Inhalt

[1 Einführung 3](#_Toc395010749)

[1.1 Warum Clean Code? 3](#_Toc395010750)

[1.2 Über dieses Handbuch 3](#_Toc395010751)

[1.3 Clean Code Wertesystem, ein Auszug 3](#_Toc395010752)

[1.3.1 Standardisierte Namensformatierung für Variablen und Funktionen 3](#_Toc395010753)

[1.3.2 Sprechende Namen für Variablen und Funktionen 3](#_Toc395010754)

[1.3.3 Eliminierung von Code Redundanzen 3](#_Toc395010755)

[1.3.4 Eine Funktion sollte nie länger als 2 Bildschirmseiten sein 3](#_Toc395010756)

[1.3.5 Code wird dort platziert wo ein fremder Entwickler diesen auch erwartet 4](#_Toc395010757)

[1.3.6 Das KISS Prinzip 4](#_Toc395010758)

[2 Code „No Go’s“ (Kategorie „rot“) 5](#_Toc395010759)

[2.1 „Functionality Code“ Redundanzen 5](#_Toc395010760)

[2.2 Noch mehr „Functionality Code“ Redundanzen.. 7](#_Toc395010761)

[2.3 „Utility Code“ Redundanzen 8](#_Toc395010762)

[2.4 „Füllstoff“ Code-Kommentare, vermeidbar bei redundanten Funktionen 9](#_Toc395010763)

[2.5 Konfigurierbare Einstellungen 12](#_Toc395010764)

[3 Optische Optimierungen (Kategorie “gelb“) 13](#_Toc395010765)

[3.1 Code Block Verschachtelungen reduzieren 13](#_Toc395010766)

[3.2 Doppelt negierte Ausdrücke vermeiden 14](#_Toc395010767)

[3.3 „if“ Statements lesbarer machen 14](#_Toc395010768)

[3.4 Wiederholende Textbezeichnungen zusammenfassen 15](#_Toc395010769)

[3.5 Bedeutungsfreie Ausdrücke verständlich umformulieren 15](#_Toc395010770)

[3.6 Lange teilredundante Bezeichner 16](#_Toc395010771)

[4 Die CKG Modul Architektur 17](#_Toc395010772)

[4.1 4-Tier Architektur 17](#_Toc395010773)

[4.2 CKG Module, Anordnung innerhalb der 4-Tier Architektur 18](#_Toc395010774)

[4.3 MVC Abbildung auf die CKG 4-Tier Architektur 19](#_Toc395010775)

[4.4 Warum ViewModels? 19](#_Toc395010776)

[4.5 MVC-VM Pattern, Anordnung innerhalb der 4-Tier Architektur 19](#_Toc395010777)

[4.6 Clean Code / MVC-VM / 4-Tier 20](#_Toc395010778)

[4.7 Beispiele für Schichtenkorrekturen ( „Falsch / Richtig“) 20](#_Toc395010779)

[4.7.1 Beispiel Schichtenkorrektur „ViewModel <-> DAL Layer“ 21](#_Toc395010780)

[4.7.2 Beispiel Schichtenkorrektur „Controller <-> ViewModel“ 23](#_Toc395010781)

[4.7.3 Beispiele View Schicht, .NET Code Fehlplatzierungen 25](#_Toc395010782)

[5 Clean Code in der System Architektur 26](#_Toc395010783)

[5.1 SoC - “Separation of concerns” 26](#_Toc395010784)

[5.2 IoC - “Inversion of control” 28](#_Toc395010785)

# Einführung

## Warum Clean Code?

Der „Clean Code Standard“ definiert ein Wertesystem für professionelle Softwareentwickler und Entwicklungsabteilungen. Ziel ist es, Software zu entwickeln die deutlich wartbarer ist sowie für andere Entwickler leicht lesbarer sein wird.

Nach fertiger Entwicklung hört der Clean Code Prozess nicht auf, Ziel ist es hier nicht nur pro-aktiv Fehler im Code zu eliminieren sondern kontinuierlich Software Code umzuformatieren (zu refakturieren) um die Lesbarkeit, Übersicht und Wartbarkeit der Software zu erhöhen.  
Wichtig ist hierbei der pro-aktive Anteil des Clean Code Entwicklers, der aus seinem Wertebewusstsein heraus selbständig bestehende Softwarte aktiv + kontinuierlich pflegt.

## Über dieses Handbuch

Dieses Handbuch stellt einen Leitfaden für die Clean Code Entwicklung in der CKG-Entwicklungsabteilung bereit.  
Anhand umfangreicher „Falsch“ / „Richtig“ Beispiele werden dem Leser die Vorteile von Clean Code veranschaulicht.   
Einige Beispiele beschreiben Clean Code auf Design- / Architekturebene,   
andere Beispiele begeben sich auf die Detailebene in den Untiefen des Software Maschinenraums, des Codes auf Modulebene.

## Clean Code Wertesystem, ein Auszug

Hier ein Auszug aus den Regeln des Clean Code Wertesystems:

### Standardisierte Namensformatierung für Variablen und Funktionen

Beispiel:   
Member Variablen beginnen mit einem Unterstrich gefolgt von einem Kleinbuchstaben  
Member (Properties) beginnen mit einem Großbuchstaben.

### Sprechende Namen für Variablen und Funktionen

Beispiel:  
Eine Variable sollte nicht einfach nur „fzgbrzlsg“ heißen, sondern vielmehr „FahrzeugbriefFuerZulassung“.  
Code Kommentare sind überflüssig, wenn die Variablen Namen bereits leserlich + sprechend sind.  
Unterbegriffe innerhalb der Namen beginnen dabei stets mit einem Großbuchstaben:   
Z.B.: „**F**ahrzeugbrief**F**uer**Z**ulassung“

### Eliminierung von Code Redundanzen

Unter-Code der sich innerhalb einer Code Datei mindestens 1x wiederholt, sollte sofort in eine Funktion ausgelagert werden – es gibt hierbei keine Ausnahme oder Ausrede!  
Strikte Konsequenz + Akribie sind ein Merkmal eines Clean Code Developers!

### Eine Funktion sollte nie länger als 2 Bildschirmseiten sein

Längere Funktionen erschweren die Lesbarkeit und die Wartbarkeit immens.  
Es gibt Programme in denen sich Funktionen befinden, die sich über jeweils 10 und mehr Bildschirme strecken.  
Die Folge: Erweiterbarkeit, Testbarkeit und Wartbarkeit sind hier nahezu unmöglich geworden.

### Code wird dort platziert wo ein fremder Entwickler diesen auch erwartet

Beispiel:   
Codezeilen die XML-Inhalte bearbeiten haben in einer Softwarebibliothek die ausschließlich Dienste für Grafikbearbeitungen bereitstellt nichts zu suchen!  
Stattdessen sollten diese in einer separaten XML Bibliothek platziert werden.  
Über Schnittstellen nach dem SOA Standard (Service orientierte Architektur) können dann beide Modulbibliotheken einander benutzen, ohne die Implementierung des jeweils anderen Moduls kennen zu müssen.

### Das KISS Prinzip

Das KISS Prinzip („Keep it simple, stupid”) orientiert sich an dem Leitspruch   
„*Alles sollte so einfach wie möglich gemacht werden, aber nicht einfacher*".   
Für die Evolvierbarkeit des Codes ist zwingende Voraussetzung, dass der Code verständlich ist.   
Eine einfache, klare und leicht verständliche Lösung sollte daher immer bevorzugt werden. Wenn man seinen eigenen Code nach kurzer Zeit schon nicht mehr versteht, sollten die Alarmglocken klingen. Noch wichtiger aber ist, dass auch andere Entwickler den Code schnell verstehen können. Dabei helfen regelmäßige Reviews und Pair Programming. Sie dienen der Kontrolle, ob tatsächlich die einfachste Lösung verwendet wurde.

Gerade in technischen Details steckt die Versuchung, eine komplizierte Lösung anzustreben. Das Bekannte, naheliegende ist manchmal zu "langweilig" – und schon hat sich eine komplizierte Lösung eingeschlichen. Wenn die einfache Lösung auch funktioniert, sollte ihr Vorrang gewährt werden. Das gleiche gilt für Datenstrukturen. Wenn ein IEnumerable reicht, sollte keine ICollection oder sogar IList verwendet werden.

