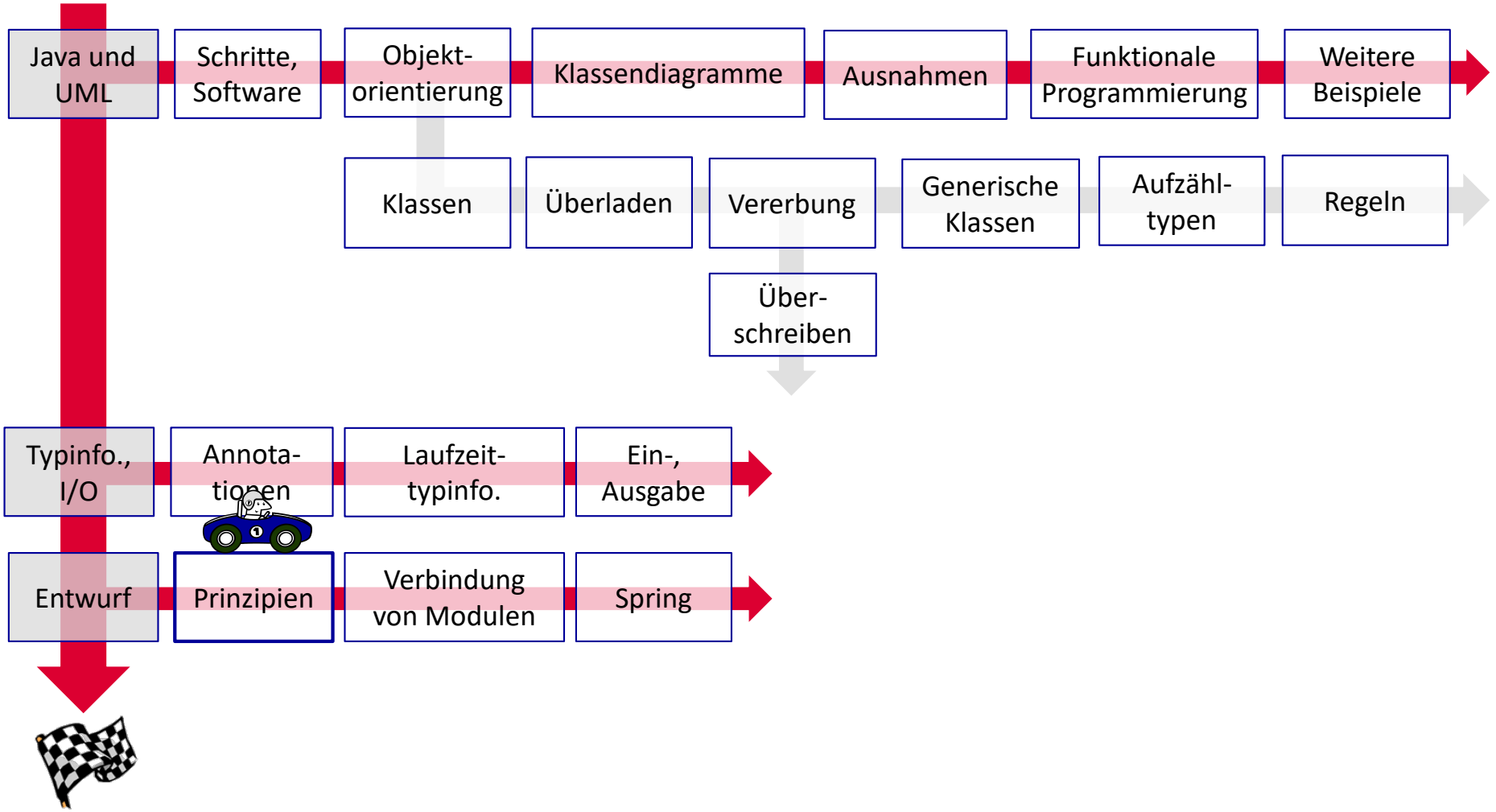
Informatik 2
Objektorientierter Entwurf

Prof. Dr.-Ing. Holger Vogelsang
holger.vogelsang@hs-karlsruhe.de

Inhaltsverzeichnis

Prinzipien der Modularisierung

Übersicht



Prinzipien der Modularisierung

Motivation

- *Was ist ein Modul?*
- *Wie werden Module logisch miteinander verknüpft?*
- *Welche technischen Hilfsmittel unterstützen die Modularisierung?*
- *Aufstellung einiger einfacher Regeln zur Beherrschung der Komplexität von Softwaresystemen.*
- *Vorbereitung der Software, so dass zukünftige Änderungswünsche einfacher eingebaut werden können.*

