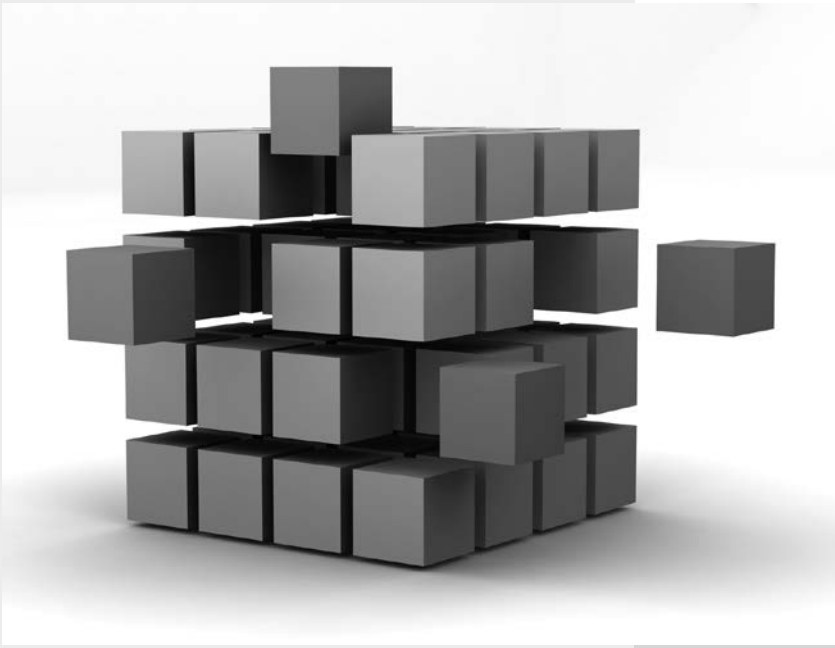
Kapitel 1

Einführung



Gleich herein: Ich habe mir das allgemein übliche Herumgerede am Anfang eines Lehrbuches gespart. Sie sollen hier einen Überblick über die verschiedenen Elemente erhalten, die Sie zur Programmierung erlernen müssen, damit Sie erst einmal eine Grundstruktur des Wissens haben.

Sie haben es als Programmierer mit zahlreichen neuen Dingen zu tun. Das verwirrt häufig und führt zu Missverständnissen. Meist sind es nur Begriffsverwechslungen, die nicht wirklich schlimm sind. Wenn Sie dann aber im Internet nachforschen wollen oder Fragen in Foren stellen, ist es schwierig, an die richtige Information zu kommen. Daher hier erst einmal die Grundstruktur und wesentliche Gedanken:

Jeder Handwerker hat zwei Dinge in seiner Werkstatt: das Material, das er bearbeitet, und die Werkzeuge, mit denen er es bearbeitet. Und Programmieren ist viel, viel Handwerksarbeit. Daher will ich mit Ihnen zunächst einen kleinen Rundgang durch die von Ihnen neu erworbene Werkstatt machen.

Das Material, die Programmiersprache »Objective-C« und das Framework »Cocoa« bespreche ich als Erstes, wobei ich ganz abstrakt bleibe, also nur die Grundkonzepte vorstelle.

Danach geht es an die Aufgabengebiete der Werkzeuge, der sogenannten Developer-Tools.

1.1 Die Sprache und das Framework

Computer werden mit Programmiersprachen programmiert. Die Programmiersprachen stellen also das Material dar, aus dem wir später unser Programm bauen. Aber bei modernen Programmiersprachen verhält es sich so, dass sie gleichermaßen nur eine leere Hülle bilden. Inhaltliche Funktionalität gibt erst das »Framework«, eine Art Grabbelkiste vorgefertigter Elemente. Man kann also vereinfachend die Programmiersprache Objective-C als »Grammatik« und das Framework Cocoa als »Vokabular« bezeichnen. Dabei können Sie an der Programmiersprache nichts ändern. Ihre Programmiertätigkeit liegt vielmehr darin, das Vokabular ständig zu erweitern.

Der Buchtitel »Objective-C und Cocoa« vermittelt dabei eine Zweiteilung. In Wahrheit geht es aber um drei Komponenten:

1.1.1 Objective- …

Die Programmiersprache, mit der wir hier programmieren werden, nennt sich »Objective-C«. Das steht ja auch auf dem Buchdeckel. Daher hier ein paar einleitende Worte zur Sprache und ihren Konzepten:

Bei Objective-C handelt es sich um eine sogenannte objektorientierte Sprache. Die Technologie bezeichnet man als »objekt-orientierte Programmierung« (OOP). Da der Begriff eine ganze Zeit ein Modewort war, ist er leider versaubeutelt worden. Objective-C verdient jedoch den Namen OOP so, wie er ursprünglich von *Alan Kay* Ende der 70er-Jahre

erfunden wurde, als dieser die Programmiersprache Smalltalk-80 entwickelte, den Vorläufer von Objective-C.

Kay arbeitete am Xerox Palo Alto Research Center (Xerox-PARC). Richtig, Xerox-PARC, das war das Forschungsinstitut, von dem auch Apple die ersten Ideen für eine graphische Benutzeroberfläche bekam. (Später arbeitete übrigens Kay eine Zeit lang für Apple.) Und diese Idee der graphischen Benutzeroberfläche revolutionierte nicht nur die Bedienung von Computern, sondern auch ihre Programmierung. Denn für diese neue Art des User-Interfaces waren bisherige Programmiersprachen unbequem. Um das zu verstehen, muss man sich erinnern (wenn man alt genug ist) oder lernen, wie man damals mit Computern arbeitete:

Grundsätzlich gab das Programm dem Benutzer in einem Raster vor, was wann zu tun war. Wir schreiben gleich ein Umrechnungsprogramm. Eine Sitzung mit einem solchen Programm hätte damals vermutlich wie folgt ausgesehen:

```
Geben Sie den Ausgangswert ein: 3[Enter]
Geben Sie den Umrechnungsfaktor ein: 2.54[Enter]
Das Ergebnis ist 7,62
Möchten Sie noch eine Umrechnung vornehmen (j/n):n[Enter]
```

Hier werden also 3 Zoll in 7,62 cm umgerechnet. Der Punkt ist, dass das Programm vorgibt, wann was getan wird: Ausgangswert eingeben – Umrechnungsfaktor eingeben – Ergebnis berechnen und ausgeben – Ende des Programms abfragen. Das Programm hat also gewissermaßen vier Arbeitsschritte, die im festen Raster abgearbeitet wurden.

Eine moderne Anwendung legt Sie nicht fest.

Stellen Sie sich mal eine Anwendung für OS X vor: Hier gäbe es zwei Felder zur Eingabe der Werte (Ausgangswert und Umrechnungsfaktor), einen Button oder einen Menüeintrag *Umrechnen* und einen Menüeintrag *Beenden*. Und für Sie wäre es völlig klar, dass Sie jeden dieser Arbeitsschritte in beliebiger Reihenfolge ausführen können. So könnten Sie etwa den Umrechnungsfaktor 2,54 vor dem Ausgangswert eingeben. Sie könnten jederzeit das Programm beenden. Natürlich würden Sie ganz häufig beim zweiten Mal nur noch den Ausgangswert eingeben und auf *Umrechnen* klicken, da sich der Umrechnungsfaktor nicht ändert, wenn Sie etwa eine ganze Zahlenkolonne von Zoll nach cm umrechnen. Wieso jedes Mal den Umrechnungsfaktor neu eingeben? Dann wäre also die Reihenfolge der Arbeitsschritte wieder eine andere.

Lange Rede, kurzer Sinn: Mit der Erfindung der graphischen Benutzeroberfläche gibt nicht mehr das Programm dem Benutzer die Abfolge der Arbeitsschritte vor, sondern der Benutzer dem Programm. Die Leute, die die graphische Benutzeroberfläche entwickelten, nannten diesen ersten Lehrsatz: »Don't mode me!«, übersetzt vielleicht: »Zwinge mich nicht dazu, eine bestimmte Abfolge einzuhalten.«

Und dies war für bisherige Programmiersprachen unbequem zu formulieren. Grundsätzlich denkt man beim Programmieren in Schritten, die nacheinander ausgeführt werden. Als Vergleich werden hier gerne Kochrezepte herangezogen: ein Arbeitsschritt nach dem anderen. Sie kämen ja auch nicht auf den Gedanken, zuerst die Pizza zu belegen und dann den Teig zu machen. Geht irgendwie nicht …

Nachrichten

Versetzen wir uns also in Alan Keys Situation: Er kannte Programmiersprachen, die eine feste Abfolge von Arbeitsschritten wollten, und er hatte im Nebenzimmer Gestalter sitzen, die sagten, dass der Benutzer eine freie Abfolge von Arbeitsschritten will. Und er musste das irgendwie zusammenbringen.

