go()
{
foreach (var command in _moveCommands)
command.Execute();
}

public void Rewind()
{
foreach (var command in _moveCommands.Reverse())
command.Undo();
}
}
```

Command Pattern & WPF 4

Mit der ersten Version von WPF wurde als essentieller Bestandteil die erweiterte komplexe Datenbindung eingeführt. Damit können Daten flexibler als mit Windows Forms an die GUI gebunden werden. Das MVVM Pattern war damit der Standard für die Datenpräsentation.
Mit WPF 4 wurde die Datenbindung um die Command-Bindung erweitert. Damit können nicht nur Daten sondern auch Methoden gebunden werden. Ein Modell enthält damit auch Logik, die gebunden werden kann.

```
public class MainUserControlModel
{
// property for binding
public string ButtonText { get; set; }

// command for bindung
public ICommand MessageBox { get; set; }
}
```

Es fällt wieder –oh Überraschung- die Schnittstelle ICommand auf. Die Schnittstelle ICommand aus dem WPF Namespace ist etwas aufwändiger um zusätzlich kontextbezogen Steuerelemente aktivieren und deaktivieren zu können.

```
public class MessageBoxCommand : ICommand
{
#region Implementation of ICommand

public void Execute(object parameter)
{
MessageBox.Show("Hello World");
}

public bool CanExecute(object parameter)
{
return true;
}

public event EventHandler CanExecuteChanged = (sender, e) => { };

#endregion
}
```

Anschließend kann das Model an die View mittels XAML gebunden werden.

```
<Button
Command="{Binding Path=MessageBox}"
Content="{Binding Path=ButtonText}"
Margin="16,11,0,0" Name="button1" [...] />
```

Command Pattern trifft CD-Sammlung

Hintergründe

Bereits in einem vorherigen Blog-Artikel hatte ich erwähnt, dass mit Metadaten von MP3 Dateien wichtig sind. Nach einem Daten-Crash-Super-GAU hatte ich mich entschieden, alle meine CDs noch einmal mit iTunes zu digitalisieren und zu 100% zu "taggen". Da mit iTunes nur ein Teil der Metadaten gefüllt werden, müssen die anderen Daten irgendwie anders her. Die Seite Musik-Sammler hat eine sehr ausführliche Liste an CDs und deren zugehörige EANs (Der CD-Code auf der Rückseite der CD). Mit den EANs lassen sich die entsprechenden CDs sehr schnell finden.

Divide et impera (Teile und Herrsche)

Kurz gesagt gibt es viele Schritte, die neben dem reinen Digitalisieren mit iTunes noch erledigt werden müssen. Dazu zählen:

- Erstellen einer Liste aller EANs meiner CDs
- Lesen von erweiterten Metadaten aus dem Internet (Musik-Sammler.de)
- Lesen des Covers aus dem Internet
- Linken des MP3 Verzeichnisses mit dem Album/EAN
- Verifizieren von beispielsweise
  - Sind alle Songs digitalisiert
  - Hat jedes Album ein MP3-Verzeichnis

Alle diese Punkte lassen sich bei fast 400 CDs nicht mehr händisch pflegen. * Also muss ein Skript (PowerShell) oder ein Programm her. Ich habe mich für eine Konsolenapplikation entschieden. Um einzelne Aufgaben getrennt ausführen zu können, habe ich ein Konzept wie bsp. stsadm gewählt, bei dem der erste Parameter die Aktion darstellt. Aufrufbeispiel:

```
Analyser.exe <Aktion> <Parameter für die Aktion>
Analyser.exe EanCdInfoReader c:\eans\ c:\cdinfo
Analyser.exe DownloadCdCover c:\eans
```

Command Pattern

Das -wir erinnern uns- hilft hierbei. Alle unterschiedlichen/möglichen Aktionen werden in separate Klassen ausgelagert und über die Schnittstelle ICommand erreichbar gemacht.

