Stoffabgrenzung Modulendprüfung PRG2

Objektorientierte Programmierung:

Sie können mindestens drei Vorteile sowie einen schwergewichtigen Nachteil der Vererbung erklären.

Vorteile:

- *Einfache Wiederverwendung (von Implementation)*
- *Vermeidung von Code-Duplikation*
- *häufig einfachere Wartbarkeit*
- *häufig einfachere Erweiterbarkeit*

Nachteile:

- *starke Kopplung*

Sie können zwischen statischem und dynamischem Typ unterscheiden.

Statisch bedeutet auf Ebene Source-Code bzw. zur Compilationszeit. (Konto k;)

Dynamisch bedeutet auf Ebene Ausführung bzw. zur Laufzeit. (k = new Giro();)

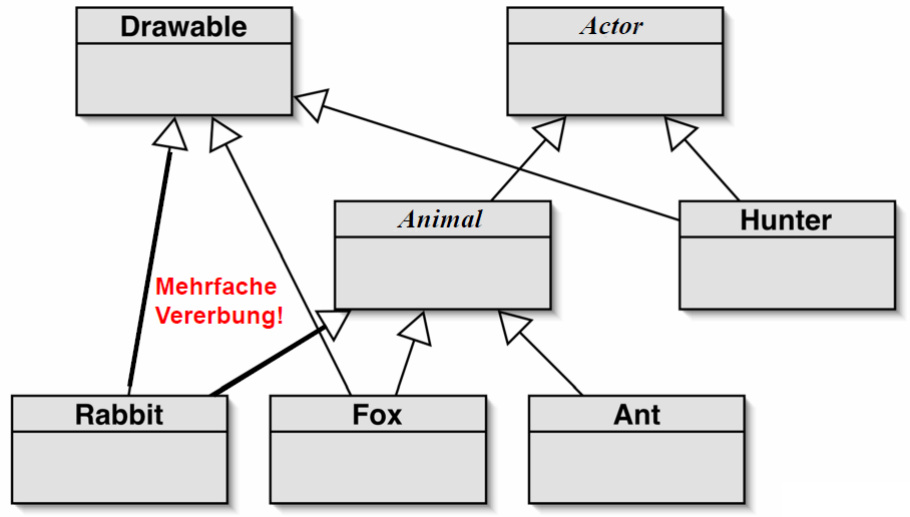
Sie können Einfach- und Mehrfachvererbung erläutern.

Eine mehrfache Vererbung ist in Java nicht möglich!

Einfache Vererbung: Schlüsselwort: extends

„Mehrfachvererbung“: Schlüsselwort: implements (Interface)

Sie können Vererbungsbeziehungen im Klassendiagramm interpretieren.



Kursiv = abstrakte Klassen

```
public interface Drawable {…}
public abstract class Actor {…}
public class Hunter extends Actor implements Drawable {…}
public abstract class Animal extends Actor {…}
public class Ant extends Animal {…}
public class Rabbit extends Animal implements Drawable {…}
public class Fox extends Animal implements Drawable {…}
```

Java - Vererbung:

Sie kennen Java spezifische Eigenschaften der Vererbung.

- *Vererbung in Java wird realisiert mit*
  - *– Basisklassen*
  - *– abstrakten Klassen*
  - *– Interfaces*
- *wichtige Konzepte sind*
  - *– Typhierarchien / Subtyping*
  - *– statischer Typ und dynamischer Typ*
  - *– Polymorphie*
- *Mehrfachvererbung von Klassen ist nicht möglich.*
- *„Mehrfachvererbung" von Interfaces (Schnittstellenklassen) ist möglich.*

Sie verstehen die Funktionsweise von abstrakten Klassen und können diese einsetzen.

- *Von abstrakten Klassen können keine Objekte instantiiert werden.*
- *Konkrete Unterklassen vervollständigen die Implementation.*
- *Abstrakte Klassen sind nicht instantiierbare Basisklassen.*
- *Abstrakte Klassen unterstützen Polymorphie (Überschreiben, Overriding) und Typprüfung.*
- *Es gibt abstrakte Klassen, die keine abstrakten Methoden enthalten.*
- *Eine abstrakte Klasse, die eine vollständige Implementation enthält, erzwingt eine Ableitung.*
- *Dadurch wird das Interface erzwungen und somit überwacht.*

```
public abstract class Animal
{
   …
   public abstract void act(Field currentField, Field updatedField, List newAnimals);
   …
}
```

Sie verstehen die Funktionsweise von abstrakten Methoden und können diese einsetzen.

- *Abstrakte Methoden haben abstract im Methodenkopf.*
- *Abstrakte Methoden haben keinen Methodenrumpf (body).*
- *Abstrakte Methoden machen auch die Klasse abstrakt.*
- *Unterklassen können durch Überschreiben (Implementieren) der abstrakten Methode konkret, d.h. instantiierbar werden.*
- *Abstrakte Methoden legen eine Schnittstelle fest und erzwingen, dass diese in einer Unterklasse konkret implementiert werden.*

Sie können ein Java Interface definieren und dieses in Klassen implementieren.

- *Ein Java Interface spezifiziert ein Verhalten (Menge von Methodenköpfen) ohne Implementationen zu beinhalten. (nur das WAS)*
- *Schlüsselwort: interface*
- *Ein Interface beinhaltet keinen Konstruktor.*
- *Alle Methoden eines Interfaces sind public abstract. Die entsprechenden Schlüsselworte können entfallen.*
- *Alle Felder eines Interface sind public static final. Die entsprechenden Schlüsselworte können entfallen.*
- *Interfaces können von einem (oder mehreren) anderen Interfaces erben (implements)*

```
public class MyClass implements MyInterface {…}
public class MyClass4 extends BaseClass implements MyInterface, MyInterface2 {…}
```

Java kennt für Klassen 'nur' einfache Vererbung, bei Interfaces ist aber Mehrfachvererbung möglich!

Sie kennen Unterschiede zwischen konkreten und abstrakten Klassen sowie zwischen abstrakten Klassen und Interfaces.

Von abstrakten Klassen können keine Objekte instanziiert werden.

Bei Interfaces sind alle darin enthaltenen Methoden abstrakt.

Bei abstrakten Klassen können nur einzelne Methoden abstrakt sein und andere implementiert.

Sie wissen, wie Sie in Java Mehrfachvererbung umsetzen können.

```
public class MyClass4 extends BaseClass implements MyInterface, MyInterface2 {…}
```

Java kennt für Klassen 'nur' einfache Vererbung, bei Interfaces ist aber Mehrfachvererbung möglich!

Sie können Polymorphie, d.h. Überschreiben und Überladen, mittels Java-Code erklären und anwenden.

- *Überladen / Overloading – „schwache Polymorphie“*
  - *Mehrere Methoden mit gleichem Namen und unterschiedlicher Signatur in einer Klasse.*
- *Überschreiben / Overriding– „starke Polymorphie“*
  - *@Override*
  - *Methode einer Oberklasse in einer Unterklasse neu bzw. spezifisch implementiert.*
  - *Eine überschriebene Methode bekommt damit Vorrang, d.h. es wird automatisch die "naheliegendste" Methode aufgerufen!*

→Polymorphie führt zu einfacherem und flexiblerem Programmcode. Sie ermöglicht die Handhabung von Objekten unterschiedlicher Klassen auf einer allgemeineren Ebene.

Sie können Substitution und Casting korrekt anwenden.

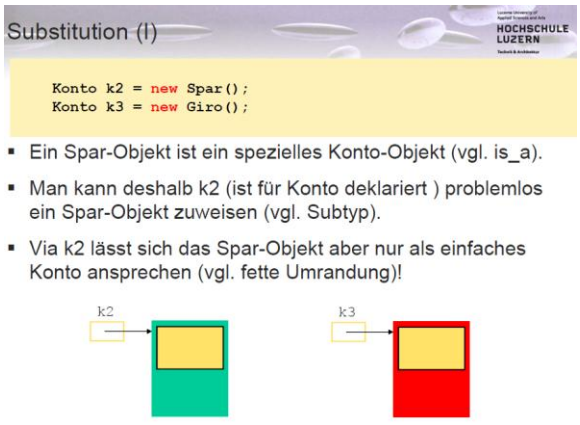
Eine Klassenhierarchie definiert gleichzeitig auch eine Typenhierarchie:

