**Daily-Scrum Meeting (Scrum Meeting)**

Wie der Name andeutet, findet das Daily Scrum Meeting an jedem Arbeitstag statt. Es dauert nicht länger als 15 Minuten (Time-Box) und dient dem Team dazu, sich abzustimmen und gegenseitig zu informieren.

**Ablauf des Daily Scrum Meeting (kurz: Daily Scrum)**

* Kurzes, tägliches Status-Meeting des Teams
* Time-boxed (15 min)
* Stand-Up-Meeting, d.h. die aktiven Teilnehmer stehen (am besten im Kreis), sie sitzen nicht
* ScrumMaster nimmt teil, notiert sich genannte *Impediments* (s.u.) auf seiner *Blocks List* und greift moderierend(!) ein, wenn unbedingt nötig
* Product Owner nimmt nach Möglichkeit auch teil, um auf dem neuesten Stand zu bleiben und bei Bedarf Fragen zu beantworten
* Nur die Team Members sprechen und berichten einander jeweils Folgendes:  
  -  Was habe ich seit dem letzten Daily Scrum getan?  
  -  Was plane ich, bis zum nächsten Daily Scrum zu tun?  
  -  Was hat mich bei der Arbeit behindert *(Impediments)*?

**Hinweise**

* Ein fruchtbares Daily Scrum erzeugt durch seine Kürze bewusst weiteren Gesprächs- und Informationsbedarf, der in anschließenden Einzelgesprächen gedeckt werden kann - ein positiver Nebeneffekt der Time-Box.
* Es ist wichtig, dass die Team Members *einander* berichten und nicht ihren evtl. anwesenden Vorgesetzten oder dem ScrumMaster oder dem Product Owner. Sie schauen einander an, alle anderen Anwesenden sind nur Zuhörer. Das *Command-and-Control*-Muster zu durchbrechen, erfordert ein wenig Übung. Die Umstellung dauert eine Weile für neue Scrum-Teams. Innerhalb kurzer Zeit sind Daily Scrums jedoch hocheffiziente und zielgerichtete Veranstaltungen.
* Dass jedes Team Member weiß, woran die anderen arbeiten, ist wichtig, damit nicht jeder in seinem Elfenbeinturm bleibt, sondern ein Gesamtbild des Projekts erhält. Wer ein Problem hat, dem kann vielleicht ein anderer helfen.
* Dieser Hinweis gilt für alle Scrum-Meetings: Es wird pünktlich begonnen, nicht fünf Minuten später. Wer zu spät kommt, hat Pech gehabt und wird, falls das Team es so festgelegt hat, entsprechend einer team-internen Regelung sanktioniert, z.B. zwei Euro in eine Kasse zahlen, aus der am Ende des Projekts für einen wohltätigen Zweck gespendet wird. Das Geld hinterher als "Belohnung" fürs Zuspätkommen gemeinsam zu "verfressen", wäre kein gutes Signal.

**Sprint Backlog**

Eine Liste von → Tasks, welche den Arbeitsumfang des → Teams für den → Sprint festlegt. Die Liste präzisiert sich während des Sprints und wird täglich von allen → Team Members gepflegt, so daß sie immer den aktuellen Bearbeitungsstand reflektiert. Der Sprint Backlog ermöglicht es dem → *ScrumMaster*, jederzeit zu erkennen, wo das Team steht und ggf. steuernd einzugreifen, damit das → Sprint Goal nicht in Gefahr gerät.

**Product Backlog**

Priorisierte Liste von Anforderungen (→ Requirements) mit Schätzwerten (Estimates), welche den jeweiligen Funktionsumfang, ggf. auch ihre Komplexität, relativ zueinander widerspiegeln. Wichtig ist, dass es um eine "gefühlte Größe" geht, nicht um eine absolute Aufwandschätzung in Personentagen o.ä. Als Einheit der Schätzwerte wird oft der sog. → Story Point - nicht zu verwechseln mit einem Function Point - verwendet.

