

SE2 Zusammenfassung

Inhaltsverzeichnis

Klausurinformationen.....	3
UML Cheatsheet.....	4
 [02] Modul, Tests, Vertragsmodell.....	5
Problemraum / Lösungsraum.....	5
Fehlerbegriffe (Mistake/Fault/Failure/Error).....	5
Defensiver Programmierstil	6
Vertragsmodell.....	6
Asserts	6
Testen	7
 [03] Polymorphie und Typhierarchien	8
Vererbung.....	8
Überladen / Redefinieren / Redeklarieren.....	8
 [04] Implementationsvererbung	9
this / super.....	9
Abstrakte Methoden / Abstrakte Klassen	9
Schablonenmuster.....	9
final.....	10
Vererbung und Konstruktoren	10
Schnittstellensichtbarkeit (public / protected / package private / private).....	10
 [05] Fehlerbehandlung, Exceptions, Module	11
package.....	11
Importieren aus Packages	11
Entwicklungsfehler / Umgebungsfehler	11
Exceptions	12
Checked / Unchecked Exceptions	12

[06] WAM, Beobachtermuster	13
Arten von Softwaresystemen	13
WAM-Ansatz.....	13
Beobachtermuster.....	14
[07] Modellierung, Entwurf	15
Entwurfsmuster	15
[08] GUI-Programmierung, Lambda-Ausdrücke.....	16
Swing-Komponenten	16
Container	17
Events	17
Lambdas	18
[09] OO Entwurfsprinzipien.....	19
Entwurf	19
Kopplung.....	19
Kohäsion	20
Law of Demeter	20
CRC-Karten	20
SOLID	21
[10] Fachwerte und Werttypen.....	22
Fachliche Werte.....	22
Gleichheit für Objekte	22
[11] Refactoring.....	23
Bad Smells	23
Refactoring	23
[12] Korrektheit und Formalisierung.....	24
Zuverlässige Software.....	24
Semantik	24
Abstrakte Datentypen	25

[13] Funktionale Programmierung	26
Threads	26
Funktionale Programmierung	26
Pure Funktion	27
Racket	27
Werte	27
Funktionen	27
Bedingte Ausdrücke	28
Namensbindung / Variablen	28
Listen	29
Map Funktion	29
Filter Funktion	29

Klausurinformationen

Die 1. Prüfung findet am 02.08.2021 um 10 Uhr statt. Die zweite Prüfung findet am 06.09.2021 um 10 Uhr statt.

Um 10:00 Uhr werden wir am Prüfungstag mit den Ansagen über das Zoom-Meeting starten. Wenn alles geklärt ist, werden die Tests freigeschaltet, und die Bearbeitungszeit von 95 Minuten beginnt. Nach der Prüfung müsst ihr die Eigenständigkeitserklärung innerhalb von 30 Minuten abgeben. Es handelt sich hierbei um ein PDF, was digital ausgefüllt werden kann. Nur dann können wir die Prüfung auch werten. Ohne diese Erklärung gilt die Prüfung dann als nicht bestanden.

Die gleichen Aufgabentypen wie im Semester, also CodeRunner, Drag & Drop, Drop-Down, Multiple-Choice usw. Es wird keine Aufgaben geben, bei denen etwas hochgeladen werden muss. Wir behalten uns vor, dass es ggf. auch Freitextfelder geben wird.

Innerhalb einer Aufgabe können aber auch Minuspunkte für falsche Teilantworten anfallen. Die minimale Punktzahl für eine Aufgabe beträgt jedoch stets 0 Punkte.

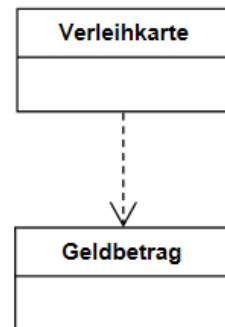
UML Cheatsheet

Allgemeine Abhängigkeit (UML: Dependency)

Jede Form von Benutzt-Beziehung; in Java: Auftreten eines Typnamens im Quelltext (vor allem bei Parametern, Rückgabetypen und lokalen Variablen)

-----> *Gestrichelte Pfeillinie, offene Pfeilspitze*

```
class Verleihkarte
{
    public Geldbetrag getMietgebuehr()
    {
        return new Geldbetrag(0);
    }
}
```

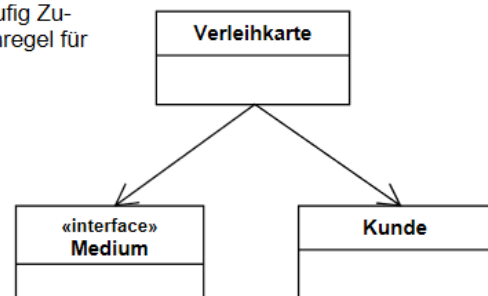


Strukturelle Abhängigkeit (UML: Association)

Eine etwas strengere Abhängigkeit als die allgemeine; modelliert häufig Zusammenhänge wie "enthält", "besteht aus" oder "verwaltet"; Daumenregel für Java: Exemplarvariablen weisen auf strukturelle Abhängigkeiten hin.

————> *Durchgezogene Pfeillinie, offene Pfeilspitze*

```
class Verleihkarte
{
    private List<Medium> _medien;
    private Kunde _kunde;
}
```

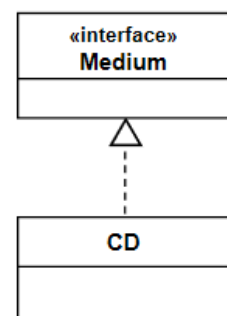


Implementation eines Interfaces (UML: Realization)

Umsetzung einer Spezifikation, Realisierung einer Schnittstelle

-----> *Gestrichelte Pfeillinie, geschlossenes Dreieck als Pfeilspitze*

```
class CD implements Medium
{
    ...
}
```

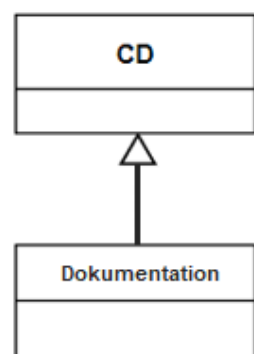


Erweiterung einer Klasse / eines Interfaces

Unterklassen enthalten Implementation ihrer Elternklassen.

————> *Durchgezogene Pfeillinie, geschlossenes Dreieck als Pfeilspitze*

```
class Dokumentation extends CD
{
    ...
}
```



[02] Modul, Tests, Vertragsmodell

Problemraum / Lösungsraum

Problemraum	Lösungsraum
<i>Konzeptionelle Beschreibung, Anforderungen („was“)</i>	<i>Technische Lösung, Realisierung („wie“)</i>
Bsp.: Speichere die Freunde einer Person ab.	Benutze zu Speicherung ein Array.