Prinzipien der Modularisierung

Modul

Definition: Modul

Ein Modul ist ein überschaubarer und eigenständiger Teil einer Anwendung:

- eine Quelltextdatei
- eine Menge von Quelltextdateien
- ein Abschnitt in einer Quelltextdatei

Ein Modul ist etwas, was ein Programmierer als eine Einheit betrachtet, die als ein Ganzes bearbeitet und verwendet wird. Jedes Modul hat nun innerhalb einer Anwendung eine bestimmte Aufgabe, für die es die Verantwortung trägt.

Prinzipien der Modularisierung

Prinzip 1: Prinzip einer einzigen Verantwortung

- **Verantwortung (Responsibility) eines Moduls**:
  - Ein Modul hat innerhalb eines Softwaresystems eine oder mehrere Aufgaben.
  - Das Modul hat die Verantwortung, diese Aufgaben zu erfüllen.

- **Prinzip einer einzigen Verantwortung (Single Responsibility Principle)**:
  - Jedes Modul soll genau eine Verantwortung übernehmen.
  - Jede Verantwortung soll genau einem Modul zugeordnet werden.
  - Die Verantwortung bezieht sich auf die Verpflichtung des Moduls, bestimmte Anforderungen umzusetzen.
  - Es gibt nur einen einzigen Grund, ein Modul anzupassen: Die Anforderungen, für die es verantwortlich ist, haben sich geändert.
- Zusammenfassung: Eine Anforderung (Aufgabe) soll nicht über mehrere Module verteilt oder in mehreren Modulen doppelt implementieren werden.

Prinzipien der Modularisierung

Prinzip 1: Prinzip einer einzigen Verantwortung

- Vorteile des Prinzips:
  - Bessere Änderbarkeit, damit bessere Wartbarkeit:
    - Ändern sich Anforderungen, muss auch die Software geändert werden.
    - Zunächst muss herausgefunden werden, welcher Programmteil die Anforderungen implementiert.
    - Da jedes Modul genau einer Aufgabe zugeordnet ist, ist die Identifikation sehr einfach.
    - Ein Durchsuchen aller möglichen Module nach winzigen Codefragmenten ist nicht erforderlich.
  - Erhöhung der Chance auf Mehrfachverwendung:
    - Ist ein Modul für eine Aufgabe zuständig, wird die Wahrscheinlichkeit, dass das Modul angepasst werden muss, verkleinert.
    - Bei einem Modul, das mehr Verantwortung als nötig trägt, ist die Wahrscheinlichkeit, dass es von mehreren anderen Modulen abhängig ist, größer. Diese andere Module lassen sich u.U. in einem anderen Kontext nicht oder nur schlecht verwenden.

Prinzipien der Modularisierung

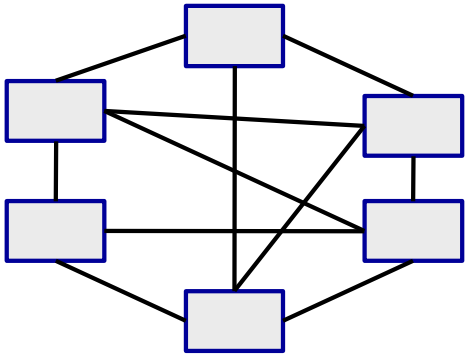
Prinzip 1: Prinzip einer einzigen Verantwortung