```
public interface ICommand
{
// args are the command line parameter (all of them)
void Execute(string[] args);
}
```

Die Command-Line Parameter werden 1:1 and die Commands durchgereicht (natürlich könnte auch der erste Parameter gelöscht werden). Es gibt also in der Assembly für alle Aktionen eine separate Klasse mit der Implementierung der Schnittstelle ICommand.

Find and Execute

Das unspannendste aber auch effizienteste ist das Instanziieren und Ausführen der Aktionen. Die Main-Methode ermittelt alle Implementierungen der Command-Schnittstelle, instanziiert die Implementierung mit dem korrespondierendem Namen und ruft die Execute Methode auf.

```
static void Main(string[] args)
{
// show actions of no parameter is given
if (args.Length == 0)
```

```
{
Console.WriteLine("you need at least one operation. available actions:");
Assembly.GetEntryAssembly().GetTypes()
.Where(t => t.GetInterface(typeof(ICommand).Name) != null && t.IsClass)
.ToList()
.ForEach(t => Console.WriteLine(" " + t.Name));
return;
}

// find class for action and create instance
var cmdType = Assembly.GetEntryAssembly().GetTypes().FirstOrDefault(f =>
f.Name.EndsWith(args[0]));
if (cmdType == null)
{
Console.WriteLine("cannot find type for given action. check for typos and upper/lower
case");
return;
}

var cmd = (ICommand)Activator.CreateInstance(cmdType);
cmd.Execute(args); // pass cmdline parameter to execute methode
}
```

Das bedeutet also, dass der Aufruf von Analyser.exe DownloadCdCover <...> eine Klasse DownloadCdCover mit der Implementierung der Schnittstelle ICommand erwartet. Ist die Klasse vorhanden, dann wird die Aktion ausgeführt. Neue Aktionen müssen sich nicht registrieren, sondern werden automatisch erkannt. Erweitern könnte man den Code noch soweit, dass er die Klassen aus allen Assemblys in dem Verzeichnis verwendet.

Pattern of the Week #15: Design Principles II

Es ist mal wieder an der Zeit, ein paar Design Principles vorzustellen. Bei Design Principles handelt es sich nicht Muster für ein Problem sondern um allgemeine Richtlinien. Diese Richtlinien beschreiben Grundgedanken und Regeln für ein gutes Design.

YAGNI

YAGNI ist ein Akronym für You Ain't Gonna Need It und bedeutet übersetzt so viel wie Du benötigst das nicht. Damit ist gemeint, dass nur diejenigen Features implementiert werden sollen, die gewünscht sind und damit auch hinreichend spezifiziert wurden. Oft werden Features implementiert, nur weil man sich sagt "Das kann ich bestimmt irgendwann gebrauchen".

YAGNI Code hat viele Nachteile

1. YAGNI Code für nie benötigte Features. Es wird Zeit in etwas investiert, das keinen Nutzen hat.
2. YAGNI Code muss getestet werden. Egal wie klein und trivial der Code ist, muss dafür mindestens ein Test geschrieben werden.
3. Wird ein Features auf "gut Glück" implementiert kann es sein, dass Code und damit die Tests neu implementiert werden müssen, da die Spezifikation anders ist, als man zunächst vermutet hat.

Hier ein kleines Beispiel. Der Kunde hat die folgende Anforderung (der Übersichtlichkeit halber sehr zusammengefasst).

- Implementieren eines LIFO Speichers mit Push und Pop
- LIFO Speicher ist generisch
- Export der Einträge als XML

Ein Ausweg hierfür ist, dass der Ansatz für die Entwicklung geändert wird. Hier helfen Methoden wie Test Driven Development (TDD) und Specification Driven Development. Beide Ansätze verfolgen den Weg, zunächst die Anforderungen in Form von Tests zu definieren. Diese Tests definieren die Erwartungen an das Framework und diese –aber auch nur diese- werden auch implementiert.