# Code „No Go’s“ (Kategorie „rot“)

## „Functionality Code“ Redundanzen

Folgender Auszug listet 2 nahezu identische Programmblöcke innerhalb 2 verschiedener Funktionen:

private Boolean **CheckGrid**(String rowUpdate)

{

[ … weiterer Code in der Function CheckGrid ]

if (m\_User.Groups[0].Authorizationright != 1)

{

var rowsPauschal = objKompletterf.tblEingabeListe.Select("ID=" + intID);

var xSteuer = rowsPauschal[0]["OhneSteuer"].ToString();

if (rowsPauschal[0]["Pauschalkunde"].ToString() == "X")

{

if (decGeb != decGebAmt)

{

var gebMat = "";

var rowsGebMat = objKompletterf.tblEingabeListe.Select("ID=" + intID + " AND id\_pos =" + intPosID);

if (rowsGebMat.Length == 1)

{

if (xSteuer == "X")

gebMat = rowsGebMat[0]["GebMatnr"].ToString();

else

gebMat = rowsGebMat[0]["GebMatnrSt"].ToString();

}

SDRelGeb = objKompletterf.GetSDRelevantsGeb(intID, intPosID, gebMat);

if (\_txtBox.Visible && **rowUpdate == "" &&** \_loeschkz != "X" && SDRelGeb != "X")//&& lblOkKZ.Visible

{

\_txtBox.BorderColor = System.Drawing.ColorTranslator.FromHtml("#bc2b2b");

lblError.Text = "Bei Pauschalkunden dürfen Gebühr und Gebühr Amt nicht unterschiedlich sein!";

bError = true;

}

}

}

}

[ … weiterer Code in der Function CheckGrid ]

}

[…]

private Boolean **CheckGridRow**(GridViewRow gvRow)

{  
  
 [ … weiterer Code in der Function **CheckGridRow** ]

if (m\_User.Groups[0].Authorizationright != 1)

{

var rowsPauschal = objKompletterf.tblEingabeListe.Select("ID=" + intID);

var xSteuer = rowsPauschal[0]["OhneSteuer"].ToString();

if (rowsPauschal[0]["Pauschalkunde"].ToString() == "X")

{

if (decGeb != decGebAmt)

{

var gebMat = "";

var rowsGebMat = objKompletterf.tblEingabeListe.Select("ID=" + intID + " AND id\_pos =" + intPosID);

if (rowsGebMat.Length == 1)

{

if (xSteuer == "X")

gebMat = rowsGebMat[0]["GebMatnr"].ToString();

else

gebMat = rowsGebMat[0]["GebMatnrSt"].ToString();

}

SDRelGeb = objKompletterf.GetSDRelevantsGeb(intID, intPosID, gebMat);

if (\_txtBox.Visible && \_loeschkz != "X" && SDRelGeb != "X")//&& lblOkKZ.Visible

{

\_txtBox.BorderColor = System.Drawing.ColorTranslator.FromHtml("#bc2b2b");

lblError.Text = "Bei Pauschalkunden dürfen Gebühr und Gebühr Amt nicht unterschiedlich sein!";

bError = true;

}

}

}

}  
 [ … weiterer Code in der Function CheckGridRow ]

}

Wir haben es hier mit mehreren fatalen Programmier-Fehlgriffen zu tun:

1. Bei Feststellung eines Bugs der in nur einer der offensichtlich duplizierten Code Blöcke auffiele, würde der Bug-Fix mit sehr großer Wahrscheinlichkeit nur im einen Code Block vorgenommen werden, da die fehlerhafte Redundanz im anderen Code Block gar nicht wahrgenommen würde.  
   Folge: Das Programm bliebe auch nach dem Bug-Fix weiterhin fehlerbehaftet!   
   Am Beispiel des Code Fragements **rowUpdate == "" &&** ist zu erkennen, dass ein Bug-Fix vermutlich im einen Code Block schon vorgenommen, im anderen aber wie erwartet übersehen wurde.
2. Dito bei Erweiterung der Funktionalität eines der Code-Blöcke:  
   Mit hoher Wahrscheinlichkeit bliebe die Erweiterung dem anderen Code Block fern…
3. Die Lesbarkeit des Programms wird durch unnötig duplizierte Code Blöcke und daraus resultierenden Code Aufblähungen entscheidend erschwert!

**Lösung:**Refakturierung durch Auslagerung der Code Redundanz in eine separate Funktion (**ProcessAuthorization**):

private Boolean **CheckGrid**(String rowUpdate)

{

[ … weiterer Code in der Function CheckGrid ]

**ProcessAuthorization(rowUpdate);**

[ … weiterer Code in der Function CheckGrid ]

}

private Boolean **CheckGridRow**(GridViewRow gvRow)

{

[ … weiterer Code in der Function CheckGridRow ]

**ProcessAuthorization(gvRow);**

[ … weiterer Code in der Function CheckGridRow ]

}

**private void ProcessAuthorization(DataRow row)**

**{**

**if (m\_User.Groups[0].Authorizationright != 1)**

**{**

**var rowsPauschal = objKompletterf.tblEingabeListe.Select("ID=" + \_intID);**

**var xSteuer = rowsPauschal[0]["OhneSteuer"].ToString();**

**if (rowsPauschal[0]["Pauschalkunde"].ToString() == "X")**

**{**

**if (\_decGeb != \_decGebAmt)**

**{**

**var gebMat = "";**

**var rowsGebMat = objKompletterf.tblEingabeListe.Select("ID=" + \_intID + " AND id\_pos =" + \_intPosID);**

**if (rowsGebMat.Length == 1)**

**{**

**if (xSteuer == "X")**

**gebMat = rowsGebMat[0]["GebMatnr"].ToString();**

**else**

**gebMat = rowsGebMat[0]["GebMatnrSt"].ToString();**

**}**

**\_sDRelGeb = objKompletterf.GetSDRelevantsGeb(\_intID, \_intPosID, gebMat);**

**if (\_txtBox.Visible && row == "" && \_loeschkz != "X" && \_sDRelGeb != "X")//&& lblOkKZ.Visible**

**{**

**\_txtBox.BorderColor = System.Drawing.ColorTranslator.FromHtml("#bc2b2b");**

**lblError.Text = "Bei Pauschalkunden dürfen Gebühr und Gebühr Amt nicht unterschiedlich sein!";**

**\_bError = true;**

**}**

**}**

**}**

**}**

**}**

## Noch mehr „Functionality Code“ Redundanzen..

Durch ein wenig Refakturierung reduziert sich auch dieser Code um sage und schreibe 2/3 !

|  |  |
| --- | --- |
| \_txtBox = (TextBox)gvRow.FindControl("txtPreis");  Decimal Preis = 0;  if (ZLDCommon.IsDecimal(\_txtBox.Text))  {  Decimal.TryParse(\_txtBox.Text, out Preis);  if (dRows.Length == 0)  {  tblData.Rows[i]["Preis"] = Preis;  }  else  {  dRows[0]["Preis"] = Preis;  }  }  else  {  if (dRows.Length == 0)  {  tblData.Rows[i]["Preis"] = 0;  }  else  {  dRows[0]["Preis"] = 0;  }  }  \_txtBox = (TextBox)gvRow.FindControl("txtGebPreis");  if (ZLDCommon.IsDecimal(\_txtBox.Text))  {  Decimal.TryParse(\_txtBox.Text, out Preis);  if (dRows.Length == 0)  {  tblData.Rows[i]["GebPreis"] = Preis;  }  else  {  dRows[0]["GebPreis"] = Preis;  }  }  else  {  if (dRows.Length == 0)  {  tblData.Rows[i]["GebPreis"] = 0;  }  else  {  dRows[0]["GebPreis"] = 0;  }  }  \_txtBox = (TextBox)gvRow.FindControl("txtGebAmt");  if (ZLDCommon.IsDecimal(\_txtBox.Text))  {  Decimal.TryParse(\_txtBox.Text, out Preis);  if (dRows.Length == 0)  {  tblData.Rows[i]["GebAmt"] = Preis;  }  else  {  dRows[0]["GebAmt"] = Preis;  }  }  else  {  if (dRows.Length == 0)  {  tblData.Rows[i]["GebAmt"] = 0;  }  else  {  dRows[0]["GebAmt"] = 0;  }  } | SaveTextBoxValueToDataRow("txtPreis", "Preis");  SaveTextBoxValueToDataRow("txtGebPreis", "GebPreis");  SaveTextBoxValueToDataRow("txtGebAmt", "GebAmt");  void SaveTextBoxValueToDataRow(string rowControlName,   string fieldName)  {  \_txtBox = (TextBox)\_gvRow.FindControl(rowControlName);  var iTextBoxValue = (decimal) 0.0;  if (ZLDCommon.IsDecimal(\_txtBox.Text))  Decimal.TryParse(\_txtBox.Text, out iTextBoxValue);  if (dRows.Length == 0)  \_tblData.Rows[\_index][fieldName] = iTextBoxValue;  else  dRows[0][\_fieldName] = iTextBoxValue;  } |

## „Utility Code“ Redundanzen

Bei genauerer Betrachtung des folgenden Code Blocks stellen wir fest, dass 5 mal im Grunde die gleiche Funktionalität programmiert wurde:   
Aus einer Text-Variablen soll jeweils 5 mal eine numerische Zahl geparst werden.