- Regeln zur Umsetzung des Prinzips:
  - **Kohäsion maximieren**: Ein Modul soll zusammenhängend (kohäsiv) sein:
    - Alle Teile eines Moduls sollten mit anderen Teilen des Moduls zusammenhängen und voneinander abhängig sein.
    - Haben Teile eines Moduls keinen Bezug zu anderen Teilen, kann man davon ausgehen, dass diese Teile als eigenständige Module implementiert werden können.
  - **Kopplung minimieren**:
    - Wenn für die Umsetzung einer Aufgabe viele Module zusammenarbeiten müssen, bestehen Abhängigkeiten zwischen diesen Modulen → Die Module sind gekoppelt.
    - Die Kopplung zwischen Modulen sollte möglichst gering gehalten werden. Dies kann oft erreicht werden, indem die Verantwortung für die Koordination der Abhängigkeiten einem neuen Modul zugewiesen wird.

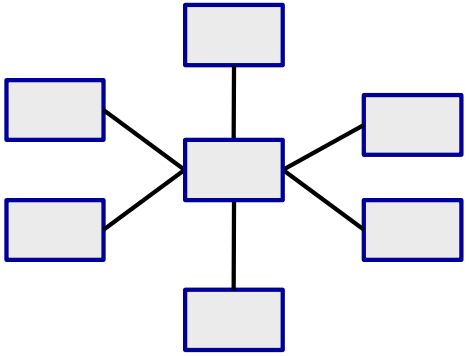
Prinzipien der Modularisierung

Prinzip 1: Prinzip einer einzigen Verantwortung

- Minimierung der Kopplung (graue Kästchen sind Module):



Hoher Kopplungsgrad



Niedriger Kopplungsgrad

- Problem: Im Extremfall koppelt immer ein zentrales Modul → Abhängigkeiten zwischen den Modulen sind nicht mehr erkennbar!
- Lösung: kommt mit den folgenden Prinzipien

Prinzipien der Modularisierung

Prinzip 2: Trennung der Anliegen

- **Anliegen (Concern) eines Moduls**:
  - Eine formulierbare Aufgabe, die zusammenhängend und abgeschlossen ist

- **Prinzip der Trennung der Anliegen (Separation of Concerns)**:
  - Ein im Programm identifizierbares Anliegen soll durch ein Modul repräsentiert werden.
  - Ein Anliegen soll nicht über mehrere Module verstreut sein.

- Sehr ähnlich zu Prinzip 1: Ein Modul soll eine Verantwortung besitzen.

- Beispiele:
  - Trennung von Benutzungsschnittstelle und auszugebender Sprache
  - Trennung von Ausgabe und Berechnung
  - Trennung von Berechnung und Speicherung in einer Datei

Prinzipien der Modularisierung

Prinzip 2: Trennung der Anliegen

- Funktioniert das immer?
  - Beispiel 1:
    - Eine Klasse mit ihren Methoden hat ein Anliegen.
    - Sie will zu Testzwecken Log-Ausgaben vornehmen. Die Log-Ausgaben sind aber ein anderes Anliegen.
    - Das Anliegen „Log-Ausgaben“ wird auf viele Methoden verteilt → Verstoß gegen das Prinzip.
  - Beispiel 2: Bei verteilten Anwendungen (z.B. Web-Anwendungen) muss auf der Server-Seite bei vielen Aufrufen geprüft werden, ob der angemeldete Benutzer überhaupt die Rechte besitzt, die Methoden aufzurufen.
  - Beispiel 3: Methoden sollen Transaktionen unterstützen: Entweder sie laufen komplett durch oder der Zustand der veränderten Objekte wird auf den Startzustand vor dem Aufruf zurückgesetzt (→ siehe später bei Datenbanken).
  - Beispiel 4: Die Ausnahmen vieler Methoden sollen zentral behandelt werden.
- Diese Querschnittsaufgaben („Cross-Cutting Concerns“) wie Protokollierung finden sich überall im Quelltext.

Prinzipien der Modularisierung

Prinzip 2: Trennung der Anliegen

- Ziel ist eine Trennung der eigentlichen Anliegen von den Querschnittsaufgaben.
- Beispiel zur Sicherheitsprüfung:

```
public void changeSemester(Student student, int semester) {
  // Sicherheitsprüfung durchführen: Hat der aktuelle
  // Nutzer Administratorrechte?
  if (!currentUser.isAdmin()) {
    throw new AccessControlException("Illegal access by user");
  }
  student.setSemester(semester);
  saveStudent(student);
}
```

- Viele andere Methoden sind im Projekt ähnlich aufgebaut.
- Die Sicherheitsprüfung ist eine Querschnittsaufgabe und hat somit ein anderes Anliegen als das Ändern der Semesternummer → gehört hier nicht in die Methode. Die Prüfung würde auch per Copy/Paste in viele Methoden eingefügt werden.

