einfach – OOP – einfach

Reden wir über Konzepte.

Ich spreche in diesem Vortrag hauptsächlich über das **Was** und das **Warum** der objektorientierten Programmierung.
Das **Wie**, also die Sprachelemente von z. B. LotusScript und die genaue Syntax, ist leicht nachzuschlagen, wenn man die grundlegenden Konzepte verinnerlicht hat.

Bild: http://www.sxc.hu/photo/998467

Thomas Bahn, 37
Diplom-Mathematiker, Universität Hannover
Mitbegründer der assono GmbH; wir sind ein IT-Dienstleister mit Hauptsitz Kiel und einem weiteren Standort bei Hamburg. Wir bieten unseren Kunden IT-Beratung, Entwicklung, Administration(sunterstützung), Schulungen an und Handeln mit Lizenzen.
Unser klarer Schwerpunkt ist IBM Lotus Notes/Domino.
Ich selbst entwickle in Java und relationalen DBMS seit 1997.
1999 bin ich dann zu Lotus Notes/Domino gekommen.
Einen großen Teil meiner Zeit entwickle ich (individuelle) Anwendungen, daneben mache ich aber auch viel Administrationsunterstützung, Beratung und hin und wieder Schulungen und Workshops.
Mein Fokus liegt dabei auf leicht zu verwendenden Notes-Client-Anwendungen, Schnittstellen zu anderen Systemen (wie z. B. RDBMS, SAP R/3) und Domino-Web-Anwendungen.
Ich bin regelmäßiger Sprecher auf nationalen und internationalen Fachkonferenzen zu Lotus Notes/Domino und schreibe gerade meinen vierten Artikel für THE VIEW.
IBM Certified Advanced Application Developer and System Administrator – Lotus Notes and Domino (R4 – 7),
IBM Certified Solutions Designer – CommonStore Email Archiving and Discovery,
Oracle Certified Professional

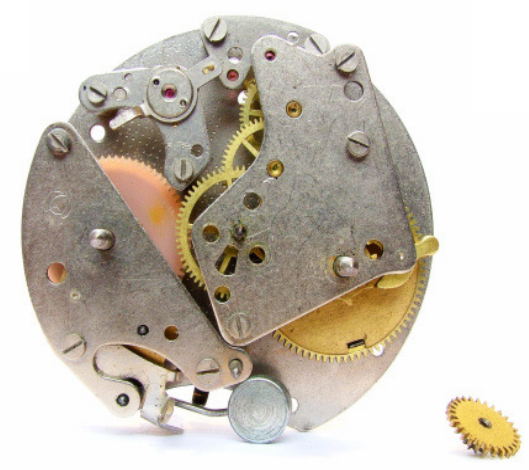
Jeder kann eine Uhr benutzen.

Natürlich.
Eine Taschenuhr hat eine sehr einfache “Benutzerschnittstelle”:

- Die Zeiger zeigen durch ihre Winkelstellung die Zeit an.
- Mit der Krone kann man die Uhr aufziehen und die Zeit einstellen. Man kann sie drehen und etwas aus der Uhr herausziehen.

Bild: http://www.sxc.hu/photo/890003

Der Hersteller muss wissen, wie man eine Uhr baut.



Der Hersteller muss detailliert wissen, wie eine Uhr im Innersten funktioniert, er kennt die Federn, Zahnräder usw. und ihre jeweilige Funktion genau.

Bild: http://www.sxc.hu/photo/839239

Vor dem Benutzer ist die Komplexität des inneren Aufbaus und der Abläufe in der Uhr aber verborgen. Und er braucht gar nicht zu wissen, **wie** die Uhr funktioniert.
Die Mechanik (oder Elektronik) ist vor ihm durch das Gehäuse versteckt (und damit auch vor unbedachten "Manipulationen" geschützt).
Alles was er kennen muss, ist die "Benutzerschnittstelle". Er muss die Zeiger ablesen und die Krone bedienen können.

In der objektorientierten Programmierung werden die versteckten, inneren Attribute und Methoden **privat** genannt, die äußeren **öffentlich**.

Bild: http://www.sxc.hu/photo/696748

Die privaten Elemente sind versteckt in einer Art und Weise, wie es sie in der normalen strukturierten Programmierung nicht gibt.

Allerdings können auch in LotusScript private Variablen und Routinen in Script-Bibliotheken auf ähnliche Art und Weise versteckt werden.

Bild: http://www.sxc.hu/photo/982557

Alles ist ein Objekt.
Alles besteht aus Objekten.

Zumindest alles, was wir sehen oder berühren können.
Sogar Menschen sind – in einem abstrakten Sinn – auch Objekte.

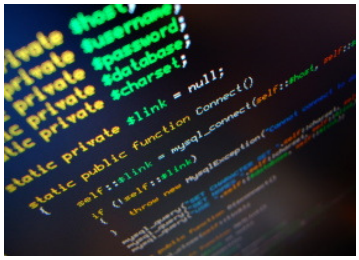
Bild: http://www.sxc.hu/photo/949574

Interagieren meine ich hier in einem sehr weiten Sinn. Auch wenn ein Blatt Papier “nur” auf einem Tisch liegt, kann man das als eine Art Interaktion verstehen. Eine sehr passive Interaktion. :-)
In vielen Fällen aber ist die Interaktion “aktiver”.

