4. Februar 2013

Kapitel 1 - Einführung

- eigene Sprachen zu schaffen und damit die Grenzen ihrer Welt neu zu setzen
- eine Programmiersprache wird unsere Gewohnheiten, Paradigmen und Weltanschauungen über die Aufgabe des Programmierens ändern
- **Konzeptuelle Modelle**: verschiedene Ansichten was Information ist und was die Verarbeitung davon bedeutet
- **Anwendungsbereiche**: neue entstehen stetig so wie sich unsere Technologie verbessert (einen Texteditor nicht in Cobol, Tex nicht für einen Basic Interpreter)
- **Defizite bekannter Sprachen**: kleinere Probleme einer Sprache lassen sich mit Revisionen beheben, Trend alte Sprachen an neue Entwicklungen anzupassen (OO in allen Sprachen), bei der Notwendigkeit der Mischung/größere Abweichungen muss eine neue Sprache entstehen
- Programmiersprache zur Abschaffung aller Sprachen kann es nicht geben (Versuche: PL/I, Ada)
- Programmiersprachen können bestimmte Methoden der Softwaretechnik in abgestufter Weise ermöglichen, unterstützen, fördern, erzwingen (z.B. Module in Modula-2)
- Fragen für gefundene Fehler:
    1. Wie wurde der Fehler gefunden (Compiler, Laufzeitunterbrechung, Betrachtung)
    2. Wurde Debugger benötigt?
    3. Wäre der Fehler in einer anderen Sprache möglich?
- Nötige Schritte zur **Einschätzung von Konzepten in Programmiersprachen**:
    1. Das fragliche Konzept auf der höchstmöglichen Abstraktionsstufe beschreiben
    2. Den entsprechenden Teil der konkreten Syntax einer Sprache in Beziehung setzen zur abstrakten Syntax, indem wir entsprechende Abbildungen definieren:
       Eine Abstraktionsabbildung von der konkreten zur abstrakten Syntax und eine Implementierungsabbildung in der umgekehrten Richtung
    3. Bewerten in welchem Umfang die Sprache das Konzept verwirklicht (Einschränkungen/Erweiterungen)

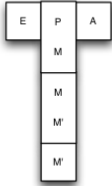
Kapitel 2 - Grundlagen von Sprachen und ihre maschinelle Behandlung

- Eine Programmiersprache hat drei Aspekte:
    - **Syntax**: definiert durch kontextfreie Grammatik, Unterscheidung zwischen konkreter und abstrakter Syntax, *(Beziehung der Zeichen untereinander)*
    - **Semantik**: die Bedeutung eines Programms als mathematisches Objekt, (*handelt von der Beziehung der Zeichen zur Welt)*
    - **Pragmatik**: Frage nach der sinngemäßen Handhabung der Sprache entsprechend ihrem Anwendungsbereich, *(handelt von der Beziehung der Zeichen zum Benutzer)*
- **Semantische Lücke**:

	Programmiersprache	Maschine
Datenstrukturen	komplizierte Datenstrukturen	primitive Maschinenworte
Kontrollstrukturen	ausgefeilte Kontrollstrukturen	bedingte und unbedingte Sprünge
Prozeduren	Lokale und globale Daten, Parameterübergabe, Rekursion	Unterprogrammsprung

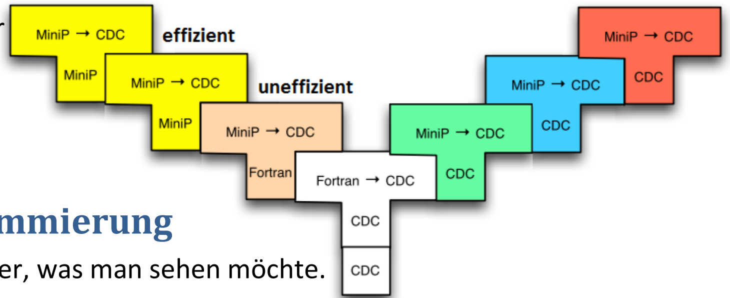
- **Interpretieren**: große Flexibilität (vor allem im Typsystem), Zuordnung von Fehlern ist einfach, allerdings fehlt es stets deutlich an Effizienz
- **Übersetzen**: Effizient, können größer und komplexer sein, starres Typsystem, Übersetzungsläufe dauern wegen Optimierungsmaßnahmen lange, komplizierter separater Debugger