Der erste Schritt zur Lösung besteht darin, die Aktionen des Benutzers (Drücken einer Taste, Klicken auf einen Button oder einen Menüeintrag usw.) als Nachricht des Benutzers an das Programm aufzufassen. Schauen Sie sich oben noch einmal den Ablauf eines »herkömmlichen« Programms an: Dort schickt das Programm Nachrichten an den Benutzer, was er jetzt zu tun habe. Jetzt machen wir es genau umgekehrt: Wir schicken Nachrichten an das Programm, was es zu tun habe. Also etwa: »Taste gedrückt: 3.«

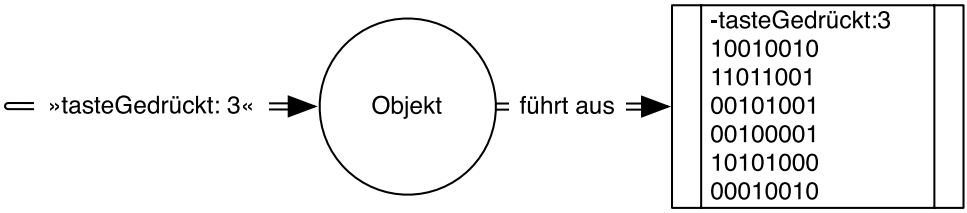
Jeder dieser Nachrichten wird dann vom Programmierer ein Stück Programm zugeordnet. Also, es gibt etwa einen Programmteil, der ausgeführt wird, wenn eine Nachricht »Taste gedrückt: 3« eintrifft. Dann wird der Programmteil tasteGedrückt: ausgeführt.

GRUNDLAGEN

Für die OOP im Sinne von Kay ist die Nachricht zentral. Es gibt auch Programmiersprachen, die Nachrichten gar nicht explizit kennen. Sie sind nicht objektorientiert in Kays Sinne.

Objekte

Jetzt gibt es da aber ein Problem: Wohin mit der 3? Die könnte ja im ersten Eingabefeld (Ausgangswert) oder im zweiten Eingabefeld (Umrechnungsfaktor) gedrückt worden sein. Und was soll mit der 3 geschehen? Sie muss ja irgendwie in den bereits bestehenden Text im Eingabefeld angehängt oder eingefügt werden oder was auch immer.



Erhält ein Objekt eine Nachricht, so führt es ein kleines Stück Programm aus.

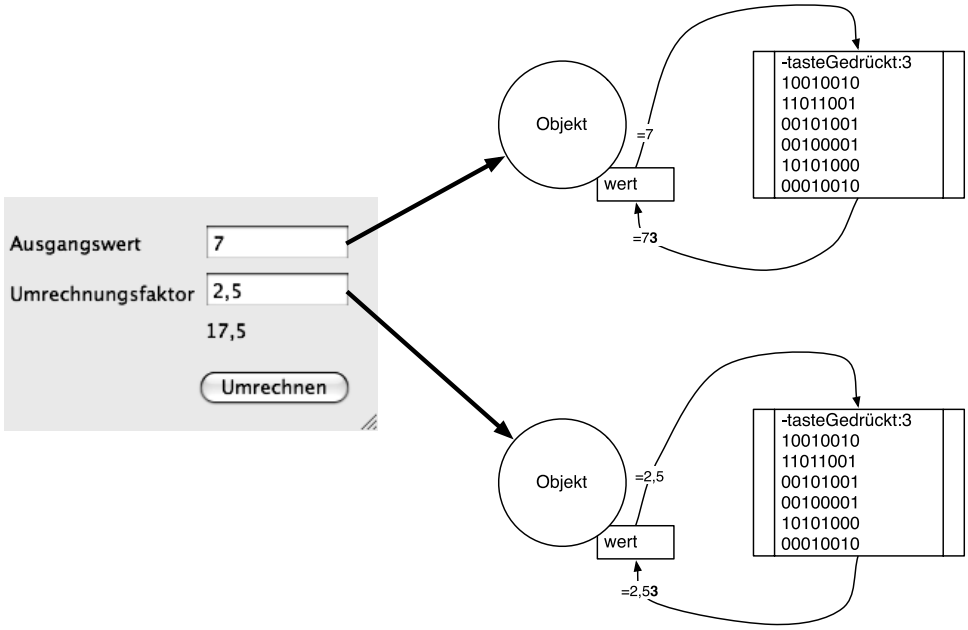
Hier kommt das zweite Konzept zum Tragen: Jede Nachricht hat einen Adressaten. Und diesen Adressaten nennt man »Objekt«. In unserem Beispiel wäre jedes Eingabefeld ein Objekt, eben ein Eingabefeldobjekt. Und so ein Objekt zeichnet sich durch zwei Dinge aus: Zum einen kann es aufgrund einer Nachricht ein bisschen Programm ausführen, wie bereits oben angedeutet. Man bezeichnet dieses bisschen Programm als »Methode«. In unserem Beispiel könnten die beiden Objekte also die Methode `tasteGedrückt:` ausführen. Dort wäre dann ein bisschen Programm, welches die Taste entgegennimmt und in den Text einfügt.

Das Zweite ist, dass ein Objekt Daten speichern kann. Nehmen Sie an, dass im ersten Eingabefeld schon der Wert 7 steht, im zweiten 2,5. Dies bedeutet, dass das erste Eingabefeld-Objekt den Wert 7 gespeichert hat und das zweite den Wert 2,5.

GRUNDLAGEN

Um dies gleich klarzustellen: Jedes Objekt kann mehrere Werte speichern, nicht nur einen. In unserem Beispiel benötigen wir jedoch lediglich einen. Andere Werte, die zu einem Eingabefeld-Objekt gespeichert sind, sind etwa die Textfarbe (fast immer schwarz), ob ein Rahmen gezeichnet werden soll usw.

Wird jetzt eine Taste im ersten Eingabefeld gedrückt, so erhält dieses erste Eingabefeld-Objekt die Nachricht »tasteGedrückt: 3« und führt daraufhin seine Methode `tasteGedrückt:` aus. Daraufhin fragt es sich selbst, welcher Wert denn bisher gespeichert ist, und erkennt 7.



Jedes Objekt kennt zudem seine Werte.

An diese 7 hängt es die 3 an und speichert 73 als neuen Wert.

Drückt der Benutzer demgegenüber die Taste, während der Cursor im zweiten Eingabefeld ist, so erhält das zweite Eingabefeld-Objekt die Nachricht »Taste gedrückt: 3« und führt die Methode `tasteGedrückt:` aus. Dort sieht die Methode, dass bisher der Wert 2,5 gespeichert ist, hängt eine 3 an und speichert das wieder als Wert 2,53.

Wieso machte das Kay auf diese Weise? Nun, wenn früher das Programm die Arbeitsschritte festlegte, konnte es keine Missverständnisse geben: Der erste Wert, der vom Benutzer eingegeben wurde, war der Ausgangswert. Der zweite Wert, der eingegeben wurde, der Umrechnungsfaktor. Die Zuordnung der Benutzereingabe zu den Speicherstellen des Programms war also fest. Jetzt jedoch ging das ja alles durcheinander. Und daher musste eine Zuordnung der Nachricht und des Speichers erfolgen.

Zusammengefasst: Ein Objekt ist eine Einheit, die Daten speichern kann und aufgrund einer Nachricht eine Operation (Methode) ausführt.

Klassen und Klassenobjekte

Ihre Arbeit als Programmierer besteht nun darin, diese Objekte zu programmieren. Nein, ganz richtig ist das nicht. Objective-C ist auch eine sogenannte klassenbasierte Programmiersprache.

Schauen wir noch einmal auf unser Programm, das wir gleich programmieren werden. Ich hatte Ihnen gesagt, dass beide Eingabefelder Objekte sind, weil sie einerseits Nachrichten empfangen können, andererseits Werte speichern.

Aber auch der weiter unten liegende Button ist ein Objekt. Er kann auch Nachrichten empfangen, etwa, wenn der Benutzer auf ihn klickt. Auch kann er Werte speichern, etwa seine Beschriftung »Umrechnen«. Also auch ein Objekt.

Aber ich muss Ihnen nicht erklären, dass die beiden Eingabefelder sehr ähnlich sind, der Button demgegenüber etwas ganz anderes. Überlegen wir uns mal, warum das richtig ist, was wir bereits ganz intuitiv fühlen:

Zum einen können beide Eingabefelder die gleiche Art von Daten speichern, nämlich den Wert, der gespeichert ist. Klar, der Wert kann in jedem Eingabefeld anders sein. Aber was überhaupt gespeichert wird, ist bei beiden Eingabefeldern gleich. Beim Button dagegen wird etwas anderes gespeichert, nämlich seine Beschriftung. Jeder Button kann wiederum eine andere Beschriftung haben. Aber die Art der Daten, die gespeichert wird, ist eben Beschriftung.

Ebenso verhält es sich bei den Nachrichten und Methoden: Beide Eingabefelder können auf die Nachricht »tasteGedrückt:« reagieren und dementsprechend die Methode `–tasteGedrückt:` ausführen. Der Button dagegen kann die Methode `–klick` ausführen.

Der Trick besteht jetzt darin, dass man gleichartige Objekte wie unsere Eingabefeldobjekte zusammenfasst zu einer Klasse. Ein ganz anderes Objekt (wie unser Button) gehört dagegen zu einer anderen Klasse. Also:

Das erste Eingabefeldobjekt ist von der Klasse `Eingabefeld`.

Das zweite Eingabefeldobjekt ist von der Klasse `Eingabefeld`.

Das Buttonobjekt ist von der Klasse `Button`.