Wenn parallel existierende Objekte interagieren, beeinflussen und ändern sie den Zustand anderer Objekte.

Bild: http://www.sxc.hu/photo/748081

OOP bildet Objekte der realen Welt ab auf Software-Objekte.



Aber nicht nur Dinge, wie Häuser oder Fische, sind Objekte; auch Abläufe, wie das Geldabheben von einem Bankkonto, oder Beziehungen zwischen Objekten, z. B. dass ein Konto einem Kunden zugeordnet ist, können wieder als Objekte betrachtet werden.

In der objektorientierten Software-Entwicklung baut man Software-Objekte, die den echten Objekten in bestimmten Punkten “ähneln”; man entwickelt Modelle der realen Welt.

Beim Programmieren startet man häufig damit, dass man die realen Objekte identifiziert, die modelliert werden müssen.

Aber bei OOP in Notes sind die meisten Klassen, die man schreibt, wesentlich abstrakter. Es sind häufig nur Hilfskonstrukte oder Werkzeuge, die nicht wirklich für Objekte der realen Welt stehen.

Bilder: http://www.sxc.hu/photo/999736, http://www.sxc.hu/photo/995000

Konzept: Kapselung (encapsulation).

Objekt = Zustand + Verhalten + Identität

In der strukturierten Programmierung gibt es getrennte Hierarchien für Daten und Routinen.
Neben den einfachen Datentypen gibt es meist auch zusammengesetzte, wie Arrays und Structs, die wiederum andere Datentypen beinhalten.
Außer den einfachsten, rufen Routinen normalerweise andere auf. Es gibt eine Art Netz von gegenseitigen Aufrufen.

Aber die (globale) Datenhierarchie und das Netz der Routinen existieren parallel und unabhängig voneinander (außer natürlich lokale Variablen und Aufrufparameter).
Aber es gibt ansonsten keine “Eigentümerschaft”.

In der OOP sind Objekte wie Kapseln. An die inneren Elemente, also privaten Attribute und Methoden, kommt niemand von außen ran. Nur die Methoden des Objekts selbst können den Wert seiner privaten Attribute und damit seinen inneren Zustand ändern.

Zustand des Objekts = Wert der Attribute, das sind seine Variablen
Verhalten des Objekts = seine Methoden, also Prozeduren und Funktionen

Identität meint hier, dass man zwei Objekte grundsätzlich immer unterscheiden können. In der realen Welt können sie niemals exakt den gleichen Platz einnehmen, und selbst wenn sie sich ähneln wie ein Ei dem anderen, sind es doch zwei unterschiedliche Objekte.
Denke zum Beispiel an zwei Uhren desselben Modells, die gleich aufgezogen sind und die gleiche Zeit anzeigen.
Auch zwei Software-Objekte, selbst wie sie gleichartig sind (von der gleichen Klasse) und den gleichen Zustand haben – die Werte der Attribute sind gleich – so sind es doch zwei unterschiedliche Objekte.

Bild: http://www.sxc.hu/photo/1006559

„Betrachte ein Objekt wie eine besondere Variable; es speichert Daten, aber zusätzlich kannst du es auffordern, Operationen auf sich selbst auszuführen.“

Bruce Eckel, Thinking in Java

Zitat aus der dritten Auflage, Seite 33:

„Think of an object as a fancy variable; it stores data, but you can „make requests“ to that object, asking it to do operations on itself.“

Modelle eines Bürostuhls

Ein Objekt der realen Welt hat sehr viele verschiedene Eigenschaften und Fähigkeiten. Bei der Abbildung in ein Software-Objekt muss man die Anzahl dieser Elemente auf das für den aktuellen Zweck Notwendige reduzieren. Man bildet ein (eingeschränktes) Modell des wirklichen Objekts.

Beispiel: Bürostuhl

Wenn der Hersteller über einen Bürostuhl nachdenkt (z. B. für seinen Produktkatalog), dann denkt er vielleicht an Höhe, Gewicht, Materialien, Farbvarianten, Einstellmöglichkeiten usw.

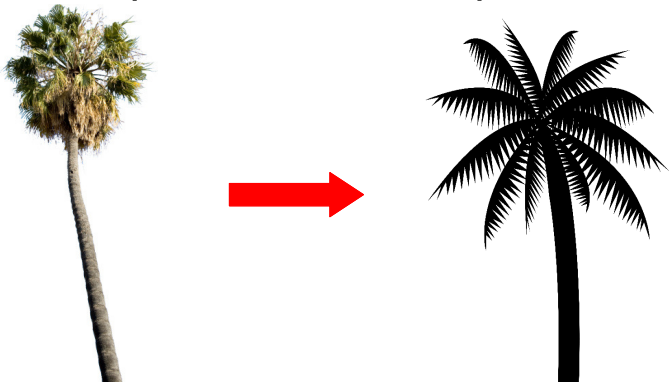
Wenn ein Buchhalter (eines Kunden) über einen Bürostuhl nachdenkt, dann denkt er vielleicht an den bezahlten Preis, das Kaufdatum, das Lieferdatum, voraussichtliche Nutzungsdauer, aktueller Buchwert, Inventarnummer usw.