- Compiler/Interpreter werden auch **Sprachprozessoren** genannt
- Zwischending zwischen Compiler & Interpreter (z.B. Java) tritt mit **virtuellen Maschinen** auf
  Compiler übersetzt das Programm zunächst in eine Zwischensprache (hebt sich aber von Maschinensprache noch deutlich ab), auf Zielmaschine wird die Sprache dann interpretiert



Bootstrapping

- Sprachprozessoren lassen sich mühelos in ihrer eigenen Sprache schreiben
- Ein Interpreter Ein Übersetzer
- Erstellung eines effizienten Compilers für MiniP der auf CDC läuft



Kapitel 3 - Paradigmata der Programmierung

- Ein Paradigma ist eine Entscheidung darüber, was man sehen möchte.
  Es macht einige Dinge zugänglicher indem es andere zurückdrängt.
  → kein einzelnes Paradigma wird zu jedem Problem passen
- Unterschiedliche Unterscheidung von Paradigmen
  Wegner: deklarativ, imperativ, interaktiv
  Jazayeri: prozedural, funktional, abstrakter Datentyp, modulbasiert, OO, generisch, deklarativ

Das imperative Paradigma (FORTRAN, Pascal, C)

- Wort-für-Wort-Verarbeitung auf einem in kleinen Einheiten gegliederten Speicher
- Programmieren durch Veränderung des Speicherzustands mit Zuweisungen
- maschinenorienterte Daten- und Programmstrukturen (dicht an der von-Neumann-Maschine)

Das funktionale Paradigma(HOPE, Haskell)

- Funktionen sind Objekte "erster Klasse" (sie können als Parameter übergeben werden)
- Baukastenprinzip zur Konstruktion von Funktionen
- Referentielle Transparenz (ein geschlossener Ausdruck hat immer den gleichen Wert)
- Polymorphie (Funktionen müssen keinen festen Typ haben)

Das relationale Paradigma (Prolog)

- Programmierer spezifiziert welche Fragen er gelöst sehen möchte (nicht aber wie)
- Keine Berechnung im klassischen Sinne
- Programmausführung wird als Ableitungs- bzw. Beweisprozess gesehen

Das objektorientierte Paradigma (in Reinform: Smalltalk)

- bestehend aus interagierenden Objekten (mit innerem Zustand (Attribute), bieten Leistungen (Methoden), senden Nachrichten (Aufrufe von Methoden))
- Hierarchische Organisation (Klassen können Oberklassen und Unterklassen haben, erben)
- Attribute-basierte Antworten: Objekte interpretieren aufgrund lokaler Eigenschaften (Instanzvariablen) und globaler Eigenschaften (Klassen)

Kapitel 4 - Entitäten, Namen und Bindungen

- Unterscheidung zwischen Variablen (in von-Neumann-Sprachen)
    1. der Speicherplatz selbst als Behältnis für Daten
    2. die in dem Speicherplatz abgelegten Daten,
    3. die Adresse des Speicherplatzes und
    4. der Name, unter denen die Speicherplatz in der Programmiersprache angesprochen werden kann.
- Sperber: Bezeichner die für Speicherplätze stehen, die Werte enthalten nennt man Variablen

Exkurs: Der Lambda Kalkül

- **Normalformen**: Ein λ-Term e' ist eine Normalform von e, wenn $e \rightarrow_{\beta}^{*} e'$ gilt und kein λ-Term e'' existiert mit $e \rightarrow_{\beta}^{*} e''$. (ein λ-Term hat höchstens eine Normalform nach Alpha-Konversion)
- **Verzweigungen**: $true \stackrel{\text{def}}{=} \lambda xy.x$, $false \stackrel{\text{def}}{=} \lambda xy.y$, $if \stackrel{\text{def}}{=} \lambda txy.t\ x\ y$
- **Natürliche Zahlen**: (komplizierter Aufbau)
- **Rekursion und Fixpunktsatz**: TODO (Seite 55ff)