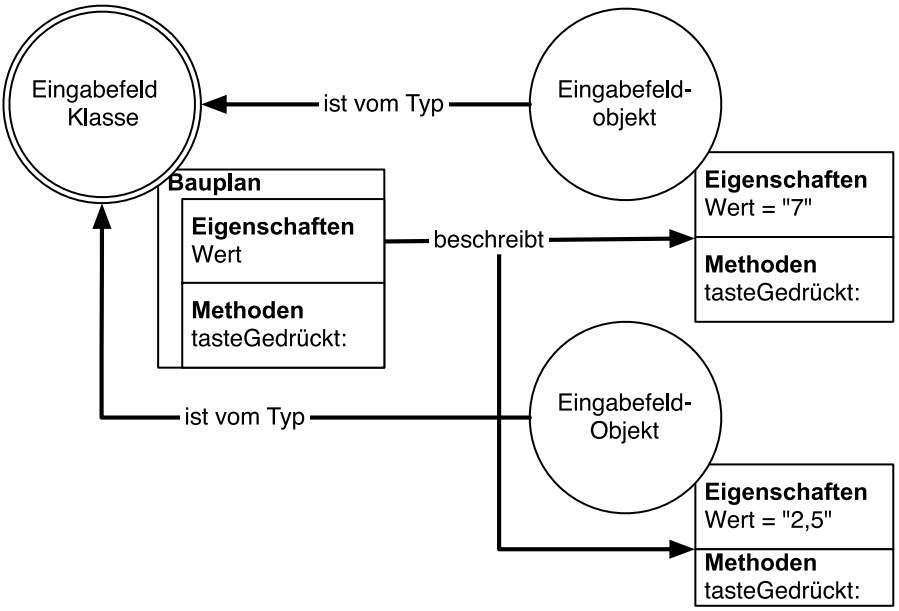
Und Ihre Aufgabe als Programmierer ist es jetzt, diese Klassen zu programmieren. Das sieht dann etwa so aus:

```
Eingabefeld
    Hat folgende Eigenschaften:
    Wert
    Textfarbe

    Hat folgende Fähigkeiten:
    tasteGedrückt:
```

```
Button
    Hat folgende Eigenschaften:
    Beschriftung

    Hat folgende Fähigkeiten:
    klick:
```

Dabei legen Sie also fest, welche Eigenschaften (Zahlen, Texte, Farbe usw.) das Objekt hat und welche Methoden aufgrund einer Nachricht ausgeführt werden können.

Später, wenn das Programm gestartet wird, liest der Computer diese Beschreibung und erstellt entsprechende Objekte in der gewünschten Zahl. Man kann also sagen, dass die Klasse eine Beschreibung der Objekte ist, deren Bauplan. Und man sagt, dass es den Typen festlegt. »Typ«, das ist das Fachwort.



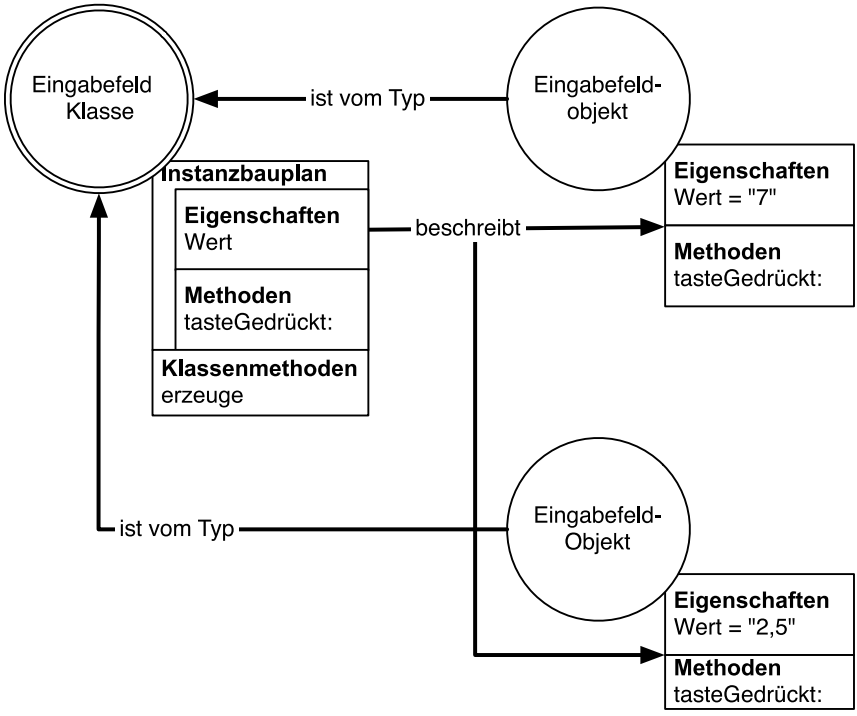
Die Klasse enthält die Beschreibung, das Objekt den konkreten Wert.

Wenn man die drei Kreise vergleicht, stellt man etwas fest: Die Eingabefeldklasse beschreibt einmalig die bei den Eingabefeldern vorhandenen Eigenschaften und Methoden. Dies ist daher bei beiden Eingabefeldobjekten gleich, Der einzige Unterschied besteht darin, dass die konkret gespeicherten Werte (7 bzw. 2,5) voneinander abweichen. Daher reicht es aus, wenn bei einem Objekt nur dieser Wert gespeichert wird, bei der Klasse der Rest. Für Sie ist es aber wichtig zu sehen, dass die Methode `tasteGedrückt:` eine Methode des Objektes, nicht der Klasse ist. Und `Wert` ist die Bezeichnung einer Eigenschaft des Objektes, nicht der Klasse. Deutlich wird das, wenn wir unser Modell erweitern:

Weil aber die Klasse der Hohe Wächter des Bauplanes ist, hat sie noch eine zweite wichtige Funktion: Sie stellt die Objekte her. Hierzu erhält sie eine Nachricht der Art: »Erzeuge mir ein Objekt nach dem bei dir gespeicherten Bauplan.« Und jetzt wird es schwierig: Wenn eine Nachricht an die Klasse geschickt wird, dann muss diese Klasse ja auch ein Objekt sein. Denn Objekte sind die Empfänger von Nachrichten. Und es muss bei der Klasse also auch eine entsprechende Methode vorhanden sein, die bei jeder Nachricht ein neues Objekt erzeugt.

Und so ist es auch: Jede Klasse wird gleichzeitig durch ein sogenanntes Klassenobjekt repräsentiert. Dieses Klassenobjekt ist, vereinfacht gesagt, bei Start des Programms einfach da, muss also nicht erst erzeugt werden. Und weil es einfach da ist, können wir es ohne weiteres benutzen. Nur eine Nachricht hinschicken, das war's. Dafür haben die Klassenobjekte einen Nachteil: Man kann in ihnen keine Daten speichern. Das ist für ihre Aufgabe aber auch nicht erforderlich.

Um das Ganze unterscheiden zu können, nennen wir die erzeugten Objekte »Instanzobjekte« oder kurz »Instanzen« und die Klassenobjekte eben so oder »Klassen«. Wobei man begrifflich schon unterscheiden sollte, dass Klasse den Typen eines Instanzobjektes bezeichnet, während Klassenobjekt den Empfänger einer Nachricht meint. Wir können unsere Graphik jetzt verfeinern:



Klassenobjekte haben auch Methoden, mit denen sich Instanzobjekte erzeugen lassen.

Um einem Missverständnis gleich vorzubeugen: Klassenmethoden müssen nicht zwingend Instanzobjekte erzeugen. Sie können auch andere Aufgaben wahrnehmen. Das ist aber seltener der Fall und hier nicht von Interesse.

Ableitung und Vererbung

Warum macht man das aber mit den Klassen? Es hat einen einfachen Grund, den man »Ableitung und Vererbung« nennt.

Nehmen wir ein Beispiel, welches wir uns im Kapitel 3 über Objective-C programmieren werden. Dort werden wir es mit Musikinstrumenten zu tun haben, mit Klavieren und Gitarren. Beide Instrumente haben Gemeinsamkeiten, nämlich etwa Preis und Alter. Das liegt daran, dass sie Instrumente sind und jedes Instrument einen Preis und ein Alter hat. Daneben haben Klaviere und Gitarren aber auch unterschiedliche Eigenschaften: Klaviere haben die Eigenschaft `Tastenanzahl`. Gitarren haben die Eigenschaft `Saitenanzahl`. Wenn wir also nach dem obigen System die Klassen für Klaviere und Gitarren schreiben, sähe das in etwa so aus:

```
Gitarre
   Hat folgende Eigenschaften:
   Alter
   Preis
   Saiten

   Hat folgende Fähigkeiten:
   …
Klavier
   Hat folgende Eigenschaften:
   Alter
   Preis
   Tasten

   Hat folgende Fähigkeiten:
   …
```

Fällt Ihnen etwas auf? Da ist etwas doppelt. Man kann das strukturieren, indem man eine Klasse Instrument erstellt.

```
Instrument
   Hat folgende Eigenschaften:
   Alter
   Preis

   Hat folgende Fähigkeiten:
   …
```

und dann sagt, dass Gitarren und Klaviere eine besondere Art von Instrumenten sind:

```
Gitarre ist ein Instrument
   Hat folgende zusätzliche Eigenschaften:
   Saiten

   Hat folgende Fähigkeiten:
   …
Klavier ist ein Instrument
   Hat folgende zusätzliche Eigenschaften:
   Tasten

   Hat folgende Fähigkeiten:
   …
```

Man sagt, dass die Klassen `Gitarre` und `Klavier` von `Instrument` abgeleitet sind. Die Klasse `Instrument` bezeichnet man als »Basisklasse«, `Gitarre` und `Klavier` als »Subklassen«. (Oder ausgehend von Gitarre wäre diese die Basisklasse und Instrument die Superklasse. Eine Frage des Startpunktes der Betrachtung.) Der Witz ist übrigens, dass auch eine nachträgliche Erweiterung der Basisklasse `Instrument` zu einer Erweiterung der abgeleiteten Klassen `Gitarre` und `Klavier` führt. Füge ich etwa in Instrument eine neue Eigenschaft Farbe ein, so haben auch die Subklassen Gitarre und Klavier diese Eigenschaften.