Nachteile einer Vermischung:

- Vorgriff auf Lösungen führt zu Detailentscheidungen,
- schränkt Freiheitsgrad bei Suche nach optimalen Lösungen ein

Vorteile einer Trennung:

- Explizite Entscheidungen mit besseren Lösungen

Fehlerbegriffe (Mistake/Fault/Failure/Error)

Mistake: Menschliche Fehlhandlung, die zu einem Fehler (*Fault*) führt

Fault: Fehler im Quelltext der Software (*Codierungsfehler, Bug*)

Failure: Versagen eines Systems aufgrund eines oder mehrerer Faults

Error: Die Realisierung entspricht nicht der Forderung



Achtung: Java-Klasse `Error`

modelliert gravierende Fehlerzustände, die nicht vom Programmierer bearbeitet werden sollten, sondern zum Programmabbruch führen

Defensiver Programmierstil

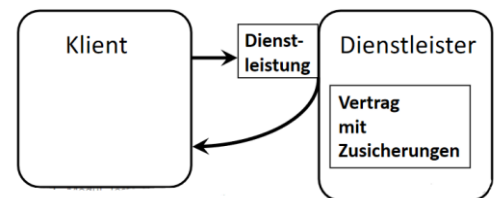
Theorie	Praxis
Risiken vermeiden	Test-first-Arbeitsweise
Anwesenheit von Fehlern immer erwarten	Vorbedingungen mit <code>assert</code> prüfen
Fehlerreaktion auf minimale Auswirkungen richten	Weiterarbeit wo möglich, in fehlerarmen Zustand gehen
Auswirkungen von Fehlern verringern / vermeiden	Tests für Nachbedingung gegen Fehlerfortpflanzung

Vertragsmodell

Ein **Klient** benutzt einen Service eines **Dienstleisters**.

Der Vertrag beschreibt:

- welche Vorleistung Klient erbringen muss
- damit Dienstleister seine Dienstleistung garantiert



Vorbedingungen:

Sollen **vor** Ausführung der Operation vom **Klienten** eingehalten werden.

Werden über `assert` geprüft und im Quelltextkommentar mit `@require` gekennzeichnet.

Nachbedingungen:

Sollen **nach** Ausführung der Operation vom **Dienstleister** erfüllt sein.

Werden über *JUnit-(Positiv-)Tests* geprüft und im Quelltextkommentar mit `@ensure` gekennzeichnet.

Invarianten:

Bedingungen, die immer gelten sollen, als Klasseninvarianten. Allgemeine Randbedingung des Vertrags, muss bei jedem Operationsaufruf gelten.

Asserts

Zwei Varianten:

`assert [Boolean];`

Wirft nur Fehler.

`assert [Boolean] : [String];`

Wirft Fehler mit Fehlermeldung.

Testen

Testfall	Beschreibung der erwarteten Ausgabedaten für bestimmte Eingabedaten
Positiv-Test	nur erwartete/gültige Eingabewerte getestet und Ausgabedaten erwartet -> erhöhen Vertrauen in Korrektheit
Negativ-Test	unerwartete/ungültige Eingabewerte getestet und Fehlerbehandlung erwartet -> erhöhen Vertrauen in Robustheit

Im Vertragsmodell werden die Nachbedingungen über **Positiv-Tests** geprüft (eigentlich brauchen wir **Negativ-Tests** gar nicht).

Positivtests müssen, **Negativtests** können sein!

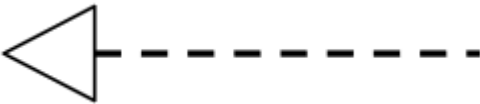
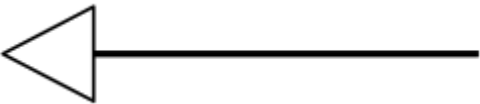
Korrektheit muss, **Robustheit** kann geprüft werden

Testgetriebene Entwicklung:

Bevor man eine Klasse oder Methode schreibt, zuerst eine zugehörige Testklasse schreiben.
Jede Klasse soll seine eigene Testklasse haben.

[03] Polymorphie und Typhierarchien

Vererbung

Typ-Vererbung	Implementationsvererbung
<i>implements [Interface]</i>	<i>extends [Klasse/Interface]</i>
Klassen können beliebig viele <i>Interfaces</i> implementieren.	Klassen können je eine <i>Klasse</i> extenden. <i>Interfaces</i> können beliebig viele <i>Interfaces</i> extenden.
Schnittstelle wird vererbt. Alle Methoden müssen implementiert werden.	Unterklassen enthalten Implementation ihrer Elternklassen .
Operationen eines Supertyps können auch an Subtypen aufgerufen werden.	Unterklassen sind Spezialisierung ihrer Elternklassen .
Variablen des Supertyps können mit Referenzen auf Objekte des Subtyps belegt werden.	Elternklassen abstrahieren von Unterschieden ihrer Unterklassen .
 <p>Klassen zeigen auf Interfaces</p>	 <p>Subtypen zeigen auf Supertypen</p>

Typen und Vererbung sollten nach der echten Welt modelliert werden.

- > bessere Verständlichkeit
- > leichter erweiterbar

Überladen / Redefinieren / Redeklarieren

Überladen	Anbieten einer neuen Operation mit gleichem Namen <pre>void run(); void run(String text);</pre>
Redefinieren	Ändern der Implementation einer Operation in einer Subklasse. Implementation der Oberklasse wird dabei überschrieben. -> Gleiche Signatur! <pre>equals()</pre> wird von Klassen häufig redefiniert.
Redeklarieren	Ändern der Signatur einer Operation in Unterklasse. Schnittstelle wird dabei geändert.

[04] Implementationsvererbung

this / super

Mit **super** kann, bei Redefinition einer Operation die Methode der Oberklasse aufgerufen werden.

Damit wird die Methode **erweitert** anstatt komplett überschrieben / redefiniert zu werden.

Es ist nur die Implementation der direkten Oberklasse erreichbar.

-> kein **super.super.run()**

super() ruft den Konstruktor der Oberklasse auf.

this beschreibt die Klasse, in der die Methode aufgerufen wird.

```
class Konto {
    public void macheJahresabschluss()
    {
        _saldo = _saldo - _gebuehren;
    }
    ...
}

class Sparbuch extends Konto {
    @Override
    public void macheJahresabschluss()
    {
        super.macheJahresabschluss();
        _saldo = _saldo + berechneZinsen();
    }
    ...
}
```

Abstrakte Methoden / Abstrakte Klassen

Abstrakte Methode

Oberklasse deklariert eine Operation, ohne eine Implementation anzugeben. Die *Oberklasse* ist damit eine **abstrakte Klasse**.

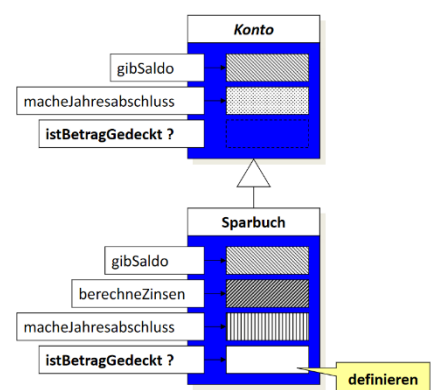
Wenn *Unterklassen* alle *abstrakten Methoden* implementieren, sind diese nicht abstrakt sondern *konkrete* („normale“) *Klassen*.