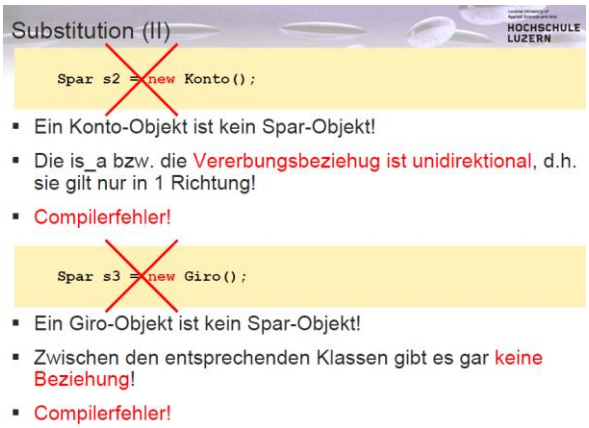
– Konto ist ein Untertyp von Object

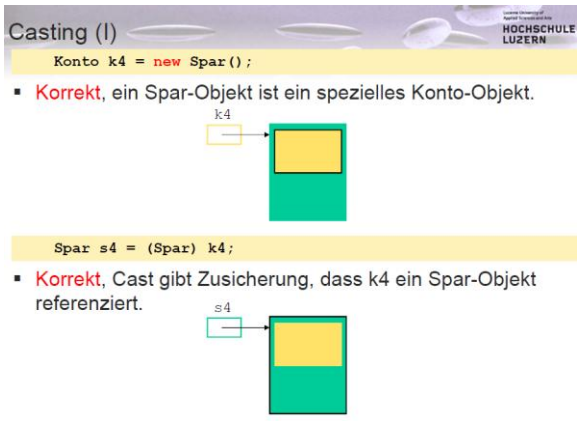
– Spar ist ein Untertyp von Konto

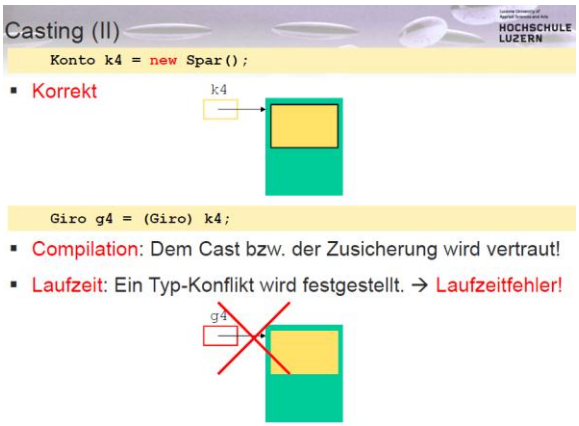
– Giro ist ein Untertyp von Konto

```
Konto k1 = new Konto();
Spar s1 = new Spar();
Giro g1 = new Giro();
```





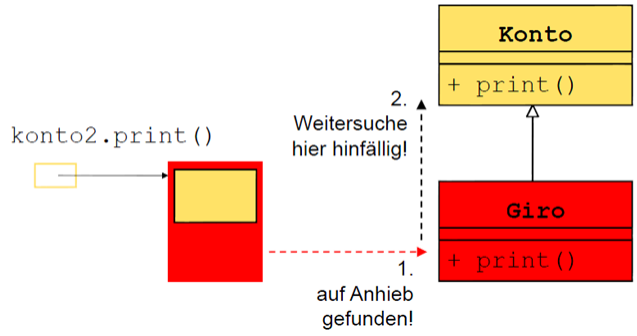




OOP1: Folie 10-17

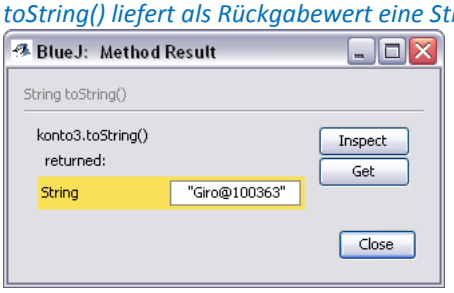
Sie können bestimmen, welche Implementation für einen bestimmten Methodenaufruf zur Ausführung gelangt.

Dynamisches Binden (Method Lookup, Method Binding, Method Dispatch): Ausgehend vom dynamischen Typ wird die Klassenhierarchie zur Laufzeit nach oben nach der auszuführenden Methode durchsucht:



Sie wissen, wie toString() arbeitet und angepasst werden kann.

- *Die Klasse Object vererbt an sämtliche Klassen eine Methode toString().*
- *toString() liefert als Rückgabewert eine String-Repräsentation des Objektes zurück, z.B.:*



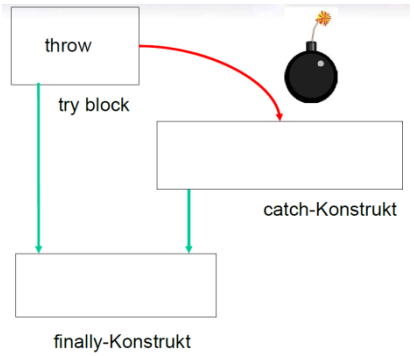
- *"Giro@100363" → Ein Objekt der Klasse Giro , welches auf dem Heap an der Adresse 100363 gespeichert ist.*
- *Liefern 2 Aufrufe dieselben Adressen zurück, handelt es sich somit auch um dieselben Objekte.*
- *Für adäquatere String-Repräsentationen wird toString() in den eigenen Klassen häufig überschrieben.*
- *Übergibt man print() und println() als Parameter eine Referenzvariable, so wird automatisch toString() aufgerufen*

Java - Exception Handling:

Sie können das Prinzip des Exception Handlings erklären.

Folgende Phasen sind zu unterscheiden:

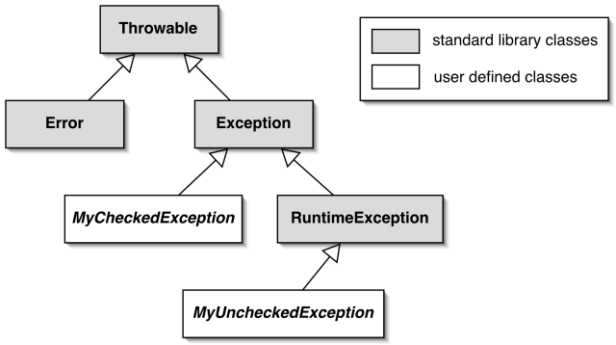
- *Definieren einer Exception*
- *Signalisieren, dass eine Exception ausgelöst werden kann: throws (nur bei checked Exceptions)*
- *Eine Exception auslösen, d.h. jetzt ist die Fehler- bzw. Ausnahmesituation eingetreten. throw*
- *Eine ausgelöste Exception behandeln. try – catch*
- *Aufräumaktionen finally*



Sie können drei Vorteile des Exception Handlings benennen.

- *Übersichtlich, die Fehlerbehandlung ist klar und einheitlich strukturiert*
- *Ausnahmesituationen können ohne Programmabbruch behandelt werden, alternative Programmausführungen sind möglich.*
- *Exceptionhandler haben zum Ziel, eine Exception zu entschärfen, d.h. eine Methode vom Ausnahmezustand in den Normalzustand zu überführen.*

Sie können verschiedene Exception Klassen benennen.



*Error (**unchecked**)*
– Systemfehler, die vom Programm nicht korrigiert werden können.

*Exception (**checked**)*
– Programmfehler, die behandelt, d.h. abgefangen werden müssen.

*Runtime Exception (**unchecked**)*
– Programmfehler, die freiwillig behandelt werden können.
– Unchecked Exceptions müssen nicht mit der throws Klausel signalisiert werden.

*Exception und Runtime Exception bilden die Basisklasse für eigene **checked** und **unchecked** Exception Klassen.*

Sie können benutzerdefinierte Exceptions implementieren.

```
public class NoOctalDigitException extends Exception
{
    public NoOctalDigitException(String s)
    {
        super("my Exception“ + s);
    }
}

public class Exc
{
    private static String[] octalDigits = {"0", "1", "2", "3", "4", "5", "6", "7"};
    public static int convertToOct(String s) throws NoOctalDigitException
    {
        for (int i = 0; i < 8; i ++) {
            if (octalDigits[i].equals(s)) {
                return i;
            }
        }
        throw new NoOctalDigitException(s + " is not an octal digit");
    }
}
```

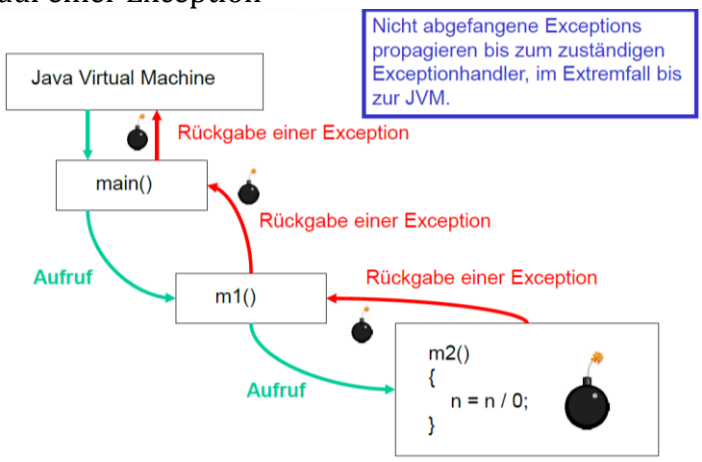
Sie können die Sprachkonstrukte try, catch, finally, throw und throws anwenden.

`throws`	*Signalisiert eine mögliche Exception*
`throw`	*Löst Exception aus*
`try`	*Behandlung des Normalfalls*
`catch`	*Behandlung des Ausnahmefalls*
`finally`	*Abschlussbehandlung in jedem Fall, also sowohl im Normalfall als auch im Ausnahmefall*

Sie können checked und unchecked Exceptions erklären.

Checked Exceptions:	*Müssen behandelt, d.h. abgefangen werden*
	Müssen mit throws im Methodenkopf signalisiert werden!
	(Compiler überprüft ob sie abgefangen werden)
Unchecked Exceptions:	*Können freiwillig behandelt werden, müssen aber nicht behandelt werden.*

Ablauf einer Exception



Java - IO:

Sie können für beide Arten von Datenströmen je zwei Anwendungen angeben.

Byte-Datenströme (8Bit Streams)

- *InputStream*
- *OutputStream*

Zeichen-Datenströme im Unicode (16Bit Streams)

- *Reader*
- *Writer*

Sie können den Unterschied zwischen binären Dateien und Text Dateien erklären.

Binäre Dateien:

- *Folge von Bytes, beliebige Daten speicherbar*

Text Dateien:

- *Folge von Zeichen, File mit Text-Editor lesbar*

Sie können den Unterschied zwischen sequentiellem und wahlfreiem Datei-Zugriff erläutern.

wahlfreier Zugriff

- *es kann direkt jede beliebige Stelle in einer Datei gelesen werden (zugriff erfolgt nicht sequentiell)*

sequentiell

- *Zugriff erfolgt „nacheinander“, d.h. fortlaufend von Anfang bis Ende*

Sie kennen überblicksmässig einige Java-Klassen für das Datei-Handling.