Prinzipien der Modularisierung

Prinzip 2: Trennung der Anliegen

- Beispiel zur Protokollierung:

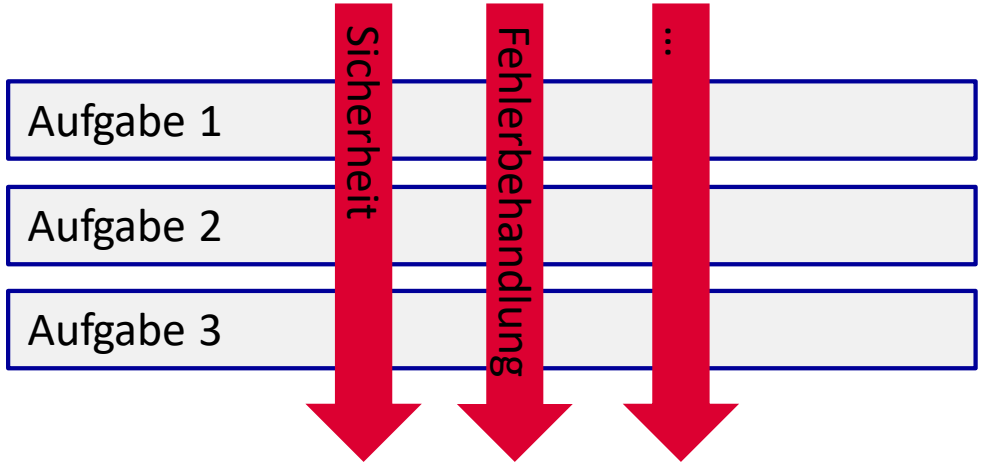
```
public long calculate() {
  // Aufruf protokollieren
  logger.log("Start");
  long result = calculateSomethingVeryComplex();
  logger.log("End");
  return result;
}
```

- Die Protokollierung ist ebenfalls eine Querschnittsaufgabe und somit ein anderes Anliegen als die Berechnung → gehört hier nicht in die Methode.

- Weitere ähnliche Probleme: Eine bestehende Methode soll während des laufenden Programms verändert werden, indem ihr Aufruf an eine andere Methode umgeleitet wird. Selbstmodifizierende Programme???? Eine Anwendung kommt gleich im Kapitel zu Spring.
- Weitere Querschnittsaufgaben: Validierung von Werten, Persistierung von Objekten in Datenbanken, Profiling von Programmen, …