Abstrakte Methoden werden mit Keyword **abstract** gekennzeichnet

Abstrakte Klassen

Klassen mit Keyword **abstract** sind *abstrakte Klassen*.

Von *abstrakten Klassen* können keine Exemplare erstellt werden.



Konto ist abstrakt, Sparbuch konkret

In UML werden konkrete Klassen normal geschrieben, abstrakte Klassen *kursiv*.

Schablonenmuster

Superklassen geben in **Schablonenmethode** (Template Method) allgemeines Verhalten fest. Dabei werden häufig **Einschubmethoden** (Hook Methods) aufgerufen, die in der Superklasse leer sind.

Die **Hook Methods** geben Stellen zum Einschieben für vorgesehene Konkretisierung und sollen von den Subklassen implementiert werden.

```
public abstract class Game {
    //Einschubmethoden
    abstract void initialize();
    abstract void startPlay();
    abstract void endPlay();

    //Schablonenmethode
    public final void play(){

        //initialize the game
        initialize();

        //start game
        startPlay();

        //end game
        endPlay();
    }
}
```

final

Der Wert einer als `final` deklarierten **Variable** kann nicht geändert werden.

Achtung: Arrays oder andere Speichertypen können immer noch ihre Belegungen ändern.

Eine als `final` deklarierte **Methode** kann nicht von einer Unterklasse redefiniert werden.

Von einer als `final` deklarierten **Klasse** können keine Unterklassen definiert werden.

Vererbung und Konstruktoren

Konstruktoren werden nicht vererbt.

Bei Erzeugung eines Objekts einer abgeleiteten Klasse werden die Konstruktoren **aller Oberklassen** aufgerufen. Dies geschieht an erster Stelle im Konstruktor und kann explizit mit `super()` gestartet werden. Wird dies nicht explizit geschrieben, wird das `super()` implizit eingefügt.

Schnittstellensichtbarkeit (`public` / `protected` / `package private` / `private`)

Schnittstellen können verschiedene Sichtbarkeit haben.

public

Schnittstelle für **Alle**.

Jede Klasse kann darauf zugreifen.

protected

Schnittstelle für **Erbende**.

Nur die Klasse selbst, erbende Klassen und Klassen im selben Package können darauf zugreifen.

Kein Keyword („package private“)

Schnittstelle für **Klassen im selben Package**.

Nur die Klasse selbst und Klassen im selben Package können darauf zugreifen.

private

Schnittstelle für **Niemanden**.

Nur die Klasse selbst kann darauf zugreifen.

	Class	Package	Subclass (same pkg)	Subclass (diff pkg)	World
<code>public</code>	+	+	+	+	+
<code>protected</code>	+	+	+	+	
<i>no modifier</i>	+	+	+		
<code>private</code>	+				

[05] Fehlerbehandlung, Exceptions, Module

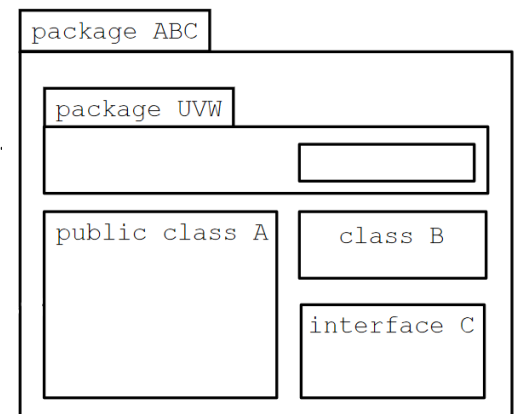
package

Pakete werden genutzt, um Programme besser zu strukturieren.
Ein Package kann **Klassen**, **Interfaces** und weitere **Packages** enthalten.

Packages sind also ein weiteres Werkzeug zur
Entkopplung durch Kapselung.

Jede Klasse kann nur zu einem Package gehören. Dies wird am
Anfang einer Übersetzungseinheit wie folgt deklariert:

```
package [NAME];
```



Importieren aus Packages

Klassen und Interfaces aus anderen Paketen können importiert werden.

```
import graphics.pixel; importiert die Klasse pixel aus dem Paket graphics.
```

```
import system.*; importiert alle Klassen aus dem Paket system.
```

Achtung! Man kann nur auf als **public** deklarierte Klassen und Interfaces aus anderen Paketen zugreifen! Diese bilden die **Export-Schnittstelle** eines Paketes.

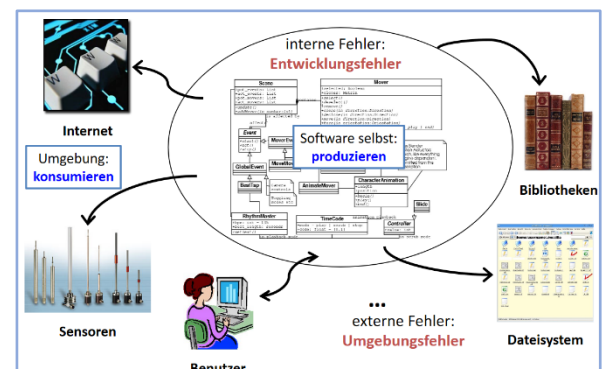
Entwicklungsfehler / Umgebungsfehler

Entwicklungsfehler

Von Entwicklern gemacht

Entwicklungsfehler

Treten in Umgebung der Ausführung auf



Exceptions

Eine **Exception** ist ein Exemplar einer **Exception-Klasse**, häufig mit eigenem Typ.

throw Löst eine *Exception* aus.

```
throw new BusinessException();
```

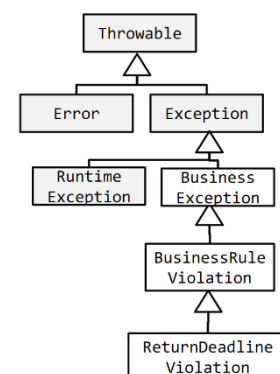
try Block, in dem mit dem Auftreten einer *Exception* gerechnet wird.

catch Block, zur Behandlung einer aufgetretenen *Exception*.

finally Block, der immer ausgeführt wird, wenn **try** betreten wurde, auch bei **catch**.
(Optional)

```
try
{
    // Anweisungen, die Exceptions auslösen können
}
catch (<Exception-Typ> e)
{
    // Behandeln der Fehlersituation
}
finally
{
    // Anweisungen, die in jedem Fall
    // ausgeführt werden sollen
}
```

throws Deklariert, dass eine Funktion möglicherweise eine Exception auslöst. In diesem Fall bricht das Programm nicht ab, sondern gibt den Fehler an die äußere Methode weiter, die diese Methode aufgerufen hat. Der Fehler muss nun in der äußeren Methode **behandelt**, oder wiederum nach außen **weitergegeben** werden.



Checked / Unchecked Exceptions

Exceptions die im Methodenkopf mit **throws** deklariert werden, sind **Checked Exceptions**.
Das Programm compiles nur, wenn diese auch behandelt werden.