Int32 \_intID = -1;

if (ZLDCommon.IsNumeric(ID.Text))

{

Int32.TryParse(ID.Text, out \_intID);  
 if (\_intID == 0)

\_intID = -1;

}

Int32 \_intPosID = -1;

if (ZLDCommon.IsNumeric(posID.Text))

{

Int32.TryParse(posID.Text, out \_intPosID);

if (\_intPosID == 0)

\_intPosID = -1;

}

Int32 intNum = -1;

if (ZLDCommon.IsNumeric(Num.Text))

{

Int32.TryParse(num.Text, out intNum);

if (intNum == 0)

intNum = -1;

}

Int32 intLfdNr = -1;

if (ZLDCommon.IsNumeric(LfdNr.Text))

{

Int32.TryParse(LfdNr.Text, out intLfdNr);

if (intLfdNr == 0)

intLfdNr = -1;

}

Int32 intAuftragsNr = -1;

if (ZLDCommon.IsNumeric(AuftragsNr.Text))

{

Int32.TryParse(AuftragsNr.Text, out intAuftragsNr);

if (intAuftragsNr == 0)

intAuftragsNr = -1;

}

**Ansatz 1, eine erste Lösung:**Refakturierung durch Auslagerung der Code Redundanz in eine separate private Funktion in diesem Modul (**ToInt**):

**var** \_intID = **ToInt(**ID.Text**);**

**var** \_intPosID = **ToInt(**PosID.Text**);**

**var** intNum = **ToInt(**Num.Text**);**

**var** intLfdNr = **ToInt(**LfdNr.Text**);**

**var** intAuftragsNr = **ToInt(**AuftragsNr.Text**);**

**private Int32 ToInt(string sNum)**

**{**

**Int32 intNum = -1;**

**if (ZLDCommon.IsNumeric(sNum))**

**{**

**Int32.TryParse(ID.Text, out intNum);  
 if (intNum == 0)**

**intNum = -1;**

**}**

**return intNum;**

**}**

**Ansatz 2, die konsequente Lösung:**Refakturierung durch Auslagerung der Code Redundanz in eine separate Standard-Funktion einer wiederverwendbaren Utility Code Bibliothek:

**var \_intID = ID.Text.ToInt();**

**var \_intPosID = PosID.Text.ToInt();**

**var intNum = Num.Text.ToInt();**

**var intLfdNr = LfdNr.Text.ToInt();**

**var intAuftragsNr = AuftragsNr.Text.ToInt();**

[ … separate wiederverwendbare Utility Code Bibliothek … ]

**public static class StringExtensions**

**{**

**[…]**

**public static int ToInt(this string stringValue, int defaultValue = -1)**

**{**

**int tmp;**

**if (!Int32.TryParse(stringValue.NotNullOrEmpty(), out tmp))**

**return defaultValue;**

**return tmp;**

**}**

**[…]**

**}**

Der Vorteil dieser Lösung liegt auf der Hand:  
Nahezu 1 Bildschirmseite Code wurde auf nur 5 Code Zeilen reduziert!  
Die Lesbarkeit wurde deutlich erhöht, der Code wurde auf das Wesentliche konzentriert.  
Häufig benötigte Standard Funktionalitäten verschwinden in einer speziell dafür vorgesehenen Bibliothek.  
Diese Standard Biblotheken können in dann in mehrere Software Projekte eingebunden und deren Funktionalität dort gemeinsam genutzt werden.  
  
Ein weiterer Vorteil ist die Schaffung eines Projekt-übergreifenden Standards für häufig benötigte Funktionalitäten, einerseits in der Funktionsweise , andererseits in der Namensgebung:

In diesem Beispiel verhält sich das Parsen einer Nummer aus einem String immer gleich und die Funktion heißt immer und überall „ToInt()“.

## „Füllstoff“ Code-Kommentare, vermeidbar bei redundanten Funktionen

Ok, Code Kommentare sind hilfreich.  
Aber Code lebt, wird gewartet, erweitert, häufig umgeschrieben oder in gar andere Dateien verschoben.  
Und das ist alles gut + richtig so.   
*Nicht* gut aber ist, dass lieb gemeinte Code Kommentare mit der Zeit veralten oder „verwaisen“.  
Im Verlauf der Zeit arbeiten verschiedene Entwickler an demselben Code,   
erweitern ihn und passen ihn an die sich mit der Zeit ändernden Business Regeln an.  
Dabei hat zwar jeder Entwickler so seinen eigenen Programmier-Stil,   
*eines* haben die meisten Entwickler jedoch gemeinsam:  
Entwickler rühren die Code Kommentare anderer Entwickler nicht an!  
Wie Heiligtümer bleiben fremde Code Kommentare unberührt, werden gar 1:1 mit dem Code verschoben, während der durch sie kommentierte Code sich laufend ändert und die Kommentare somit zum Ersten Ihre Bedeutung verlieren und zum Zweiten den lesenden Entwickler auch noch in die Irre führen!  
Pervertiert wird das Ganze noch, wenn die optische Komplexität offensichtlicher Code Redundanzen mit im Grunde ja unnötigen Kommentaren dem Leser verständlicher zu machen versucht wird.  
Bei Refakturierung bedürfte es jedoch hier nicht eines Code Kommentars!

Hier ein Beispiel aus einer bestehenden Anwendung:  
Die hervorgehobenen Stellen beschreiben die sehr wenigen Unterschiede folgender 3 Code-Funktionen:  
Mal ehrlich … brauchen wir hierfür wirklich eine Bildschirmseite Code, der zudem hauptsächlich aus unnötigem und ziemlich ermüdendem Füllstoff besteht … ?

/// <summary>

/// Das gestrige Datum in das Control txtZulDate einfügen.

/// </summary>

/// <param name="sender">object</param>

/// <param name="e">EventArgs</param>

protected void lbtnGestern\_Click(object sender, EventArgs e)

{

DateTime dDate;

String sDate;

dDate = DateTime.Today.AddDays(-1);

sDate = dDate.Day.ToString().PadLeft(2, '0') + dDate.Month.ToString().PadLeft(2, '0') + dDate.Year.ToString().Substring(2, 2);

txtZulDate.Text = sDate;

txtKennz2.Focus();

}

/// <summary>

/// Das heutige Datum in das Control txtZulDate einfügen.

/// </summary>

/// <param name="sender">object</param>

/// <param name="e">EventArgs</param>

protected void lbtnHeute\_Click(object sender, EventArgs e)

{

DateTime dDate;

String sDate;

dDate = DateTime.Today;

sDate = dDate.Day.ToString().PadLeft(2, '0') + dDate.Month.ToString().PadLeft(2, '0') + dDate.Year.ToString().Substring(2, 2);

txtZulDate.Text = sDate;

txtKennz2.Focus();

}

/// <summary>

/// Das morgige Datum in das Control txtZulDate einfügen.

/// </summary>

/// <param name="sender">object</param>

/// <param name="e">EventArgs</param>

protected void lbtnMorgen\_Click(object sender, EventArgs e)

{

DateTime dDate;

String sDate;

dDate = DateTime.Today.AddDays(1);

sDate = dDate.Day.ToString().PadLeft(2, '0') + dDate.Month.ToString().PadLeft(2, '0') + dDate.Year.ToString().Substring(2, 2);

txtZulDate.Text = sDate;

txtKennz2.Focus();

}

Die Antwort lautet: „Nein“ - und nochmals „Nein!“

Wie wäre es mit dieser Lösung, bei der wir zudem noch auf die meisten Code Kommentare verzichten können! (…denn was glauben wir wohl welches Datum gesetzt wird von einer Funktion, die den Bezeichner „Gestern“ enthält … ?)

protected void lbtnGestern\_Click(object sender, EventArgs e)

{

SetTextboxDate(DateTime.Today.AddDays(-1));

}

protected void lbtnHeute\_Click(object sender, EventArgs e)

{

SetTextboxDate(DateTime.Today);

}

protected void lbtnMorgen\_Click(object sender, EventArgs e)

{

SetTextboxDate(DateTime.Today.AddDays(1));

}

/// <summary>

/// Ein bestimmtes Datum in das Control txtZulDate einfügen.