Das Ganze gilt übrigens nicht nur für Eigenschaften, sondern auch für Methoden. Wir können also etwa bereits Instrument eine Methode `spielen` geben. Diese Methode hätten dann auch automatisch die Subklassen `Gitarre` und `Klavier`. Und man kann ebenso die Subklassen mit Methoden erweitern, etwa der Gitarrenklasse eine weitere Methode `gezupftSpielen` geben.

Wir lernen also, dass man durch Ableitung Eigenschaften und Fähigkeiten strukturieren und erweitern kann.

Überschreiben und Polymorphie

Jetzt mögen Sie sich gefragt haben, wieso es überhaupt sinnvoll sein kann, bereits der Basisklasse `Instrument` eine Methode `spielen` zu geben, weil man ja nun Gitarren und Klaviere auf ganz unterschiedliche Weise spielt. Gut, ich kann einwenden, dass dennoch diese Methode logisch vorhanden ist, weil jedes Instrument gespielt werden kann und es sich daher um eine Fähigkeit bereits des Instrumentes handelt. Aber das wird Sie wenig beruhigen, weil man einfach nicht das Programm für diese Methode schreiben kann, also abstrakt, ohne Rücksicht auf die Subklasse: Es gibt keine Musiklehrer für »Instrumente spielen«. Es gibt nur Lehrer für bestimmte Instrumente.

Und hier kommen wir zu einem weiteren Punkt: Methoden, also die Fähigkeiten, kann man in einer Subklasse nicht nur erweitern, sondern auch ändern. Der übliche Grund

dafür ist, dass zwar die Basisklasse ein sinnvolles Verhalten aufweist, man aber dennoch in einer Subklasse ein geeigneteres programmieren will.

Konzentrieren wir uns zunächst nur auf die Gitarre. Sie werden mir Recht geben, dass man die Methode `spielen` sinnvollerweise für eine Gitarre programmieren könnte. Das wäre dann das Standardgeklimper auf einer Gitarre. Wenn man jetzt eine Subklasse `Jimi-Hendrix-Gitarre` programmiert, so wird die Art des Spielens der Gitarre doch änderungsbedürftig. Niemand spielt so Gitarre wie Jimi Hendrix. Und jetzt schreiben wir uns einfach eine neue Methode spielen, die eben anders funktioniert. Die Methode `spielen` in der Klasse `Jimi-Hendrix-Gitarre` überschreibt dann die Methode `spielen` in der Klasse `Gitarre`.

Ähnlich verhält es sich bei Instrument und dessen Subklassen. Ich kann vermutlich gar keine sinnvolle Methode schreiben. Das ist aber nicht schlimm. Ich lasse sie einfach leer und übertrage damit den Subklassen `Gitarre` und `Klavier` die Verantwortung, dort etwas Sinnvolles hinein zu schreiben. Man nennt eine solche Methode wie `spielen` in der Klasse `Instrument` eine »virtuelle Methode«. Sie gehört zwar logisch zum Instrument, weil man jedes Instrument spielen kann, aber ist inhaltlich noch nicht da, man kann die konkreten Schritte zum Spielen eines Instrumentes eben nur in Bezug auf ein bestimmtes Instrument programmieren.

Nun stellt sich aber die Frage, welche Methode ausgeführt wird, wenn eine Nachricht spielen an ein Objekt geschickt wird. Die einfache Antwort: die desjenigen Objektes, das Empfänger ist. Empfängt also ein Instanzobjekt der Klasse `Gitarre` diese Nachricht, so führt es die Methode aus, die in seiner Klasse `Gitarre` vorgegeben ist. Erhält ein Instanzobjekt der Klasse `Jimi-Hendrix-Gitarre` diese Nachricht, so wird die Methode ausgeführt, die bei der Klasse `Jimi-Hendrix-Gitarre` angegeben ist. Dieselbe Nachricht kann also in Abhängigkeit vom Empfänger zu verschiedenen Methoden, Programmstücken führen. Diesen Effekt bezeichnet man als »Polymorphie« (Vielgestaltigkeit).

Wenn übrigens eine Klasse nicht von der Möglichkeit Gebrauch macht, eine Methode zu überschreiben, so wird einfach in der übergeordneten Klasse nach einer passenden Methode gesucht. Gibt es dort keine entsprechende Methode, dann bei der nächsthöheren und so weiter. Hat man sich auf diese Weise bis zur höchsten Klasse gehangelt und diese verfügt immer noch über keine entsprechende Methode, so erzeugt das Computerprogramm einen Fehler.

Dabei ist aber ein Aspekt wichtig, den wir später noch besprechen werden: Derjenige, der die Nachricht versendet, weiß möglicherweise gar nicht, dass er es mit einer Subklasse zu tun hat. Stellen Sie sich eine Schreckenscombo vor, die aus einem Klavier und einer Gitarre besteht. Bei der Erzeugung der Instanzen muss natürlich angegeben werden, was ich will:

```
Instrument1 ist ein Instrument
Speichere in Instrument 1 eine Gitarre
Instrument2 ist ein Instrument
Speichere in Instrument2 ein Klavier
```

Sie sehen hier schon, dass ich ohne Weiteres eine Gitarre bzw. ein Klavier in einer Speicherstelle ablegen darf, die Instrumente speichert. Klar: Gitarren und Klaviere sind ja Instrumente. Wieso sollte das verboten sein? Und ein ganz anderer Programmteil, der von all diesen Subklassen nichts weiß, startet jetzt das Konzert:

```
Instrument1 spielen
Instrument2 spielen
```

Im ersten Fall wird eine Gitarre gespielt, im zweiten ein Klavier, obwohl dort nichts von Gitarren und Klavieren steht. Die in `Instrument1` gespeicherte Instanz hat sich gemerkt, dass sie eine Gitarre ist, und führt daher die Methode `spielen` von `Gitarre` aus. Entsprechendes gilt für das Klavier.

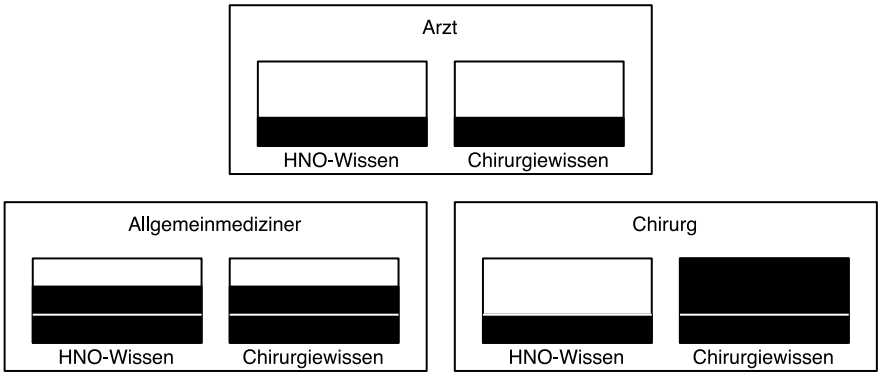
Also, zusammengefasst: Eine Subklasse kann bestehende Methoden einer Basisklasse überschreiben, was bedeutet, dass bei Empfang einer entsprechenden Nachricht anstelle des Programmteiles in der Basisklasse der Programmteil in der Subklasse ausgeführt wird. Wird sie nicht überschrieben, so wird die Methode aus der Basisklasse gewählt. Gibt es in der gesamten Ahnengalerie keine passende Methode, so erzeugt dies einen Fehler.

Erweiterung oder Spezialisierung

Widersprechen sich die Begriffe Spezialisierung und Erweiterung in einer Subklasse eigentlich nicht? Ist es nicht vielmehr so, dass eine Erweiterung etwas allgemeiner macht, also das Gegenteil von einer Spezialisierung ist?

Kommt drauf an, und zwar letztlich darauf, dass die Erweiterung eine Spezialisierung ist: Stellen Sie sich die Klasse Arzt vor. Die Objekte dieser Klasse hätte Fähigkeiten, die jeder Arzt hat, die man so in einem medizinischem Grundstudium erlernt. Dann gibt es Fachärzte als Spezialisten. Die können mehr auf ihrem Spezialgebiet, haben zusätzliches Wissen. Ihre Spezialisierung liegt also gerade in dem Mehr. Sie stimmen mir sofort zu, dass das logisch ist: Sie sind Spezialfälle, weil sie eine Erweiterung sind.