Auswertungsmodelle

- **Linksaußen-Reduktion** ($\rightarrow_{\beta o}$): nur auf β-Redexe die möglichst weit links außen stehen
  Wenn e' eine Normalform von e ist, so gilt $e \rightarrow_{\beta o}^{*} e'$. (findet JEDE Normalform (nicht effizient))
- *Abstraktionen heißen schwache Kopfnormalformen* ??
- **Call-by-Name-Auswertung** ($\rightarrow_{\beta n}$): wie oben, nur dass bereits bei schwacher Kopfnormalform aufgehört wird (findet immer Kopfnormalform (auch nicht effizient))
- **Linksinnen-Reduktion** ($\rightarrow_{\beta i}$): ugs. es werden nur Werte eingesetzt (Endlosschleifen)
- **Call-by-Value Reduktion** ($\rightarrow_{\beta v}$): w steht für Wert und e für Nichtwert. $(\lambda v.e)\ w \rightarrow_{\beta v} e[v \mapsto w]$
  (Darf nur in einem Gesamtterm angewendet werden, wenn nicht in schwacher KopfNF)
- *Call-by-Value Sprachen/strikte Sprachen: C, Java, Pascal, ML, ...*

Bindungen in Programmiersprachen

- **Zeitpunkt der Bindung**:
    1. Laufzeit (zu beliebigen Zeiten während der Ausführung, beim Start, Eintritt in Block)
    2. Übersetzungszeit (vom Programmierer oder Compiler gewählt)
    3. Sprachimplementierungszeit
    4. Sprachdefinitionszeit
- **Dauer der Bindung**:
    1. statisch (lexikalisch)
    2. dynamisch

Zeitpunkt	*Dauer*
Beliebiger Zeitpunkt	beliebig kurz
Eintritt in einen Block	bis zum Austritt aus dem Block
Start des Programms	bis zum Ende des Programms
Übersetzungszeit	über alle Programmläufe
Sprachimplementierungszeit	über alle Compilationen dieses Systems
Sprachdefinitionszeit	über alle Systeme und Compilationen

- Üblicherweise werden Typen zur Übersetzungszeit statisch gebunden (C, Pascal)
  Werte allerdings meist dynamisch

Attribute von Variablen, Deklarationen

- implizite Deklarationen durch Namenskonventionen (z.B. alles was nicht auf $ endet ist Real (BASIC))
- vorgegebene Deklarationen: man kann Typen auch neudefinieren in Modula-2
- explizite Deklarationen

Sichtbarkeitsbereich (Scope): Bereich in denen der Bezeichner bekannt ist. (Betrifft Variablen und Prozeduren), Unterscheidung zwischen statischen (lexical scope) und dynamischen Sichtbarkeitsbereichen.

- statische Sichtbarkeitsbereiche: einfache Handhabung für Programmierer, ohne Ausführung nachvollziehbar, wirft Probleme bei Implementierung auf
- dynamische Sichtbarkeitsbereiche: leicht zu implementieren, schwierig zu handhaben

Lebenszeit (extent): das Zeitintervall einer Variablen in der ihr eine Adresse zugeordnet ist

- statische Allokation: Allokation zur Übersetzungszeit (Lebenszeit ist gesamte Ausführung)
- dynamische Allokation: Allokation zur Laufzeit
    - stapelbasiert: Variablen werden beim betreten der Prozedur alloziert
    - frei programmierbar: beliebig vom Programmierer festlegbar

Konstanten: Konstanten als Präprozessor die einfach deren Inhalt einsetzen ohne ihn Auszuwerten bescheren erhebliche Probleme.