Je höher die Priorität einer Anforderung, desto feingranularer sind tendenziell die Schätzwerte, da das Product Backlog im Wesentlichen nach absteigender Priorität abgearbeitet wird. Das Product Backlog ändert sich im Lauf der Zeit, ist also keine endgültige, vorab festgelegte Spezifikation, sondern eine Anforderungsliste, die im Fluss ist und sich den Gegebenheiten des Projekts kontinuierlich anpasst : In einem → Sprint realisierte oder verworfene Requirements werden entfernt, Schätzwerte aufgrund Teilbearbeitung oder neuer Erkenntnisse aktualisiert, neue Anforderungen aufgenommen, Prioritäten verändert, grob geschätzte Themen oder Funktionsblöcke verfeinert, also in kleinere heruntergebrochen etc.

**Burndown Chart**

Graphik, die den Projektfortschritt eines Produktes, → Sprints oder → Releases in einer Kurve visualisiert. Die Kurve gibt für jeden Punkt auf dem horizontalen Zeitstrahl an, wieviel Arbeit nach jeweils aktueller Schätzung zu jedem Zeitpunkt noch übrig war/ist, um das Ziel zu erreichen. Mittels einer auf den vergangenen Schätzwerten basierenden Trendlinie durch die Kurve lässt sich das voraussichtliche zeitliche Ende des Projektes vorhersagen bzw. können sich andeutende Abweichungen vom Zeitplan innerhalb eines Sprints prognostiziert und Gegenmaßnahmen ergriffen werden.

**–––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––**

**Scrum-Rollen**

**Scrum-Rollen - Pigs & Chickens 🡪** In Scrum werden direkt am Prozeß beteiligte Personen Pigs (Schweine) genannt, Außenstehende heißen Chickens (Hühner). Das hat seinem Ursprung in einem englischen Witz.

**Scrum-Rollen - Product Owner 🡪**Immer, wenn Software entwickelt wird - egal ob im Kundenprojekt oder für ein Standardprodukt - gibt es einen Auftraggeber, der die fachliche Sicht vertritt, Anforderungen stellt und die spätere Umsetzung seiner Wünsche im Hinblick auf Funktionalität, Benutzbarkeit (Usability), Performanz und Qualität beurteilt.

**Scrum-Rollen – ScrumMaster 🡪** Agile Prozesse zeichnen sich - deshalb auch diese Bezeichnung - durch eine hohe Dynamik aus. Damit der Prozeß zielgerichtet verläuft und aus Dynamik nicht Chaos wird, gibt es den ScrumMaster. Er ist sozusagen die Seele des Prozesses und sorgt dafür, daß die Regeln eingehalten werden.

**Scrum-Rollen – Team 🡪** Das Scrum-Team, meistens kürzer einfach Team genannt, steht im Zentrum des Scrum-Prozesses, weil es für die Umsetzung der Anforderungen in Produktfunktionalität sorgt. Das Team insgesamt ist einer von drei Managern in Scrum.

**Windows Presentation Foundation** (kurz WPF), auch bekannt unter dem Codenamen Avalon, ist ein Grafik-Framework und Teil des .NET Frameworks von Microsoft, das mit Windows Vista, Windows 7 und Windows 8 ausgeliefert wird, sich aber auf Windows XP (bis zur Version 4.0) und Server 2003 nachinstallieren lässt.

Bei WPF handelt es sich „um eine 2006 gänzlich neu eingeführte Bibliothek von Klassen, die zur Gestaltung von Oberflächen und zur Integration von Multimedia-Komponenten und Animationen dient. Sie vereint die Vorteile von DirectX, Windows Forms, Adobe Flash, HTML und CSS“ [1]

WPF stellt ein umfangreiches Modell für den Programmierer bereit. Dabei werden die Präsentation und die Geschäftslogik getrennt, dies wird vor allem durch die Auszeichnungssprache XAML (basierend auf XML) unterstützt. XAML beschreibt Oberflächen-Hierarchien deklarativ als XML-Code. WPF-Anwendungen können sowohl Desktop- als auch Web-Anwendungen sein und benutzen, wenn möglich, auch Hardwarebeschleunigung. Das Framework versucht, die verschiedenen Bereiche, die für die Präsentation wichtig sind (Benutzerschnittstelle, Zeichnen und Grafiken, Audio und Video, Dokumente, Typographie), zu vereinen.

Zudem gibt es mit Microsoft Silverlight (ehemals WPF/E) eine Plattform, die explizit für das Web entwickelt wurde. Silverlight setzt dabei nicht auf das .NET-Framework auf, sondern benutzt eine reduzierte Version dessen. Silverlight kann auch mit XAML entwickelt werden.