Wenn eine Umzugsfirma über einen Bürostuhl nachdenkt, dann denkt er vielleicht an seine Abmessungen, wie viel Platz der Stuhl im Lastwagen benötigt, ob man ihn auseinander nehmen kann, von welcher Lokation (Adresse, Gebäude, Etage, Raum) zu welcher anderen Lokation er transportiert werden muss usw.

Es gibt nicht das eine Modell eines Objekts der realen Welt, sondern viele verschiedene. Welches davon in Software „gegossen“ werden muss, hängt ab vom Kontext und dem Verwendungszweck.

Bild: http://www.sxc.hu/photo/588853

Konzept: Abstraktion (abstraction).



Reduziere Komplexität durch das Ignorieren unnötiger Details.

In einem bestimmten Kontext und für einen bestimmten Verwendungszweck sind nur einige der vielen Merkmale eines Objekts notwendig.

Um Komplexität zu reduzieren, sollten Modelle nur diese Details enthalten, die für den beabsichtigten Zweck wichtig sind.

Bild: http://www.sxc.hu/photo/996245, http://www.sxc.hu/photo/999574

Nur durch diese Konzentration auf das Wesentliche, die Abstraktion, werden Objekte der realen Welt überhaupt in Software abbildbar.

Bild: http://www.sxc.hu/photo/974531

Konzept: Klasse (class).

Eine Klasse ist wie ein Stempel oder ein Bauplan, um Objekte zu bauen.

Jede einzelne Uhr ist ein Objekt. Der Hersteller hat die Konstruktionspläne, um diese einzelnen Objekte zu erstellen. Die Pläne beschreiben die Einzelteile, die Arbeitsschritte und bestimmen damit auch die Eigenschaften des Endprodukts, also die möglichen Zustände (Attribute) und das Verhalten (Methoden).

Alle Objekte, die nach den gleichen Plänen gebaut werden, haben dieselben Eigenschaften. Das gleiche gilt für Software-Objekte, die von der gleichen Klasse erzeugt werden.

In der OOP wird das Erstellen von Objekten „Instanziieren“ genannt. Objekte heißen auch Instanzen der Klasse.

Bild: http://www.sxc.hu/photo/770543

Teile und herrsche: Entwirf Klasse für Klasse

Objektorientierte Programmierung besteht mindestens aus den folgenden zwei Schritten:

- Identifiziere die benötigten Objekte und deren Eigenschaften und definiere davon ausgehend die benötigten Klassen
- Implementiere jede Klasse – eine nach der anderen

Dies ist das klassische „Teile und herrsche“-Prinzip (divide et impera).

Bild: http://www.sxc.hu/photo/443042

Indem man die Software-Entwicklung so aufteilt, kann sich der Programmierer im ersten Schritt auf die Gesamtsicht des Systems konzentrieren:

- Welche Teile gibt es?
- Welche Interaktionen?

Im zweiten Schritt konzentriert er sich dann bei der Implementierung einer Klasse auf die Details, auf die internen Zustände und Abläufe.

Bild: http://www.sxc.hu/photo/343423

Alle Fahrzeuge haben bestimmte gemeinsame Eigenschaften und Fähigkeiten:
Alle haben Bremsen zum langsamer werden, ein Gaspedal zu beschleunigen, ein Steuerrad und einige Anzeigeinstrumente wie z. B. ein Tachometer. Alle haben eine aktuelle Geschwindigkeit, ein Gewicht, usw.
Es gibt unterschiedliche Arten von Fahrzeugen, wie Lastkraftwagen oder Rennwagen.
Auch diese haben die oben genannten Eigenschaften und Fähigkeiten, aber zusätzlich weitere: Ein Lastkraftwagen z. B. hat die Fähigkeit, große Lasten zu transportieren, und damit eine Ladekapazität, die Art der Ladung (Stückgut, Flüssigkeiten, Streugut usw.).

In der OOP kann man eine Klasse bauen für Fahrzeuge, in der die gemeinsamen Attribute und Methoden modelliert sind. Dann kann man eine Klasse für Lastkraftwagen erstellen und definieren: ein Lastkraftwagen **ist ein** Fahrzeug und hat deshalb alle Attribute und Methoden des Kraftfahrzeugs. Dann kann man noch weitere Attribute und Methoden hinzufügen, die nur für Lastkraftwagen gelten.

Dies wird Vererbung genannt. In diesem Fall ist Lastkraftwagen eine Unterklasse von Fahrzeug, oder andersherum ist Fahrzeug die Oberklasse von Lastkraftwagen.
Man kann natürlich mehrere Unterklassen von einer Klasse ableiten (Rennwagen, Panzer, ...) und eine Unterklasse kann gleichzeitig auch wieder Oberklasse für weitere Klassen sein (Tanklastwagen, PKW-Transporter, etc.)

Eine Variable vom Typ Fahrzeug kann Fahrzeug-Objekte beinhalten (klar), aber genauso auch Lastkraftwagen-, Rennwagen- oder Tanklastwagen-Objekte (oder Objekte jeder anderen Unterklasse von Fahrzeug).