Exceptions die im Methodenkopf nicht mit **throws** deklariert werden, sind **Unchecked Exceptions**.

[06] WAM, Beobachtermuster

Arten von Softwaresystemen

Interaktive Softwaresysteme

- Häufige Interaktion zwischen System und Benutzerin / Benutzer.
- Benutzerschnittstelle mit großer Bedeutung: Human Computer Interface HCI.
- Rechenaufwand des Computers für eigentliche Antwort gering; aufwändig ist eher die Darstellung

Software zur Batchverarbeitung

Rechenaufwendige Prozesse die nach Start selbstständig über längere Zeit laufen.

Bsp.: Videorendern, Physiksimulation, ...

Eingebettetes System

Software als Teil eines technischen Systems mit eingeschränkter Nutzerschnittstelle; kommuniziert mit der Umgebung vorrangig über Ereignisse und Signale.

Bsp.: Steuerungscomputer einer Kaffeemaschine

WAM-Ansatz

Interaktive Systeme bestehen aus **vier** Elementtypen:

Werkzeuge

Grafische Schnittstelle für Benutzer, bearbeiten von Materialien.
Bestehen aus **Werkzeugklasse** und **UI-Klasse**.

Services

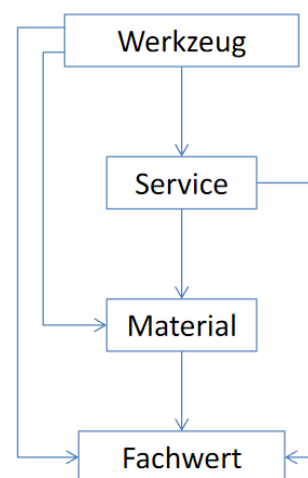
Fachliche Dienstleistungen, Materialübergreifend, Systemweit.
Es gibt jeweils ein Exemplar eines Services.

Materialien

Veränderliche, anwendungsfachliche Gegenstände.
Bsp.: CD, DVD

Fachwerte

Anwendungsfachliche Werte; unveränderlich.
Programmiertechnisch durch **Werttypen** realisiert.
Bsp.: Kontonummer, Geldbetrag



Werkzeuge erhalten ihre **Materialien** als *Konstruktorparameter*, über *Setter* oder von *Services* bereitgestellt.

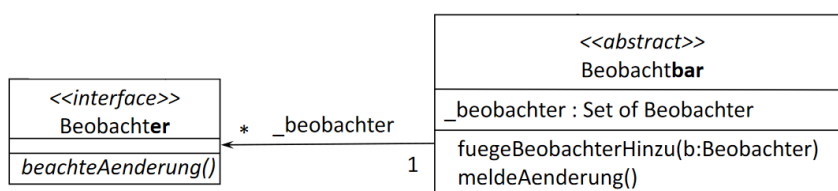
Werkzeugen werden benötigte **Services** als *Konstruktorparameter* übergeben.

Werkzeuge erzeugen ihre zugehörige **UI-Klasse** im eigenen *Konstruktor*.

Die UI-Klasse eines Werkzeugs hat die Aufgaben, die **GUI-Komponenten** zu *erzeugen*, zu *layouten* und zu *verwalten*.

Beobachtermuster

Wird genutzt, um bessere Kommunikation zwischen Klassen zu gewährleisten.



Dabei gibt es **Beobachter** und **Beobachtbare** Klassen.

Ein **Beobachter** kann sich bei **Beobachtbaren** Objekten anmelden (`fuegeBeobachterHinzu(b:Beobachter)`). Bei Events, die von **Beobachtbaren** Objekten angestoßen werden (`meldeAenderung()`), ruft das **Beobachtbare** Objekt eine spezielle Methode bei allen **Beobachtern** auf (`beachteAenderung()`), die sich bei ihm angemeldet haben (`_beobachter`).

Jede Klasse, die eine andere Klasse beobachten möchte, muss **Beobachter** implementieren.

Jede Klasse, die beobachtbar sein will, erbt von **Beobachtbar**.

Vorteile	Nachteile
Flexibilität und Modularität durch Entkopplung Änderungsaufwand verringert	Gefahr von Aktualisierungs-Kaskaden
Zustand des Gesamtsystems konsistent gehalten	Gefahr, Abmeldung vom Beobachter zu vergessen
Kompatibel zu Schichtenmodell	Superklasse vergeben - nicht für Fachliches verfügbar

[07] Modellierung, Entwurf

Entwurfsmuster

- Vorlagen für erfolgreiche Problemlösungen
- Vereinheitlichte, wiedererkennbare Lösungen
- Höhere Qualität von Entwürfen

Strukturierter Text:

Name des Entwurfsmusters,

Problem, das mit Hilfe des
Musters gelöst werden soll,

Kontext, in dem sich das Problem
stellt,

Lösung als Struktur (Organisation
von Klassen in einer
Klassenhierarchie) und / oder
Verhalten (Gestaltung ihrer
Interaktion)

Konsequenzen (positive und
negative) Auswirkungen der
Musteranwendung

Ziel, Motivation

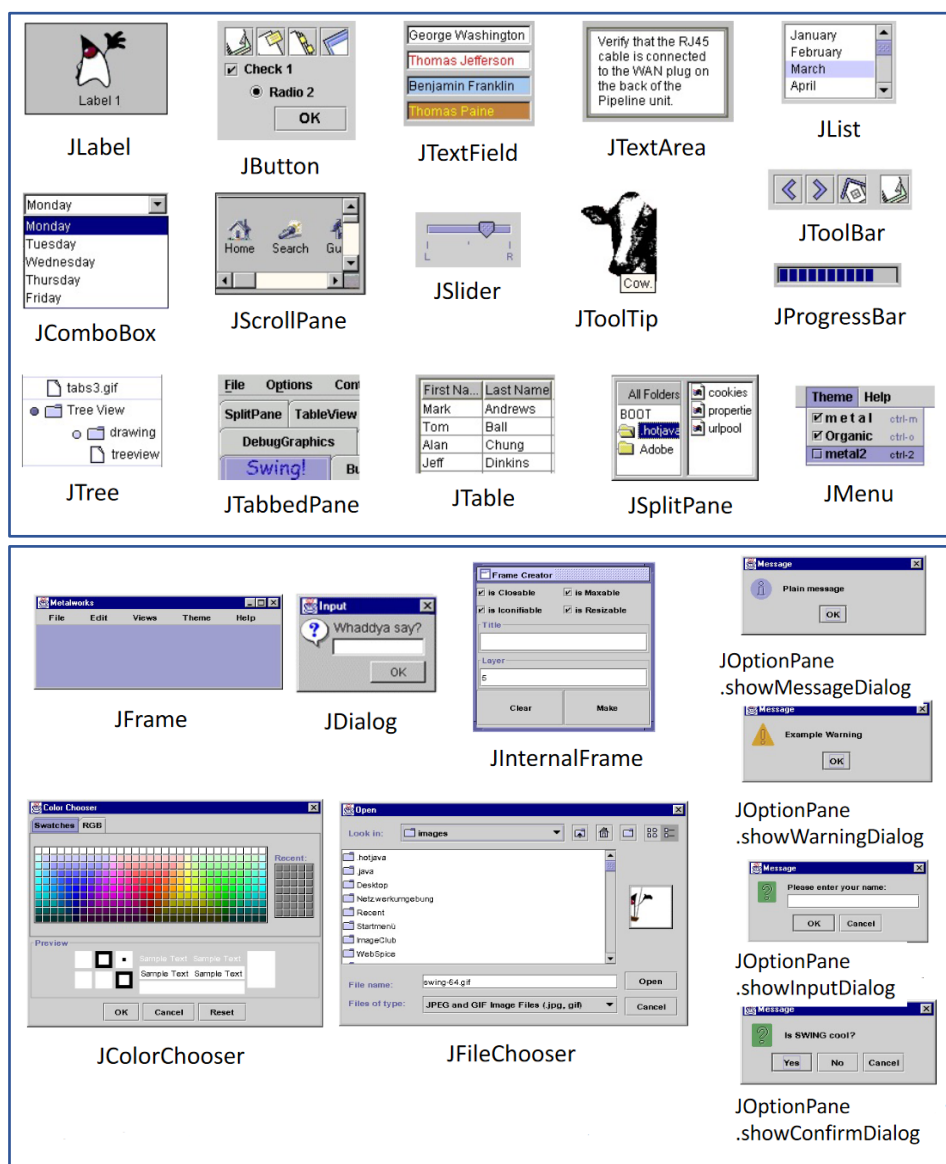
Struktur, Teilnehmer,
Zusammenarbeit,
Implementation