Kapitel 5 - Laufzeitorganisation

- Betriebssysteme sind eher Großhändler für Speicher, sie geben diesen nicht variablenweise ab, sondern nur in größeren Stücken
- Programm muss auf allozierte Variablen zugreifen können. Dies sollte von der Sprache und nicht vom Betriebssystem abhängen.
- **Linker**: kennt alle Bindeeinheiten und vergibt Adressen relativ zum Programmanfang
- **Compiler**: kennt Adressen innerhalb einer bestimmten Übersetzungseinheit (Offset)
- **Lader**: passt nach fertigem linken des Programms noch einmal die Adressen im Hauptspeicher an, lädt DLLs (dynamic link libraries) nach und benötigt dazu die Fähigkeiten des Linkers

P-Code

- Der Instruction Pointer (IP) zeigt jeweils auf die nächste auszuführende Instruktion im Programmspeicher (P).
- Im Laufzeitstapel (runtime stack (RS)) werden die Aktivierungsblöcke der Funktionen gespeichert, sie enhalten lokale Variablen. Der "frame pointer" FP zeigt auf den gerade relevanten Aktivierungsblock. Der "stack pointer" SP zeigt auf den ersten freien Platz des RS.
- Rechnungen werden auf dem Operandenstapel (OS) durchgeführt

Stapelbasierte Allokation

Kümmern sich um die Verwaltung der dynamischen Verweise beim betreten und verlassen von Prozeduren.
Prolog: Anlage eines neuen Aktivierungsblocks, alten FP in Stapel (DL) retten.
Epilog: Wiederherstellen des alten SP aus dem FP dann wiederherstellen des alten FP aus dem DL (dynamic link)

Haldenbasierte Allokation

Durch Freigabe von Speicher kommt es zwangsweiße zu Lücken im Speicher.

- **first fit**: alloziere den ersten Speicherblock der passt
- **best fit**: alloziere den Speicherblock der am besten passt (keine Lücken hinterlässt)

Haldebasierte Allokation wird durch nebenläufige Programmierung oder Objektorientierung "erzwungen". Einmal halten die Daten länger und einmal werden Aktivierungsblöcke nicht unbedingt in der selben Reihenfolge freigegeben.

Speicherverwaltung

- Probleme: Durch häufiges Allozieren und Deallozieren werden viele Fragmente erzeugt, Nicht mehr erreichbare Speicherblöcke → Garbage Collection
- Varianten der **Garbage Collection**
    1. *Mark and sweep*: Markierungsphase (erreichbare Blöcke werden markiert), Bereinigung (alle nicht markierte werden freigegeben)
    2. *Reference counting*: es wird für jeden Block vermerkt wie viel im Moment darauf zeigen
    3. *Copying garbage collection*: es gibt eine aktive und eine inaktive Halde, aktive Blöcke werden in die inaktive Halde kopiert und die inaktive als aktiv markiert, alle Zeiger werden dementsprechend angepasst

Aktivierungsblöcke

Es gibt eine Unterscheidung:

- Aktivierungsblöcke **ohne dynamische Felder**: alle Aktivierungsblöcke eines Blocks P sind gleich groß, alle lokalen Variablen in P befinden sich in jedem Block an derselben Stelle (prozedurstatische Variablen genannt)
- Aktivierungsblöcke **mit halbdynamischen Feldern**: zur Allokation ist die Größe des Blocks berechenbar, ebenso die Offsets der dynamischen Felder und bleiben während der Lebensdauer des Blocks konstant
- Aktivierungsblöcke **mit dynamisch veränderbarer Größe**: entsteht zum Beispiel durch flexibel lange Arrays, stapelbasierte Allokation ist hier nicht mehr möglich → Halde

Zugriff auf nichtlokale Größen

- **statische Verweiskette**: es gibt einen Verweis auf den Aktivierungsblock der die Funktion aufgerufen hat, damit kann man beliebig "zurückhüpfen", Variablen sind vom Compiler durch Paare beschrieben (Niveaudifferenz, Offset). (bei aktuellem Block Niveaudifferenz = 0)
- **globales Display**: Variablen beschrieben durch (Niveaustufe, Offset), Prozedurprolog erfolgt nun durch enter i, z (i für Schachtelungstiefe, z für die Anzahl an Variablen), Epilog leave i
- **lokales Display**: Aktivierungsblock erhält Adressen der anderen Aktivierungsblöcke darüber (eine Art Caching), der Prolog wird dadurch aufwendiger, Epilog bleibt gleich