/// </summary>

private void SetTextboxDate(DateTime dDate)

{

String sDate;

sDate = dDate.Day.ToString().PadLeft(2, '0') + dDate.Month.ToString().PadLeft(2, '0') + dDate.Year.ToString().Substring(2, 2);

txtZulDate.Text = sDate;

txtKennz2.Focus();

}

Die neu ausgelagerte Funktion SetTextboxDate übernimmt den Löwenanteil der Arbeit.  
Das Verständnis beim Lesen wird zusätzlich durch die Markierung des private Modifizierers erleichtert, weil sich dem Leser sofort erschließt dass diese Fuktion nur in dieser Code Klasse Bedeutung hat, also nur hier verwendet wird.

Aber warum wird derart schlechter Code überhaupt produziert?

Ausreden wie „*Die Zeit war knapp..*“   
oder „*Der Kunde hat uns wieder unter Dampf gesetzt!*“  
sind inakzeptabel!

Mit den heutigen modernen Entwicklungsumgebungen wie „Eclipse“ oder „Visual Studio“ sowie leistungsfähigen Add Ons wie „Resharper“ sind Auslagerungen von Code Redundanzen in separate Funktionen nur 1 einzigen Mausklick entfernt!

Das Entscheidende aber ist, es *wirklich* *zu tun* !  
Ein guter Clean Code Entwickler nimmt sich die (wenige) Zeit um seinen Code übersichtlich + lesbar zu gestalten und zu *halten*.

Ein guter Clean Code Entwickler liebt seine Arbeit und pflegt liebevoll seinen Code!  
Er ist auch bei Erweiterungsarbeiten an älteren Programmen ständig bemüht, den Code sauber zu halten!

## Konfigurierbare Einstellungen

Konfigurierbare Einstellungen wie Pfade und URLs sollten niemelas „hart“im Code verdrahtet werden.

if (m\_User.IsTestUser)

{

sPath = "\\\\192.168.10.96\\test\\portal\\zld\\lieferscheine\\";

}

else

{

sPath = "\\\\192.168.10.96\\prod\\portal\\zld\\lieferscheine\\";

}

**Lösung:**  
Stattdessen werden derartige Einstellungen in einer Konfigurationsdatei abgelegt und im Code darauf verwiesen:

if (m\_User.IsTestUser)

{

sPath = **ConfigurationManager.AppSettings["LieferscheinePfadTest"]**;

}

else

{

sPath = **ConfigurationManager.AppSettings["LieferscheinePfadProd"]**;

}

# Optische Optimierungen (Kategorie “gelb“)

## Code Block Verschachtelungen reduzieren

Betrachten wir unser inzwischen Clean Code bereinigtes Code Beispiel aus dem ZLD-Portal von oben:

**private void ProcessAuthorization( ref Boolean bError, ref String \_sDRelGeb, Int32 \_intID,   
Int32 \_intPosID, String \_loeschkz, TextBox \_txtBox,   
Decimal \_decGeb, Decimal \_decGebAmt, DataRow row )**

**{**

**if (m\_User.Groups[0].Authorizationright != 1)**

**{**

**var rowsPauschal = objKompletterf.tblEingabeListe.Select("ID=" + \_intID);**

**var xSteuer = rowsPauschal[0]["OhneSteuer"].ToString();**

**if (rowsPauschal[0]["Pauschalkunde"].ToString() == "X")**

**{**

**if (\_decGeb != \_decGebAmt)**

**{**

**var gebMat = "";**

**var rowsGebMat = objKompletterf.tblEingabeListe.Select("ID=" + \_intID + " AND id\_pos =" + \_intPosID);**

**if (rowsGebMat.Length == 1)**

**{**

**if (xSteuer == "X")**

**gebMat = rowsGebMat[0]["GebMatnr"].ToString();**

**else**

**gebMat = rowsGebMat[0]["GebMatnrSt"].ToString();**

**}**

**\_sDRelGeb = objKompletterf.GetSDRelevantsGeb(\_intID, \_intPosID, gebMat);**

**if (\_txtBox.Visible && row == "" && \_loeschkz != "X" && \_sDRelGeb != "X")//&& lblOkKZ.Visible**

**{**

**\_txtBox.BorderColor = System.Drawing.ColorTranslator.FromHtml("#bc2b2b");**

**lblError.Text = "Bei Pauschalkunden dürfen Gebühr und Gebühr Amt nicht unterschiedlich sein!";**

**bError = true;**

**}**

**}**

**}**

**}**

**}**

**Lösung:** Optische Optimierung durch Umstellung der „if“ Statements:   
 **private void ProcessAuthorization( ref Boolean bError, ref String \_sDRelGeb, Int32 \_intID,   
Int32 \_intPosID, String \_loeschkz, TextBox \_txtBox,   
Decimal \_decGeb, Decimal \_decGebAmt, DataRow row )  
{   
 if (m\_User.Groups[0].Authorizationright == 1)  
 return;  
  
 var rowsPauschal = objKompletterf.tblEingabeListe.Select("ID=" + \_intID);  
 var xSteuer = rowsPauschal[0]["OhneSteuer"].ToString();  
 if (rowsPauschal[0]["Pauschalkunde"].ToString() != "X")  
 return;  
 if (\_decGeb == \_decGebAmt)  
 return;**

**var gebMat = "";**

**var rowsGebMat = objKompletterf.tblEingabeListe.Select("ID=" + \_intID + " AND id\_pos =" + \_intPosID);**

**if (rowsGebMat.Length != 1)**

**return;**

**if (xSteuer == "X")**

**gebMat = rowsGebMat[0]["GebMatnr"].ToString();**

**else**

**gebMat = rowsGebMat[0]["GebMatnrSt"].ToString();**

**\_sDRelGeb = objKompletterf.GetSDRelevantsGeb(\_intID, \_intPosID, gebMat);**

**if (\_txtBox.Visible && row == "" && \_loeschkz != "X" && \_sDRelGeb != "X")//&& lblOkKZ.Visible**

**{**

**\_txtBox.BorderColor = System.Drawing.ColorTranslator.FromHtml("#bc2b2b");**

**lblError.Text = "Bei Pauschalkunden dürfen Gebühr und Gebühr Amt nicht unterschiedlich sein!";**

**bError = true;**

**}**

**}**

Resultat: - Die Anzahl der Verschachtelungen wurde von 4 auf 2 Ebenen reduziert.  
- Code mit weniger Verschachtelungen liest sich flüssiger, nach jedem „return“ ist die  
 Bedingung abgehakt, der Kopf ist „frei“ für die nächsten Code Zeilen.

## Doppelt negierte Ausdrücke vermeiden

In folgendem Fall beinhaltet die Bedeutung im Variablennamen bereits eine Negierung:   
„NotValid“.  
Soweit ok.   
Eine erneute Negierung durch den zusätzlichen Negierungs-Operator „!“ führt zu doppelten Negierungen, die grundsätzlich schwerer zu lesen sind.

if (!NotValid)

{

}  
else  
{  
}

**Lösung**: Vertauschung der Code Blöcke „if“ „else“

if (NotValid)

{

}  
else  
{  
}

## „if“ Statements lesbarer machen

In folgendem kommt vor einem schlanken, einzeiligen „else“ Zweig ein komplexerer „if“ Zweig.  
Beim Lesen des schlankeren „else“ Zweiges fehlt der schnelle optische Zugriff auf die zugehörige „if“ edingung, weil diese durch den vorangehenden komplexen Bedingsteil verdeckt wird, bzw. optisch wieder „in Vergessenheit“ gerät.

if (!NotValid)

{

FillBarabhebung();

if (Sendmail())

{

objBarabhebung.Change(Session["AppID"].ToString(), Session.SessionID, this);

if (objBarabhebung.Status != 0)

{

lblError.Text = "Fehler: " + objBarabhebung.Message;

}

else

{

lblError.Text = "Daten erfolgreich gespeichert!";

ClearForm();

if ((objBarabhebung.PDFXString != null) && (objBarabhebung.PDFXString.Length > 0))

{

Session["PDFXString"] = objBarabhebung.PDFXString;

ResponseHelper.Redirect("Printpdf.aspx", "\_blank", "left=0,top=0,resizable=YES,scrollbars=YES");

}

else

{

lblError.Text += "PDF-Generierung fehlgeschlagen.";

}

}

}

}

**else**

**lblError.Text = "Prüfen Sie die rot markierten Felder!";**

**Lösung**: Auch hier Vertauschung der Code Blöcke „if“ „else“

if (NotValid)

**lblError.Text = "Prüfen Sie die rot markierten Felder!";**

else  
{   
 […]

## Wiederholende Textbezeichnungen zusammenfassen

if (istRechnung)

{

sText = "Bitte geben Sie eine Referenzbezeichnung für die angeforderte Rechnung ein!";

}

else

{

sText = "Bitte geben Sie eine Referenzbezeichnung für die angeforderte Auftragsbestätigung ein!";

}

**Lösung 1:** Zusammenfassung der Bezeichnungen  
  
var meldung = "Bitte geben Sie eine Referenzbezeichnung für die angeforderte {0} ein!";  
if (istRechnung)