Anwendbarkeit

[08] GUI-Programmierung, Lambda-Ausdrücke

Achtung! Die GUI-Programmierung geschieht nur in den UI-Klassen der Werkzeuge.

Swing-Komponenten

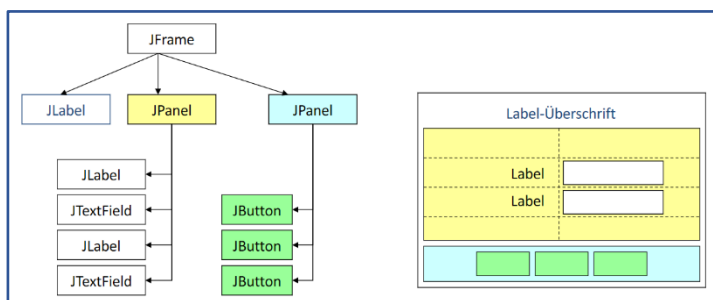


Container

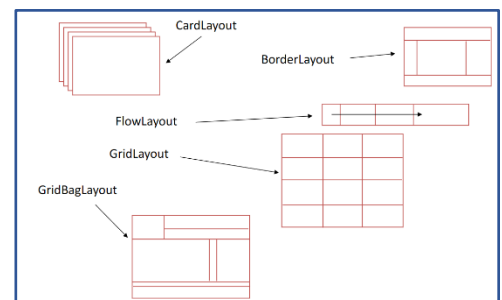
Container können atomare *Swing Komponenten* und weitere *Container* beinhalten. Diese können mit *Containern* hierarchisch angeordnet werden. Die Container-Schnittstelle `.add()` erlaubt das Hinzufügen von *Komponenten*.

Das Layout der *Komponenten* kann über einen **Layout-Manager** gesetzt und abgefragt werden. Jeder *Container* besitzt als Default einen *Layout-Manager*.

```
void setLayout( LayoutManager );
LayoutManager getLayout();
```



Beispiel für eine Container Hierarchie



Verschiedene Layout-Manager Typen

Events

Mausbewegung, Mausklick, Tastendruck, Buttonpress, ... werden vom Systemcode der GUI registriert und es wird ein **Ereignis** (Event) für sie erzeugt. Diese Events werden an alle Teile des Systems verschickt, die sich für diese Aktion bei der GUI-Komponente **angemeldet** haben (siehe Beobachtermuster).

Dafür bietet Java bereits vorprogrammierte **Listener**, bspw. ActionListener.

```
public interface ActionListener
{
    void actionPerformed(ActionEvent event);
}
```

Eine Klasse implementiert dann **Listener**, und legt in `actionPerformed()` fest wie es auf Änderungen reagiert. Die Klasse meldet sich dann bei GUI-Komponente als Listener an.

Bei einem Event erzeugt die GUI-Komponente ein **Event-Objekt** und übergibt dieses als Parameter in `actionPerformed(ActionEvent event)` an alle Listener weiter. Das Event-Objekt enthält alle Informationen über das Ereignis (auslösende Komponente, ...).

Wichtige Listener:

JRadioButton	ActionListener, ItemListener
JList	ActionListener, ListSelectionListener
JComboBox	ActionListener, ItemListener
TextField	ActionListener

Lambdas

Interface mit nur einer Operation. Erlauben, Methoden als Parameter zu übergeben.

```
MathOperation subtraction = (a, b) -> a - b;
result = myObject.operate(10, 5, subtraction)
```

- Typ-Deklaration darf fehlen
- Klammern um Parameter () dürfen fehlen
- Geschweifte Klammern {} dürfen fehlen
- Return darf fehlen

Listeners können gut mit Lambdas implementiert werden.

```
_myButton.addActionListener( (event) ->
    {
        _zaehler++;
        _label.setText(_zaehler + "mal gedrückt");
    } );
```

Benötigt wird nur ein Name für den Parameter und die Implementation der Methode.

[09] OO Entwurfsprinzipien

Entwurf

Überführen aus Problembereichsmodell in technische Lösung. Dabei wird methodisch und strukturiert vorgegangen (Bsp. WAM, SCRUM).

Folgendes ist wichtig für guten Entwurf:

Lose Kopplung

- Geeignete Schnittstellen wählen
- Entwurfsentscheidungen kapseln
- Geheimnisprinzip
- Zyklen vermeiden
- Law of Demeter

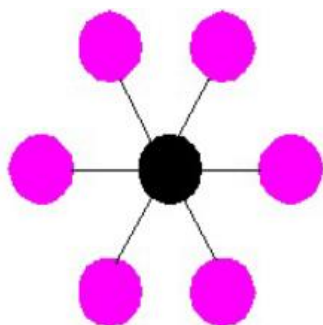
Hohe Kohäsion

- für Klassen und für Methoden
- Code-Duplizierung vermeiden
- Geeignete Bezeichner wählen
- Große Einheiten vermeiden

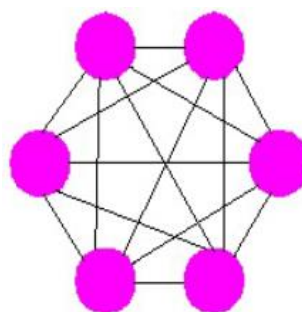
Kopplung

Kopplung beschreibt den **Grad der Abhängigkeiten** (*Benutzt-Beziehung/Vererbung*) zwischen Einheiten eines Softwaresystems. Wenn eine Klasse mit allen anderen Klassen verbunden ist und kommuniziert, ist das ihre **maximal mögliche Kopplung**. Gut ist eine möglichst lose Kopplung, d.h. wenn eine Einheit mit möglichst wenig anderen Einheiten kommuniziert.