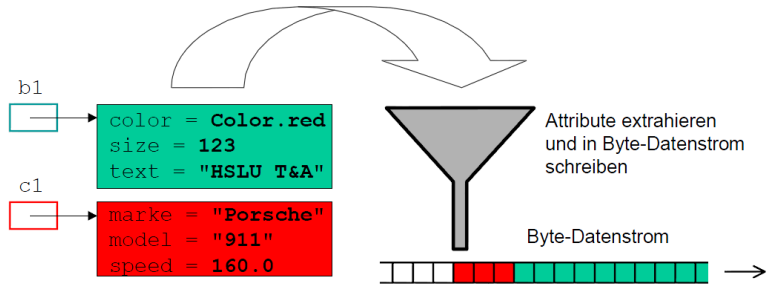
File stellt plattformunabhängige Betriebssystemfunktionen im Zusammenhang mit Dateien und Verzeichnisse und Verzeichnissen bereit.

Beispiel:

```
FileInputStream aFileInputStream = new FileInputStream("data.bin");
DataInputStream aDataInputStream = new DataInputStream(aFileInputStream);
double value = aDataInputStream.readDouble();


PrintWriter aPrintWriter = new PrintWriter(System.out);
BufferedWriter aBufferedWriter = new BufferedWriter(aPrintWriter);
aBufferedWriter.write("ZeichenDatenströme");
...
aBufferedWriter.flush(); // Puffer proaktiv "spülen"
```

Sie kennen das Prinzip und die Eigenheiten der Objekt-Serialisierung.

- *Die ObjektSerialisierung ermöglicht es, Objekte mit Hilfe von Datenströmen übertragen zu können, z.B. in eine Datei oder via Internet auf einen anderen Rechner.*
- *Bei der Serialisierung eines Objektes werden seine Attribute in einen ByteDatenstrom geschrieben (Klasse ObjectOutputStream) und umgekehrt aus einem ByteDatenstrom gelesen (Klasse ObjectInputStream).*
- *Transiente Attribute (Modifizierer transient) werden nicht serialisiert, weil sie nur momentane Gültigkeit haben oder nicht übertragen werden sollen (z.B. Passwort).*
- *Klassenvariablen (statische Attribute) werden ebenfalls nicht serialisiert, da sie nicht zum Objekt gehören!*



Sie kennen das Konzept eines Marker Interfaces.

Serializable ist ein so genanntes Marker-Interface. Es spezifiziert keine Methode und dient nur zum "Markieren"!

Java - Threads:

Sie können die Begriffe Prozess und Thread sowie Nebenläufigkeit erklären.

Prozess

- *Eine dynamische Folge von Aktionen.*
- *Zum Ablauf ist das Speicherabbild des ausführbaren Programms (Code), Speicher für die Daten, weitere vom Betriebssystem bereitgestellte Betriebsmittel (Ressourcen) und ein Prozessor notwendig. Ein Prozess nutzt diese Ressourcen exklusiv. Ein solcher Prozess wird in der Literatur oft auch "schwergewichtig" genannt.*

Thread

- *Ein Thread ist Teil eines Prozesses. Thread = Aktivitätsträger, leichtgewichtiger Prozess.*
- *Ausführungsstrang mit minimalem Kontext (Stack und Register) in der Abarbeitung eines Programms.*
- *Alle Threads, die zu ein und demselben Prozess gehören, benutzen denselben Adressraum sowie weitere Ressourcen (Betriebsmittel) dieses Prozesses gemeinsam.*

Nebenläufigkeit

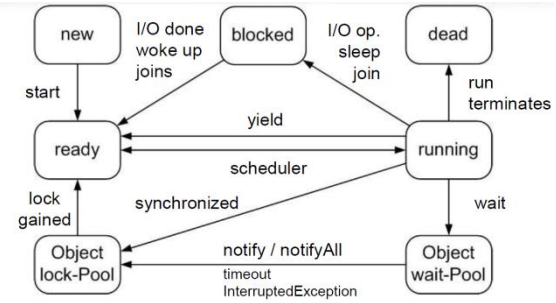
- *Mehrere Vorgänge heissen nebenläufig, wenn sie weitgehend voneinander unabhängig bearbeitet werden können.*
- *Mit der Nebenläufigkeit von Prozessen findet zugleich ein Wettstreit um notwendige Ressourcen statt.*

Sie können den Unterschied von synchronen und asynchronen Messages erläutern.

Bei synchronen Messages warten die Kommunikationspartner aufeinander, d.h. die Kommunikation ist blockierend.

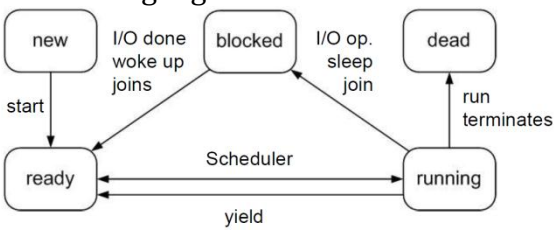
Bei asynchronen Messages warten die Kommunikationspartner nicht.

Sie kennen die wesentlichen Zustände und Zustandsübergänge von Java Threads.



Jedes Objekt hat genau einen Object wait-pool.

Sie können den Java Thread Lebenszyklus zeichnen, der die vorgestellten Zustände und Zustandsübergänge beinhaltet.



new	*Der Thread ist erzeugt, aber noch nicht gestartet.*
ready	*Der Thread ist gestartet, lokaler Speicher (stack) ist zugeteilt, er wartet nur noch auf die Zuweisung des Prozessors.*
running	*Die Anweisungen eines Threads werden auf einem Prozessorkern ausgeführt.*
blocked	*Der Thread muss warten bis eine Bedingung erfüllt ist. Dies sind: Warten bis eine I/O Ressource verfügbar ist, warten bis eine gewisse Zeit abgelaufen (sleep) ist oder warten auf das Ende eines anderen Threads (join).*
dead	*Der Thread existiert nicht mehr und kann vom Garbage Collector entfernt werden, wenn er nicht mehr referenziert ist.*

Sie können die Problematik beim Zugriff mehrerer Threads auf gemeinsame Ressourcen erklären.

Alle Threads eines schwergewichtigen Prozesses teilen dessen Adressraum

- *die Threads können auf gemeinsame Daten zugreifen*
- *die Kommunikation über gemeinsame Daten ist möglich*

Aber: die nebenläufige Manipulation gemeinsamer Datenstrukturen ist problematisch!

- *eine Synchronisation der Aktivitäten ist in der Regel erforderlich!*

Bei nebenläufigen Zugriffen auf eine gemeinsame Ressource kann es zu Konflikten kommen. Solche Zugriffe sollten vom Scheduling System nicht unterbrochen werden, sondern atomar ausgeführt werden können.

Idee: Die Threads reservieren einen Codebereich für sich.

Diesen Vorgang nennt man wechselseitigen Ausschluss (mutual exclusion).

Sie können eine Anwendung mit Threads implementieren.

```
import java.awt.Component;

public class Balloon extends Component implements Runnable {
    private int position;
    private int altitude;
    private Thread workerThread;

    public Balloon() {
        position = 0;
        altitude = 0;
    }

    private void newLocation(int wind) {
        position += wind;
        altitude++;
        System.out.println("Alt: " + altitude + " - Position: " + position);
    }

    public void run() {
        for (int i = 1; i <= 1000; i++) {
            newLocation(5);
            try {
                Thread.sleep(1000);
            } catch(InterruptedException e) {}
        }
    }

    public void start() {
        if (workerThread == null) {
            workerThread = new Thread(this);
            workerThread.start();
        }
    }
}
```

Sie können das Monitorkonzept in Java (synchronized) anwenden.

Durch synchronized kann entweder eine komplette Methode oder ein Block innerhalb einer Methode geschützt werden.

Ein "lock" kann immer nur einmal zu einem gegebenen Zeitpunkt aktiv sein. Wenn mehrere Threads in den kritischer Abschnitt eintreten wollen, werden sie vom Monitor in eine Warteschlange gestellt.

```
public class Test
{
  int count;
  synchronized void bump()
  {
      count++;
  }
  static int classCount;
  static synchronized void classBump()
  {
      classCount++;
  }
}
```

Monitor für das Objekt selber

Monitor für die Klasse

```
public class Counter
{
  private int count = 0;

  public int nextNumber()
  {
    synchronized (this)
    {
      count++;
      return count;
    }
  }
}
```

```
public class Counter
{
  private int count = 0;

  public synchronized
  int nextNumber()
  {
    count++;
    return count;
  }
}
```

erwerbe Monitor für das aktuelle Objekt vom Typ Counter

Die beiden Codeteile bewirken den identischen Schutz der gemeinsamen Ressource `count`.

Sie können die Methoden wait() und notify() anwenden.