Bild: http://www.sxc.hu/photo/396573

Durch Vererbung kann man sehr einfach Code wieder verwenden. Nur dadurch, dass man Lastkraftwagen zur Unterklasse von Fahrzeug macht, erbt es den kompletten Code von Fahrzeug: alle Variablen, Prozeduren, Funktionen!

In der ersten Demonstration werde ich eine Klasse Watch und ihre Unterklasse WatchWithSeconds zeigen.

Bild: http://www.sxc.hu/photo/789140

Konzept: Polymorphie (polymorphism).

Unterklassen können neue, weitere Eigenschaften und Verhalten (Methoden) haben. Aber sie können auch das Verhalten, das sie von ihrer Oberklasse geerbt haben, ändern. Ein Rennwagen kann z. B. ein anderes Beschleunigungsverhalten haben als ein normales Fahrzeug.

Wenn zwei Klassen wie Fahrzeug (Oberklasse) und Rennwagen (Unterklasse) jeweils eine Methode mit dem gleichen Namen (und der gleichen Signatur = Rückgabetyp und Parametertypen) haben – wie z. B. beschleunige(), und man ruft diese Methode auf einem Objekt auf, dann muss zur Laufzeit entschieden werden, welche Methode tatsächlich ausgeführt wird: Ist das Objekt ein Rennwagen, dann wird die Methode der Rennwagen-Klasse ausgeführt, ist es ein anderes Fahrzeug, oder ein Objekt einer anderen Unterklasse, wird die Methode in der Fahrzeug-Klasse ausgeführt.

Bild: http://www.sxc.hu/photo/297733

Indem man der Unterklasse eine Methode gleichen Namens hinzufügt, wird die Methode der Oberklasse „überschrieben“. (Tatsächlich bleibt diese natürlich unverändert, sie wird aber für Objekte der Unterklasse „überdeckt“).

Ein Beispiel: Jemand hat eine Klasse Motor entwickelt. Diese wurde ausgiebig getestet und ist nun seit einiger Zeit im produktiven Einsatz. Jetzt wird eine neue Variante benötigt, die z. B. ein etwas anderes Beschleunigungsverhalten aufweist. Man kann einfache eine Unterklasse Rennmotor von Motor ableiten und nur eine einzige Methode, beschleunige() überschreiben.

In der Anwendung gibt es eine (oder mehrere) Variablen vom Typ Motor. Man kann dieser Variable ein Objekt der Unterklasse Rennmotor zuweisen anstelle eines normalen Motor-Objekts.

Jedesmal wenn dann die beschleunige()-Methode auf dem in der Variablen gespeicherten Objekt aufruft, wird die Methode aus der Rennmotor-Klasse ausgeführt. Ansonsten verhält sich der Rennmotor wie jeder andere Motor.

Auf diese Weise hat man dann das Verhalten verändert (eine andere Methode wird ausgeführt), ohne eine einzige Zeile der Oberklasse Motor geändert zu haben!

Bild: http://www.sxc.hu/photo/206507

Da also der Code von Motor unverändert geblieben ist, braucht man ihn nicht wieder zu testen. Das Testen kann man auf den neuen Code beschränken, also z. B. auf die Rennmotor-Klassen (und die Stellen im Code, die man geändert hat, um Rennmotor- statt Motorobjekte zu erzeugen).

Das ist echte Wiederverwendung von Code!

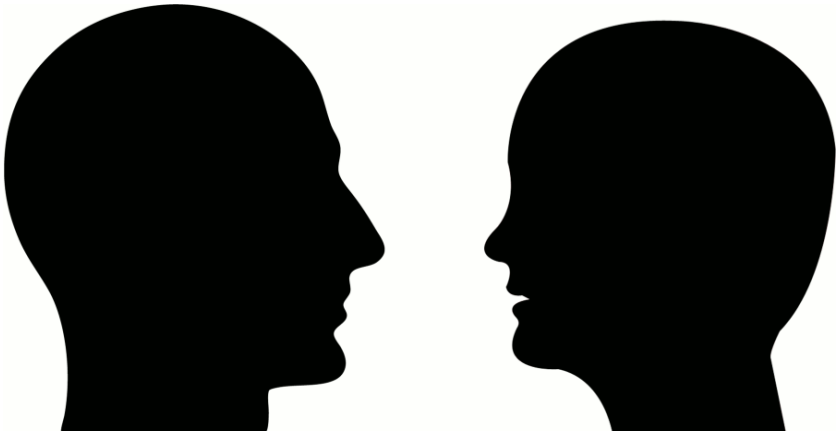
Bild: http://www.sxc.hu/photo/289829

Jeder trägt seine Verantwortung. Auch jedes Objekt.

Im Fall eines Fahrzeugs hat der Motor die „Verantwortung“ oder Aufgabe, das Fahrzeug zu beschleunigen.

Bild: http://www.sxc.hu/photo/321961

Konzept: Delegation (delegation).



Der Motor ist Teil eines Fahrzeugs. Umgekehrt kann man sagen, ein Fahrzeug **hat ein** Motor. Und es benutzt diesen Motor. Es „delegiert“ die Aufgabe Beschleunigen an diesen Motor.