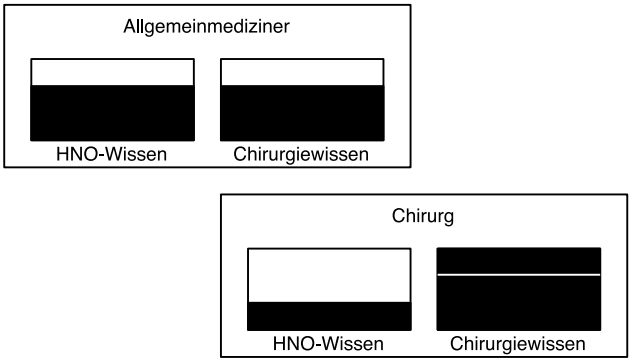
Jetzt haben Mediziner aber eine geniale Idee gehabt: den Facharzt für Allgemeinmedizin. Ist das nicht irre? Ein Spezialist für das Allgemeine! Ideen haben die Leute … Ist das jetzt ein allgemeiner Arzt (Basisklasse `Arzt`), wie das Wort Allgemeinmedizin vermittelt? Oder ist das ein Spezialist (Subklasse von `Arzt`) wie das Wort Facharzt vermittelt? Überlegen wir uns das einmal: Auch ein Facharzt für Allgemeinmedizin kann mehr als andere Ärzte, wenn auch auf breitem Gebiet. Es gibt sicherlich Dinge, mit denen er sich auskennt, die ein Facharzt für Chirurgie nicht einmal mit der Zange anfassen würde. Er ist also ein Spezialist. Und daher bildet er nicht die Basisklasse Arzt, sondern davon eine Subklasse Allgemeinmediziner, wie die anderen Fachärzte auch.



Behauptungen über das Mindestwissen von Ärzten werden in der Subklasse bestätigt.

Es gibt ein System, um zu überprüfen, ob die Anordnung von Klassen untereinander richtig ist. Sie müssen dazu einfach Behauptungen über die Basisklasse aufstellen. Hier etwa: »Hat medizinisches Wissen aus dem Grundstudium.« Und dann müssen Sie in einem zweiten Schritt überprüfen, ob jede einzelne dieser Behauptungen auch für die Subklassen gilt. Das wäre hier der Fall. Denn bei beiden Subklassen liegt in allen Bereichen mindestens das Wissen der Basisklasse vor.

Anders wäre es, wenn wir den Allgemeinmediziner zur Basisklasse erhoben hätten, was ja zunächst naheliegt. Denn gäbe es die Behauptung: »Hat breites, aber nicht tiefes Spezialwissen.« (So stelle ich mir in etwa einen Facharzt für Allgemeinmedizin vor.) Der Chirurg als Subklasse würde diese Behauptung nicht mehr erfüllen, da er zwar vermutlich auf seinem Gebiet tieferes Spezialwissen hat, dafür aber außerhalb seines Gebietes viel weniger Wissen als der Allgemeinmediziner. Wir hätten also eine Behauptung, die für die Basisklasse wahr ist, für die Subklasse nicht mehr: Fehler! Gehen Sie ruhig mal durch Ihre Wohnung und klassifizieren Sie zu Übungszwecken verschiedene Gegenstände.



Falsch: Wäre der Allgemeinmediziner Basisklasse, würde uns das HNO-Wissen des Chirurgen enttäuschen.

BEISPIEL

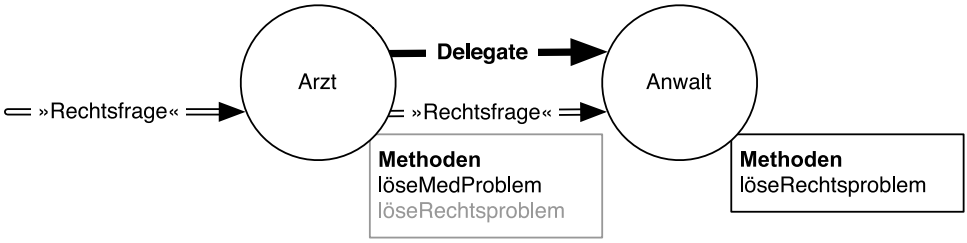
Apple liefert uns übrigens in Cocoa ein wunderbares Beispiel dafür, wie man es falsch macht. Sie werden im Kapitel 4 Container kennenlernen, so etwas wie Datenhalden. Davon gibt es häufig zwei Varianten: Eine unveränderliche Datenhalde, die die immer selben Daten hält, und eine veränderliche Variante. Die veränderliche Variante ist eine Subklasse, also Spezialisierung der unveränderlichen. Das ist falsch, denn ich kann über die Basisklasse die Behauptung aufstellen »Verändert sich nicht!« Diese Behauptung wird in der Subklasse unwahr. Tja …

Delegieren: alternative Spezialisierungen

Zuletzt sei Ihr Augenmerk noch kurz auf etwas gerichtet: Bisher hatten wir Spezialisierungen durch Subklassen hergestellt. Das hat bestimmte Nachteile, so dass man sich fragen muss, ob es nicht Alternativen gibt.

GRUNDLAGEN

Der Nachteil liegt darin, dass eine Subklasse umfangreiche Kenntnisse über den Aufbau der Basisklasse erhält. Häufig will man aber gerade den verschweigen, um sich nicht festzulegen. Das ist dann schwierig.



Delegation: Das Spezialwissen wird nicht durch eine Subklasse, sondern durch ein Objekt einer anderen Klasse besorgt.

Kommen wir wieder zu den Ärzten. Sie haben da dieses Jucken und gehen zu Ihrem Hausarzt. Der doktort an etwas herum und stellt dann fest, dass es nicht so eine übliche Krankheit ist, sondern etwas ganz Spezielles. Also schickt er sie zu einem Spezialisten, einer (anderen) Subklasse von Arzt. Das hatten wir bisher.

Es gäbe aber noch eine Möglichkeit: Der Arzt, nicht Sie, konsultiert einen Spezialisten, erklärt ihm den Fall und holt sich einen Ratschlag ein. Mit dem behandelt er Sie dann. Sie wissen nun gar nichts mehr von Fachärzten, haben es einfach mit einem einzigen Arzt zu tun, der sich sein Wissen zusammenklaubt. Der Patient, der vorhin noch von einem Facharzt zum nächsten geschickt wurde, merkt davon gar nichts. Für ihn gibt es nur einen Arzt.

Es gibt aber noch einen Trick. Derjenige, der konsultiert wird, muss gar kein Arzt sein. (Bei einer Subklasse wäre er ja automatisch ein Arzt.) Wenn Sie bei einem Plausch mit Ihrem Arzt also auf eine rechtliche Frage stoßen, so kann der Arzt sagen: »Da frage ich meinen Freund, den Anwalt.« Telefonanruf, und schwupps bekommen Sie von Ihrem Arzt eine rechtliche Auskunft. Wir verschieben also das Problem: Die Spezialisierung erfolgt nicht mehr von einer Basisklasse in eine Subklasse, sondern von dem Objekt einer Klasse zu einem Objekt einer gänzlich anderen Klasse. Und damit umgehen wir das obige Problem, dass die Basisklasse Interna bekannt machen muss. Anstelle von Hierarchien erstellen wir also Collagen. Und vor allem: Für diese Spezialisierung müssen wir nicht Anwälte zu Subklassen von Ärzten machen – was sie einfach nicht sind.

So etwa gibt es auch in der OOP. Man nennt es »Delegating«, wobei der konsultierte Rechtsanwalt das Delegate ist. Und ich stelle Ihnen das hier schon vor, weil Apple, sehr löblich, von dieser zweiten Möglichkeit der Spezialisierung durch Konsultation in Cocoa regen Gebrauch macht. Sie werden dem im Laufe dieses Buches häufig begegnen.

GRUNDLAGEN

Es gibt noch weitere Fälle, bei denen ein Objekt eine Nachricht an ein anderes Objekt schickt, um seine Aufgaben zu bearbeiten. Das ist ja auch klar, wenn man bedenkt, dass es bei der OOP darum geht, Nachrichten zu versenden. Das Besondere am Delegating ist, dass zwischen den Partnern (hier: Arzt und Anwalt) ein sehr genauer Vertrag dazu besteht, welche Anfragen weitergeleitet werden, welche davon der Anwalt bearbeiten und welche er nicht bearbeiten muss. Dies ist ein weiterer Vorteil: Beim Delegating wird genau definiert, welche Nachrichten das Delegate erhält. Beim Ableiten kann in Objective-C prinzipiell jede Methode überschrieben werden. Man sieht also nicht so genau, was spezialisiert ist und was nicht.

1.1.2 … C …

Diese Nachrichten, Objekte und Klassen bilden die große Struktur des Programms. Aber jede Nachricht führt ja dazu, dass eine Methode ausgeführt wird. Und diese einzelnen Methoden funktionieren klassisch so, wie man das vor Alan Kay kannte: Schritt für Schritt wird die Aufgabe erledigt, wie in einem Rezept. Das ist sozusagen der mikroskopische Blick auf Ihre Arbeit.

Als *Brad Cox*, der Entwickler von Objective-C, Kays Ideen aufnahm, legte er also den objektorientierten Teil von Objective-C fest. Dann musste er aber noch diesen Kleinkram festlegen? Nö, musste er nicht, denn das gab es ja schon zuhauf. Also nahm er einfach die klassische Programmiersprache C, um diesen Kleinkram zu erledigen. Das Ganze vermischt ergibt dann Objektive-C.

GRUNDLAGEN

Aber, dies sei auch gesagt: C inkorporiert einige Konzepte, die in Objective-C schlicht überflüssig sind. Wenn Sie ein C-Recke sind, so werden Sie etwa nur an ganz obskuren Ecken C-Arrays und Pointer-Arithmetik finden. Falls Sie kein C-Recke sind, so werden Sie nicht verstehen, was ich gerade gesagt habe. Das ist nicht schlimm. Zwar soll dieses Buch auch Einsteigern helfen, programmieren zu lernen. Und daher werde ich auch – ohne das zu trennen – C vermitteln. Aber eben nur immer so viel C, wie es für Objective-C nützlich ist. C ist hier also ein reines Hilfsmittel für Objective-C.