Wichtig: *Sowohl wait als auch notify dürfen nur an dem Objekt aufgerufen werden, das bereits gesperrt ist, also nur innerhalb eines synchronized-Blocks für dieses Objekt.*

```
public synchronized guardedJoy()
{
    while ( ! joy) {
        try {
            wait();
        } catch (InterruptedException e) {
        }
    }
    System.out.println("Joy has been achieved!");
}


synchronized (lock)
{
    lock.notify();
}
```

wait()

public final void wait() throws InterruptedException

- *Bei Aufruf von wait wird der aufrufende Thread in einen Wartezustand versetzt, und gleichzeitig wird der Lock auf diesen Abschnitt freigegeben.*
- *Es führen genau drei Wege aus dem Warte-Zustand wieder heraus:*
  1. *Ein anderer Thread signalisiert den Zustandwechsel mittels notify bzw. notifyAll.*
  2. *Die im Argument angegebene Zeit (timeout) ist abgelaufen.*
  3. *Ein anderer Thread ruft die Methode interrupt des wartenden Threads auf.*

notify(), notifyAll()

public final void notify()

- *notify() weckt genau einen Thread im wait-Zustand auf. Falls mehrere Threads warten, ist nicht vorhersehbar oder bestimmbar, welcher Thread aufgeweckt wird.*
- *der Thread wartet noch einmal, bis er den Lock auf den wait-Abschnitt erhält. Erst dann wird er wieder "ready", d.h. bereit zur erneuten Ausführung.*

public final void notifyAll()

- *notifyAll() weckt alle an diesem Objekt wartenden Threads auf.*

Sie verstehen das Singleton Pattern und können es umsetzen.

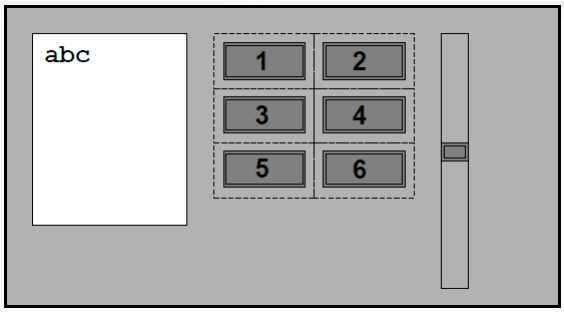
```
public final class Singleton
{
    private static Singleton theInstance;

    private Singleton()
    {
        //…
    }

    public static Singleton getInstance()
    {
        if (theInstance == null) {
            theInstance = new Singleton();
        }
        return theInstance;
    }
}
```

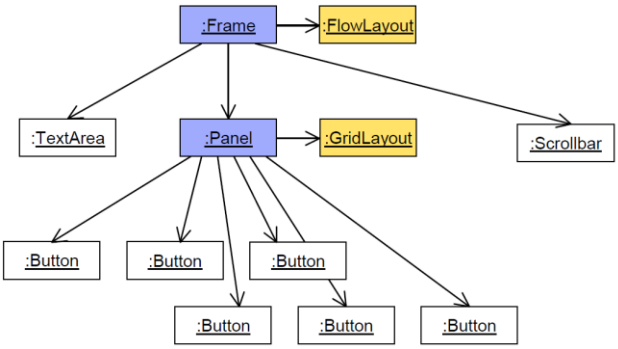
Java - GUI Programmierung:

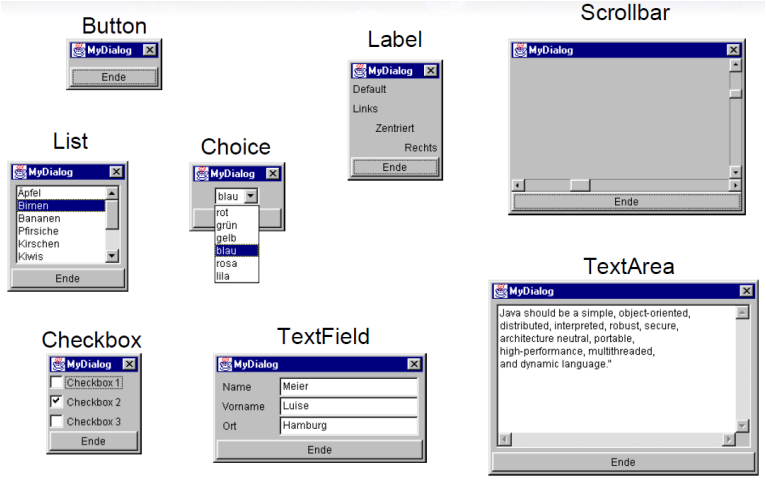
Sie können prinzipiell den Aufbau eines GUI erklären.

GUIs …

- *sind "State of the Art".*
- *sind meistens ergonomisch und intuitiv (Usability).*
- *geben dem Benutzer/der Benutzerin grösstmögliche Freiheit.*
- *sind mit OOP gut umsetzbar. (Jede GUI-Komponente entspricht einem Objekt.)*

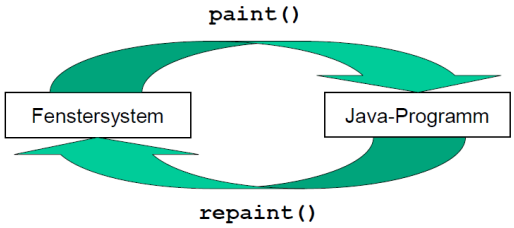






Sie können prinzipiell die Bildschirmausgabe in Windowssystemen erklären.

- *Jede Component hat eine Methode public void paint(Graphics g) zum Zeichnen der GUIKomponente.*
- *Das Betriebssystem bzw. das Fenstersystem ruft paint() automatisch auf, sobald die GUIKomponente neu gezeichnet werden soll, z.B. wenn die FensterGrösse mit der Maus verändert wird.*

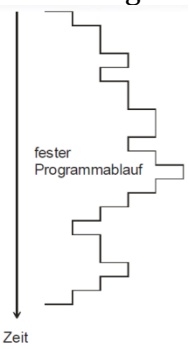


- *Ein JavaProgramm (vgl. Frame, Applet, JFrame, JApplet) kann ein Neuzeichnen via Aufruf von repaint() initiieren, z.B. wenn eine Grafik zu aktualisieren ist. repaint() bewirkt indirekt einen paint()Aufruf.*

Sie kennen die Merkmale eines sequentiellen und ereignisgesteuerten Programms.

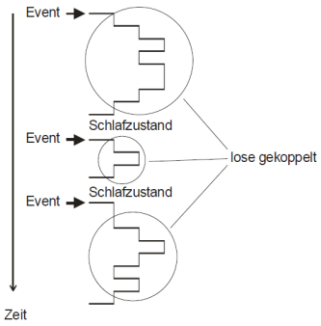
Sequentielles Programm:

- *Aneinanderreihung von Anweisungen*
- *laufen "immer genau gleich" der Reihe nach ab*
- *Programm fordert Benutzer/in zu Eingaben auf*
- *Benutzer/in kann nicht anders in den Ablauf eingreifen*
- *Beispiel: SBB Billett-Automat*



Ereignisgesteuertes Programm:

- *Im Normalfall "schläft" das Programm.*
- *Erst wenn ein Ereignis (Event) eintrifft, geschieht etwas.*
- *Was geschieht/ausgeführt wird, hängt vom Event ab.*
- *Nach der Ausführung fällt das Programm erneut in den "Schlafzustand".*
- *Beispiel: Zeichnungsprogramm*

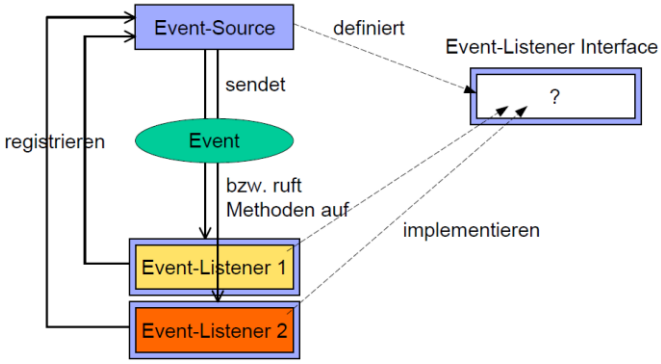


Sie können das Zusammenspiel zwischen Event-Quelle und Event-Listener erklären.

Bei einer Aktion erzeugt die GUI-Komponente ein Event-Objekt, z.B. ein ActionEvent-Objekt.

Die GUI-Komponente leitet das Objekt an alle interessierten Stellen weiter, die sich vorgängig bei der GUI-Komponente registriert haben, z.B. an ein ActionListener-Objekt.

Events werden von Listener-Objekten verarbeitet. Die Event-Source (GUI-Komponenten) unterhält eine Liste mit an diesem Event interessierten Listener (sie wurden vorher registriert). Die Event-Source leitet nun an jeden registrierten Listener den Event weiter.



Sie können mindestens je zwei elementare GUI-Komponenten, Container und Layout-Manager benennen und identifizieren.

GUI-Komponente:

Button, List,Checkbox, Choice, Label, Scrollbar, TextField, TextArea

Container:

Panel, Frame

Layout-Manager:

BorderLayout, FlowLayout, GridLayout, setLayout (null)

Sie kennen Vor- und Nachteile von Swing.

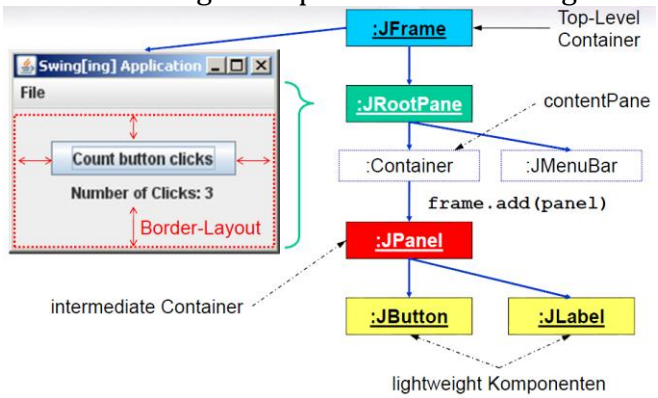
Vorteile:

- *Meilenstein einer neuen GUI-Generation*
  - *unabhängig vom GUI-API des Betriebssystems*
  - *komplette grafische Benutzerschnittstelle mit vielen vordefinierten Komponenten*
- *Herausragende Features*
  - *Plugable Look & Feels: anpassbare grafische Oberfläche (Windows, Motif, Mac)*
  - *vereinfachte Model-View-Controller-Architektur*

Nachteile:

- *Swing ist ressourcenintensiv*
  - *braucht schnelle Prozessoren*
  - *braucht Swing-konformen Web-Browser (Applet)*

Sie kennen wichtige Komponenten von Swing und deren Einsatzmöglichkeit.