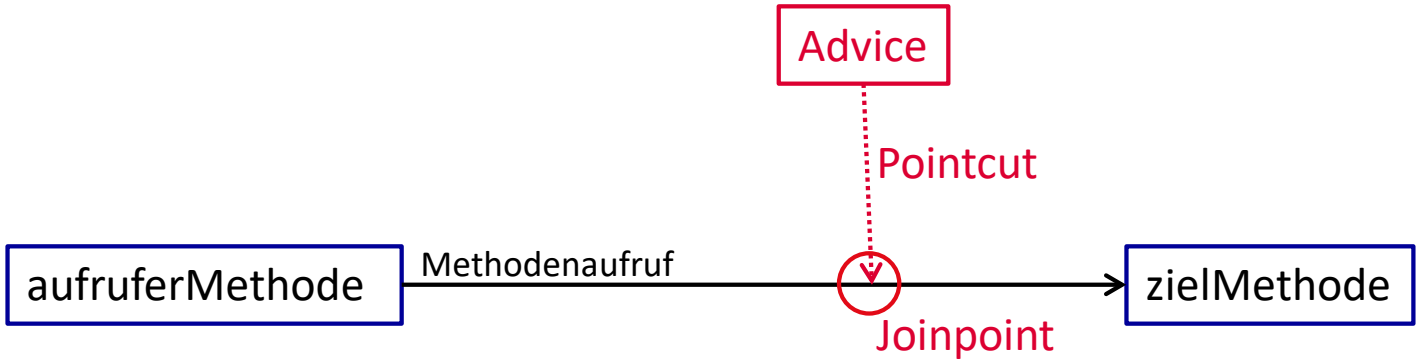
Prinzipien der Modularisierung

Prinzip 2: Trennung der Anliegen



- Wie kommen solche Querschnittsaufgaben denn sonst in den Programmcode?
- Mittels aspekt-orientierter Programmierung (AOP) können Querschnittsaufgaben („Aspekte“) automatisch zur Laufzeit oder Übersetzungszeit in Methoden eingefügt werden.
- AspectJ ist eine Implementierung für Java, die die Aspekte zur Laufzeit in den Bytecode einfügt.

- Ein Aspekt besteht aus den drei rot markierten Teilen:



- Advice:
  - Welcher Code soll ausgeführt werden?
  - Wann soll der Code ausgeführt werden (hier am Beispiel einer Methode)?
    - **`before`**: bevor der Rumpf der Ziel-Methode betreten wird (z.B. für eine Sicherheitsprüfung oder den Start einer Zeitmessung)
    - **`after`**: nachdem die Ziel-Methode verlassen wurde (z.B. für die Beendigung einer Zeitmessung)

  - **`after-returning`**: nachdem die Ziel-Methode erfolgreich beendet wurde (z.B. zum Speichern des veränderten Objektes)
  - **`after-throwing`**: nachdem die Ziel-Methode durch eine Ausnahme beendet wurde (z.B. zur Speicherung des Fehlers)
  - **`around`**: bevor die Ziel-Methode betreten und nachdem die Ziel-Methode verlassen wurde (z.B. für die Protokollierung von Aufrufen)
- Joinpoint: Stelle, an der ein „Advice“ prinzipiell eingefügt wird. AspectJ unterstützt Methoden, Konstruktoren, Ausnahmen und Arrays. Beispiele:
  - in die Methode **`setX`** der Klasse **`Point`**: **`void Point.setX(int)`**
  - bei Auftritt der Ausnahme **`ArrayOutOfBoundsException`**
- Pointcut: Menge von Joinpoints zusammen mit der Angabe, wie deren Advices aktiv werden. Beispiele:
  - beim Aufruf jeder Methoden **`setX`** und **`setY`** der Klasse **`Point`**:

    `call(void Point.setX(int)) || call(void Point.setY(int))`
  - beim Auftreten der Ausnahme: **`handler(ArrayOutOfBoundsException)`**

Prinzipien der Modularisierung

Prinzip 2: Trennung der Anliegen

- Genaueres Beispiel anhand zweier sehr einfacher Klassen **`Fraction`** und **`Vector`**: (Projekt **`AspectJ`**)

```
public class Fraction {
  private long counter;
  private long denominator;
  public Fraction(long counter, long denominator) {
    this.counter = counter;
    this.denominator = denominator;
  }
  public long getCounter() {
    return counter;
  }
  public void setCounter(long counter) {
    this.counter = counter;
  }
  public long getDenominator() {
    return denominator;
  }
  public void setDenominator(long denominator) {
    this.denominator = denominator;
  }
}
```

Prinzipien der Modularisierung

Prinzip 2: Trennung der Anliegen

```
public class Vector<E> {
  private Object[] values;

  public Vector(int size) {
    values = new Object[ size ];
  }

  @SuppressWarnings("unchecked")
  public E getValue(int index) {
    return (E) values[ index ];
  }

  public void setValue(int index, E value) {
    values[ index ] = value;
  }
}
```

Prinzipien der Modularisierung

Prinzip 2: Trennung der Anliegen

- Es wird hier ein Aspekt mit dem Namen **ExampleAspect** erstellt:

```
public aspect ExampleAspect {
  pointcut setter() : call(void set*(..));

  before(): setter() {
    System.out.println("##AJ## Vor Setter " + thisJoinPoint.getSignature());
  }

  after() returning: setter() {
    System.out.println("##AJ## Nach Setter " + thisJoinPoint.getSignature());
  }
}
```

- Der Pointcut mit dem Namen **setter** verwendet einen Joinpoint:
  - Dieser soll beim Aufruf jeder Setter-Methode (fängt mit **set** an) aktiv werden.
  - Der Pointcut gilt für alle Setter-Methoden aller Klassen.
- Die beiden Advices werden vor bzw. nach den Methodenaufrufen ausgeführt, die der Pointcut **setter** beschreibt.