**–––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––**

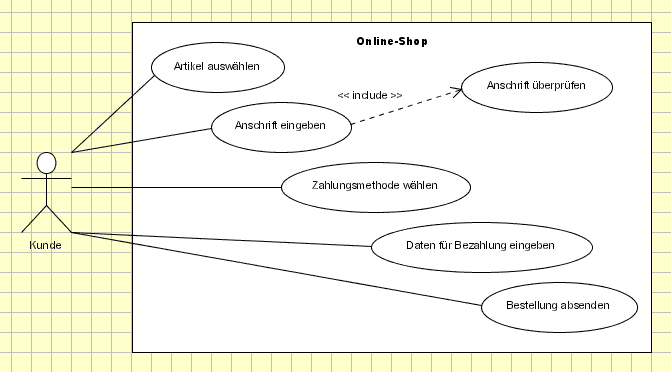
**MVC: Model View Controller**

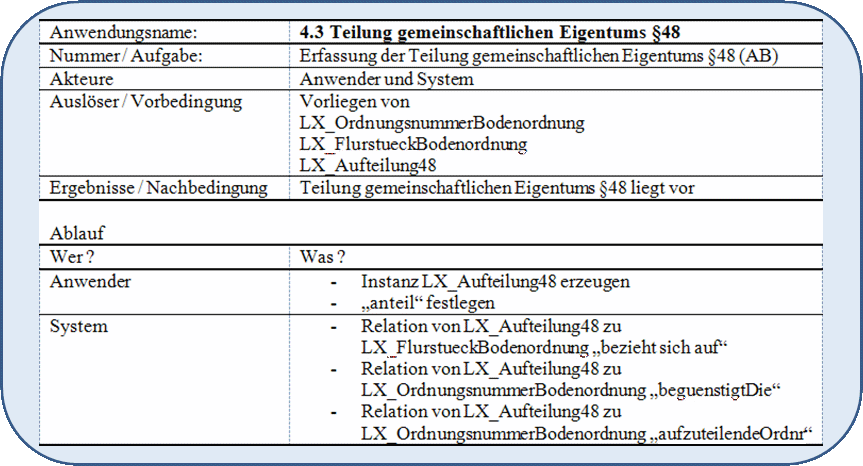
Der englischsprachige Begriff model view controller (MVC, englisch für Modell-Präsentation-Steuerung) ist ein Muster zur Strukturierung von Software-Entwicklung in die drei Einheiten Datenmodell (engl. model), Präsentation (engl. view) und Programmsteuerung (engl. controller). Manche Autoren stufen es als Architekturmuster ein, andere als Entwurfsmuster.[1] Ziel des Musters ist ein flexibler Programmentwurf, der eine spätere Änderung oder Erweiterung erleichtert und eine Wiederverwendbarkeit der einzelnen Komponenten ermöglicht. Es ist dann zum Beispiel möglich, eine Anwendung zu schreiben, die dasselbe Modell nutzt und es dann für Windows, Mac, Linux oder für das Internet zugänglich macht. Die Umsetzungen nutzen dasselbe Modell, nur Controller und View müssen dabei jeweils neu implementiert werden.

**–––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––**

**Modell (model) 🡪** Das Modell enthält die darzustellenden Daten (und in manchen Umsetzungen des MVC-Musters auch die Geschäftslogik). Es ist von Präsentation und Steuerung unabhängig. Die Bekanntgabe von Änderungen an relevanten Daten im Modell geschieht nach dem Entwurfsmuster „Beobachter“. Das Modell ist das zu beobachtende Subjekt, auch Publisher, also „Veröffentlicher“, genannt.