Wenn eine Klasse ein Attribut (also Variable) vom Typ einer Klasse hat, das also ein Objekt als Wert speichert, kann sie die Methoden dieses Objekts aufrufen. Die Methoden des Objekts erledigen dann Aufgaben für die Klasse, die also die Erledigung der Aufgaben an das in ihr gespeicherte Objekt delegiert.

Bild: http://www.sxc.hu/photo/980826

Das ist die Stärke von OOP.

Durch diese objektwertigen Attribute und Delegation kann man eine ansonsten komplexe Klasse mit vielen Verantwortlichen aufteilen und vereinfachen, indem man mehrere Klassen schreibt, die jede nur eine oder wenige Verantwortlichkeiten haben. Diese Klassen sich einfacher zu schreiben, einfacher zu verstehen und dadurch weniger fehleranfällig. Außerdem sind einfache Klassen einfacher abzuleiten und zu erweitern.

Wieder eine Anwendung des „**Teile und Herrsche**“-Paradigmas.

Desweiteren kann man die enthaltenen Objekte austauschen, genauso wie man in einen Wagen einen neuen Motor einbauen kann – z. B. einen Rennmotor. Durch Ableiten der Klasse eines inneren Objekts und Überschreiben von dessen Methoden kann das Verhalten des Gesamtsystems verändert werden.

Man kann auch durch das Schreiben einer Unterklasse das Verhalten ändern. Aber das geht nur zum **Kompilierzeitpunkt**. Man kann die Klasse eines einmal erzeugten Objekts nicht mehr ändern.
Dem gegenüber kann man aber neue innere Objekte erzeugen und den Attributen der umgebenden Klasse zuweisen – neuen Rennmotor einbauen. Das geht jederzeit zur **Laufzeit**.
Das macht Delegation zu einem noch flexibleren Verfahren als Vererbung.

Bild: http://www.sxc.hu/photo/722800

Erstellen und Zerstören von Objekten.

Wenn man mit einer Klasse ein Objekt erstellt, wird eine spezielle Methode der Klasse ausgeführt – der **Konstruktor**. Diese Methode wird normalerweise dazu genutzt, den Zustand des Objekts sinnvoll zu initialisieren, also seine Attributwerte zu setzen und ggf. weitere innere Objekte zu erzeugen, Dateien, Datenbankverbindungen oder andere Ressourcen zu öffnen u. ä.
Ein Konstruktor kann auch Parameter haben (wie jede andere Methode) und so kann man gezielt „bestimmte“ Objekte erstellen.

Das Gegenteil des Konstruktors ist ein **Destruktor**, der dazu genutzt werden kann, reservierte Ressourcen wieder freizugeben, wenn das Objekt nicht mehr benötigt wird. Er wird aufgerufen, bevor ein Objekt zerstört wird, also z. B. wenn das Objekt gelöscht wird oder die Ausführung der Anwendung den Sichtbarkeitsbereich der das Objekt speichernden (lokalen) Variable verlässt.

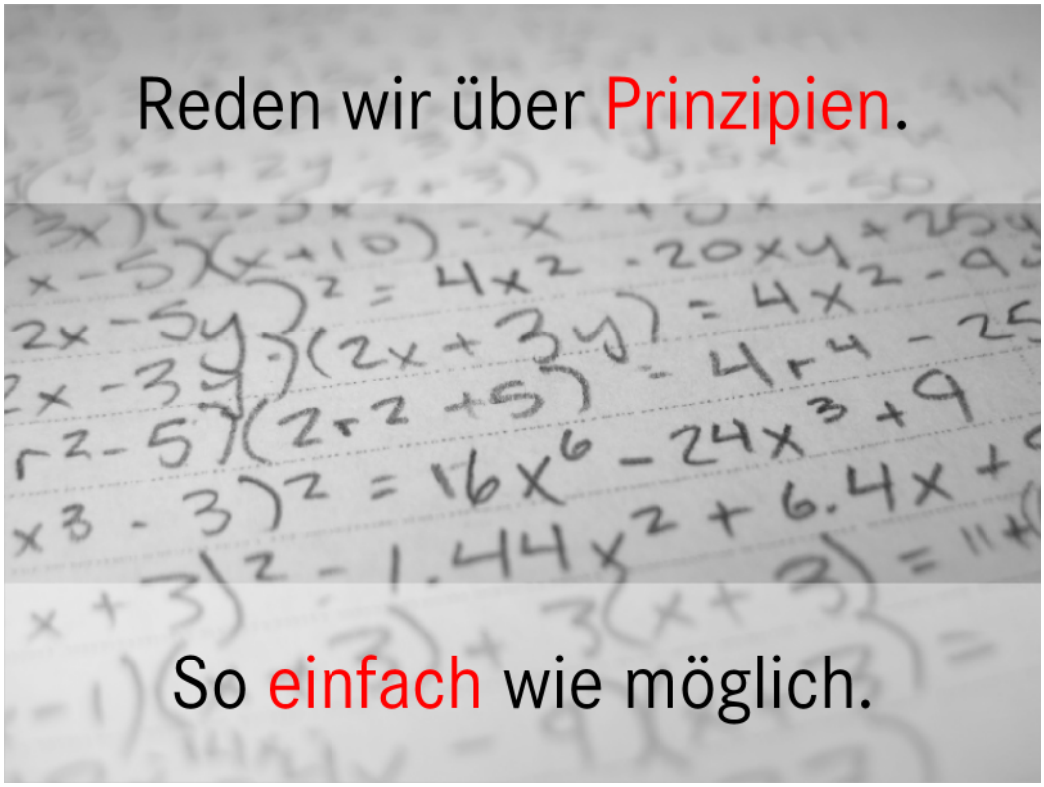
In LotusScript heißt der Konstruktor:

```
Public Sub New
```