	Stat. Verweis	Glob. Display	Lok. Display
Prolog	$1+d$	3	$2(m-1)$
Variablenzugriff	$d+1$	2	2
Epilog	0	2	0

Funktionen als Werte, Closures

Closures: Eine Art Funktion die in einer Variablen gespeichert und weitergereicht werden kann. Sie wird als Paar aus Zeiger auf dem Code und Zeiger auf Umgebung repräsentiert. *(Verbesserung: es werden Variablen dir nur von der Closure gebraucht werden mit in die Closure kopiert, damit ist schellerer Zugriff beim Benützen möglich und die Garbage Collection kann einfacher erfolgen)*

Kapitel 6 - Prozedurale Abstraktion

Parameter-Zuordnungsmethoden

Wie werden Parameter einander zugeordnet?

1. Zuordnung durch Stellung innerhalb der Argumentliste (üblich)
2. Zuordnung durch Namen
   Vorteil: verhindert Irrtümer, weggelassene Parameter können mit Defaultwerten gefüllt werden, beliebige Reihenfolge der Parameter möglich

Parameter-Übergabemechanismen

In welcher Gestalt werden Parameter an die Prozedur übergeben und Resultate zürckgegeben?

1. **Übergabe durch Referenz**:
   - Vorteile: Einfachheit, Effizienz, Uniformität
   - Nachteile: Konstanten als Parameter nicht möglich
2. **Übergabe durch Kopie**:
   - Call by value: (wie in C)
   - Call by result: es wird eine Variable angegeben in die beim Rücksprung ein Resultatwert kopiert wird
   - Call by value-result: Der Wert der übergebenen Variablen wird kopiert und danach zurückkopiert (das schützt die Referenzen) (ACHTUNG: ist nicht call by refrence!!)
3. **Übergabe durch Namen**: (Call by name)
   - Jedes Vorkommen eines formalen Parameters im Rumpf einer Prozedur führt dazu, dass dieser Parameter erneut ausgewertet werden muss
   - Jede Auswertung erfordert Prozeduraufruf
   - Jede Auswertung kann anderes Ergebnis liefern

Parameter-Auswertungsstrategien

Wann und in welcher Reihenfolge werden sie ausgewertet?
Call by value wertet Ausdrücken von innen her aus. Call by Name verbietet dies.

Impementierung der Parameterübergabe

Register: schnellste Methode, nur geringe Anzahl möglich, wenn eine Prozedur andere aufruft, müssen die Werte gerettet werden und die Effizienz geht Flöten

Parameterblock: Es wird ein Vektor im Hauptspeicher angelegt. (früher bei IBM-Großrechnern üblich)

Stapel: Heutzutage die bevorzugte Methode in Modernen Architekturen.

Pascal-Methode: Der erste Parameter wird zuerst geschoben. Die aufgerufene Prozedur entfernt die Parameter (mit ihren Lokalen Variablen inkl. Verwaltungsinformationen)

C-Methode: Der letzte Parameter wird zuerst geschoben. Die aufgerufene Prozedur entfernt die Parameter wieder vom Stapel nachdem die aufgerufene Prozedur zurückgekehrt ist. (*wäre in C nicht anderst möglich, da die Anzahl der Parameter nicht festgelegt ist und nur die aufrufende Prozedur weiß wie viel Parameter sie übergeben hat*)

Prozeduren als Parameter

Probleme: Variablen in übergebenen Prozeduren benötigen besondere Beachtung. Hier ist der statische Verweis zwingend von Nöten.
Damit eine Prozedur p in der Prozedur q übergeben werden kann, muss gelten:

1. p wurde selber als Parameter übergeben: Hier kann es einfach weiterkopiert werden
2. p wurde nicht als Parameter übergeben: Schachtelungsdifferenz von p und q wird berechnet und mit der Verfolgung der statischen Verweiskette der richtige Aktivierungsblock ermittelt und dessen Adresse als statischer Verweis übergeben.

most-recent-Eigenschaft: ??

funcarg-Problem: Es wurde dadurch gelöst indem ein eigenes Schlüsselwort function eingeführt wurde, um Parameter von ansonsten gleich aussehenden Listen zu unterscheiden.