Der Grund dafür war einfach: C ist sehr verbreitet, C ist sehr gut dokumentiert, es gibt zahllose Bücher, Tutorials, was weiß ich für C. C ist eben ein Standard.

Nun gut, Nachrichten, Objekte und Klassen waren der Objective-Anteil an Objective-C. Was ist der C-Anteil? Im Wesentlichen geht es um drei Dinge, die wir von C benutzen werden:

- mathematische Berechnungen
- Verwendung von Datentypen
- Kontrollstrukturen

Klingt gut, nicht wahr? Richtig freakig. Eine ganz kurze Einleitung:

In unserem Programm müssen wir später eine Berechnung durchführen, nämlich den Ausgangswert mit dem Umrechnungsfaktor multiplizieren, um das Ergebnis zu erhalten. Der entsprechende Teil des Computerprogramms sieht so aus:

```
result = input * factor;
```

Das ist C. Reines C. Da tauchen keine Objekte auf, da werden keine Nachrichten ausgetauscht usw. Sie haben auch keine Fähigkeiten. Ein solcher Wert kann eben gespeichert und wieder gelesen werden. Das war es dann aber auch. Es werden oben einfach zwei Werte multipliziert. Der Mikrokosmos eben.

BEISPIEL

Sie sehen den Unterschied zu OOP nicht? In einer reinen OOP-Sprache würde diese Multiplikation sinngemäß lauten: »input-Objekt, bitte multipliziere dich mit dem factor-Object.« Also eine Nachricht an ein Objekt. Es gibt Programmiersprachen, die so funktionieren. Man kann das auch in Objective-C so machen. Aber Objective-C lässt es eben auch zu, dass man es »klassisch« macht.

Das Zweite sind diese (einfachen) Datentypen. Ich hatte ja bereits geschrieben, dass Instanzobjekte Daten speichern. Wenn man das in Objective-C machen möchte, so muss man in der Regel sagen, was für eine Art von Daten gespeichert wird, also etwa Text,

ganze Zahlen (-3, 5, 8 usw.) oder Brüche (2,54, 3,7, -4,8 usw.). Auch dieses System der Datentypen ist von C gestohlen.

Schließlich, und damit möchte ich diese kleine Einführung abschließen, werden nicht immer alle Arbeitsschritte nacheinander ausgeführt. Ich habe es mir da bisher ein bisschen leicht gemacht. Auch das kennen Sie bereits von Rezepten: Manchmal steht da etwas, was wiederholt werden soll, zum Beispiel:

```
Solange, bis der Teig Blasen wirft,
   Kneten Sie den Teig
```

Der Arbeitsschritt »Kneten Sie den Teig« wird also wiederholt, bis eine sogenannte Abbruchbedingung erfüllt ist. Man nennt dies eine »Schleife«. Eine andere wichtige Kontrollstruktur ist die Verzweigung. Manchmal liest man so etwas in Rezepten:

```
Falls Sie das Gericht im Ofen zubereiten wollen,
   …
andernfalls
   …
```

Das Rezept sieht hier also zwei Arten der Zubereitung – im Ofen und sagen wir: auf dem Herd – vor, und die Arbeitsschritte, die Sie erledigen müssen, unterscheiden sich dann. (Bei einer Zubereitung im Ofen muss dieser etwa erst vorgeheizt werden, während man das bei Herden eher selten macht.)

Dies ist alles C. Wie bereits angekündigt, werde ich das im weiteren Verlauf des Buches aber nicht weiter unterscheiden. Sie sollen Objective-C lernen, so, wie es jetzt ist.

1.1.3 … und Cocoa

Das letzte Element des Buchtitels bildet Cocoa. Hierbei handelt es sich um ein sogenanntes Framework. Ein solches Framework hat vornehmlich zwei Funktionen: Zum einen ist es eine Art Bauteilkiste, in der wir uns bedienen können, ohne selbst programmieren zu müssen. Zum anderen ist es so etwas wie ein warmes Plätzchen für unser Programm.

GRUNDLAGEN

Objective-C und Cocoa sind so eng verzahnt, dass es zuweilen Haarspalterei ist, eine Technologie Cocoa oder Objective-C zuzuordnen. Es kommt auch immer wieder vor, dass eine Technologie vom Framework in die Sprache verschoben wird. Ich werde das hier deshalb nicht immer haarscharf trennen. Aber Sie merken sich bitte, dass Objective-C die Programmiersprache ist und Cocoa das Framework. Ich wiederhole es gerne: Objective-C ist eine Grammatik und Cocoa das Vokabular. Erst beides zusammen hat einen sinnvollen Einsatzzweck.

Cocoa als Library

Wie bereits erwähnt ist Cocoa zunächst eine Bauteilkiste. Denken Sie etwa noch einmal an unsere Anwendung Umrechner. Hier hatten wir zwei Eingabefeld-Objekte. Sie werden in der Abbildung erkannt haben, dass diese ganz normal aussehen, wie Sie es schon bei x Anwendungen beobachtet haben. Und wenn Sie eine Taste drücken, verhalten sich diese Objekte so, wie Sie es aus x anderen Programmen gewöhnt sind. Da wäre es eine Schande, wenn jeder Programmierer jede dieser Eingabefeld-Klassen neu programmieren und jedes Mal dasselbe schreiben müsste, damit bei einem Tastendruck ein Zeichen eingefügt wird. Daher hat Apple diese Objekte bereits für uns programmiert. Wenn wir in Kapitel 2 unser erstes Programm herstellen, werden Sie sehen, dass wir uns nur bei Apple bedienen müssen und entsprechend vorgefertigte Objekte holen. Man kann das also mit einer Standard-Bauteilkiste vergleichen, bei der wir uns bedienen, um etwas herzustellen. Fertighausbauweise …

Und dies ist ein großer Vorteil: Verwenden Sie immer, wenn es möglich ist, Standardbauelemente von Cocoa. Sie ersparen sich dadurch nicht nur die Arbeit, so etwas selbst programmieren zu müssen, sondern partizipieren auch gleich noch an der Horde hochqualifizierter Programmierer bei Apple.

Cocoa selbst besteht aus drei Themenbereichen:

- Foundation: Hierbei handelt es sich um grundlegende Klassen für Objective-C-Programme.
- AppKit (Application Kit): die Elemente einer Anwendung mit graphischer Benutzeroberfläche
- Core Data: Elemente für die Datenspeicherung

Alle Klassen des Frameworks beginnen mit NS, was für »NextStep« steht. NextStep ist der Vorfahr von Cocoa.

Daneben existieren noch zahlreiche weitere Frameworks, die man optional einbinden kann. Mit diesen weiteren Frameworks werden wir uns nicht beschäftigen.

Cocoa als Umgebung

Daneben stellt uns Cocoa wichtige grundlegende Funktionen zur Verfügung, die unser Programm überhaupt erst lauffähig machen. Zum einen geht es dabei um handwerkliche Dinge. Zum anderen erschafft uns Cocoa eine Welt, die zu Objective-C passt. Was nämlich im Computer wirklich vor sich geht, ist alles andere als objekt-orientiert. Damit wir also überhaupt sinnvoll programmieren können, muss uns zunächst jemand aus der kalten Welt des Computers eine Illusion der OOP schaffen. Und auch dies ist eine Aufgabe von Cocoa.

Insgesamt ist also Cocoa eine nährende Mutter, die uns viele Dinge vorbereitet, und ein beschützender Vater, der uns die heile Welt vorgaukelt.

1.2 Xcode Developer-Tools

Der nächste wichtige Grundbegriff, von dem ich hier sprechen möchte, ist Xcode. Wie Sie sicher schon gehört haben, kann man mit einem Computer nichts anfangen, wenn man kein passendes Programm hat. Und so ist es auch, wenn es ums Programmieren geht: Wir benötigen zuerst ein Programm. Es sind die Xcode Developer-Tools. Dabei handelt es sich genau genommen um zahlreiche Programme. Aber Xcode ist das Programm, das Sie unmittelbar kennenlernen und bedienen werden. Man nennt so etwas ein »Front-End«. Die Programme, die im Hintergrund arbeiten, nennt man »Back-End«.

1.2.1 Installation der Developer-Tools

Sie installieren sich jetzt bitte Xcode. Sie erhalten das Programmpaket im App Store auf gewohnte Weise. Es ist kostenlos. Während des nicht so ganz kurzen Zeitraumes, den das in Anspruch nehmen wird, können Sie hier schon weiterlesen.

1.2.2 Xcode Integrated Developer Envirement

Ein wichtiger Teil der Developer-Tools ist das Integrated-Developer-Envirement (IDE) Xcode. Wie der Name bereits ausdrückt, handelt es sich um die Schaltzentrale des Programmierers, hinter der sich viele Programme und Programmbestandteile verbergen. Sie starten daher stets Xcode selbst.

Es verhält sich seit Xcode 4 so, dass auch das Programm zur Gestaltung der Benutzerschnittstelle (vormals Interface Builder) in Xcode integriert wurde.

Mit der Bedienung von Xcode werden wir uns im nächsten Kapitel intensiv beschäftigen. Eine ganz besondere Einsteigerhürde will ich aber schon hier nehmen: die Sonderzeichen.

Bei der Programmierung mit Objective-C und Cocoa haben wir es mit geschweiften und eckigen Klammern zu tun. Und die sind auf einer deutschen Tastatur nicht aufgedruckt. Sie verstecken sich bei gedrückter Wahltaste hinter den Tasten für die Ziffern 5 bis 9, jedoch nur im normalen Tastenfeld, nicht im Zahlenblock.