```
[Test()]
public void PopPushTest()
{
MyStack<int> sut = new MyStack<int>();

var expected = 23;
sut.Push(expected);
var actual = sut.Pop();

Assert.AreEqual(expected, actual, "stack returned wrong value");
}

[Test()]
public void ToXmlTest()
{
MyStack<int> sut = new MyStack<int>();

string xml = sut.ToXml();

Assert.AreEqual(string.Empty, xml);
}
```

Ein weiterer Aspekt von YAGNI ist es, nicht mit der sprichwörtlichen Kanone auf Spatzen zu schießen. Aus der Anforderung "den Stack in XML zu exportieren", können beliebig viele Featuers abgeleitet werden.

- Methode "Export" zum Exportieren
- Methode "Import" um den Export importieren zu können
- Serializer Factory um unterschiedliche Serialisierungsverfahrung unterstützen zu können
- Plugin Mechanismus für zusätzliche Serialisierungsverfahren
- Interceptor Pattern um in die Serialisierung eingreifen zu können

Aus der Anforderung, einen Stack in ein XML zu exportieren, kann man sehr sehr viele Features ableiten. Aber man muss sich immer die Frage stellen, was war der Auftrag. In der Spezifikation (who, what, why) steht vielleicht "Der Entwickler möchte den Stack nach XML exportieren um die Daten in Excel darstellen zu können". Welches der Features aus der obigen Liste werden benötigt?

Tell, don't Ask

Wer Kinder hat wird das Problem kennen: Bei langen Autofahrten wird ständig gefragt "Sind wir bald da?". Die Antwort ist dann so etwas wie "noch 150 Kilometer". Steht man im Stau sind die Fragen noch lästiger, da die Entfernung sich nicht ändert. Wie wäre es denn alternativ mit einen Status alle 20 Kilometer? Das Kind muss (müsste) nicht mehr fragen und bekommt regelmäßig einen Status, aber auch nur dann, wenn es sinnvoll ist. Ebenso verhält es sich in Software. Es hat keinen Nutzen, ein Objekt ständig nach seinem Status zu fragen. Man bekommt auf diesem Weg irgendwann mit, das sich etwas geändert hat, aber nicht wann. Die Fragenden bekommen aber nicht mit, wenn sich wirklich etwas ändert.
Der Ausweg für dieses Problem ist das Observer Pattern oder in der .NET Sprache Events . An das Event können sich alle fragenden Objekte (Listener) registrieren und wieder deregistrieren. Das Quellobjekt benachrichtig alle fragenden Objekte –aufrufen der übergebenen Methode. Das beste Beispiel für eine solche Implementierung ist die Schnittstelle aud dem .NET Framework INotifyPropertyChanged.

```
public class AClassWithSomeState : INotifyPropertyChanged
{
// INotifyPropertyChanged implementation
public event PropertyChangedEventHandler PropertyChanged = (sender, e) => { };

private string _name;

public string Name
{
get { return _name; }
set
{
_name = value;
PropertyChanged.Invoke(this, new PropertyChangedEventArgs("Name"));
}
}
}
```

Damit kann man sich von Statusänderungen benachrichtigen lassen, wenn sie wirklich auftreten.

```
var acwss = new AClassWithSomeState();

for (int i = 0; i < 10; i++)
{
int i1 = i; // remember: "i1" is an closure, "i" NOT
acwss.PropertyChanged +=
(sender, e) => Console.WriteLine("acwss has changed. event listerner #" + i1 +
((AClassWithSomeState)sender).Name);
}
```

```
acwss.Name = "New Name";
```

Ein weiterer Schritt, um die Aufrufe an die Klasse zu reduzieren ist es, dem Listener bei dem Methodenaufruf den Status mitzugeben. Damit wird zusätzlich die Kopplung zwischen Klasse und Listener entkoppelt. Beide hängen nur noch von den EventArgs als Schnittstelle ab.