{

sText = string.Format(meldung, "Rechnung");

}

else

{

sText = string.Format(meldung, "Auftragsbestätigung");

}

**Lösung 2:** Zusätzliche Eleminierung des „if“ Statements  
  
  
var meldung = "Bitte geben Sie eine Referenzbezeichnung für die angeforderte {0} ein!";

sText = string.Format(meldung, istRechnung ? "Rechnung" : "Auftragsbestätigung");

**Resultat**:

* Übersicht erhöht
* Fehleranfälligkeit durch Vermeidung doppelter Texte minimiert.
* Lesbarkeit verbessert

## Bedeutungsfreie Ausdrücke verständlich umformulieren

Während die erste Bedingung im folgenden Statement noch einigermaßen interpretierbar ist, ist bei der rechten Bedingung (gelb unterlegt) die inhaltliche Bedeutung nicht mehr nachvollziehbar.

if (dRow["WebMTArt"].ToString() == "D" && (Int32)dRow["ID\_POS"] == 10) **???**

{  
}

**Lösung:**Auslagern der Teilbedingungen in separate sowie „sprechende“ boolsche Variablen.  
Sogar Code Kommentare sind hier dann auch nicht mehr nötig:  
var **istDeutsch** = dRow["WebMTArt"].ToString() == "D";  
var **istKopfzeile** = (Int32)dRow["ID\_POS"] == 10);  
if (istDeutsch && **istKopfzeile**)   
{  
}

## Lange teilredundante Bezeichner

Clean Code schützt auch die Augen des Lesers vor semamtisch redundanten Teilbezeichnern.   
In diesem Beispiel besagt der Namespace bereits, dass es sich um Objekte der Thematik „Insurance“ handelt.

namespace CkgDomainLogic.Insurance.ViewModels

{

public class InsuranceSchadenakteViewModel : CkgBaseViewModel

{

[XmlIgnore]

public IInsuranceSchadenakteDataService DataService   
 { get { return CacheGet<IInsuranceSchadenakteDataService>(); } }

public Schadenakte Schadenakte { get; set; }

public InsuranceSchadenakteDocsViewModel DocsViewModel { get; set; }

}

}

Die erneute Verwendung des Teilbezeichners „Insurance“ in Typ- und Property-Bezeichnern ist von daher überflüssig und erschwert zudem das Lesen des gesamten Codes.  
Auch die Unterscheidung zwischen den einzelnen Bezeichnern wird deutlich erschwert.  
Dass „InsuranceSchadenakteViewModel“ etwas anderes ist als „InsuranceSchadenakteDataService“, erschließt sich halt nicht sofort auf dem schnellen Blick.

Die Begriffe „SchadenakteViewModel“ + „SchadenakteDataService“ hingegen sind bereits schon beim flüchtigen Lesen deutlich besser voneinander zu unterscheiden:

namespace CkgDomainLogic.Insurance.ViewModels

{

public class SchadenakteViewModel : CkgBaseViewModel

{

[XmlIgnore]

public ISchadenakteDataService DataService   
 { get { return CacheGet<ISchadenakteDataService>(); } }

public Schadenakte Schadenakte { get; set; }

public SchadenakteDocsViewModel DocsViewModel { get; set; }

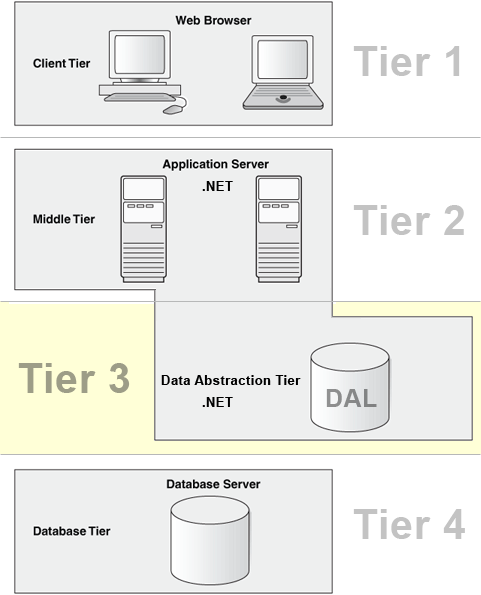
}

}

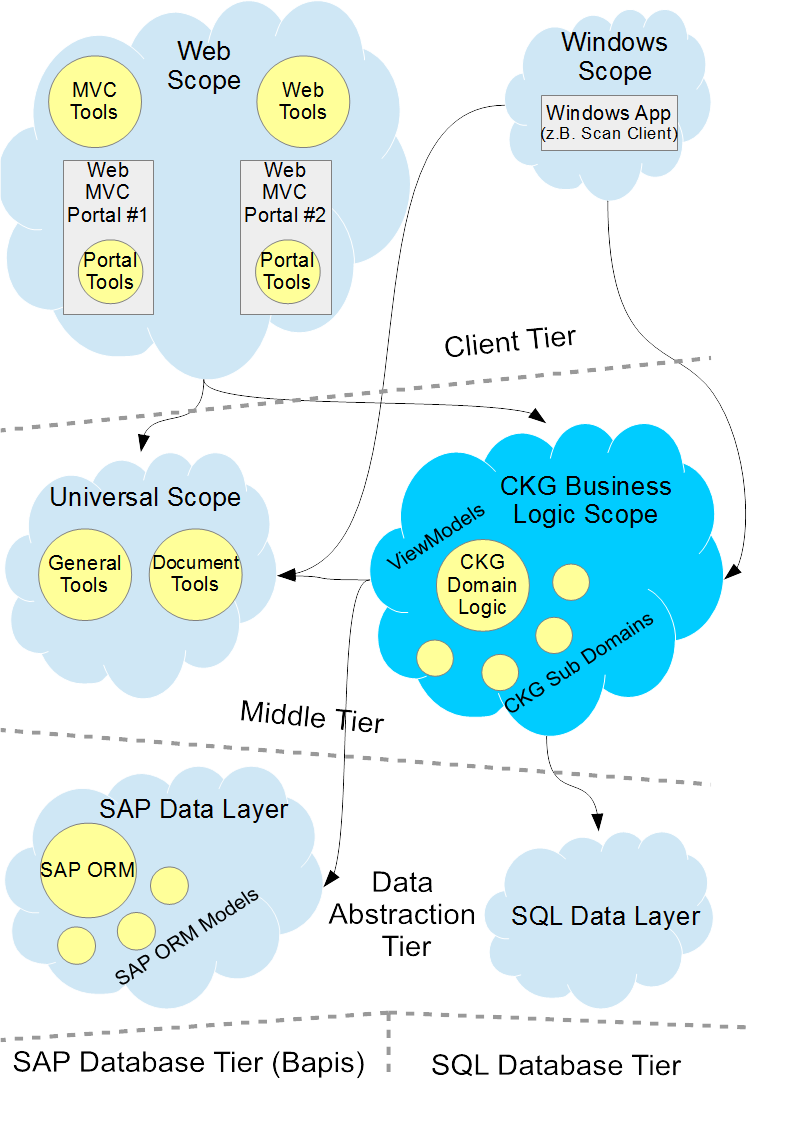
# Die CKG Modul Architektur

## 4-Tier Architektur

|  |  |
| --- | --- |
| Die Module der CKG .NET Anwendungen erweitern das 3-Tier Archtitektur Modell, das Anwendungen auf 3 Schichten in unterschiedlichen Rechnersystemen verteilt. |  |

  
  
  
  
  
  
Zwischen der „Middle Tier“ und der „Database Tier“ haben wir bei CKG noch eine zusätzliche Schicht für die **Daten Abstraktion** eingeführt. Die urpsünglich 3. Schicht „Database Tier“ wird nun zur „Tier 4“.  
Die neue Schicht „Tier 3“, der sogenannte Data Abstraction Layer (DAL), ermöglicht uns den kompletten Austausch der Database Tier (Tier 4), z. B. von SAP auf SQL für Simulations- bzw. Prototyping Zwecke.  
Hinweis: Tier 2 + 3 befinden sich auf denselbem Rechnersystem und sind in .NET implementiert.

## CKG Module, Anordnung innerhalb der 4-Tier Architektur



## MVC Abbildung auf die CKG 4-Tier Architektur

Die Abbildung des MVC Patterns auf unsere Architektur bei CKG erfolgt streng genommen über eine eigene Erweiterung in Form eines MVC-**VM** Pattern.  
Das **VM** steht für das **V**iew**M**odel Pattern das schon in WPF Anwendungen unter Windows Systemen erfolgreich zum Einsatz kommt.