Eine zu AWT äquivalente Komponente in Swing hat den gleichen Namen mit dem Präfix "J"

- *Frame →JFrame*
- *Label →JLabel*
- *etc.*

Sie können ein Java-Programm mit GUI (AWT und Swing) und Event-Handling analysieren und interpretieren.

```
import javax.swing.*; // Package für Swing
import java.awt.*; // AWT wird häufig für
import java.awt.event.*; // Ereignisverarbeitung benötigt
public class SwingApplication extends JFrame
{

    private int numClicks = 0;

    public static void main(String[] args)
    {
        JFrame frame = new SwingApplication();
    }

    public SwingApplication()
    {
        super("Swing[ing] Application");
        setDefaultCloseOperation(EXIT_ON_CLOSE); // Anstelle CloseListener
        // Lightweight Components
        JButton button = new JButton("Count button clicks");
        JLabel label = new JLabel("Number Clicks: " + numClicks);
        // ... hinzufügen
        setLayout(new GridLayout(0, 1)); // In JFrame überschriebene Methode
        add(button);
        add(label);
        // Platzbedarf ermitteln und anzeigen
        pack();
        setVisible(true);
    }
}
```

Sie verstehen das Konzept der inneren Klassen und können diese für die Implementierung ereignisgesteuerter Java-Applikationen einsetzen.

- *Innere Klasse: Die Klasse wird in eine andere Klasse hinein genommen.*
- *Anonyme innere Klasse: Ein namenloses Objekt einer namenlosen Klasse wird direkt beim Anmelden des Listeners erzeugt.*
  - *vereinfachte Variante der inneren Klasse*
  - *bei langen Ereignis-Handlern unübersichtlich*

Innere Klasse:

```
public class InnerClassDemo extends JFrame
{
  private JButton button1;

  public InnerClassDemo()
  {
    button1 = new JButton("Drück mich");
    button1.addActionListener( new MyActionListener() );
    add(button1);
  }

  private class MyActionListener implements ActionListener
  {
    public void actionPerformed(ActionEvent e)
    {
      button1.setText("Ich wurde gedrückt");
    }
  }
}
```

Anonyme innere Klasse:

```
public class AnonymousClassDemo extends JFrame
{
  public AnonymousClassDemo()
  {
    JButton button1 = new JButton("Drück mich");
    button1.addActionListener(
      new ActionListener()
      {
        public void actionPerformed(ActionEvent e)
        {
          onButton1Pressed();
        }
      }
    );
    add(button1);
  }

  private void onButton1Pressed()
  {
    …
  }
}
```

// Innere Klasse

Java - Ordnung, Gleichheit und Hash:

Sie können beispielhaft die Unterscheidung von "natürlicher Ordnung" und "spezieller Ordnung" erläutern.

*Mit "**natürlicher Ordnung**" ist jene Ordnung gemeint, die nahe liegend ist bzw. die normalerweise gilt, vgl.*

- *Zahlenwerte: … -2 < -1 < 0 < 1 < 2 < 3 < …*

*Atypisch kann auch mal eine "**spezielle Ordnung**" zum Zug kommen, vgl.*

- *Zahlenwerte: 1 < 7 < 2 < 0 < … (gemäss Länge des Linienzuges)*
- *Zahlenwerte: (0 = 3 = 6 = 8 = 9) < (1 = 2 = 7) < (4 = 5) (gemäss Anzahl "Ecken")*

Sie können die Beziehung zwischen der Methode equals() und den Interfaces Comparable und Comparator erläutern.

*Eine Klasse kann eine **natürliche Ordnung** festlegen, indem sie das Interface **Comparable**<T> implementiert.*

- *Das Interface spezifiziert einzig folgende Methode:*
  - *int compareTo(T other) // Compares this object with the specified object for order.*

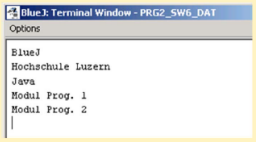
*Mit Hilfe des Interfaces **Comparator**<T> lassen sich beliebig viele **spezielle Ordnungen** festlegen.*

- *Das Interface spezifiziert folgende 2 Methoden:*
  - *int compare(T o1, T o2) // Compares its two arguments for order.*
  - *int equals(Object obj) // Indicates whether some other object is "equal to" this Comparator.*

Sie können die Interfaces Comparable und Comparator für einfache Fälle implementieren.

Comparable

```
import java.util.Set;
import java.util.TreeSet;    // Red-Black-Tree  (vor Java 5 als AVL-Baum)

public class Balloon implements Comparable<Balloon>
{
    ...
    public int compareTo(Balloon other)
    {
        if (this == other)                // 1. Test auf Identität
            return 0;
        return text.compareTo(other.text); // 2. Vergleich relevanter Felder
    }

    public static void main(String[] args)
    {
        Set<Balloon> set = new TreeSet<Balloon> ();
        set.add(new Balloon("Hochschule Luzern"));
        set.add(new Balloon("Hochschule Luzern"));
        set.add(new Balloon("Modul Prog. 2"));
        set.add(new Balloon("Modul Prog. 1"));
        set.add(new Balloon("Java"));
        set.add(new Balloon("BlueJ"));
        for (Balloon b : set){
            System.out.println(b);
        }
    }
}
```



Comparator

```
import java.util.Comparator;


public class SizeUpComparator implements Comparator<Balloon>
{
    public int compare(Balloon o1, Balloon o2)
    {
        return (o1.getSize() - o2.getSize());
    }

    public boolean equals(Object obj)
    {
        if (obj == null){
            return false;
        }
        return (this.getClass() == obj.getClass());
    }

}
```

Sie können die Wichtigkeit von equals() begründen und die Methode equals() gemäss Vier-Schritt-Schema adäquat implementieren.

- *Die Methode equals() ist in der Klasse Object so implementiert, dass sie auf gleiche Identität prüft. Sie versteht ihre "Prüflinge" defaultmässig als Entity Types!*
- *Bei Value Types ist dieses Verhalten falsch! Dann muss man equals() überschreiben!*

Implementations-Schema:

1. *Test auf Identität (Alias-Prüfung)*
2. *Test auf null*
3. *Test auf Typen-Vergleichbarkeit*
4. *Vergleich aller relevanter Felder:*
   - *gewöhnliche Variablen: Vergleich mit ==*
   - *Referenzvariablen: Vergleich wiederum mit equals()*

```
public boolean equals(Object other) // gleicher Text, dann gleiche Ballone
{
    if (this == other)                      // 1. Test auf Identität
        return true;
    if (other == null)                      // 2. Test auf null
        return false;
    if (other.getClass() != this.getClass())// 3. Test auf Vergleichbarkeit
        return false;
    if (!text.equals(((Balloon)other).text))// 4. Vergleich relev. Felder
        return false;
    return true;
}
```

Sie können die drei Punkte des hashCode()-Contractes erläutern und die Methode hashCode() für einfache Fälle implementieren.

Der hashCode()-Contract gibt Grundsätzliches vor:

1. *bestimmte Programmausführung und bestimmtes Objekt*
   → immer derselbe Rückgabewert
   (Falls Persistenz gefordert, dann für beliebige Programmausführungen immer derselbe int-Wert!)
2. *Zwei gleiche Objekte im Sinne von equals()*
   → dieselben Rückgabewerte
3. *Zwei ungleiche Objekte im Sinne von equals()*
   → nicht zwingend verschiedene Rückgabewerte (idealerweise schon)

```
public int hashCode()
{
    return text.hashCode();     // hashCode() von String wird aufgerufen
}                               // (siehe auch nächste Seite)
```

Datenstrukturen:

Sie können mindestens vier Eigenschaften von Datenstrukturen benennen und erklären.

- *Allgemeine elementare Zugriffsmöglichkeiten: Einfügen, Entfernen, Suchen etc. Zugriffsmöglichkeiten im Sinne einer Ordnung: Nachfolgendes/vorangehendes Datenelement etc.*
- *Zugriffsaufwand in Abhängigkeit der n Elemente in der Datenstruktur*
- *Statisch vs. dynamisch: Dynamische Datenstrukturen haben keine feste Grösse und lassen sich zur Laufzeit vergrössern.*
- *Explizit vs. implizit: Bei expliziten Datenstrukturen werden die Beziehungen zwischen den Datenelementen explizit mit Referenzen festgehalten (z.B. Liste, Baum). Bei impliziten Datenstrukturen ist das nicht der Fall (z.B. Array).*
- *Direkt vs. indirekt: Direkte Datenstrukturen bieten direkten Zugriff auf alle Elemente (auf Array via Index).*
- *Indirekte Datenstrukturen bieten keinen direkten Zugriff (bspw. beim Set mit Iterator und next()).*

Sie können an einem Beispiel die gegenseitige Abhängigkeit von Algorithmus und Datenstruktur aufzeigen.

Beispiel:

- *unsortiertes Array: O(1) für das Einfügen, O(n) für das Entfernen und Suchen*
- *Hashtabelle: praktisch O(1) für das Einfügen, Entfernen und Suchen.*

Sie können den Aufwand für die elementaren Zugriffsoperationen für Arrays (sortiert und unsortiert) und Listen (sortiert und unsortiert) angeben.