Lose Kopplung macht einzelne Klassen besser verständlich und Änderungen einfacher durchzuführen.



Die schwarze Einheit hat maximale Kopplung, alle anderen eine niedrige



Jede Einheit hat maximale Kopplung

Kohäsion

Kohäsion beschreibt den **Grad** (Anzahl und Verschiedenheit) **der Aufgaben**, die eine Softwareeinheit zu erfüllen hat. Wenn eine Einheit nur für genau eine Aufgabe zuständig ist, dann sprechen wir von **höher Kohäsion**.

Wir unterscheiden zwischen Kohäsion von Methoden und Kohäsion von Klassen. Beide sollten möglichst hoch sein.

Law of Demeter

Richtlinie auf Code-Ebene für **lose Kopplung**.

Eine **Methode m** eines **Objektes o** sollte ausschließlich Methoden von folgenden Objekten aufrufen:

- von **o** selbst
- von *Parametern* von **m**
- von *Objekten*, die **m** erzeugt
- von *Exemplarvariablen* von **o**

Erlaubt wäre beispielsweise nicht:

```
student.getRecord().getExamEntry("SE2/2006").addResult(93);
```

Besser wäre:

```
student.scored(markedExam);
```

CRC-Karten

Klassen- / Komponentename (Class / Component Name)

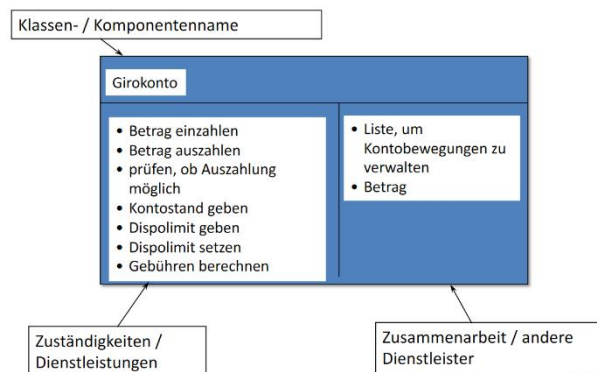
- Wichtiger Begriff des Anwendungsgebiets bzw. des Anwendungssystems
- Teil der Fachsprache.

Zuständigkeiten (Responsibility)

- Von der Klasse / Komponente erbrachte Dienstleistungen
- In sich zusammenhängendes Angebot an potenzielle Klienten (Kohäsion)

Zusammenarbeit (Collaboration)

- Andere Klassen / Komponenten, an die Teile der Dienstleistung delegiert werden



SOLID

SRP: Single Responsibility Principle

Aufteilung nach Zuständigkeiten, Separation of Concerns

OCP: Open Closed Principle

Offen für Erweiterungen, Änderungen wirken sich nicht auf Klienten aus

LSP: Liskov Substitution Principle

Supertyp durch Subtyp ersetzbar

ISP: Interface Segregation Principle

Klienten-gerechte Interfaces

DIP: Dependency Inversion Principle

Klienten und Dienstleister hängen von Abstraktionen ab und nicht voneinander

Ziel: niedrige Kopplung und hohe Kohäsion

[10] Fachwerte und Werttypen

Fachliche Werte

Gehören zu Anwendungsbereich und haben eine fachliche Bedeutung (Bsp. Postleitzahl, Geldbetrag). Sie haben oft Operationen, die von den „normalen“ mathematischen Operationen abweichen.

Kontonummern *konkatenerieren* möglich, *subtrahieren* nicht

Nicht alle fachlichen Werte sind durch Zahlen sinnvoll abbildbar.

Werte sind Elemente, die aus einer **Wertemenge** ausgewählt werden können. Sie werden nicht erzeugt oder verbraucht, sondern nur ausgewählt.

Werte sind **abstrakt**, **zeitlos** und **unveränderlich**.

abstrakt

Werte sind immateriell und abstrahieren von Konkreten Kontexten (Bsp. die Zahl 2). Fachliche Werte sind häufig Abstraktionen von Dingen, um diese zu identifizieren (Kontonummer, Postleitzahl).

zeitlos

Begriffe wie *Zeit* und *Dauer* sind auf Werte nicht anwendbar. Sie werden nicht erzeugt oder zerstört.

In Ausdrücken entstehen keine Werte und werden auch nicht verbraucht.

$$40 + 2 = 42$$

unveränderlich

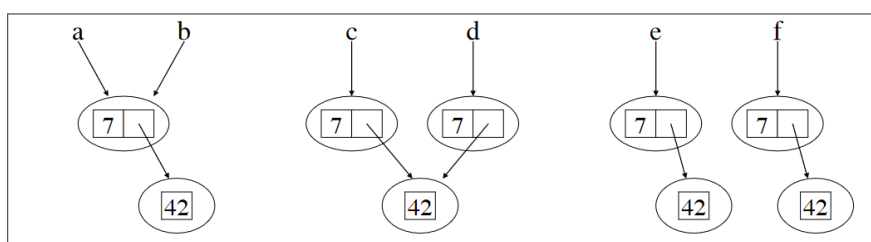
Werte können berechnet und auf andere Werte bezogen werden, aber nicht verändert werden. Funktionen können auf Werte angewandt werden, um andere Werte zu berechnen.

$$40 + 2 = 42 \quad \text{Die 42 wird nicht **erzeugt**, sondern **ausgewählt**.}$$

Gleichheit für Objekte

Wir unterscheiden zwischen 3 Formen.

- Referenzgleichheit, Identität (identity)
- Einfache strukturelle Gleichheit (shallow equality)
- Rekursive strukturelle Gleichheit (deep equality)



identity

impliziert shallow equality

impliziert deep equality

shallow equality

impliziert deep equality

deep equality

[11] Refactoring

Bad Smells

Änderungen werden erschwert, wenn der Code schlecht ist:

- niedrige Verständlichkeit
- unnötige Komplexität
- Fehleranfälligkeit (nicht robust)
- Verletzung von üblichen Entwurfsregeln

Häufige Dark-Patterns (Patterns, die man vermeiden sollte):

- **Code-Duplikate** Bei Wartung sind mehr Änderungen nötig
- **Gott-Klasse** Eine Klasse, die alles steuert, führt zu hoher Kopplung
- **Verletzte Kapselung** Klassen haben nutzen Methoden anderer Klassen mehr als der eigenen.

Diese können durch **Refactoring** verändert/verbessert werden.

Refactoring

Verändern der **internen Struktur** einer Software zur *Verbesserung von Wartbarkeit und Verständlichkeit*, ohne ihr beobachtbares Verhalten zu verändern.

