Geschichte der Programmierung

1784 **Addiermaschine**, erbaut von Johann Helfrich von Müller
Menschen haben also schon seit langem das Bedürfnis, Rechnen maschinell zu erledigen. „to compute“ bedeutet „berechnen“, moderne Computer sind im Grunde nichts als sehr schnelle Rechner, die simple Grundrechenarten milliardenfach pro Sekunde ausführen.
Müller beschrieb außerdem die Idee einer Differenzmaschine.

1823 **Difference Engine**, ein mechanischer Computer, entworfen von Charles Babbage
Unterstützt von Mitteln der britischen Regierung, scheiterte jedoch, weil die nötigen Teile damals nicht präzise genug gefertigt werden konnten.
Die Differenz Engine konnte programmiert werden, unterschiedliche mathematische Funktionstabellen zu berechnen. Anwendungsbereiche in Militär und Wirtschaft.

1938 **Z1**, der erste **mechanische**, frei programmierbare Computer, erbaut von Konrad Zuse
Wie die Differenzmaschine konnte er Funktionstabellen berechnen.
Die deutsche Nazi-Regierung hatte (glücklicherweise) kein großes Interesse an Computern.
Nachfolgemodell Z3 (1941) mit elektromechanischen Relais

1945 **ENIAC**, der erste **elektronische** Computer (US-Army / University of Pennsylvania)
Zweck: Berechnung ballistischer Kurven von Artilleriegeschossen
Wurde u.a. von John von Neumann zur Entwicklung der Wasserstoffbombe verwendet.
Kosten: ca. 500.000 $, in heutiger Währung ca. 7 Mio. $
Verwendete Elektronenröhren und wog 27 Tonnen

1967 **FORTRAN**, eine verbreitete **imperative** Programmiersprache
von IBM für Wissenschaft und Technik entwickelt, mehrere Jahrzehnte in rechenintensiven Bereichen eingesetzt (z.B. Wettervorhersage, Kristallographie etc.)
Ein „imperatives“ Programm beschreibt, was Computer in welcher Reihenfolge tun soll.
Verwendet Kontrollstrukturen wie Schleifen u. Verzweigungen, Prozeduren

1971 **Pascal**, die erste **strukturierte** Programmiersprache von Niklaus Wirth (ETH Zürich)
Ziel: Vermeidung von Sprunganweisungen („GOTOs“), die zu schwer verständlichem Code führen. Pascal war eine Reaktion auf Softwarekrise 1960er: große Softwareprojekte scheiterten wegen hoher Kosten aufgrund der wachsenden Komplexität der Programme (dieses Problem ist bis heute nicht behoben).

1980 **Smalltalk**, die erste **objektorientierte** Programmiersprache
Entwickelt am Xerox PARC von Alan Kay, beeinflusste spätere Entwicklungen wie C++.

1985 **C++**, auch heute noch eine der verbreitetsten objektorientierten Programmiersprachen
Entwickelt von Bjarne Soustrup
Verbindet abstrakte Konzepte der OOP mit systemnaher, ressourcenoptimierter Programmierung. Wird besonders in der Systementwicklung eingesetzt (z.B. Linux).

1995 **Java**, eine **plattformunabhängige** OOP-Sprache, entwickelt von SUN Microsystems
Java-Programme laufen auf unterschiedlichen Betriebssystemen, z.B. Windows, Android
Heute noch sehr verbreitet, z.B. an Universitäten, für Android-Apps, Spiele (Minecraft)
Java und seine Entwicklungswerkzeuge sind kostenfrei verfügbar.

2000 **C# / .NET**, eine Weiterentwicklung von C++ und Java von Microsoft
Vieles automatisiert, was man bei Java noch „per Hand“ programmieren muss.
Allerdings weitgehend auf Windows beschränkt, daher Java immer noch weiter verbreitet.

Rollen in der Softwareentwicklung

In den ersten Jahrzehnten der Programmiersprachen wurden Computerprogramme meist für militärische und wissenschaftliche Anwendungen entwickelt, bei denen es um die Berechnung komplexer mathematischer Probleme ging. Programme wurden in der Regel von Einzelpersonen erstellt und von wenigen Menschen benutzt – die Entwickler waren selbst die Anwender ihrer Programme. Die Programme produzierten meist reine Rechenergebnisse, und brauchten nicht komfortabel bedienbar zu sein.

Heute sind Softwareprojekte oft sehr viel umfangreicher und komplexer. Systeme wie z.B. Google Maps werden von Millionen von Anwendern benutzt und verarbeiten gigantische Datenmengen. Für die Entwicklung so einer Software haben viele Menschen viele Jahre gebraucht, und die Entwicklung geht auch nach Jahren immer noch weiter.