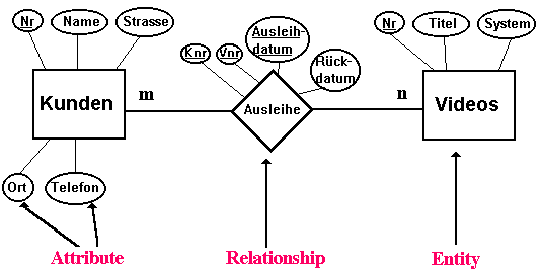
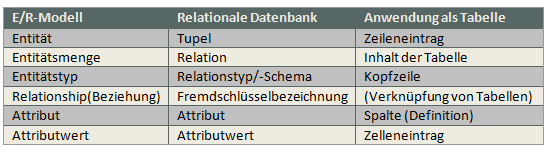
**Präsentation (view) 🡪** Die Präsentationsschicht ist für die Darstellung der benötigten Daten aus dem Modell und die Entgegennahme von Benutzerinteraktionen zuständig. Sie kennt sowohl ihre Steuerung als auch das Modell, dessen Daten sie präsentiert, ist aber nicht für die Weiterverarbeitung der vom Benutzer übergebenen Daten zuständig. Im Regelfall wird die Präsentation über Änderungen von Daten im Modell mithilfe des Entwurfsmusters „Beobachter“ unterrichtet und kann daraufhin die aktualisierten Daten abrufen. Die Präsentation verwendet oft das Entwurfsmuster „Kompositum“.

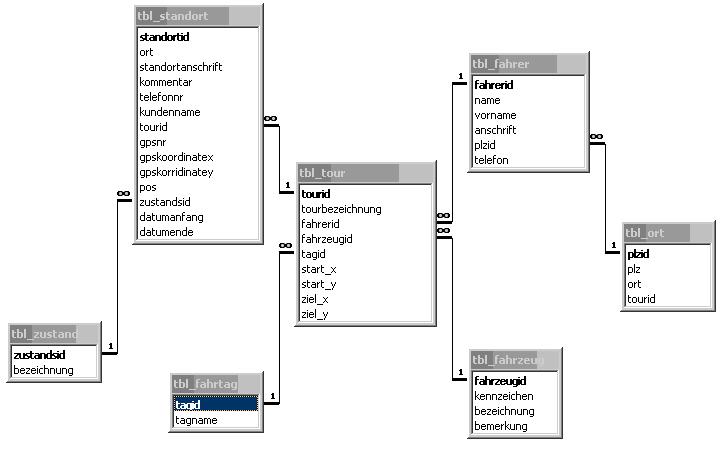
**Steuerung (controller) 🡪** Die Steuerung verwaltet eine oder mehrere Präsentationen, nimmt von ihnen Benutzeraktionen entgegen, wertet diese aus und agiert entsprechend. Zu jeder Präsentation existiert eine eigene Steuerung. Die Steuerung sorgt dafür, dass Benutzeraktionen wirksam werden, z. B. durch Änderung der Präsentation (z. B. Verschieben des Fensters) oder durch Weiterleiten an das Modell (z. B. Übernahme von Eingabedaten oder Auslösen von Verarbeitungen). Als es noch keine Objektorientierung gab, bestand ein Modell nur aus Daten, und die Steuerung hat die Daten oft direkt aktualisiert. In einer objektorientierten Umgebung ist es dagegen besser, wenn das Modell die Geschäftsobjekte enthält und die Steuerung sich darauf beschränkt, Benutzereingaben (Daten und Methodenaufrufe) weiterzuleiten, von der Präsentation an das Modell. Die Steuerung enthält weiterhin Mechanismen, um die Benutzerinteraktionen der Präsentation einzuschränken. Die Steuerung kann in manchen Implementierungen ebenfalls zu einem „Beobachter“ des Modells werden, um bei Änderungen der Daten die Präsentation direkt zu manipulieren.