Kapitel 7 - Typen und Typsysteme

- **Datentypen**: Mengen von Werten zusammen mit Mengen von Operationen auf diesen Werten. (bei abstrakten Datentypen ist sogar die innere Struktur der Werte nicht relevant)
- **Typsysteme**: sind logische Kalküle, die sicherstellen, dass bei Funktionsaufrufen die Argumente zu den Operationen passen.
- **Sicherheitsgurte für Programmierer**:
  Vorteile: Sicherheit vor unerlaubten Manipulationen, Verbesserung der Portabilität, Möglichkeit der Überprüfung des korrekten Variablengebrauchs schon zur Übersetzungszeit, Auflösung überladener Operatoren
  Nachteile: extra Deklarationen nötig (zusätzliche Planung), Programme werden ineffizient, Typensystem verhindert Flexibilität bei Entwicklung von Prototypen.
  Gegenargumente: extra Deklaration braucht man sowieso, Compiler optimieren besser als Coder, getypte Sprachen können auch Polymorphie anbieten

Typen als Strukturbeschreibung

- **untypisierte Sprachen**: mit Grundoperationen kann Datenstruktur beliebig manipuliert werden (Assembersprachen)
- **latent typisierte Sprachen**: kein sichtbares Typsystem, weißt aber zur Laufzeit Typen zu und wirft bei fehlerhafter Verwendung Fehler (Skriptsprachen)
- **statisch typisierte Sprachen**: Typen werden beim Übersetzen bereits untersucht, trotzdem können noch Fehler auftreten. Erst mit der Garantie, dass ein Programm später keine Operationen auf ungeeigneten Eingabedaten ausführt ist das erreicht. Solche Sprachen nennt man typsicher (bzw. stark typisiert). Beispiele sind ML und Haskell. Ohne Casts würde auch Java als typsicher gelten.

Strukturen von Werten: Elementare, Zusammengesetzte, Gemischte, Rekursiv definierte Daten

Typsprachen und Deskriptoren

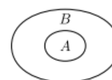
Typdeskriptor (TD):

- TD=TA: Zielsprache ist Hochsprache
- TA→TD im Backend: Große Teile des Compilers sind unabhängig vom tatsächlichen Speicherlayout, daher kann die Erzeugung eines Typdeskriptor bis in die Code-Generierungsphase verschoben werden.
- TD zur Laufzeit: Werte enthalten einen Verweis auf den Deskriptor ihres Typs.

Taxonomie von Datentypen

1. **Primitive Datentypen**: boolean, state, enumerated, character, ordinal, time, integer, rational, scaled, real, complex, void (hierarchische Ordnung, ab ordinal unendlicher Wertebereich)
2. Untertypen und erweiterte Typen (subtype)
3. Erzeugte Datentypen (nach ISO): Auswahl (choice), Zeiger (pointer), Prozedur (procedure), Aggregat-Datentypen (record, class, set, bag, sequence, array, table)
4. Definierte Datentypen (benutzerspezifisch)

Einzelne Datentypen



Subtypen: Formal ist A ein Subtyp wenn sein Definitionsbereich eine Teilmenge vom Supertyp B ist.

Coercion: Das sind automatisch eingefügte Konversionen die den Subtyp s an den Supertyp t in der Darstellungsweiße (Bitmuster) anpassen. (c: $s \rightsquigarrow t$) (Transitiv)

Casts: Zwingt den Compiler den Typ zur Übersetzungszeit zu ändern. (in Java werden Coercions auch als Cast bezeichnet)

Referenzen und Zeiger: Notwendig um indirekt zu adressieren und um rekursive (dynamische) Datentypen zu ermöglichen (beides fragwürdige Argumentationen). Übliche Fehler:

- *Fehlende Initialisierung*
- *Ungültige Zeiger*: Wenn ein Nullzeiger dereferenziert werden will wird Assertion geworfen
- *Memory Leaks*: Speicher wird nicht freigegeben, irgendwann ist der Speicher voll
- *Dangling References*: Zeiger zeigt auf ungültiges Datenobjekt
- *Zeigerarithmetik*: out-of-bounds Zeiger zu weit geschoben

Overloading: Wir schreiben + für die Addition und das konkatenieren von Strings. Jeweils nur aus dem Kontext ersichtlich welche Variante im Moment gewählt werden soll. (Überladung)

Polymorphie: Ein Wert hat mehrere Typen. (Überladung, Coercion, ...)