	Einfügen	*Suchen*	*Entfernen (inkl. Suchen)*	*Nachfolger*	*Vorgänger*	*Sortierte Ausgabe*
Array unsortiert	*O(1)*	*O(n)*	*O(n)*	*Nein*	*Nein*	*Nein*
Array sortiert	*O(n)*	*O(log (n))*	*O(n)*	*Ja*	*Ja*	*Ja*
Liste unsortiert	*O(1)*	*O(n)*	*O(n)*	*Nein*	*Nein*	*Nein*
Liste sortiert	*O(n)*	*O(n)*	*O(n)*	*Ja*	*Nein*	*Ja*

Sie können begründen, weshalb Generics insbesondere bei Datenstrukturen vorteilhaft sind.

Ohne Generics können Datenstrukturen nur Objekte einer Klasse aufnehmen, sind also höchst spezifisch. Für jedes Objekt wäre eine neue Implementierung nötig. Mit Generics muss die Struktur einmal implementiert werden und garantiert die Typsicherheit.

Sie können generische Klassen einsetzen.

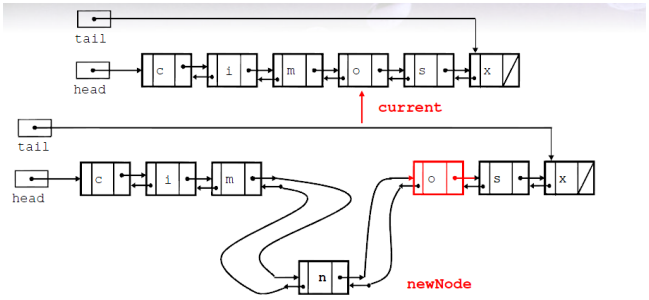
```
public class ListNode<T>
{

    private T data;

    public void setData(T d)
    {
        data = d;
    }
}
```

Sie können eine einfache Liste selber implementieren.

```
public class ListNode
{

    private String data; // Daten; hier ein String
    private ListNode next; // Referenz zum nächsten ListNode-Objekt, rekursiv!
    public ListNode(ListNode n, String d)
    {
        next = n;
        data = d;
    }

    public void setData(String d)
    {
        data = d;
    }

    public String getData()
    {
        return data;
    }

    public void setNext(ListNode n)
    {
        next = n;
    }

    public ListNode getNext()
    {
        return next;
    }
}
```

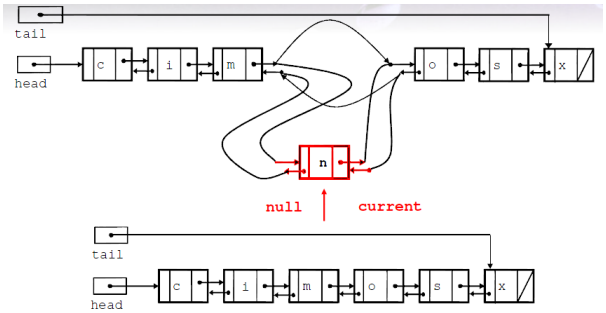
Sie können Einfügen und Löschen in einer doppelt verketteten Liste mit einer Skizze detailliert erläutern.

Einfügen bei doppelt verketteter Liste:




```
newNode.next = current;
newNode.prev = current.prev;
current.prev.next = newNode;
current.prev = newNode;
```

Entfernen bei doppelt verketteter Liste:




```
current.prev.next = current.next;
current.next.prev = current.prev;
current = null;
```

Sie wissen, wann doppelt verkettete Listen vorteilhaft sind.

- *Eine einfach verkettete Liste kann nur in einer Richtung effizient durchlaufen werden.*
- *Abhilfe: Links symmetrisch implementieren, d.h. jeder Knoten erhält eine Referenz next und previous.*
- *Als grundlegende Struktur für Stack, Queue, usw.*
  - *wenn die Anzahl der Elemente a priori unbekannt ist*
  - *wenn die Reihenfolge/Position relevant ist*
- *Wenn Einfügen und Löschen von Elementen in der Mitte häufig ist*

```
public class ListNode<E>
{
    private E data;
    private ListNode<E> next;
    private ListNode<E> prev;
    …
}
```

Sie können einen Stack implementieren.

Stack mit Array implementiert (ohne Fehlerbehandlung)

```
public class Stack
{   private int size;
    private int top = 0;
    private Object[] stack;

    public Stack(int s)
    {   size = s;
        stack = new Object[size];
    }

    public void push(Object o)
    {   stack[top] = o;
        top++;
    }

    public Object pop()
    {   top--;
        return stack[top];
    }

    public boolean isEmpty()
    {
        return(top == 0);
    }

    public boolean isFull()
    {
        return(top == size);
    }
}
```

Array als eigentlicher Stapel

Konstruktor erwartet maximale Anzahl Elemente

top zeigt auf den nächsten zu bearbeitenden Platz

Stack mit ArrayList implementiert

```
import java.util.ArrayList;

public class StackArrayList
{
    private ArrayList stack = new ArrayList();

    public void push(Object o)
    {
        stack.add(o);
    }

    public Object pop()
    {
        return stack.remove(stack.size() - 1);
    }

    public boolean isEmpty()
    {
        return stack.isEmpty();
    }

    public boolean isFull()
    {
        return false;
    }
}
```

Sie können eine Queue implementieren.

Queue Array-Implementation (ohne Fehlerbehandlung)

```
public class Queue
{
    private int size;
    private int nbrElt = 0;
    private int in = 0;
    private int out = 0;
    private Object[] queue;

    public Queue(int s)
    {
        size = s;
        queue = new Object[size];
    }

    public void enqueue(Object o)
    {
        nbrElt++;
        if (in == size) {
            in = 0;
        }
        queue[in] = o;
        in++;
    }

    public Object dequeue()
    {
        nbrElt--;
        if(out == size) {
            out = 0;
        }
        Object o = queue[out];
        out++;
        return o;
    }

    public boolean isEmpty()
    {
        return (nbrElt == 0);
    }

    public boolean isFull()
    {
        return (nbrElt == size);
    }
}
```

Queue List-Implementation

```
import java.util.LinkedList;

public class QueueLinkedList
{
    private LinkedList queue = new LinkedList();

    public void enqueue(Object obj)
    {
        queue.addLast(obj);
    }

    public Object dequeue()
    {
        return queue.removeFirst();
    }

    public boolean isEmpty()
    {
        return queue.isEmpty();
    }

    public boolean isFull()
    {
        return false;
    }
}
```

Sie kennen Anwendungen für die Datenstrukturen Stack und Queue.

Anwendungen für Stacks

- *Parameterübergabe an Methoden*
- *Programmausführung (Programmzähler)*
- *Test auf korrekte Klammersetzung*

Anwendungen für Queues und Warteschlangen

- *ein Drucker für viele Anwender*
- *Allgemein dort, wo der Zugriff auf irgendeine Ressource nicht parallel erfolgen darf, sondern gestaffelt (sequentiell) erfolgen muss.*

Sie können einen Baum definieren. Sie können den Grad und die Höhe eines Baumes und das Niveau eines Knotens bestimmen.

- *Ordnung (order): Maximale Anzahl Kinder, die ein Knoten haben darf (z.B. mind. Ordnung 4).*
- *Grad (degree): Die effektive Anzahl Kinder eines Knotens (nur direkte Kinder).*
- *Höhe (height): Tiefe des am weitesten von der Wurzel entfernten Knotens.*
- *Niveau (level): Knoten mit der gleichen Tiefe (Wurzel hat das Niveau 1, anschliessend nach unten zählen 2,3...)*

Sie können die Funktionen insert, print und search eines binären Suchbaumes implementieren und kennen das Prinzip des Löschens in einem binären Suchbaum.

Binäre Suchbäume besitzen keine, eines oder zwei Kinder.

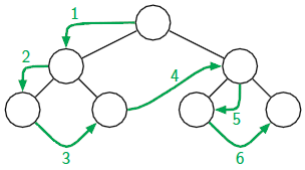
Jeder Knoten weist einen Schlüssel auf (key), der die Daten kennzeichnet.

Jeder Schlüssel im linken Teilbaum eines Knotens ist kleiner als der Schlüssel im Knoten selbst.

Jeder Schlüssel im rechten Teilbaum eines Knotens ist grösser oder gleich dem Schlüssel im Knoten selbst.

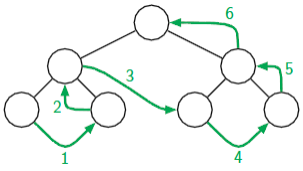
Preorder-Reihenfolge

Bei der Preorder-Reihenfolge (Hauptreihenfolge) wird ein Knoten jeweils vor seinem linken und rechten Teilbaum durchlaufen.



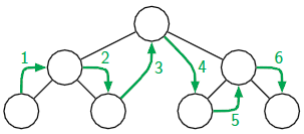
Postorder-Reihenfolge

Bei dieser Durchlaufordnung wird jedoch ein Knoten v nach seinem linken und rechten Teilbaum besucht.



Inorder-Reihenfolge

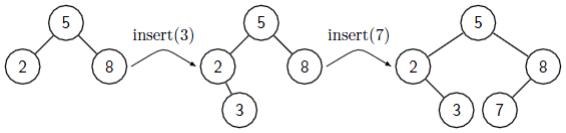
Bei der Inorder-Reihenfolge (symmetrischen Reihenfolge) wird ein Knoten v zwischen dem Durchlaufen seines linken und rechten Teilbaumes besucht.