und der Destruktor:

```
Public Sub Delete
```

Bild: http://www.sxc.hu/photo/600354



Allein die Verwendung von Klassen und Objekten macht eine Anwendung noch nicht „besser“. Es ist nur eine notwendige Grundlage dafür.

Um das meiste aus OOP herauszuholen und wirklich bessere Programme zu schreiben, sollte man noch ein paar Prinzipien beherzigen – das heißt aber eigentlich nur, dass man seinen gesunden Menschenverstand anwenden sollte.

Das aus meiner Sicht wichtigste Prinzip ist es, seine Programme so einfach wie möglich zu halten, während sie natürlich noch ihre Funktion erfüllen. Sie komplexer zu machen als unbedingt nötig bedeutet in erster Linie: längere Entwicklungszeit, längeres Testen, mehr Fehlermöglichkeiten, erschwerte Wartung und Lesbarkeit.

Bild: http://www.sxc.hu/photo/614682

Die öffentlichen Attribute und Methoden einer Klasse haben jeweils einen Namen, ggf. Parameter und Rückgabewert, und eine bestimmte Semantik, die durch den Namen und hoffentlich durch Kommentare beschrieben wird.

Der Benutzer einer Klasse – also ein Entwickler, der Objekte der Klasse erstellt und benutzt – kann und muss sich auf diese Charakteristiken verlassen können.

Wie einen Vertrag muss der Entwickler der Klasse dieses **Versprechen** erfüllen.

Bild: http://www.sxc.hu/photo/204756

Wenn man ein öffentliches Attribut oder eine öffentliche Methode löscht oder ändert, also z. B. umbenennt oder die Parametersignatur ändert, kann Code, der diese Klasse benutzt, ungültig werden. Dementsprechend müssten alle Benutzer der Klasse benachrichtigt werden und ihren Code auf notwendige Änderungen prüfen und diese dann durchführen. Da ihr Code dann verändert wurde, muss er natürlich auch neu getestet werden.

Neue Attribute und Methoden sind aber unkritisch und können nicht solche Auswirkungen haben.

Daraus folgt, dass man das öffentliche Interface einer Klasse zwar jederzeit erweitern, aber nie mehr verkleinern oder ändern kann, ohne das es viel Arbeit macht. Indem man die öffentliche Schnittstelle von Anfang an so schlank wie möglich hält, reduziert man die Wahrscheinlichkeit, dass man die Schnittstelle später verkleinern oder ändern muss.

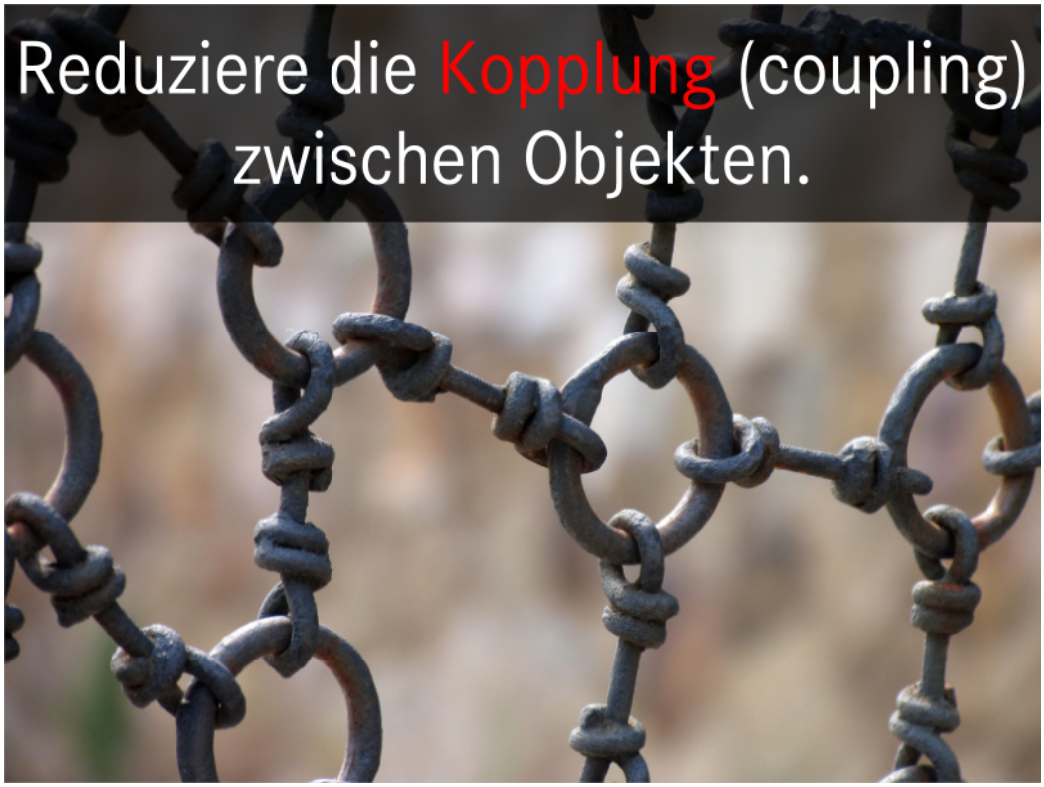
Man sollte deshalb nicht weniger Methoden oder Attribute erstellen, sie aber wenn möglich als privat deklarieren.