|  |  |
| --- | --- |
| Innovator Help | **System** Das Rechteck stellt das geplante System dar. Der Name gibt den Namen des Systems an. Ein Use Case Diagramm kann auch mehrere Systeme enthalten. Dadurch kann ein System in Teilsysteme gegliedert werden. |
| https://www.fbi.h-da.de/fileadmin/Labor/case/uml/useCase/UC_useCase.gif | **Use Case (Anwendungsfall)**  Eine Ellipse stellt einen Anwendungsfall des Systems dar. Ein Anwendungsfall ist ein in sich abgeschlossener Vorgang, der für einen oder mehrere Akteure ein beobachtbares Ergebnis liefert. Er beschreibt aus Sicht der Akteure welche Leistungen das System für den Anwender zur Verfügung stellt. Ein Use Case stellt somit einen Teil der Gesamtfunktionalität des Systems dar. In UML 2.0 kann auch ein Rechteck, das mit einer Ellipse markiert wird, als Use-Case-Symbol verwendet werden. Der Name kann innerhalb oder außerhalb des Symbols stehen. |
| https://www.fbi.h-da.de/fileadmin/Labor/case/uml/useCase/UC_akteur.gif | **Akteur**  Ein Akteur ist ein Element, das nicht zum geplanten System ge­hört. Er kann eine Person sein, die auf das System zugreift, oder ein anderes System, das mit dem geplanten System kommuniziert. Die UML erlaubt mehrere Symbole zur Darstellung eines Ak­teurs. Er kann als Strichmännchen dargestellt werden. Es ist optio­nal erlaubt ein Klassensymbol zu verwenden, das mit dem Stereotyp <<actor>> markiert wird. Zusätzlich können eigene Symbole verwendet werden um nicht menschliche Akteure darzu­stellen. |
| https://www.fbi.h-da.de/fileadmin/Labor/case/uml/useCase/UC_beziehung.gif | **Assoziation**  Eine Linie stellt eine Assoziation zwischen einem Akteur und einem Use Case dar. Sie beschreibt den Zugriff des Akteurs auf die Funktionalität, die das System in diesem Use Case zur Verfü­gung stellt, bzw. eine Antwort des Systems an einen Akteur. |
| https://www.fbi.h-da.de/fileadmin/Labor/case/uml/useCase/UC_include.gif | **include-Assoziation** Bei der **include** Beziehung verwendet ein Use Case die Funktionalität, die ein anderer Use Case zur Verfügung stellt. Der includierte Use Case wird **immer** ausgeführt.  Der Use Case a importiert die Funktionalität des Use Case b. |
| extend | **extend-Assoziation** Die **extend** Beziehung beschreibt die Erweiterung der Funktionalität eines Use Cases durch einen anderen Use Case. Man kann dadurch **optionales** Verhalten beschreiben, bzw. Funktionen modellieren, die nur unter bestimmten Bedingungen ausgeführt werden. |

**E/R-Model (ERM)**

**Datenbankmodell**



**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**EF LINQ**

Der nachfolgende Codeausschnitt zeigt die Daten, welche anschliessend in allen Beispielen verwendet werden.

    1 private List<Customer> \_customers;

    2 private List<Order> \_orders;

    1 \_customers = new List<Customer>()

    2 {

    3     new Customer() { Identifier = 1, FirstName = “Hans”, LastName = “Meier” },

    4     new Customer() { Identifier = 2, FirstName = “Uwe”, LastName = “Mueller” },

    5     new Customer() { Identifier = 3, FirstName = “Sandra”, LastName = “Schmidt” }

    6 };

    7

    8 \_orders = new List<Order>()

    9 {

   10     new Order() { Identifier = 101, CustomerIdentifier = 2, TotalPrice = 750 },

   11     new Order() { Identifier = 102, CustomerIdentifier = 2, TotalPrice = 80 },

   12     new Order() { Identifier = 103, CustomerIdentifier = 2, TotalPrice = 120 },

   13     new Order() { Identifier = 104, CustomerIdentifier = 3, TotalPrice = 55 }

   14 };

Es werden zwei Listen erzeugt. Eine Kundenliste und eine Liste mit Bestellungen. In den Bestellungen wird jeweils die Kundennummer gespeichert. So lassen sich Verknüpfungen zwischen den beiden Listen erzeugen.

**Einfache Datenabfragen**

Nun möchte Ich sie nicht länger auf die Folter spannen und Ihnen die erste LINQ Abfrage zeigen. Es sollen alle Bestellungen abgefragt werden, welche Herr Müller getätigt hat. Dabei wird direkt die Identifikationsnummer des Kunden genutzt, also noch keine Verbindung zwischen den beiden Listen hergestellt.

    1 var results = from order in \_orders

    2               where order.CustomerIdentifier == 2

Die Anwendung erzeugt folgende Ausgabe:  
750  
80  
120

    3               select order.TotalPrice;

    4

    5 foreach (var result in results)

    6 {

    7     Console.WriteLine(result);

    8 }

Die deklarative Syntax in LINQ ist eigentlich nur syntaktischer Zuckerguss um den Softwareentwickler die Arbeit etwas zu versüssen. Im Hintergrund wird die Abfrage in funktionalen Code umgewandelt. Statt der deklarativen Syntax kann die Abfrage daher auch mittels Funktionen geschrieben werden. Der nachfolgende Codeausschnitt zeigt die gleiche Abfrage wie zuvor, aber diesmal in funktionaler Schreibweise.