Refactoring ist nicht:

- Das Einfügen zusätzlicher Funktionalität
- Das Beheben von Fehlern
- Eine Änderung am Layout der GUI ohne Änderung der Funktionalität

Refactoring-Prinzipien:

1. Keep it small

Folge kleiner, inkrementeller Veränderungen – jede kann leicht geprüft und rückgängig gemacht werden (geringes Risiko)

2. Auslösen durch Kunden-Bedarf

Im Vorfeld von (wichtigen) funktionalen Erweiterungen

3. Team-Zusammenhalt

Verständnis, collective code ownership. Jeder ist auf dem Gleichen Stand.

4. Transparenz der Kosten

Kunde kennt Kosten und akzeptiert.

Achtung! Da Refactoring auch Geld kostet sind viele Kunden häufig dagegen.

Empfehlung ist zu refactorn, aber Kunden nicht zu sagen (grundsätzlich mit jeder Veränderung Refactoring verbinden)

[12] Korrektheit und Formalisierung

Zuverlässige Software

Wir wollen nicht nur *wiederverwendbare*, *erweiterbare* und *änderbare*, sondern auch **zuverlässige** Software entwickeln.

Zuverlässigkeit ist nach Meyer gekennzeichnet durch:

Korrektheit

bezeichnet die Fähigkeit von Software, ihre Aufgabe genauso zu erfüllen, wie es die Spezifikation vorschreibt. Eine Software ist also nur in Relation zu ihrer Spezifikation korrekt.

Robustheit

bezeichnet die Fähigkeit von Software, angemessen auf Fälle zu reagieren, die nicht in der Spezifikation definiert sind.

Semantik

In der theoretischen Informatik kann mittels **formeller Semantik** das Verhalten von Programmen beschrieben werden.

1. Denotationale Semantik

Änderungen im Speicher betrachten, die ein Programm bewirkt. Eine Anweisung liefert dabei einen neuen Speicherzustand, basierend auf dem vorherigen Speicherzustand.

$$f(A): Z \rightarrow Z, \quad f(z) = z'$$

A beschreibt eine Anweisung

Z beschreibt die Menge aller möglichen Speicherzustände

z beschreibt einen Zustand, **z'** einen Folgezustand

2. Operationale Semantik

Schrittweise Veränderung des Zustands einer abstrakten Maschine durch ein Programm betrachten. Ein **konkretes Programm** wird durch ein semantisch gleiches, aber **abstraktes Programm** ersetzt.

Dieses Programm wird von einer abstrakten Maschine interpretiert. Die Wirkung eines Programms ist die schrittweise Veränderung dieser abstrakten Maschine, d.h. ihrer Variablenbelegung.

Anweisungsfolge vor Ausführung	alter Zustand	neuer Zustand	Anweisungsfolge nach Ausführung
$i_1 := i_2 + 1; a$	z	$z \langle i_1 \leftarrow (z(i_2) + 1) \rangle$	a

Dabei ist die Annahme, dass die Veränderung der abstrakten Maschine ist gleich der eines konkreten Computers ist.

3. Axiomatische Semantik

Legt für einzelne Programmelemente **Vor-** und **Nachbedingungen** fest. Dabei wird die Änderung von Programmvariablen betrachtet.

Hoare Kalkül

Semantik einer **imperativen Anweisung a**, wobei **P** und **Q** **Prädikate** (Bedingungen, logische Ausdrücke) sind. Die Form ist das Tripel:

$$(\{P\}, a, \{Q\})$$

Wenn **P** vor Ausführung von **a** gilt, dann gilt **Q** nach der Ausführung.

Bsp.: $(\{0 < x < 1000\}, x := x + 1, \{1 < x < 1001\})$
In diesem Fall sind **P** und **Q** nur ein logischer Ausdruck.

Abstrakte Datentypen

Datentypen und ihre zulässigen Operationen werden gemeinsam definiert.

Die Datentypen sind nur durch die Operationen, die auf ihn anwendbar sind und durch die Bedingungen ihrer Anwendung beschreiben. Es wird keine genaue Implementation definiert.

Bsp.: Algebraische Spezifikation der natürlichen Zahlen

TYPE (der Typ Nat wird definiert)

Nat

FUNCTIONS (die Signaturen der Funktionen legen die Schnittstelle von Nat fest)

$0: \rightarrow \text{Nat}$

$\text{suc}: \text{Nat} \rightarrow \text{Nat}$

$\text{add}: \text{Nat} \times \text{Nat} \rightarrow \text{Nat}$

AXIOMS (Einschränkungen der Verwendung der Funktionen)

For any $i, j: \text{Nat}$

$\text{add}(i, 0) = i$

$\text{add}(i, \text{suc}(j)) = \text{suc}(\text{add}(i, j))$

Mehr Beispiele sind im Skript, **Vorlesung 12, Folie 21f.**

[13] Funktionale Programmierung

Bisher haben wir **imperativ** programmiert. Ein Programm ist dabei eine Folge von zustandsändernden Anweisungen. Der Gegensatz dazu ist die **Funktionale Programmierung**.

Threads

Ein Thread ist eine **eigenständige Aktivität in einem Prozess** mit eigenem Kontext (z.B. Variablen, Ressourcen), also ein eigener Rechenprozess.

Mehrere Threads können gleichzeitig ausgeführt werden. Dies kann jedoch zu Problemen führen, wenn mehrere Threads auf die gleiche Ressource zugreifen.

Race Condition

Wenn zwei Threads auf dieselbe Ressource zugreifen und gleichzeitig lesen und schreiben, können unerwünschte Zustände entstehen.

Über **Locking-Mechanismen** können die Zustände von Ressourcen geschützt werden, sodass bspw. nur ein Thread gleichzeitig auf sie zugreifen kann.

- Mutex (mutual exclusion)
- Semaphoren
- Java keyword synchronized

Gemeinsames Objekt: Konto	
Abheben-Thread	Einzahl-Thread
Gib Saldo von Konto Antwort: Saldo = 100	
Saldo >= Betrag?	
	Gib Saldo von Konto Antwort: Saldo = 100
Hebe 50 von Konto ab (Resultat Saldo = 50)	
	Zahle 100 auf Konto ein (Resultat Saldo = 200)

*Ein Beispiel für eine Race Condition.
Beide Threads greifen auf dasselbe Konto zu*

Dies ist jedoch Fehleranfällig und von vielen Programmierparadigmen verboten. An dieser Stelle ist es sinnvoll, **funktionale Programmierung** zu nutzen.

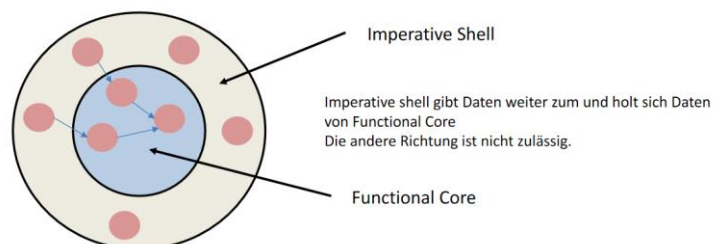
Funktionale Programmierung

Bei der **funktionalen Programmierung** werden Funktionen ähnlich zur Mathematik betrachtet. Ein Programm ist dabei ein **Ausdruck**, der einen Wert als Ergebnis liefert (wie eine Rechnung).