Search:

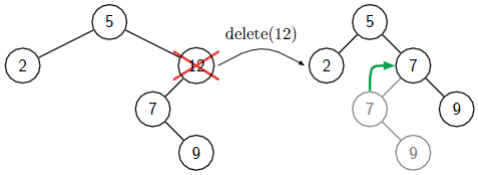
```
algorithm search(v, k)
    //Im Baum mit Wurzel v wird der Schlüssel k gesucht}
    if v 6= null then
        if k < key(v) then
            search(left(v), k) {Suche im linken Teilbaum left(v)}
        else
            if k > key(v) then
                search(right(v), k) {Suche im rechten Teilbaum right(v)}
            else
                Beende Suche {Suche war erfolgreich}
            end if
        end if
    else
        Beende Suche {Suche war erfolglos}
    end if
end algorithm
```

Insert:

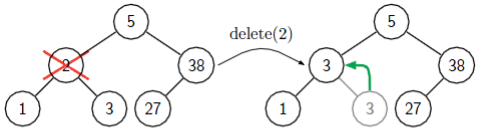


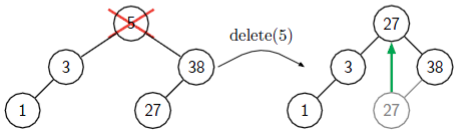

```
algorithm insert(v, k)
    //Im Baum mit Wurzel v wird nach einem Platz gesucht,
    //an dem der neue Knoten mit Schlüssel k gespeichert werden kann.
    if v 6= null then
        if k < key(v) then
            insert(left(v), k) {Knoten muss im linken Teilbaum gespeichert werden}
        else
            insert(right(v), k) {Knoten muss im rechten Teilbaum gespeichert werden}
        end if
    else
        Füge hier den neuen Knoten mit Schlüssel k ein.
    end if
end algorithm
```

Delete:



Delete, Beispiel 2:



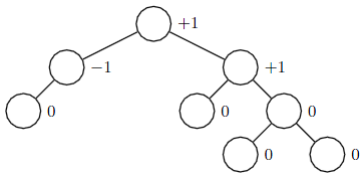


Sie können einen AVL Baum definieren und bestimmen, ob ein gegebener Baum ein AVL Baum ist.

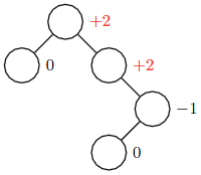
AVL-Bäume sind höhenbalancierte binäre Suchbäume.

Ein binärer Suchbaum ist ein AVL-Baum (oder AVL ausgeglichen), wenn für alle Knoten v des Baums gilt, dass sich die Höhe des linken Teilbaumes von v höchstens um 1 von der Höhe des rechten Teilbaums unterscheidet.

AVL-Baum



***KEIN** AVL-Baum*

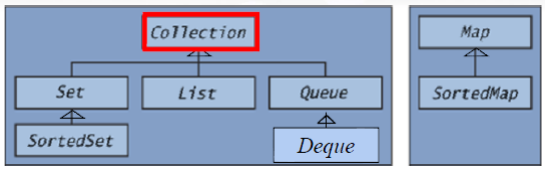


(Balancefaktor ist grösser als 1 resp. -1)

Ein Knoten ist ausgeglichen, wenn sein Balancefaktor -1, 0, oder 1 ist.

Sind alle Knoten ausgeglichen, ist auch der Baum AVL-ausgeglichen.

Sie können den Collection-Begriff erläutern und können mindestens zwei Vorteile nennen, die der Einsatz des Java Collection Frameworks mit sich bringt.



Vorteile:

Geringerer Programmieraufwand:

Datenstrukturen und zugehörige Funktionalität müssen nicht selbst implementiert werden.

Schnell und gut:

Die konkreten Implementationen des Frameworks funktionieren effizient und zuverlässig.

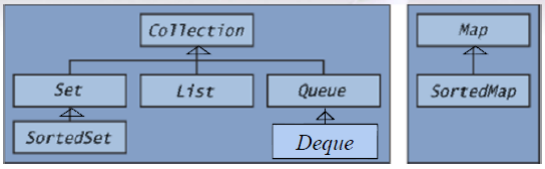
Unterstützt Interoperabilität:

Der Einsatz des Frameworks vereinfacht die Zusammenarbeit zwischen Datenstrukturen.

Fördert die Wiederverwendung:

Der Einsatz standardisierter Interfaces erleichtert die Wiederverwendung von Funktionalitäten und Datenstrukturen.

Sie kennen die Klassenstruktur der Java Collection Interfaces und können diese nutzen.



Sie kennen die verschiedenen Collection Views auf eine Map.

```
for(KeyType key : m.keySet()) {
    System.out.println(key);
}
```

for-each key
für values sinngemäss

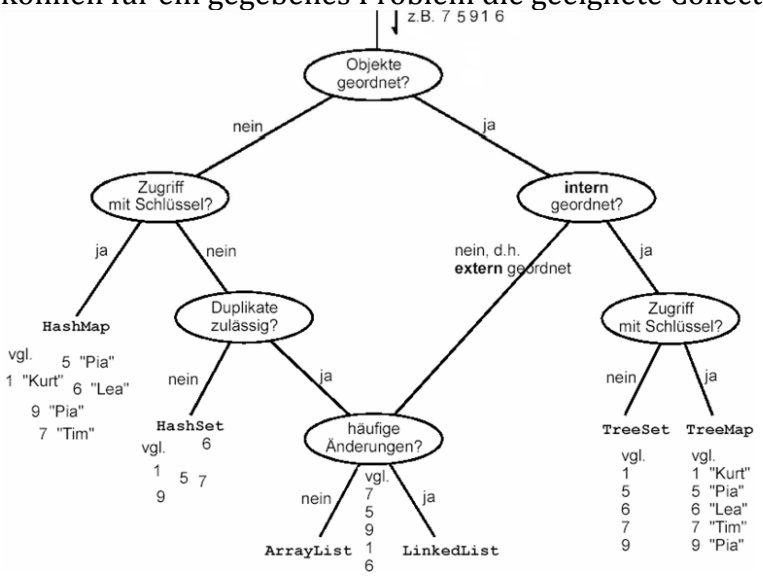
```
Iterator<KeyType> i;
for(i = m.keySet().iterator(); i.hasNext(); ) {
    if(!cond(i.next())) {
        i.remove();
    }
}
```

key Iterator
für values sinngemäss

for-each entrySet

```
for(Map.Entry<KeyType, ValType> e : m.entrySet()) {
    System.out.println(e.getKey() + e.getValue());
}
```

Sie können für ein gegebenes Problem die geeignete Collection-Klasse auswählen.



Software Engineering:

Sie können den Begriff Software-Engineering umschreiben.

Software-Engineering ist Gestaltungsprozess der Projektdurchführung und Projektmanagement.

Sie erkennen Symptome von Software-Entwicklungsproblemen und können wichtige Gründe nennen, wie es dazu kommen kann.

- *(nicht alle) Anspruchsgruppen einbezogen*
- *zu viele Änderungen / moving targets*
- *Kostenüberschreitung wegen „explodierenden" Anforderungen*
- *Fehlendes KnowHow*
- *späte Aufdeckung schwerwiegender Fehler / Konzeptfehler*
- *schlechte Performance / nicht richtig getestet*

Sie kennen wichtige Qualitätsaspekte aus Anwender- und Entwicklungssicht.

Anwendersicht:

- *Funktionserfüllung*
- *Zuverlässigkeit*
- *Effizienz*
- *Benutzbarkeit / Usability*
- *Sicherheit*

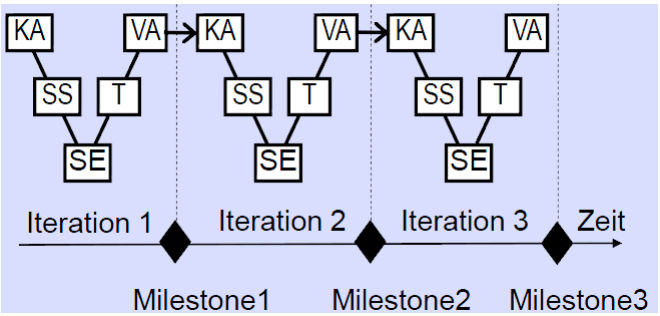
Entwicklungssicht:

- *Portierbarkeit*
- *Wartbarkeit*
- *Erweiterbarkeit*
- *Wiederverwendbarkeit*

Sie können die Begriffe iterativ, inkrementell und agil im Zusammenhang mit Software-Entwicklungsprozessen erklären.

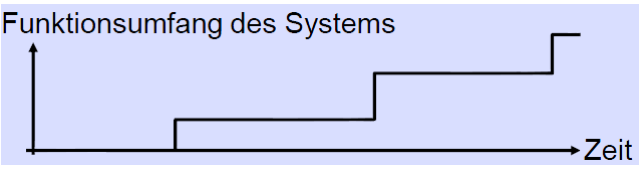
Iterativ:

- *Die Aktivitäten des Entwicklungsprozesses werden mehrmals durchlaufen, wiederholt.*
- *Erstellung von Prototypen, funktionsumfang kann jeweils getestet werden*



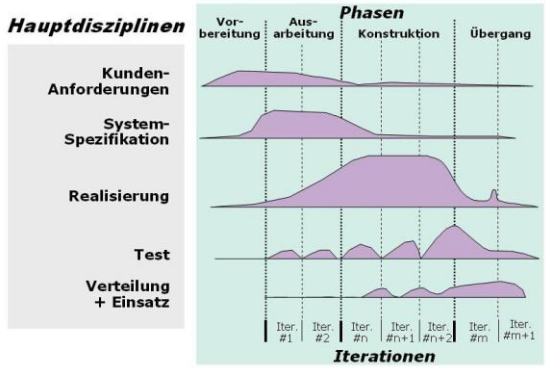
Inkrementell:

- *Jeder Release enthält einen grösseren Funktionsumfang*



Am Ende jeder Iteration steht ein lauffähiger Prototyp bzw. Release.