Bild: http://www.sxc.hu/photo/841825

Die privaten Methoden und Attribute und die „innere“ Implementierung kann man aber jederzeit ändern. Da sie sowieso von außen nicht sichtbar sind, kann es keine Referenzen oder Abhängigkeiten geben, die durch Änderungen gebrochen werden könnten.

Das kann man damit vergleichen, dass der Hersteller einer Uhr ihr Innenleben komplett ändern kann, also z. B. eine mechanische Uhr zu einer batteriebetriebenen machen kann. Der Benutzer der Uhr kann sie trotzdem noch genauso stellen und ablesen, da sich sein „Interface“ nicht geändert hat.

Bild: http://www.sxc.hu/photo/921108



Man nennt Abhängigkeiten zwischen Klassen auch Kopplung. Eine Klasse hängt von einer anderen ab, wenn sie deren Attribute liest oder ändert oder ihre Methoden aufruft.

Indem man die Benutzung von anderen Klassen reduziert, verringert man auch die Kopplung. In der Praxis heißt dass meistens, dass man versucht, von weniger Klassen abzuhängen und nicht, deren Attribute und Methoden weniger zu benutzen.

Viele Abhängigkeiten zwischen Objekten zu haben, macht ein Programm unflexibler – genauso wie diesen Metallzaun. Wenn man versucht, ein Element zu ändern, zu bewegen, muss man unweigerlich alle anderen gekoppelten Element auch bewegen.

Auf OOP übertragen bedeutet dass, das die Änderung einer Klasse die Änderung vieler Klassen notwendig machen kann.

Bild: http://www.sxc.hu/photo/711284

Das ist jetzt wirklich nichts Neues: Teste früh, teste oft.

Klassen bilden eine ideale Einheit für Tests, weil es relativ kleine Komponenten sind (sein sollten!) und sie möglichst abgeschlossen sind (sein sollten!)

Nachdem man die Klassen einzeln getestet hat, kann man sie zu größeren Komponenten integrieren, die man wieder testet. Das wiederholt man, bis man das gesamte Programm erstellt und getestet hat.