Probleme werden dabei durch Definition und Aufruf von Funktionen gelöst.

Es ist nicht sinnvoll, **komplett objektorientiert** oder **funktional** zu programmieren. Beide Paradigmen haben ihre Stärken und Schwächen.

Eine Idee ist, dass ein Teil der Anwendung funktional gehalten wird, während der andere Teil zustands- und seiteneffektbehaftet sein darf.



Pure Funktion

Eine pure Funktion ist eine Funktion, die immer das **gleiche Ergebnis für die gleiche Eingabe** liefert. Sie besitzt dabei keine Abhängigkeit zum globalen Zustand der Anwendung und erzeugt keine Seiteneffekte.

Pure Funktion	Keine Pure Funktion
<pre>int add(int a, int b) { return a+b; }</pre>	<pre>LocalDateTime getTime() { return LocalDateTime.now(); }</pre>

Racket

Racket ist eine multi-paradigmatische Programmiersprache. Wir benutzen Racket um funktional zu Programmieren.

Programm	eine Menge von Funktionsdefinitionen im mathematischen Sinne
Ausdruck	Kombination von Funktionsaufrufen
Berechnung	Auswertung eines Ausdrucks => liefert einen Wert
Wert	Element eines bestimmten Datentyps (String, integer, Liste etc.)
Variable	Name für einen unbekannten Wert wie in mathematischen Formeln – keine Zuweisung zu einem Speicherplatz, wie wir es aus der imperativen Programmierung kennen. Der Wert einer Variable kann <u>nicht</u> geändert werden.

Werte

Elementare Datentypen:

Zahlen	251, 420, 5
Wahrheitswerte	#t, #f
Character	#\a, #\A, #\space, #\newline, #\077

Oktalzahl
jedes ASCII-
Zeichens

Funktionen

Syntax: ([OPERATION] e1 e2 ... en)

Bsp. Addition

```
(+ 200 220)
(+ (+ 1 2) 3)
```

Die Auswertung erfolgt rekursiv.

Folgende Operationen sind vordefiniert: + - * / < > <= >= =

Bedingte Ausdrücke

Syntax: `(if e1 e2 e3)`

Wertet zuerst **e1** zu **v1** aus, und prüft ob **v1** **#f** ist.

Wenn **v1** **#f** ist, wird **e3** ausgewertet und zurückgegeben.

Wenn **v1** nicht **#f** ist, wird **e2** ausgewertet und zurückgegeben.

Namensbindung / Variablen

Syntax: `(define <bezeichner> <variable>)`

Bezeichner

Variablen wie in mathematischen Formeln: Name für einen unbekannten Wert
(nicht für einen reservierten Speicherplatz, wie in Java)

Wir schreiben $a \mapsto 3$, wenn eine Variable einen Wert annimmt (in diesem Fall nimmt a den Wert 3 an).
Diese Variablen werden der Umgebung hinzugefügt und können in späteren Operationen verwendet werden.

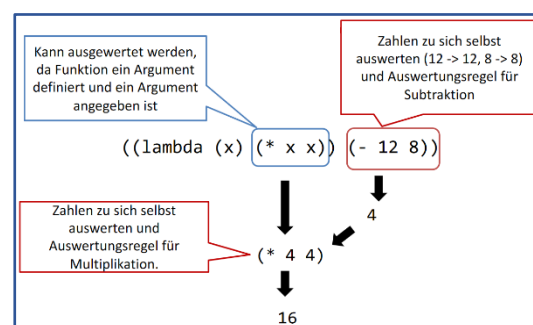
Bsp:	<u>Ausdrücke</u>	<u>Variablen in der Umgebung</u>
	<code>(define x (+ 1 2))</code>	$(x \mapsto 3)$
	<code>(define y (* 4 x))</code>	$(x \mapsto 3, y \mapsto 12)$
	<code>(define diff (- y x))</code>	$(x \mapsto 3, y \mapsto 12, \text{diff} \mapsto 9)$
	<code>(define test (< x diff))</code>	$(x \mapsto 3, y \mapsto 12, \text{diff} \mapsto 9, \text{test} \mapsto \#t)$

Wir können über das Keyword `lambda` unsere eigenen Funktionen definieren.

Syntax: `(lambda (x1 ... xn) e)`

Parameter **x1** bis **xn** sind Bezeichner

Rumpf **e** ist ein beliebiger Ausdruck



Mit `lambda` wird eine Funktion definiert und direkt mit dem Parameter 4 aufgerufen.

Diese selbsterstellten Funktionen können wir auch an Namen binden und später aufrufen.

`(define square (lambda (x) (* x x)))`

`(define x (square 4))` $(x \mapsto 16)$

Listen

Listen bestehen aus zwei Teilen:

- Dem ersten Element
- Dem Rest der Liste

Ausgangspunkt ist die leere Liste

- `empty` oder
- `'()`

```
(cons 'Karl empty)
;; => (Karl)
```



```
(cons 'Rosa (cons 'Karl empty))
;; => (Rosa Karl)
```



```
(cons 'Klara (cons 'Rosa (cons 'Karl empty)))
;; => (Klara Rosa Karl)
```



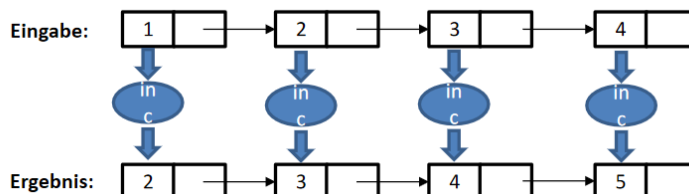
Die Funktion `cons` fügt einer Liste ein neues erstes Element hinzu. Die Funktion nimmt zwei Argumente: Das neue erste Element und eine Liste.

Map Funktion

Nimmt eine Funktion `f` und eine Liste `xs`, und wendet `f` auf jedes Element aus `xs` an.

```
(define (inc x) (+x 1))
(map inc (list 1 2 3 4))

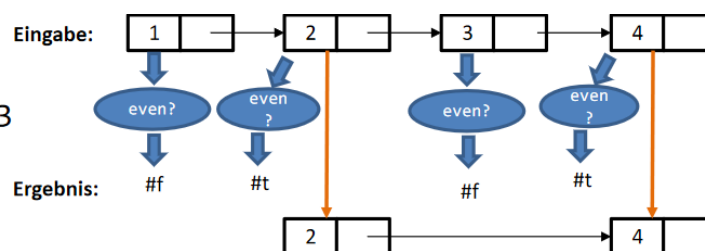
Ergebnis: '(2 3 4 5)
```



Filter Funktion

Nimmt eine Funktion `f` und eine Liste `xs`, und gibt jedes Element aus `xs` zurück, für das `(f x)` nicht zu `#f` ausgewertet wird.

```
(define (even? x)
  (= 0 (modulo x 2)))
(filter even? (list 1 2 3 4))
```



Geschrieben von David Rath

david.rath@studium.uni-hamburg.de