    1 var results = \_orders.Where(order => order.CustomerIdentifier == 2)

    2                      .Select(order => order.TotalPrice);

Wie anhand der beiden Beispiele ersichtlich ist, dürfen Sie in LINQ sowohl deklarativ als auch funktional programmieren. Die deklarative Syntax ist aber klar vorzuziehen da sie einem besser lesbaren Quellcode resultiert. Eine abgeschlossene deklarative Anweisung kann aber mit funktionalem Code erweitert werden. Das nachfolgende Beispiel zeigt solch eine gemischte Schreibweise.

    1 var results = (from order in \_orders

    2                where order.CustomerIdentifier == 2

    3                select order.TotalPrice)

    4                .ToArray();

**Verknüpfen von Datenmengen mittels *Join***

Die beiden Datenlisten wurden bisher ausschliesslich getrennt voneinander betrachtet. Die Liste mit Bestellungen enthält aber die Kundennummer. Damit lässt sich eine Verbindung zwischen beiden Datenmengen herstellen. Betrachten wir dazu folgendes Beispiel: Es sollen alle Preise der Bestellungen und der jeweils zugehörende Kunde aufgelistet werden. Der nachfolgende Quellcode zeigt eine mögliche Implementierung.

    1 var results = from customer in \_customers

    2               join order in \_orders on

    3                 customer.Identifier equals order.CustomerIdentifier

Ausgabe:  
Mueller, Uwe: 750  
Mueller, Uwe: 80  
Mueller, Uwe: 120  
Schmidt, Sandra: 55

    4               select new

    5               {

    6                   Name = customer.LastName + “, “ + customer.FirstName,

    7                   order.TotalPrice

    8               };

    9

   10 foreach (var result in results)

   11 {

   12     Console.WriteLine(result.Name + “: “ + result.TotalPrice);

   13 }

In Zeile 2 wird mittels *join/in/on/equals* angegeben wie die beiden Listen verknüpft werden sollen. In diesem Fall durch erfolgt die Verknüpfung mittels der beiden Identifier Werte. Anschliessend werden die Ergebnisse in ein neues anonymes Objekt selektiert.

**Sortierung des Abfrageergebnisses**

Genau wie bei SQL stellt sich auch bei LINQ die Frage wie die Ergebnisse sortiert sind. Wenn Sie keine Sortierung vorgeben dann sind die Ergebnisse in beiden Fällen zufällig sortiert. Wenn Sie die Ergebnisse in einer bestimmten Reihenfolge erwarten, dann müssen Sie dies in der LINQ Anweisung angeben. Dazu dient der *orderby* Befehl. Im nachfolgenden Beispiel werden die Daten zuerst mit aufsteigendem und danach mit absteigendem Preis sortiert.

    1 var resultsAsc = from order in \_orders

    2                  where order.CustomerIdentifier == 2

    3                  orderby order.TotalPrice

    4                  select order;

    5

    6 var resultsDesc = from order in \_orders

    7                   where order.CustomerIdentifier == 2

    8                   orderby order.TotalPrice descending

    9                   select order;

**Gruppierung**

Analog zu SQL lassen sich auch in LINQ die Ergebnisse von Datenabfragen zu Gruppen zusammenfassen. Betrachten wir dazu die Liste mit Bestellungen. Ein typischer Anwendungsfall könnte lauten: Ermittle wie viele Bestellung jeder Kunde getätigt hat. Der nachfolgende Quellcode zeigt die entsprechende LINQ Anweisung bei welcher die Schlüsselwörter *group/by/into* eingesetzt werden um die Ergebnisse zu Gruppieren.

    1 var results = from order in \_orders

    2               group order by order.CustomerIdentifier into g

Die Anwendung erzeugt folgende Ausgabe:

2: 3

3: 1

    3               select new { Identifier = g.Key, Count = g.Count() };

    4

    5 foreach (var result in results)

    6 {

    7     Console.WriteLine(result.Identifier + “: “ + result.Count);

    8 }