- Das mach der AspectJ-Compiler daraus (am Beispiel des Vektors, die Bruchklasse wird analog behandelt):

```
public class Vector<E> {
  private Object[] values;

  public Vector(int size) {
    values = new Object[ size ];
  }
  @SuppressWarnings("unchecked")
  public E getValue(int index) {
    return (E) values[ index ];
  }

  public void setValue(int index, E value) {
    System.out.println("##AJ## Vor Setter void de.hska.iwii.i2.Vector.setValue(int, "
                       + "Object)");
    values[ index ] = value;
    System.out.println("##AJ## Nach Setter void de.hska.iwii.i2.Vector.setValue(int, "
                       + "Object)");
  }
}
```

```
public aspect ExampleAspect {
  pointcut setter() : call(void set*(..));

  before(): setter() {
    System.out.println("##AJ## Vor Setter " + thisJoinPoint.getSignature());
  }

  after() returning: setter() {
    System.out.println("##AJ## Nach Setter " + thisJoinPoint.getSignature());
  }
}
```

Prinzipien der Modularisierung

Prinzip 2: Trennung der Anliegen

- Ein Advice hat auch Zugriff auf die Übergabeparameter:

```
public aspect ExampleAspect {
  pointcut fractionSetter(Fraction f): target(f) &&
            (call(void setCounter(long)) ||
             call(void setDenominator(long)));

  before(Fraction f): fractionSetter(f) {
    System.out.println("##AJ## Vor Fraction-Setter: " + f);
    Object[] args = thisJoinPoint.getArgs();
    for(Object arg: args) {
      System.out.println("  " + arg);
    }
  }
}
```

- Der Pointcut fängt Aufrufe der Methode **`setCounter`** und **`setDemoniator`** der Klasse **`Fraction`** ab.
- Die Referenz auf das aktuelle Bruchobjekt wird dem Advice übergeben.
- Das Advice kann mit Hilfe einer API die Übergabeparameter auslesen.
- **`thisJoinPoint`** ist eine Referenz auf den Joinpoint, der den Aufruf der Advice verursacht hat.

Prinzipien der Modularisierung

Prinzip 2: Trennung der Anliegen

- Zusammenfassung:
  - Mit aspekt-orientierter Programmierung lassen sich Querschnittsaufgaben von den eigentlichen Ausgaben trennen → saubere Trennung der Anliegen
  - Nachteil: etwas höherer Laufzeitaufwand

- Verwendung in Eclipse:
  - AspectJ über den Eclipse Marketplace herunterladen
  - AspectJ-Projekt erstellen, dann wird der zusätzliche Compiler vor dem Aufruf des Kava-Compilers aktiv.
  - Der AspectJ-Compiler kann auch nachträglich zu einem Projekt hinzugefügt werden.

Prinzipien der Modularisierung

Prinzip 3: Wiederholungen vermeiden

- **Wiederholungen vermeiden (Don't repeat yourself)**:
  - Eine identifizierbare Funktionalität eines Softwaresystems sollte innerhalb dieses Systems nur einmal umgesetzt sein.

- Was passiert, wenn man an verschiedenen Stellen ähnlichen Code hat?
  - Aufspürung und Entfernung solcher Redundanzen

- Beispiele:
  - Statt fester Zahlen immer Konstanten oder Aufzähltypen im Programm verwenden: Fragen wie „Welche Bedeutung hat die Zahl 5 im Quelltext" lassen sich sonst schwer beantworten, besonders dann, wenn die 5 mehrere Bedeutungen hat (Anzahl Werktage in der Woche, Typ eines Werkstücks, …).
  - Häufig wiederkehrende Codestücke müssen in eigene Methoden verschoben werden.
  - Copy/Paste!!!!