Durch Drücken der Wahltaste erreichen wir geschweifte und eckige Klammern.

Auf amerikanischen Tastaturen sind die geschweiften und eckigen Klammern auf Kosten der deutschen Sonderzeichen wesentlich leichter zu erreichen. Da beim Programmieren kein Deutsch verwendet wird, setzen einige Entwickler amerikanische Tastaturen ein. Man verwechselt dann allerdings leicht [y] und [z].

1.2.3 Compiler, Linker und Build

Die Programmiersprache Objective-C, in der wir programmieren werden, wird vom Computer nicht verstanden. Daher können Objective-C-Programme nicht auf dem Computer gestartet werden.

Super Sache! Was ich Ihnen hier also beibringen möchte, ist völlig nutzlos? Nein, so dramatisch ist es nun nicht ganz. Wie Sie vielleicht schon wissen, verstehen Computer nur 1'en und 0'en. Sie kennen nichts anderes. Daher besteht ein letztlich lauffähiges Programm immer aus solchen 1'en und 0'en, die Befehlsfolgen in sogenannter Maschinensprache enthalten.

So werden wir etwa gleich ein kleines Programm in Objective-C schreiben, in dem – wie bereits erwähnt – folgende Anweisung vorkommen wird:

```
result = input * factor;
```

Sie werden das sicher nicht verstehen, da wir mit dem Programmieren noch gar nicht angefangen haben. Aber Sie werden vermutlich irgendwie nachvollziehen können, dass in dieser Zeile eine Variable `input` mit einer Variablen `factor` multipliziert und das Ergebnis in der Variablen `result` gespeichert wird. Wenn man das Sternchen als Multiplikationszeichen liest (was es ist), dann ist das ja fast wie normale Mathematik. Diesen Programmtext in der Ausgangssprache nennt man »Source«, »Sourcetext« oder »Sourcecode«.

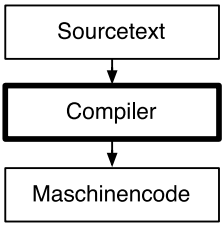
Dies sind aber offenkundig keine 1'en und 0'en. Was der Computer ausführt, sieht so aus:

```
11110010 00001111 00010000 01000101 11100000
11110010 00001111 01011001 01001001 11101000
11110010 00001111 00010001 01000101 11110000
```

Sofort verständlich, nicht wahr? Und noch schlimmer: Das hier abgedruckte Programm ist indessen nur auf Intel-Macs ausführbar. Dieses Produkt in der Sprache des konkreten Computers nennt man »Maschinencode« oder »Objectcode«. Unsere Anweisung in Objective-C ist aber unabhängig davon, ob das Programm auf einem Intel-Mac oder einem PPC-Mac ausgeführt werden soll. Was hat die obige Gleichung mit Prozessoren zu tun?

Wir lernen daraus also, dass die Sprache, die wir benutzen, zwar leichter zu verstehen und vom Computermodell unabhängig ist, jedoch gar nicht ausgeführt werden kann. Deshalb

muss jedes Programm, das wir hier schreiben, in die Sprache des Computers übersetzt werden. Dies geschieht mit einem Programm, welches Sie niemals wirklich zu Gesicht bekommen: dem Compiler. Und der bei den Developer-Tools mitgelieferte Standardcompiler heißt clang/LLVM.

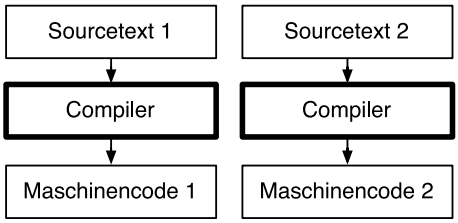


Der Compiler macht aus dem Sourcetext einen ausführbaren Maschinencode.

GRUNDLAGEN

Dieser Begriffswirrwarr hat eine Bedeutung: Der Compiler clang erzeugt entgegen meinen obigen Ausführungen Maschinencode nur für einen gedachten, imaginären (virtuellen) Computer, der sich LLVM (Low-Level-Virtual-Machine) nennt. Daraus wird dann mittels eines weiteren Compilers Maschinencode für das Zielsystem erzeugt. Insgesamt ist das aber ein für Sie unbemerkter Vorgang, der deshalb nicht weiter interessiert. In der tatsächlich von Xcode ausgelieferten Version geschieht dies alles in einem Übersetzungsvorgang. Theoretisch könnte aber der zweite Schritt erst auf einem anderen Computer ausgeführt werden, so dass dieser einen ganz anderen Prozessor eingebaut haben könnte.

Allerdings erfolgt da noch mehr. Man kann Programme zur Übersichtlichkeit in verschiedene Module gliedern. Jedes dieser Module besteht aus »seinem« Sourcecode und wird unabhängig von den anderen übersetzt. Sie können sich das wie dieses Werk vorstellen, das in zwei Bänden geschrieben ist. Jeder Band erhält unabhängig voneinander sein Layout, wird unabhängig voneinander gedruckt usw.

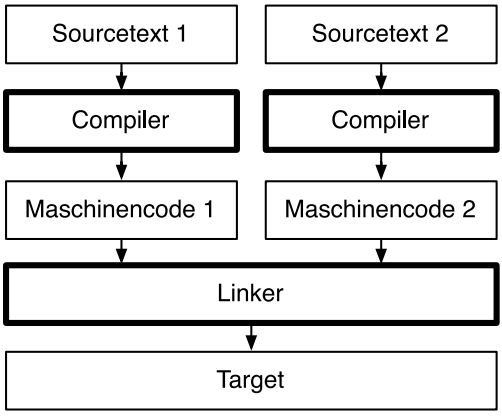


Mehrere Module werden unabhängig voneinander übersetzt.

GRUNDLAGEN

Im Übrigen gilt das nicht nur für die von Ihnen eingegebenen Sourcetexte. Auch andere Bestandteile der fertigen Anwendung, insbesondere Dateien der graphischen Benutzerschnittstelle, werden mit dafür vorgesehenen Compilern übersetzt.

Hierbei tauchen allerdings zwei Probleme auf. Wenden wir uns dem ersten zu: Wie Sie der Graphik entnehmen können, haben wir jetzt zweimal Maschinencode. Wir wollen aber am Ende ein Programm haben. Dies bedeutet, dass wir beide Maschinencode-Dateien zu einer Datei verschmelzen müssen. Dies macht der sogenannte Linker. Das Endergebnis nennt man das »Target«.



Der Linker fasst die Module zu einem Programm zusammen.

BEISPIEL

Ich habe mich dazu entschlossen, dieses Werk in zwei Bänden zu schreiben, also zwei Modulen. Diese können unabhängig voneinander ein Layout bekommen, also »übersetzt« werden. Wenn Sie jedoch das gesammelte Wissen über Objective-C und Cocoa haben wollen, benötigen Sie beide Bände. Daher werden sicherlich verschiedene Büchereien ein Paket mit beiden anbieten. Die beiden Bände sind dann wieder zu einem Werk gelinkt.

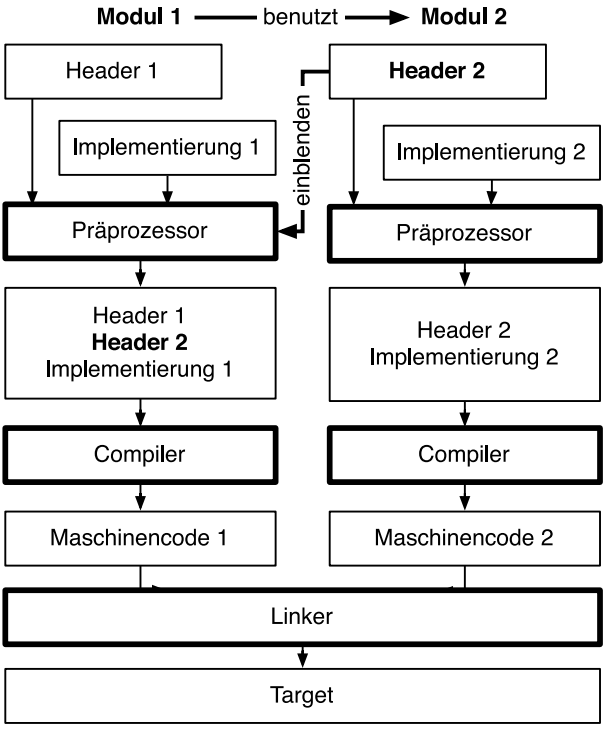
Das zweite Problem besteht darin, dass ich ja aus Gründen der Übersichtlichkeit zwei Module gemacht hatte. Natürlich gehören die aber zusammen. Es kann also passieren, dass ich für ein Modul die Funktionalität des anderen Moduls benötige. Denken Sie etwa, dass im obigen Beispiel die Klasse `Rechtsanwalt` den bei der Klasse `Arzt` definierten Delegating-Vertrag kennen muss, um sich organisatorisch darauf einzurichten, dass da komische Fragen weitergeleitet werden. Der Compiler würde sich aber bei Übersetzung des einen Moduls weigern, irgendetwas des anderen Moduls zu verwenden, weil er dies ja gar nicht kennt. Die Übersetzungen laufen ja unabhängig voneinander.