## Warum ViewModels?

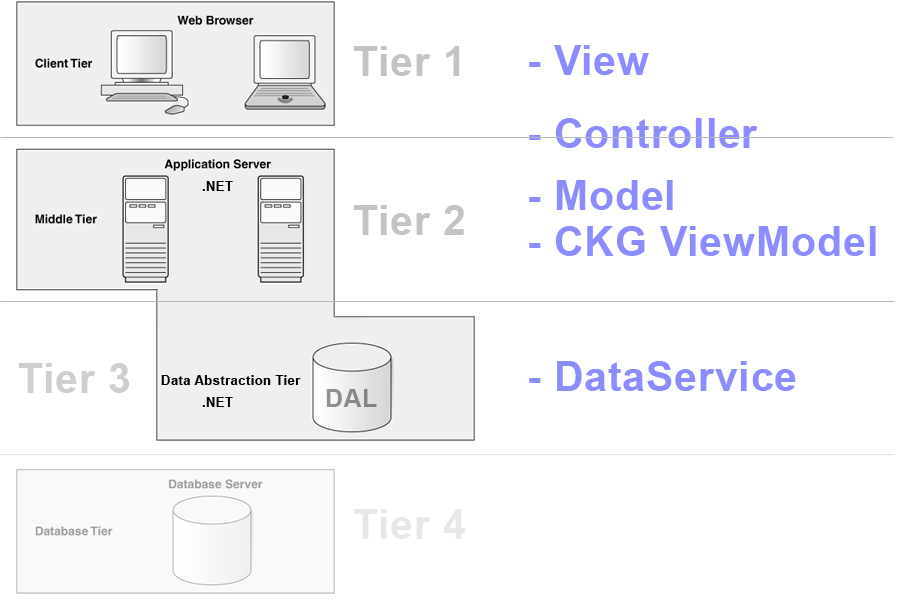
Die ViewModel Ebene wird zwischen den DAL (Data Abstraction Layer) und der Controller Schicht gezogen und hat den Sinn + Zweck, sämtliche Business Logic in einer einzigen Schicht zu konzentrieren, sowie separate, extern gesteuerte Tests der Business Logic auf Modulebene zu ermöglichen (Unit Testing).

Logiken, Filterregeln, Business Rules, etc, werden deshalb nur in den ViewModels platziert – und bitte nur dort!  
Denn nur so können Redundanzen auf DAL Ebene („Tier 3“) vermieden werden.  
Der DAL sollte nur die reinen Datenzugriffs-Logiken beherbergen.

Zudem abstrahiert die über der ViewModel Schicht liegende Controller Schicht den Front End Typ.  
Das heißt, dass sämtliche Business Logik wiederverwendbar ist, und zwar unabhängig von der Wahl des Front Ends.  
Z. B. können dann Windows Anwendung mit denselben ViewModel Modulen arbeiten wie Web Anwendungen.

## MVC-VM Pattern, Anordnung innerhalb der 4-Tier Architektur

Die MVC-VM Implementierungen verteilen sich ausschließlich auf Tier 1 + 2.  
Hervorzuheben ist, dass die Controller Schicht als Bindeglied zwischen Tier 1 + 2 fungiert.



## Clean Code / MVC-VM / 4-Tier

Hier schließt sich der Kreis.   
Wir erweitern wie folgt den Clean Code Gedanken um unsere firmeneigenen Bedürfnisse bei CKG:

***Clean Code im Zusammenhang mit dem MVC-VM Pattern und der CKG 4-Tier Architektur bedeutet die konsequente Platzierung des .NET Codes in die Schichten in die er letztendlich hingehört!***

Das heißt dass ein CKG Entwickler per se nur dann Clean Code erstellt, wenn er sich auch konsequent an diese Architektur-Regeln hält.

## Beispiele für Schichtenkorrekturen ( „Falsch / Richtig“)

Zum besseren Verständnis werden im folgenden einige Beispiele gelistet, wie man verkehrt platzierten Code an die richtige Stelle rückt.

Dabei geht es hauptsächlich um verkehrt platzierten Code in den Schichten direkt ober- und unterhalb der ViewModel Ebene.   
Meist landet Business Code im DAL Layer, der aber gar nichts in der Datenschicht zu suchen hat und stattdessen ins ViewModel gehört.  
Analog ist der Entwickler häufig versucht, schnell mal eben Business Regeln im Controller zu implementieren, statt - wie richtig wäre - den Code ins ViewModel zu platzieren.

### Beispiel Schichtenkorrektur „ViewModel <-> DAL Layer“

**Falsch:**

public List<TerminSchadenfall> BoxBlockerTermine

{

get   
 {   
 return PropertyCacheGet(() => EventsDataService  
 .TermineGet()  
 .ToList());   
 }

}

public List<TerminSchadenfall> TermineGet(Schadenfall schadenfall = null, int boxID = 0)

{

[…]

return AppModelMappings.Z\_DPM\_TAB\_ZEVENT\_TERMIN\_01\_GT\_TERMIN\_To\_TerminSchadenfall  
 .Copy(sapList)  
 .Where(item => item.LoeschDatum == null  
 **&& item.IsBlockerDummyTermin)** .ToList();

}

**Richtig:**

public List<TerminSchadenfall> BoxBlockerTermine

{

get   
 {   
 return PropertyCacheGet(() => EventsDataService  
 .TermineGet()  
 **.Where(t => t.IsBlockerDummyTermin)** .ToList());   
 }

}

public List<TerminSchadenfall> TermineGet(Schadenfall schadenfall = null, int boxID = 0)

{

[…]

return AppModelMappings.Z\_DPM\_TAB\_ZEVENT\_TERMIN\_01\_GT\_TERMIN\_To\_TerminSchadenfall  
 .Copy(sapList)  
 .Where(item => item.LoeschDatum == null)  
 .ToList();

}

**Erläuterung:**

In diesem Beispiel wurde die Business Filterregel für so genannte „Box-Blocker Termine“ aus dem DAL Layer in die ViewModel Ebene hoch geschoben.

Grund für diese Korrektur ist, dass ansonsten eine Redundanz in allen DAL Layern programmiert werden müsste.  
Z. B. müsste diese Logik in einem SAP DAL Layer und zusätzlich ein weiteres Mal in einem SQL Layer programmiert werden. (**Code Redundanz**!)

Ein weiterer Vorteil ist zudem, dass die DAL Layer möglichst schlank gehalten werden, also wenig Code beinhalten.  
Idealer Weise enthalten die DAL Layer nur den Minimal Code der für den reinen properitären Datenzugriff nötig ist!

Es stellt sich an dieser Stelle ggfls. noch die Frage, warum die Bedingung   
item => item.LoeschDatum == null trotzdem im DAL verbleibt.

Die Antwort ist dass diese Regel vom DAL abhängig ist!  
Denn nur der DAL allein entscheidet, ob gültige Werte kein Löschdatum haben und ungültige mit einem solchen versehen werden.  
In diesem Beispiel gilt diese Regel nur für den SAP DAL Layer.  
Ein anderer DAL Layer hingegen könnte u. U. gar kein Lösch-Datum zur Verfügung stellen, weil er ein Löschen u. U. gar nicht unterstützt oder ein Löschen humorlos ganz ohne Setzen eines Lösch-Datums durchführt!

### Beispiel Schichtenkorrektur „Controller <-> ViewModel“

**Falsch:**

public ActionResult VersEventOrtBoxDetailsFormSave(VersEventOrtBox model)

{

**if (model.BoxArt == "RE")**

**{**

**// bei BoxArt "Techniker" die Versicherung "löschen"**

**model.VersicherungID = null;**

**if (model.TechnikerName.IsNullOrEmpty())**

**{**

**ModelState.AddModelError (m => m.TechnikerName, "Bitte geben Sie einen Techniker an");**

**return;**

**}**

**}**

EventsViewModel.VersEventOrtBoxValidate(model, ModelState.AddModelError);

[…]

return PartialView("Konfigurator/VersEventOrtBoxDetailsForm", model);

}

public void VersEventOrtBoxValidate(VersEventOrtBox model,   
 Action<Expression<Func<VersEventOrtBox, object>>, string> addModelError)

{

if (model.BoxArt == "GU")

{

[…]

}  
  
 […]

}

**Richtig:**

public ActionResult VersEventOrtBoxDetailsFormSave(VersEventOrtBox model)

{

EventsViewModel.VersEventOrtBoxValidate(model, ModelState.AddModelError);

[…]

return PartialView("Konfigurator/VersEventOrtBoxDetailsForm", model);

}

public void VersEventOrtBoxValidate(VersEventOrtBox model,   
 Action<Expression<Func<VersEventOrtBox, object>>, string> addModelError)

{

if (model.BoxArt == "RE")

{

// bei BoxArt "Techniker" die Versicherung "löschen"

model.VersicherungID = null;

if (model.TechnikerName.IsNullOrEmpty())

{

addModelError(m => m.TechnikerName, "Bitte geben Sie einen Techniker an");

return;

}

}

if (model.BoxArt == "GU")

{

[…]

}  
  
 […]

}

**Erläuterung:**

In diesem Beispiel wurden Business Regeln für bestimmte Model Validierungen leider direkt im Controller implementiert.  
Nach den CKG Clean Code Regeln sollte der Controller aber nur den Minimal Code enthalten den er für die Steuerung (Controlling) der Views benötigt, zudem bitteschön auch nur Web abhängigen .NET Code.