Prinzipien der Modularisierung

Prinzip 4: Offen für Erweiterung, geschlossen für Änderung

- **Prinzip der Offenheit für Erweiterungen und Geschlossenheit für Änderungen (Open-Closed Principle)**:
  - Ein Modul soll für Erweiterungen offen sein:
    - Durch die Verwendung des Moduls zusammen mit Erweiterungsmodulen lässt sich die ursprüngliche Funktionalität des Moduls anpassen.
    - Die Erweiterungsmodule enthalten nur die Abweichungen der gewünschten von der ursprünglichen Funktionalität.
  - Ein Modul soll für Änderungen geschlossen sein:
    - Es sind keine Änderungen des Moduls notwendig, um es erweitern zu können.
    - Das Modul hat definierte Erweiterungspunkte, an die sich die Erweiterungsmodule anknüpfen lassen.
    - Ziel: Änderungen an bereits verwendeten Modulen führen möglicherweise zu Problemen, weil sich Anwendungen auf Eigenschaften verlassen.

Prinzipien der Modularisierung

Prinzip 4: Offen für Erweiterung, geschlossen für Änderung

- Beispiel:
  - Taste („Button“) in einer grafischen Benutzungsoberfläche
  - Wie wird ein Modul informiert, wenn die Taste gedrückt wird?
  - Änderung am Button:
    - Für jeden Einsatz muss die Button-Klasse verändert werden → keine Änderung am Original → Kopie der Klasse mit Änderungen
    - Nicht sinnvoll implementierbar!
  - Lösung:
    - Das Modul, das informiert werden möchte, meldet sich am Button an und wird bei einem Tastendruck informiert.
    - Erweiterung: Jemand kann sich anmelden (registrieren).
  - Konzept: Beobachter-Muster (schon bekannt vom JavaFX-Abschnitt her)

Prinzipien der Modularisierung

Prinzip 5: Trennung der Schnittstelle von der Implementierung

- **Prinzip der Trennung der Schnittstelle von der Implementierung (Program to Interfaces)**:
  - Jede Abhängigkeit zwischen zwei Modulen sollte explizit formuliert und dokumentiert sein.
  - Ein Modul sollte immer von einer klar definierten Schnittstelle des anderen Moduls abhängig sein und nicht von der Art der Implementierung der Schnittstelle.
  - Die Schnittstelle eines Moduls sollte getrennt von der Implementierung betrachtet werden können.
  - → siehe Beispiel von der instabilen Basisklasse

- Schnittstelle beinhaltet:
  - bereitgestellte Operationen und deren Parameter
  - komplette Spezifikation, die festlegt, welche Funktionalität ein Modul anbietet