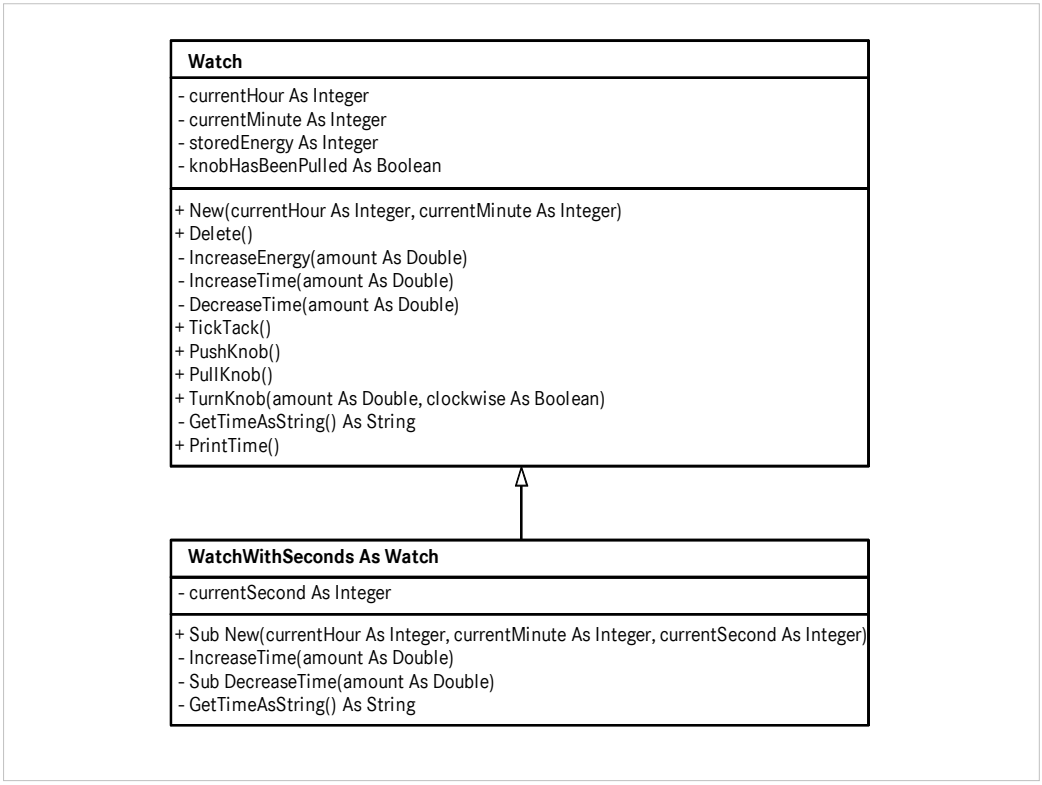
Bild: http://www.sxc.hu/photo/213004

Auch wenn ich jetzt gleich die Beispiele in LotusScript zeigen werde, gehe ich mit der LotusScript-spezifischen Syntax nicht ins Detail. Man kann sie schnell und bequem in der Designer-Hilfe nachschlagen. Dort stehen die exakten Bezeichner, Reihenfolgen, Parameter und Beschreibungen...

Bild: http://www.sxc.hu/photo/948591

Wir sehen uns jetzt die Watch-Klasse in der „Class Watch“-Skriptbibliothek der Demo-Datenbankan, und deren Unterklasse WatchWithSeconds. Der Agent „Demo Watch Classes“ enthält Testcode und zeigt, wie man diese Klassen verwendet.

Bilder: http://www.sxc.hu/photo/607060, http://www.sxc.hu/photo/955323



Anmerkungen:

- Alle Attribute sind privat.
- Beachte den Konstruktor (New) und Destruktor (Delete) der Watch-Klasse.
- Es gibt drei interne, also private Methoden.
- TickTack sollte eigentlich einmal pro Minute aufgerufen werden, aber die NotesTimer-Klasse funktioniert nicht in Agenten.
- PushKnob und PullKnob ändern den Zustand der Krone.
- TurnKnob benutzt ausschließlich interne Methoden. Welche das sind, hängt von dem internen Zustand der Attribute ab.
- GetTimeAsString gibt die Zeit als formatierten String zurück.
- PrintTime gibt diesen String aus, also die aktuelle Uhrzeit der Watch.

Die Klasse WatchWithSeconds hat nur ein Attribut und 4 Methoden, benutzt aber 4 Attribute und 7 Methoden ihrer Oberklasse – Watch. Ihr Konstruktor hat einen weiteren Parameter und ruft als erstes den Konstruktor von Watch auf. Ich überschriebe nur einige interne Methoden, lasse die öffentlichen unverändert.

Es ist absolut einfach, diese Klassen zu benutzen.
Für die Schleife rufe ich TickTack 15 mal auf, obwohl die Uhr nur „Energie“ für 10 Minuten hat. Achte auf die Ausgabe. Die Uhr ist stehen geblieben.

Jetzt sehen wir uns die Klasse „Email“ in der „Class Email“-Skriptbibliothek der Demo-DB an. Der Agent „Demo Email Class“ enthält Testcode und zeigt die Verwendung der Klasse.

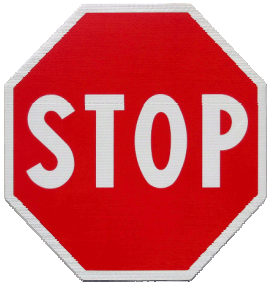
Für die Kommentare nutze ich die ls.doc-Konvention (siehe http://www.lsdoc.org/). So kann ls.doc die Kommentare automatisch extrahieren und – wie javadoc – formatierte Webseiten daraus erstellen.

Zuerst hatten wir eine Prozeduren zum Versenden von E-Mails (ähnlich @SendMail) mit SendTo, Subject und Message als Parameter. Dann kamen noch CopyTo und BlindCopyTo dazu. Dann eine Dokumentverknüpfung mit Kommentar. Dann brauchten wir einen „besseren“ Zugriff auf den Body, um Formatierungen und mehrere Verknüpfungen hinzuzufügen. Dann sollte man noch den Absender (Sender und Principal) beliebig setzen können. Als Ergebnis hatten wir sehr viele Methoden mit unterschiedlichen Signaturen.
Dann haben wir diese Klasse erstellt, die alle unsere Anforderungen abdeckt. Später haben wir der Klasse sogar noch mehr Fähigkeiten verliehen, wozu wir jeweils nur ein paar Attribute und Methoden hinzufügen mussten. Insgesamt wurde die Verwendung der „Versende E-Mails“-Funktionalität so viel einfacher und wartbarer.

Für die häufigsten Anwendungsfälle kann man dann wieder öffentliche Prozeduren schreiben, die intern die Klasse benutzen. So hätte man für den einfachen Fall wieder nur einen Aufruf und könnte gleichzeitig für komplexere Anforderungen die Klasse direkt benutzen.

Bilder: http://www.sxc.hu/photo/607060, http://www.sxc.hu/photo/890671

Das Ende...



Stelle deine Fragen jetzt – oder später

Blog: www.assono.de/blog
E-Mail: tbahn@assono.de
Telefon: +49/4307/900-401

Bild: http://www.sxc.hu/photo/200984