Deshalb wurde der Business Code in die ViewModel Schicht geschoben, wo er letztlich auch hingehört!

Vorteil 1:  
Genau wie beim DAL Layer bleibt auch im Controller Layer der Code schlank, weil der Controller nur den Code enthält auf den er „spezialisiert“ ist.

Vorteil 2:  
Beim Switch des Client Tiers von einem Web Client auf einen Windows Client oder gar ein anderes Front End muss der Business Code nicht dupliziert werden!  
Denn wenn der Code korrekt im ViewModel platziert ist, kann das ViewModel Modul vom anderen Client Front End aus einfach wiederverwendet werden, ganz ohne Code Redundanz!

### Beispiele View Schicht, .NET Code Fehlplatzierungen

Auch in der View-Schicht gilt:  
Lesbarkeit und Übersicht wird durch Clean Code gefördert.

#### Unverständliche Code Bedingungen im View mit Model Properties kapseln

Im folgenden Code Beispiel werden unverständliche Bedingungen in sprechende boolesche Model Properties gekapselt. Die unverständlichen Bedingungen wandern ins Backend des Models.  
Einem unbeteiligter Entwickler erschließtmdich die Bedeutung der Bedingungen in der View Schicht sofort, auch ohne ins Model Backend schauen zu müssen.

**Falsch:**

@model CkgDomainLogic.Fahrer.Models.FahrerAuftrag

<div class="grid-auftrags-command-bar">

<a href="GetAuftragsPdfBytes?auftragsNr=@Model.AuftragsNr" class="btn blue mini">  
 <i class="icon-download-alt"></i>  
 </a>

@if (**Model.FahrerStatus.NotNullOrEmpty().Trim().IsNullOrEmpty() || Model.FahrerStatus.NotNullOrEmpty() == "NO"**)

{

<a href="#" onclick="return SetFahrerAuftragsStatus('@Model.AuftragsNr', 'OK');" class="btn green mini">  
 <i class="icon-ok"></i>  
 </a>

}

@if (**Model.FahrerStatus.NotNullOrEmpty().Trim().IsNullOrEmpty() || Model.FahrerStatus.NotNullOrEmpty() == "OK"**))

{

<a href="#" onclick="return SetFahrerAuftragsStatus('@Model.AuftragsNr', 'NO');" class="btn red mini">  
 <i class="icon-remove"></i>  
 </a>

}

</div>

**Richtig:**

@model CkgDomainLogic.Fahrer.Models.FahrerAuftrag

<div class="grid-auftrags-command-bar">

<a href="GetAuftragsPdfBytes?auftragsNr=@Model.AuftragsNr" class="btn blue mini">  
 <i class="icon-download-alt"></i>  
 </a>

@if (**Model.AuftragIstNeu || Model.AuftragIstAbgelehnt**)

{

<a href="#" onclick="return SetFahrerAuftragsStatus('@Model.AuftragsNr', 'OK');" class="btn green mini">  
 <i class="icon-ok"></i>  
 </a>

}

@if (**Model.AuftragIstNeu || Model.AuftragIstAngenommnen**)

{

<a href="#" onclick="return SetFahrerAuftragsStatus('@Model.AuftragsNr', 'NO');" class="btn red mini">  
 <i class="icon-remove"></i>  
 </a>

}

</div>

Public class FahrerAuftrag   
{   
 [...]

public bool AuftragIstNeu { get { return FahrerStatus.NotNullOrEmpty().Trim().IsNullOrEmpty(); } }

public bool AuftragIstAbgelehnt { get { return FahrerStatus.NotNullOrEmpty() == "NO"; } }

public bool AuftragIstAngenommnen { get { return FahrerStatus.NotNullOrEmpty() == "OK"; } }  
}

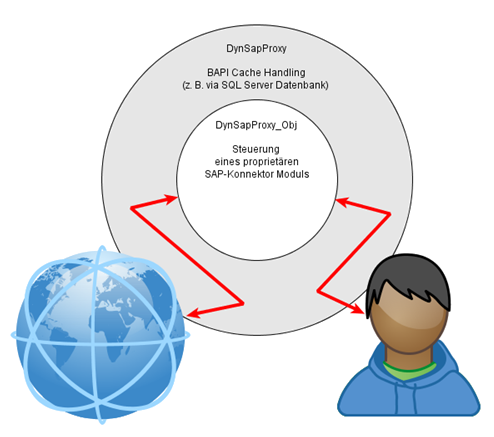
# Clean Code in der System Architektur

## SoC - “Separation of concerns”

Eine der wichtigsten Regeln in der System Archtiketur ist die Trennung von Verantwortlichkeiten innerhalb der Module, die „separation of concerns“.  
Entscheidender Grundsatz hierbei ist, jedem Code Modul möglichst nur eine Aufgabe zuzuweisen (Single Responsibility Prinzip).

Warum?  
Wenn eine Codeeinheit keine klare Aufgabe hat, ist es schwer sie zu verstehen, sie anzuwenden und sie ggf. zu korrigieren oder zu erweitern!

Separation of Concerns führt darüber hinaus auch zu gut testbaren Komponenten. Denn wenn der Zweck einer Codeeinheit fokussiert ist, muss weniger breit getestet werden.  
Unit Tests sind einfacher zu erstellen und jederzeit ohne größere Abhängigkeiten automatisiert durchführbar.

Hier ein Paradebeispiel für die Verletzung des SoC Prinzips aus den älteren CKG Basis Modulen.   
In der CKG Entwicklung umgangsprachlich „Base“ genannt (CKG.Base.dll):

**Falsch:**

Die bisherige Architektur im so- genannten „DynSapProxy“ stellt eine Kernklasse „DynSapProxy\_Obj“ (weißer Ring in der Grafik) bereit, die die proprietäre Konnektivität zum SAP ERP-Konnektor Version 3.5 kapselt.  
Die von dieser Kernklasse „DynSapProxy\_Obj“ abgeleitete Klasse „DynSapProxy“ (grauer Ring) regelt im Grunde nur das Datenbank Caching der SAP BAPI Strukturen.

Folgende Nachteile:  
  
1. Der äußere Ring stellt mit dem BAPI Caching eine Standard-Funktionalität zur Verfügung, die im Grunde unabhängig von der Kern-Funktionalität implementiert sein sollte.  
Möchte man ein weiteres Kernmodul mit abweichender Kern-Funktionalität parallel entwickeln, müsste der Code des äußeren Kerns per „Cut Copy Paste“ dupliziert werden.

2. Der innere Kern ist mit Abhängigkeiten zu externen Funktionalitäten wie z. B. Web- und User-Kontexten behaftet.

Hiermit ist es nicht möglich, die eigentliche Kernfunktionalität – nämlich der SAP Zugriff – eigenständig und modular zur Verfügung zu stellen.

Eine Verwendung der Kernfunktionalität z. B. in einer Windows-Anwendung ist somit nicht möglich, weil intern z. B. auf einen Web-Kontext verwiesen wird.