Kapitel 8 - Modulkonzept

- **Geheimnisprinzip**: Verhindert, dass innerhalb einer Übersetzungseinheit Details einer anderen Einheit verwendet werden, die nicht in der Schnittstellenbeschreibung aufgeführt sind.
- Unterscheidung zwischen Unabhängigen Übersetzungen (keine Prüfungen über die Grenzen von Übersetzungseinheiten hinweg) und Getrennten Übersetzungen
- Dynamisches Linken:
    - Jedes Modul wird erst geladen wenn es wirklich gebraucht wird, damit wird der Speicherverbrauch potentiell reduziert (allerdings kann es später zu Verzögerungen durch das Nachladen kommen)
    - Ohne inneren Zustand können Bibliotheken zwischen verschiedenen Tasks aufgeteilt werden (bei statisch würde jeder eine eigene Kopie benötigen)
- **Konsistenz von Schnittstelle und Implementierung**: Wenn sich das Modul ändert das eingebunden wird, so treten Inkonsistenzen auf die dadurch gelöst werden, dass Timestamps oder Prüfsummen verglichen werden.

Kapitel 9 - Objektkonzept

Mehrfachvererbung: Ist im Prinzip nicht nötig. Das was Java mit Interfaces kann ist das, was die Mehrheit sich von Mehrfachvererbung verspricht, nur dass es weit weniger problematisch ist. (Probleme: gleiche Methoden/Variablennamen, wer überschreibt?)

Rest trivial.

Kapitel 10 - Ausnahmebehandlung

Ausnahme/Exception: Eine Situation, die es für das Programm unmöglich macht mit der normalen Verarbeitung fortzufahren.

Klassifikation von Ausnahmen:

- **Harte Ausnahmen**: Ausnahmen die von der Hardware/Betriebssystem geworfen werden. (z.B. Speicherschutz-Verletzung, ungültiger Instruktionscode) Vorrausgegangen ist meist, dass Daten überschrieben wurden.
- **Sprach-Ausnahmen**: Typfehler, Indexüberschreitung, ...
- **Bibliotheks-Ausnahmen**: (z.B. Platte voll, Speicher erschöpft, Zieloperand zu klein)
- **Anwendungs-Ausnahmen**: (z.B. eine Indexdatei ist inkonsistent, eine erwartete Zahleneingabe enthält Buchstaben)

Arten der Reaktion auf Ausnahmen: Programm abschießen, Ausnahme ignorieren

- **Benutzer-Information**: Benutzer ist nicht in der Lage Fehler zu interpretieren. Daher sollte das Programm über weiteres Vorgehen "verhandeln".
- **Sanfter Tod**: Bei brutalem Stopp verbleiben Programmteile im Schwebezustand. (z.B. offene Dateien, Datenbanktransaktionen, gehaltene Modemverbindungen, verwendete Ressourcen)
- **Robustheit des Programms**: Es muss in der Lage sein bestimmte Transaktionen zu wiederholen und den Programmlauf fortfahren!

Modelle bei Ausnahmen:

- **Terminationsmodell**: Der Block, in dem die Ausnahmesituation auftritt wird unwiderruflich beendet und Ausnahmebehandler reagiert nach Programmiererwillen.
- **Wiederaufnahmemodell**: Der Block wird nur unterbrochen und nach Ausnahmebehandlung fortgesetzt. (Unterscheidung zwischen Fortsetzungs- und Wiederholungsmodell)

Programmieren ohne Ausnahmebehandlung: Benötigt extensives Testen, Beweisen von Programmeigenschaften, Erfolgsparameter, Haken