Sie können am Beispiel von HTAgil die Elemente eines modernen Vorgehensmodells erläutern.



Vorbereitung: *Kundenanforderungen, Grober Projektplan*
Ausarbeitung: *UseCase-Modell, Software-Architektur, Projektplan verfeinert*
Konstruktion: *Implementation Software Komponenten, Testplanung, Handbücher*
Übergang: *Deployment, Abnahme, Beta-Tests, Feedback*

Sie können den Begriff Meilenstein definieren und können für ein Projekt Meilensteine für eine Grobplanung vorschlagen.

Ein Meilenstein ist ein geplanter Punkt im Projektablauf, an dem vorher festgelegte (Zwischen-)Ergebnisse vorliegen, die es erlauben, den Projektfortschritt zu messen.

Zu jedem Meilenstein gehören Artefakte wie Dokumente und Software-Komponente.

Der Meilenstein ist erreicht, wenn die erforderlichen Dokumente vorliegen und ihre Überprüfung erfolgreich war.

Sie können mindestens drei wichtige Merkmale guter Anforderungen nennen.

***Adäquatheit**: Das beschreiben, was der Kunde will bzw. braucht.*
***Vollständigkeit**: Alles beschreiben, was der Kunde will bzw. braucht.*
***Widerspruchsfreiheit**: Sonst ist die Spezifikation nicht realisierbar.*
***Verständlichkeit**: Sowohl für Kunde als auch für Entwickler/innen.*
***Eindeutigkeit**: Fehler durch Fehlinterpretationen vermeiden.*
***Prüfbarkeit**: Anforderungen müssen überprüft werden können.*

Sie kennen das Konzept der Stakeholders.

Stakeholder bezeichnen die am Projekt in irgendeiner Form beteiligten Personen.

Im Prozess des Erfassens und Dokumentieren von Anforderungen besteht der erste Schritt in der Identifikation der Stakeholder. Das können z.B. sein: Auftraggeber, Produktempfänger, Benutzer und Entwickler, etc.

Viele Fehler und Mängel, die unmittelbar nach der produktiven Einführung eines Systems auftreten, lassen sich auf unbeachtete Interessenhalter zurückführen.

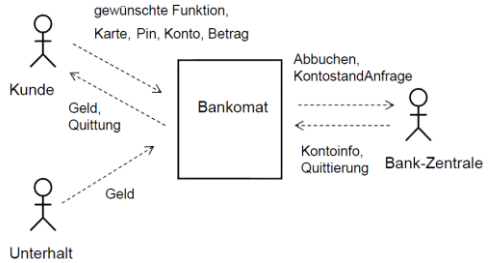
Sie können eine Systemidee formulieren und Ihr System abgrenzen.

Systemidee = „Systembeschreibung auf einem Produktkarton“.

Kurze Formulierung der Produktidee schriftlich unter Berücksichtigung der wichtigsten Eigenschaften, Merkmale, Rahmenbedingung und Voraussetzungen.

Was leistet die Software für den Benutzer? Welche Hard- und Software-Voraussetzungen müssen gegeben sein?

Stakeholder müssen die Systemidee kennen und vorbehaltlos unterstützen. Ein Kontext-Diagramm zeigt die Systemgrenzen auf.



Sie können Anforderungen mittels Use Cases dokumentieren.

Name	*Callcenter reserviert Fahrzeug*
Kurzbeschreibung	*Das Callcenter reserviert für einen Kunden für einen definierten Zeitraum ein Auto.*
Akteure	*Kunde, Callcenter*
Auslöser	*Der Anrufer möchte ein Auto reservieren.*
Vorbedingungen	*Keine.*
Eingehende Infos	*Kundennr., Kundenname, Anrufername, Reservierungswunsch.*
Ergebnisse	*Reservierungsbestätigung.*
Nachbedingung	*Für den Kunden wurde ein Auto reserviert.*
Ablauf	1. *Kunde identifizieren: Der Anrufer nennt seinen Namen, seine Kundennr. und den Kundennamen. Anhand der Kundennr. wird der Kunde gesucht, der gespeicherte Kundenname muss mit dem genannten Kundennamen übereinstimmen.* 2. *Reservierungswunsch aufnehmen: Der Anrufer gibt seinen Reservierungswunsch an, d.h. Fahrzeug-Typ, Reservierungszeitraum, Abhol- und Rückgabeort und besondere Ausstattungsmerkmale.* 3. *Reservierungsmöglichkeit prüfen: Das System ermittelt, ob der gegebene Reservierungswunsch erfüllt werden kann.* 4. *Fahrzeug reservieren: Für den Kunden wird wie gewünscht ein Auto reserviert, er erhält zur Bestätigung eine Reservierungsnr. Es wird kein konkretes Fahrzeug reserviert, sondern nur mengenmässig ein Exemplar des gewünschten Typs.* 5. *Reservierung bestätigen: Dem Anrufer wird die Reservierung mündl. bestätigt, ihm wird die Reservierungsnr. mitgeteilt.*
Nicht-funktional	*Die Prüfung der Reservierungsmöglichkeit muss weniger als 1 s dauern*
Änderungen	*12.8.2004 intern abgestimmt*

Sie wissen, wie Sie Ihr Projekt strukturieren und in Iterationen unterteilen können.

Das Ziel besteht in der klaren Gliederung der Gesamtaufgaben in einzelne plan- und kontrollierbare Arbeitsaspekte (Teilaufgaben). Die Projektabwicklung wird überschaubar.

Die Arbeitspakete sind die Grundbausteine der Planung und Kontrolle.

Sie sollten innerhalb einer Iteration zu erledigen sein.

Sie könne Risiken identifizieren und im Projekt damit umgehen.

Ein Risiko ist ein Problem, dass erst noch auftreten muss. Ein Problem ist ein Risiko, das eingetreten ist.

Risiken identifizieren:

- *Brainstorming Sitzung*
- *Studium ähnlicher Projekte, Interviews mit Projektbeteiligten*
- *Besichtigungsanalyse*
- *Organisationsanalyse*
- *Anforderungsanalyse, Dokumentationsanalyse*

Risiken, die bei vielen Projekten gemeinsam sind:

- *fehlerhafter Zeitplan*
- *Inflation der Anforderungen*
- *Mitarbeiterfluktuation*
- *Spezifikationskollaps*
- *mangelnde Arbeitsleistung*
- *fehlendes Know-how*

Ein Risiko verfügt über eine Eintrittswahrscheinlichkeit, Schaden und Auswirkung. Risiken lassen sich in Risikomatrizen einordnen und bewerten.

Eine Risiko-Neubeurteilung erfolgt in jeder Iteration.

Sie verstehen das Konzept des Projektcontrollings.

Das Projektcontrolling dient zur Überprüfung von gesetzten Teilzielen und zur Kontrolle des Arbeits-Fortschritts. Es geht um den Stand der Arbeit (Fortschritt), um Ressourcen (Kosten, Vergleich geplant und geleistet) und Risiken (neue, sind eingetreten, haben sich verändert?).

➔ *Aussagen des Projekt-Controlling sind im Projektplan zu dokumentieren!*

Sie können erklären, wie beim iterativ-inkrementellen Vorgehen in der Dokumentation mit Änderungen umgegangen wird.

Dokumente werden mit einer Versionsverwaltung versehen (Version, Datum, Autor, Bemerkung, Status).

Ein Konfigurations-Management dient zudem zur Handhabung der Versionen unterschiedlicher Systemteile.

Sie kennen die Bedeutung von Version, Konfiguration und Release.

Version: *Eine Version ist ein spezifisches, identifizierbares Artefakt auf einem bestimmten Entwicklungsstand.*

Konfiguration: *Die Konfiguration bezeichnet eine bestimmte Kombination von Versionen der Systemteile (Hardware, Firmware, Software, Dokumentation).*

Release: *Ein Release ist eine getestete und freigegebene Konfiguration.*

Sie wissen wann und wie Sie ein Refactoring durchführen.

In jeder Iteration wird der Funktionsumfang des Systems gesteigert. Evtl. genügt die Struktur nicht mehr den neuen Anforderungen.

Voraussetzungen sind getestete, lauffähige Systeme.

1. *Falls nicht vorhanden Testspezifikation schreiben und den Code testen*
2. *Code re-designen ohne funktionale Änderungen*
3. *Code entsprechend testen*

Sie wissen, wie Sie die Entwicklung Ihres Softwaresystems dokumentieren.

Die Schnittstellen des Systems sind aus dem Kontextdiagramm ersichtlich. Sie werden in Anforderungs- und Systemspezifikations-Dokument festgehalten.

Auch die Beschreibung der inneren Schnittstellen ist in der Systemspezifikation vorzunehmen.

Sie kennen verschiedene Testarten und das zugehörige Einsatzgebiet.

Testarten:

- *Unit Tests*
- *Integrationstests*
- *Systemtests (testen, ob die Anforderungen erfüllt sind)*
- *Abnahmetests*

Anwendungsfälle sind eine gute Quelle für die Testplanung.

- *Aus jedem Anwendungsfall (Use Case) lassen sich einfach Testfälle (Test Cases) ableiten.*

Ein Testplan definiert und plant die Testaktivitäten.

Ein Testprotokoll hält die Ergebnisse einer Testdurchführung fest.