Und deshalb hat man sich etwas Feines ausgedacht: Der Sourcetext von jedem Modul wird in zwei Dateien aufgeteilt. Der eine Teil, genannt »Header«, enthält nur eine Art Lieferschein, ein Inhaltsverzeichnis. In diesem Header wird also nur gesagt: »Ich verspreche, dass dieses Modul die Methode X enthält.« In einer zweiten Datei, der »Implementierung«, wird dann die Methode erst programmiert.

BEISPIEL

Wenn ich in Band 1 auf ein Kapitel in Band 2 verweisen möchte, so muss ich ja zunächst wissen, dass dort das entsprechende Kapitel existiert. Daher habe ich mir vorher zwei Zettel geschrieben, die einfach eine Liste der Kapitel enthalten. So kann ich anhand dieses Inhaltsverzeichnisses von Band 2 sicher im Text von Band 1 verweisen. Geschrieben haben muss ich das Kapitel in Band 2 aber dazu noch nicht.

Und jetzt kommt der Trick: Ich kann mir in Modul 1 den Header (also das »Inhaltsverzeichnis«) von Modul 2 einblenden. Man nennt diesen Vorgang »importieren«. Damit weiß der Compiler, was sich dort findet, und übersetzt mein Modul 1 einwandfrei auch dann, wenn ich die Methode aus Modul 2 verwende. Aber jetzt gibt es ein Problem: Damit ich Modul 1 übersetze, muss der Compiler drei Dateien haben: den Header von Modul 1 selbst, den Header vom fremden Modul 2 und die Implementierung von Modul 1 selbst. Die drei Dateien müssen also dem Compiler zusammengeklebt übergeben werden. Hierum kümmert sich der sogenannte Präprozessor.



Die getrennten Teile eines Moduls können mit anderen Headern verschmolzen werden.

GRUNDLAGEN

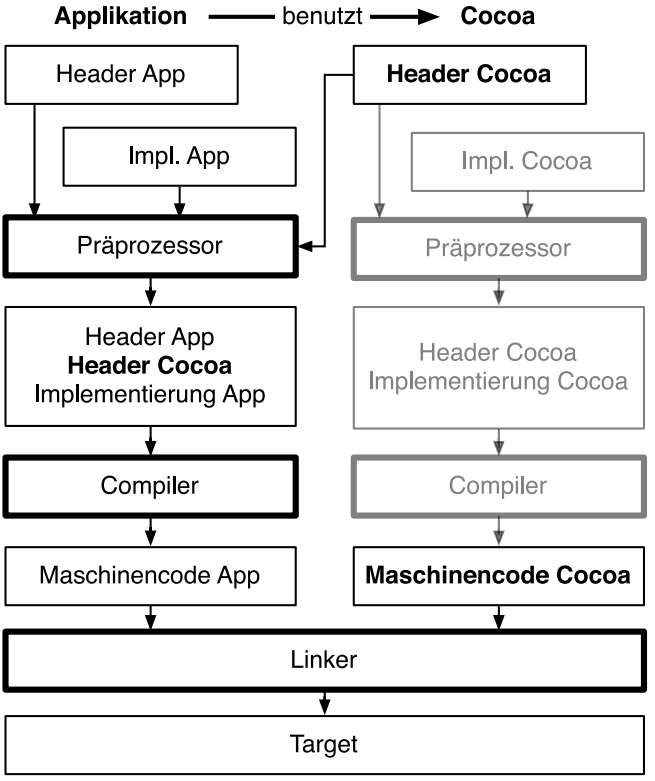
Der Präprozessor kann noch mehr. Diese Importfunktion ist aber das, was wir vor allem benötigen werden.

Hierbei passiert übrigens noch etwas: Der Compiler weiß ja nur über den Header von Modul 2, dass dort die Methode von Modul 2 vorhanden ist. Er kennt aber noch nicht die »Seitenzahl« dieser Methode. Die ist erst bekannt, wenn auch das zweite Modul übersetzt ist. Daher hinterlässt der Compiler bei der Übersetzung des Modules 1 an der entsprechenden Stelle nur einen Hinweis: »Hier soll die Methode X von Modul 2 genutzt werden.« Der Linker löst diesen Hinweis auf und setzt dafür ein, wo sich denn genau die Methode bei Modul 2 befindet. Erst dies macht das Programm lauffähig.

GRUNDLAGEN

Was passiert eigentlich, wenn Modul 1 sich auf den Header von Modul 2 verlässt und die dort genannte Methode aber nie in Implementierung 2 programmiert wird? Modul 1 vertraut ja einfach auf den Header von Modul 2 und das dort enthaltene Versprechen. Bei der Übersetzung von Modul 2 überprüft der Compiler, ob auch wirklich jedes Versprechen aus dem Header eingelöst wurde. Ist dies nicht der Fall, beschwert er sich bei Ihnen sinngemäß mit einer Fehlermeldung: »Du hast Modul 2 nicht vollständig programmiert, weil die Methode X fehlt, die du im Header versprochen hattest.« Der Linker wird dann gar nicht mehr gestartet und kein Programm erzeugt.

Ein letztes Wort: Ich hatte Ihnen ja davon erzählt, dass Cocoa ein Framework ist, welches unter anderem die Aufgabe hat, Ihnen zahlreiche vorgefertigte Funktionalitäten zu bieten. Bei diesem Framework handelt sich um nichts anderes als viele zusammengefasste Module. Dies bedeutet, dass, wenn wir Cocoa nutzen wollen, wir den Header von Cocoa importieren müssen, was wir auch tun. Das sehen Sie gleich. Aber es gibt einen Unterschied zu eigenen Modulen. Bei Cocoa sind nur die Header und der fertige Maschinencode mitgeliefert worden. Die Implementierung existiert nur bei Apple. Das funktioniert auch, wenn wir uns das mal graphisch anschauen:



Header und Maschinencode von Cocoa sind da, der Rest ist bei Apple.

Sie können es sehen: Wir können unser Modul kompilieren, weil wir den Header von Cocoa haben, so dass für den Compiler alles Notwendige da ist. Der Linker kann auch seine Arbeit tun, weil er den Maschinencode von Cocoa hat. Was genau in Cocoa die Implementierung programmiert, bleibt geheim, das weiß nur Apple.

Abschließend sei noch gesagt, dass dieser Build-Prozess hier keinesfalls vollständig dargestellt wurde. Vielmehr habe ich mich auf die reine Programmierarbeit beschränkt. Tatsächlich gehört aber zu einem Programm noch mehr, wie Sie gleich lernen werden, so dass in der Horizontalen noch andere Dinge erledigt werden. Und zu einem vollständigen Build gehören auch noch zusätzliche Schritte, so dass die vollständige Graphik auch in der Höhe noch wächst. An dieser Stelle soll das aber nicht weiter ausgeführt werden, weil wir es nicht zum Verständnis benötigen.

1.2.4 Debugger

Ein weiterer wichtiger Bestandteil der Developer-Tools ist der »Debugger«. Ein Debugger ist ein Programm, welches es Ihnen erleichtert, Fehler zu finden und zu beseitigen. Der mitgelieferte Debugger nennt sich LLDB, was eben der Debugger für die LLVM ist.

GRUNDLAGEN

Mit »Bug« bezeichnet man bei der Softwareentwicklung einen Fehler. Dies geht angeblich darauf zurück, dass sich vor geraumer Zeit ein Käfer (englisch Bug) in einen Großcomputer verirrte und dort ein Relais blockierte. Das funktionierte dann nicht mehr, hatte einen Fehler, einen Bug eben.

Im nächsten Kapitel werden wir den Debugger benutzen. Ich erspare mir hier Ausführungen, da das Verständnis dafür bereits einige Programmierkenntnisse voraussetzt. Das Grundprinzip sei aber schon erläutert: Sie können mit dem Debugger dem Programm bei der Arbeit zuschauen und auf diese Weise Fehlentwicklungen und deren Ursache erkennen.

1.3 Zusammenfassung und Ausblick

Sie haben in diesem Kapitel einen ersten Eindruck davon bekommen, was Ihre Tätigkeit als Softwareentwickler ausmacht. Sie haben rudimentäre theoretische Kenntnisse darüber gewonnen, was objekt-orientierte Programmierung mit Klassen in der Weise, wie es Objective-C umsetzt, ausmacht. Sie werden in Kapitel 2, bei einem kurzen Demo-Projekt, die praktische Umsetzung dieser Ideen und Konzepte erfahren. In Kapitel 3 erfolgt dann die tiefer gehende Auseinandersetzung mit Objective-C.

Ihnen wurde das Framework Cocoa vorgestellt, welches die Standardklassen einer Mac-Anwendung mitbringt. Der Löwenanteil dieses Buches befasst sich ab Kapitel 4 mit der Funktionalität von Cocoa und damit, wie Sie diese einsetzen und abwandeln können.

Außerdem haben Sie einen Überblick über die wichtigsten Werkzeuge erhalten und erfahren, dass Sie keinesfalls ein fertiges Programm schreiben. Vielmehr sind, auch nachdem Ihre Arbeit getan ist, zahlreiche Schritte notwendig, um ein lauffähiges Programm zu erstellen. Die wichtigsten Arbeitsschritte bei der Bedienung von Xcode und Interface Builder erläutere ich gleich im Anschluss in Kapitel 2. Weitere Kniffe und Tricks gibt es dann ganz am Ende in dem Xcode-Kapitel.

Und jetzt keine Müdigkeit vortäuschen und die Seite umschlagen.