```
// new event definition
public event EventHandler<AClassWithStateEvtStateEventArgs>
PropertyChanged = (sender, e) => { };

// event args definition
public class AClassWithStateEvtStateEventArgs : EventArgs
{
private readonly string _property;
private readonly object _name;

public AClassWithStateEvtStateEventArgs(string property, object name)
{
_property = property;
_name = name;
}

public object Name
{
get { return _name; }
}

public string Property
{
get { return _property; }
}
}
```

Mockups

Es wird gepredigt, die Kopplung der Komponenten möglichst gering zu halten und Schnittstellen zu definieren. Das erhöht die Wartbarkeit und reduziert die Fehleranfälligkeit. Ein weiterer wichtiger Vorteil ist allerdings die Testbarkeit.
Das größte Problem beim Testen ist die Kopplung zwischen den Komponenten. Selbst wenn eine Komponente nur von einer Schnittstelle abhängt, hängt sie dennoch davon ab. Befriedigt man diese Abhängigkeit indem man eine andere Komponente des "produktiven" Code übergibt, kann man in Komponente A auf einen Fehler treten, die durch Komponente B entsteht. Der Test ist damit nicht aussagekräftig und falsch.
Mockups und Schnittstellen lösen dieses Problem und bieten einen weiteren Benefit: Aufruftests. Beginnt man nun eine Klasse zu mocken, stellt sich dennoch die Frage, ob eine Methode des Mocks wirklich aufgerufen wurde oder nicht. Hat man eine Schnittstelle IEmail und eine Klasse SmtpSender wäre es gut, wenn die Property Body auch mindestens einmal aufgerufen wurde.
Mocking Frameworks gibt es wie Sand am Meer. Da ich für meine Tests normalerweise NUnit einsetze, benutze ich für mein Mocking gerne die Klasse DynamicMock. Letztlich ist es egal, welches Mocking Framework man nutzt, da die Essenz dieser Frameworks gleich ist: Objekte zu erstellen, die eine Schnittstelle implementieren und als Atrappen für die "produktiven" Objekte stehen.

```
// the interface definition
public interface IWantsToBeMocked
{
string Name { get; set; }
```

```
string Address { get; set; }
}

// the interface "productive" implementation
public class WantsToBeMocked : IWantsToBeMocked
{
public string Name { get; set; }
public string Address { get; set; }
}

// the class which uses the interface
public class SystemUnderTest
{
public void WriteNameWithoutAddress(IWantsToBeMocked instance)
{
Console.WriteLine("Name: " + instance.Name);
}
}
```

Der Test erstell ein Mock-Objekt (mittels DynamicMock) und reicht dieses an das System unter Test (sut) weiter. Damit entstehen Implementierungen für eine Schnittstelle, die nur das nowendige implementiert und die Rückgabewerte beim Mocking fest definiert werden. Ein Bonuspunkt ist, dass die Erwartung an die Methodenaufrufe ebenfalls definiert werden.

```
// create mock object
var aMockFactory = new DynamicMock(typeof(IWantsToBeMocked));
aMockFactory.ExpectAndReturn("get_Name", "Thomas");
aMockFactory.SetReturnValue("get_Address", "Bramsstr. 8");

IWantsToBeMocked mock = (IWantsToBeMocked)aMockFactory.MockInstance;

// create sud and test
var sut = new SystemUnderTest();
sut.WriteNameWithoutAddress(mock);

// check if mock was called correctly
aMockFactory.Verify();
```

Die Methodennamen get_* liegen daran, dass hier Getter von Properties aufgerufen werden, die intern diese Namenskonvention haben. Zwar hängt das SUT weiterhin von der Schnittstelle ab, es ist aber durch das Mocking auszuschließen, dass der Test fehlschlägt, weil die abhängigen Klasse fehlerhaft ist. Man sieht, dass Mocking notwendig ist, um korrekte Tests zu definieren. Tests, in denen nur eine Methode des SUT getestet wird und nichts anderes.