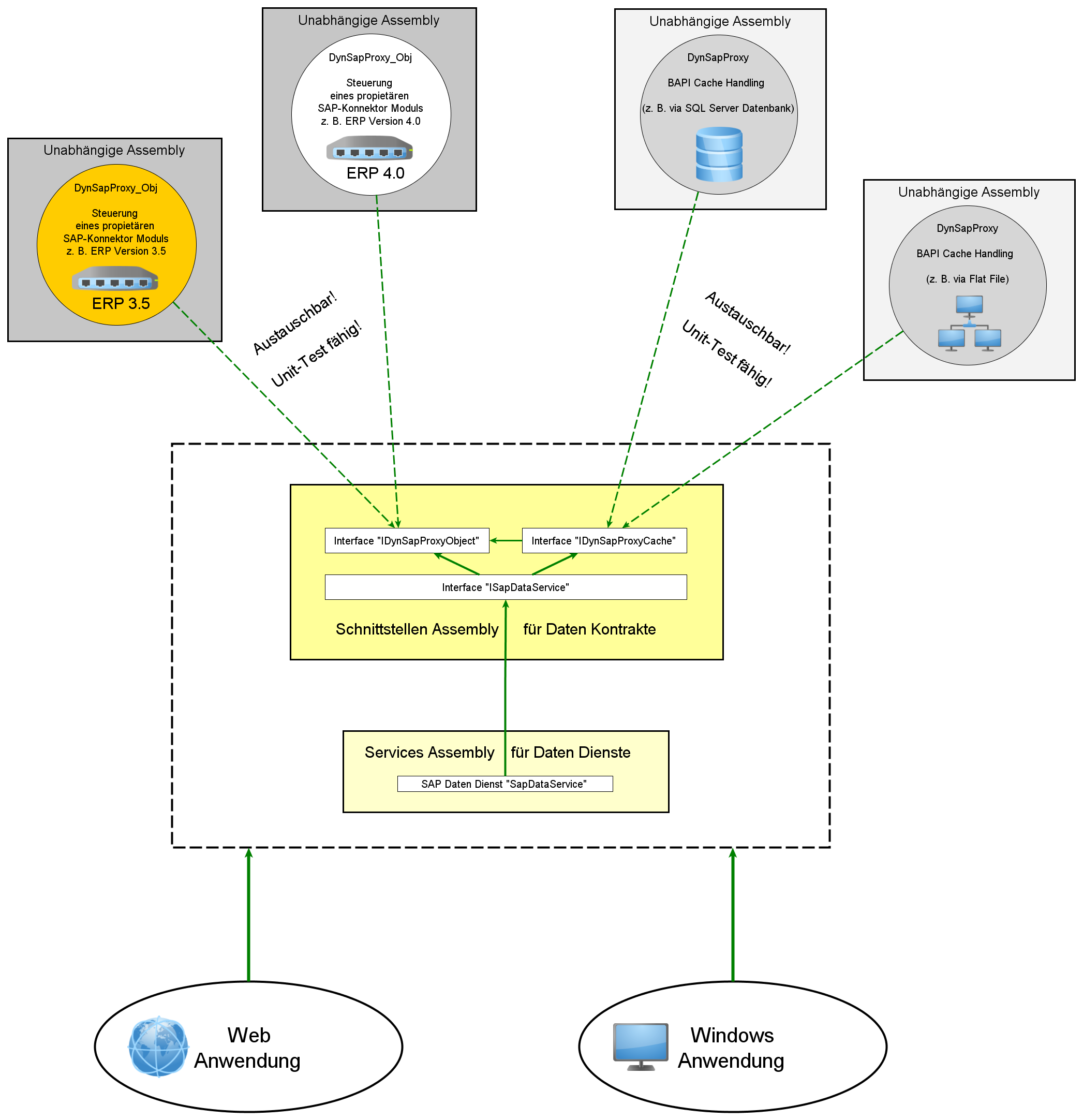
**Richtig:**

In der neuen Architektur wurden folgende Veränderungen vorgenommen:

1. Auslagerung der beiden abhängigen Ringe aus Abbildung 1 in zwei voneinander unabhängige Funktionsgruppen.

2. Programmierung der Funktionalität gegen eine Daten-Schnittstelle bzw. einen Daten-Kontrakt. (Verwendung von .NET Interfaces)

3. Eliminierung aller Abhängigkeiten zu sämtlichen „Nicht SAP Funktionalitäten“, wie Web-, User- und Kunden-Kontexte.



## IoC - “Inversion of control”

Für punktgenaues Testen ist die Isolation von Klassen zwingende Voraussetzung.   
Isolation entsteht, wenn Klassen keine Abhängigkeiten von Implementationen mehr enthalten – weder zur Laufzeit, noch zur Übersetzungszeit. Konkrete Abhängigkeiten sollten deshalb so spät wie möglich entschieden werden. Am besten zur Laufzeit.  
Genau hier kommt IoC „Inversion of Control“ ins Spiel:  
Statt dass die Anwendung den Kontrollfluss steuert, wird die Steuerung der Ausführung bestimmter Unterprogramme an das Framework abgegeben.

Warum nun das Ganze?  
Gegenfrage: Wie würden wir eine Großstadt bauen?

Könnten wir alle Details einer Stadt selber managen?   
Sehr wahrscheinlich nicht.  
Das Verwalten und das Erweitern einer Stadt ist halt zuviel für 1 Person.  
Trotzdem - Städte funktionieren (meistens zumindest).  
Sie funktionieren, weil Städte Personal in Teams beschäftigen die bestimmte Teile selbstständig und unabhängig voneinander betreuen, wie die Versorgung mit Elektrizität, Wasser, Straßenbau, Telefoninfrastruktur, Computervernetzung, Internet, etc.

Städte funktionieren auch, weil sie angemessene Abstraktionsebenen und Modularität hervorgebracht haben, die es dem Personal und den „Komponenten“ die es betreut ermöglicht, effektiver zu arbeiten auch ohne das „Große Ganze“ zu verstehen.

***Obwohl Software Teams ähnlich wie die Teams arbeiten die unsere Städte in Betrieb halten, wurden Software Systeme leider oft genug ohne dieselben Abstraktionsebenen und ohne ausreichende Modularität entwickelt.***

***Clean Code hilft uns Software Archtitekten und Entwicklern genau eben das zu erreichen!***

Beachtet man, auch bei “Mini” Entwicklungen, diese Entwicklungspraktiken, spart man sich als Entwickler sehr viele Nachbesserungs- und Anpassungsarbeiten.

Die Idee hinter „Inversion of Control“, auch unter „Dependency Injection“ bekannt, ist die Anwendung des sogenannten “Hollywood Prinzips” => “Don’t call us, we call you!”

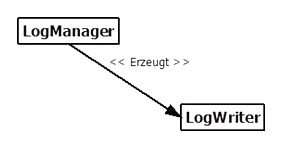
Was bedeutet das für die Software Entwicklung?   
In einem klassischen Ansatz werden Klassen und deren Abhängigkeiten hard-codiert. Das heißt es wird mit konkreten Klassen gearbeitet, welche über Konstruktoren oder Properties Abhängigkeiten erhalten. Somit wird schon zur Kompilierzeit für jedes konkrete Objekt festgelegt, welche Interaktionen mit anderen Objekten bestehen. Einfach gesagt, werden alle Objekte fix miteinander “verdrahtet”. Änderungen sind nicht mehr möglich (ausser durch Umbauarbeiten im Sourcecode), ein dynamisches Beeinflussen von Interaktionen ist gar nicht möglich.

Abbildung 1:   
LogManager ist direkt abhängig von einer konkreten Implementierung von LogWriter

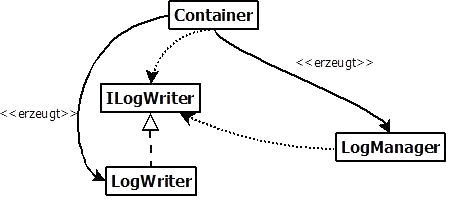


Abbildung 2: Der Container erzeugt Instanzen und Injiziert Abhängigkeiten

Besser ist eine genaue Umkehrung dieser Vorgehensweise.   
Also eine Komponente (Container in Abbildung 2), die Objekte erzeugt und selbst zur Laufzeit entscheidet, wie die einzelnen Abhängigkeiten “befriedigt” werden und welche konkreten Implementierungen zu verwenden sind, übernimmt die Kontrolle; daher => „Inversion of Control“.   
Die Verwendung diese Prinzips führt implizit schon zu besserer Software, da man mit Abstraktionen wie Interfaces und Abstrakten Klassen arbeiten muss um dieses dynamische Verhalten zu erreichen.

**Beispiel**

Hier ein Beispiel aus einer bestehenden Anwendung:

public class UeberfuehrungController : CkgDomainController

{

#if NO\_SAP

private IUeberfuehrungDataService UeberfuehrungDataService   
 { get { return new **UeberfuehrungDataServiceTest**(AppSettings, LogonContext); } }

#else

private IUeberfuehrungDataService UeberfuehrungDataService   
 { get { return new **UeberfuehrungDataServiceSAP**(S.AP, AppSettings, LogonContext); } }

#endif

protected override ILogonContext CreateLogonContext()

{

#if NO\_SAP

return new **LogonContextTest**();

#else

return new **LogonContextDataServiceSqlDatabaseAutohaus**();

#endif

}

[…]

}

Hier derselbe Code Bereich nach erfolgtem „IoC“ konformen Umbau:

public class UeberfuehrungController : CkgDomainController

{

public UeberfuehrungController(**ILogonContextDataServiceAutohaus** logonContext,   
 **IUeberfuehrungDataService** ueberfuehrungDataService)   
 : base(logonContext)

{

[…]

}  
  
 […]  
}