Für große Softwareprojekte muss der Arbeitsprozess sehr viel strukturierter sein, als wenn ein Wissenschaftler für sich selbst ein Programm schreibt. In modernen Entwicklungsteams gibt es daher eine Reihe von unterschiedlichen Aufgaben und Rollen:

- Ein **Systemanalytiker** erfasst zunächst eine Liste von **Anforderungen** an eine neue Software. Er formuliert präzise, welche Aufgaben die Software erledigen soll und wie die Anwender diese bedienen werden. Anforderungsanalyse findet oft in Gesprächen mit Kunden statt und bildet eine Brücke zwischen den Vorstellungen der Kunden und den Programmierern.
  Die Anforderungsanalyse kann in Form eines Dokuments („Pflichtenheft“), oder mithilfe einer speziellen Software erstellt werden.
- Ein **Software-Architekt** entwickelt ein **Modell** für die neue Software. Er stellt in Diagrammen dar, wie die Software strukturiert ist, aus welchen Teilen (Klassen) sie besteht und wie diese Teile zusammenhängen.
- **Programmierer** schreiben anhand des Modells den **Quellcode**, also das eigentliche Programm in einer Programmiersprache.
- **Software-Tester** prüfen regelmäßig die Arbeit der Programmierer – denn es ist sehr schwierig, fehlerfreie Programme zu erstellen. Durch Tests soll die Zahl der Fehler möglichst gering gehalten werden.
- **Grafiker** erstellen Grafiken für die Software. Je nach Art der Software kann das ein kleiner oder auch ein sehr großer Anteil der gesamten Arbeit sein. Insbesondere bei Spielen kommen auch noch 3D-Modellierer hinzu.
- Ein **Projektmanager** organisiert den Arbeitsprozess dieser Mitarbeiter und ist dafür verantwortlich, dass der Zeitplan für die Entwicklung des Projekts eingehalten wird. Entwicklungsprozesse sind heute standardisiert, ein Beispiel ist SCRUM.
- Das **Support-Team** ist für verschiedene Bereiche zuständig:
  Hilfe bei der Installation der Software, Schulung von Mitarbeitern des Kunden, Erstellen von Handbüchern und Webseiten, Hilfe bei Schwierigkeiten und Problemen.

Vorteile der objektorientierten Programmierung (OOP)

Objektorientierte Programmiersprachen wurden entwickelt, um folgende Dinge zu unterstützen:

- **Modellierung** von Anwendungsbereichen und Programmteilen
  Eine Software hat oft einen Anwendungsbezug zu einer realen Situation, z.B. die Buchhaltung einer Firma. Das Programm muss Aspekte dieser Realität (Produkte, Kosten, Lieferanten usw.) abbilden. Außerdem besteht eine komplexe Software aus vielen Programmteilen.

  - OOP-Sprachen unterstützen, reale Aspekte und Softwarekomponenten abstrakt als Modelle darzustellen.

- **Aufteilen der Arbeit** an einem Softwareprojekt auf viele Mitarbeiter
  Eine Software wird heute meist von vielen Mitarbeitern erstellt, die an unterschiedlichen Teilen des Projektes arbeiten, z.B. an der graphischen Oberfläche, Netzwerk, Server-Software usw.

  - Durch die Unterteilung von Programmen in Module („Klassen“) wird ermöglicht, dass ein Programmierer seinen Teil des Projekts leicht mit einen anderen Teilen verknüpfen kann. Klassen definieren einfach verständliche „Schnittstellen“, so dass ein Programmierer sich nicht allzu sehr mit dem Quellcode seiner Kollegen beschäftigen muss. Das spart Zeit und vermeidet Fehler und Missverständnisse.

- Ermöglichen von **Änderungen** und **Erweiterungen**
  Softwareprojekte benötigen oft Jahre an Entwicklungszeit, und entwickeln sich auch danach laufend weiter. Während dieser langen Zeit werden oft von Seiten der Kunden bzw. Anwender Änderungen gewünscht.

  - OOP-Sprachen unterstützen Änderungen bzw. Erweiterungen dadurch, dass die einzelnen Module (Klassen) im Idealfall nur geringe Abhängigkeit voneinander haben. So kann man einzelne Klassen ändern / austauschen, ohne die ganze Software überarbeiten zu müssen.

- **Wiederverwendung von Quellcode**
  Bestimmte Teile von Software werden immer wieder in anderen Projekten benötigt. Durch den modularen Aufbau können diese Teile in anderen Projekten wiederverwendet werden, was Entwicklungszeit und damit Geld spart.