Prinzipien der Modularisierung

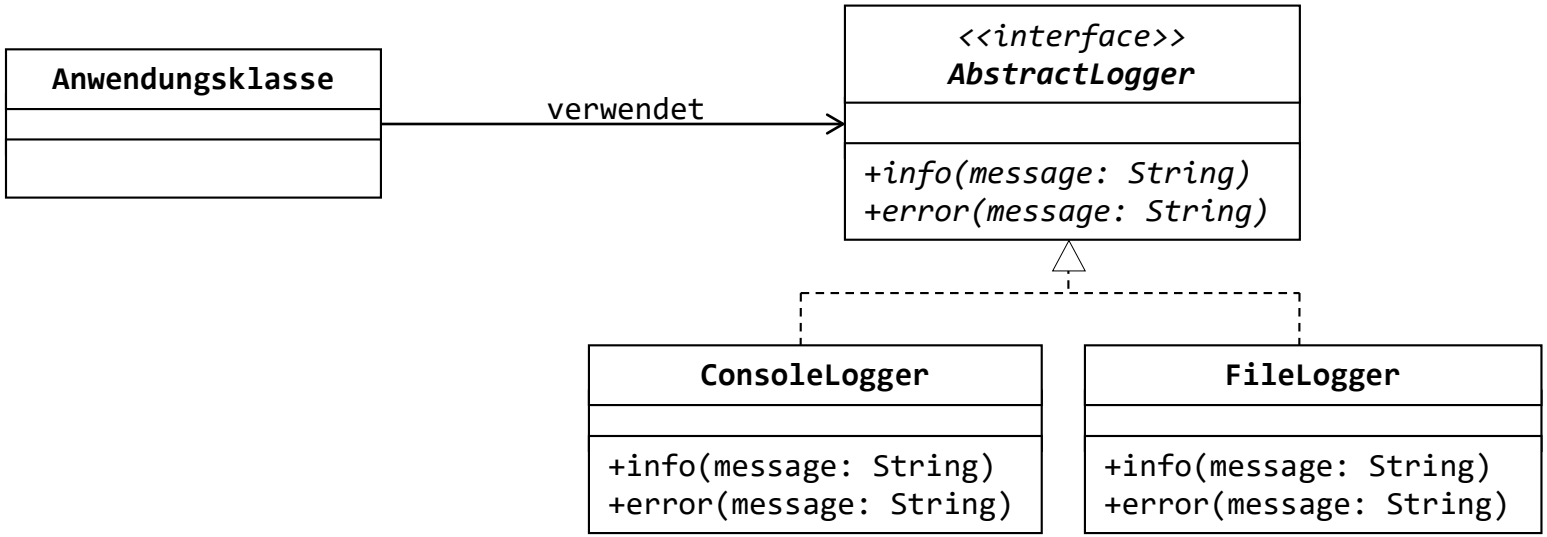
Prinzip 5: Trennung der Schnittstelle von der Implementierung

- Beispiel:
  - Ein Modul möchte Log-Meldungen ausgeben.
  - Dummy-Ansatz: Textmeldungen auf der Konsole ausgeben
  - Besser:
    - Das Modul verwendet ein Log-Modul.
    - Das Log-Modul wird als Schnittstelle bzw. als Klasse mit nur abstrakten Methoden implementiert.
    - Alle Methoden des Log-Moduls besitzen ein genau definiertes Verhalten.
  - Vorteil:
    - Es können beliebige Log-Module verwendet werden: Datenbank-Speicherung, Textdatei, Konsole, …

Prinzipien der Modularisierung

Prinzip 5: Trennung der Schnittstelle von der Implementierung

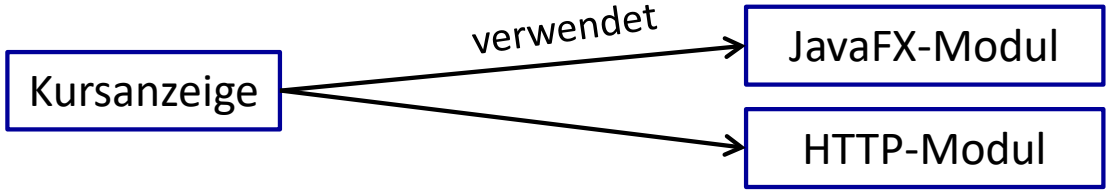
- Die Anwendungsklasse kennt nicht die konkrete Implementierung, sondern nur deren Schnittstelle:



Prinzipien der Modularisierung

Prinzip 6: Umkehr der Abhängigkeiten

- **Abstraktion**
  - beschreibt das in einem gewählten Kontext Wesentliche eines Gegenstandes oder eines Begriffes.
  - blendet Details aus, die für eine bestimmte Betrachtungsweise nicht relevant sind.
  - ermöglicht es, unterschiedliche Elemente zusammenzufassen, die unter einem bestimmten Gesichtspunkt gleich sind.
- Beispiel:
  - Modul, das Börsenkurse anzeigt
  - Die Kurse stammen aus dem Internet und werden mit JavaFX angezeigt.
  - Lösung mit einem klassischen Top/Down-Entwurf nach Prinzip der „einzigen Verantwortung“ und Prinzip „Trennung der Anliegen“:



Prinzipien der Modularisierung

Prinzip 6: Umkehr der Abhängigkeiten

- Probleme:
  - Die Daten sollen nach einer Erweiterung auch aus anderen Quellen (Datenbank, …) kommen.
  - Die Daten sollen auch anders visualisiert werden.

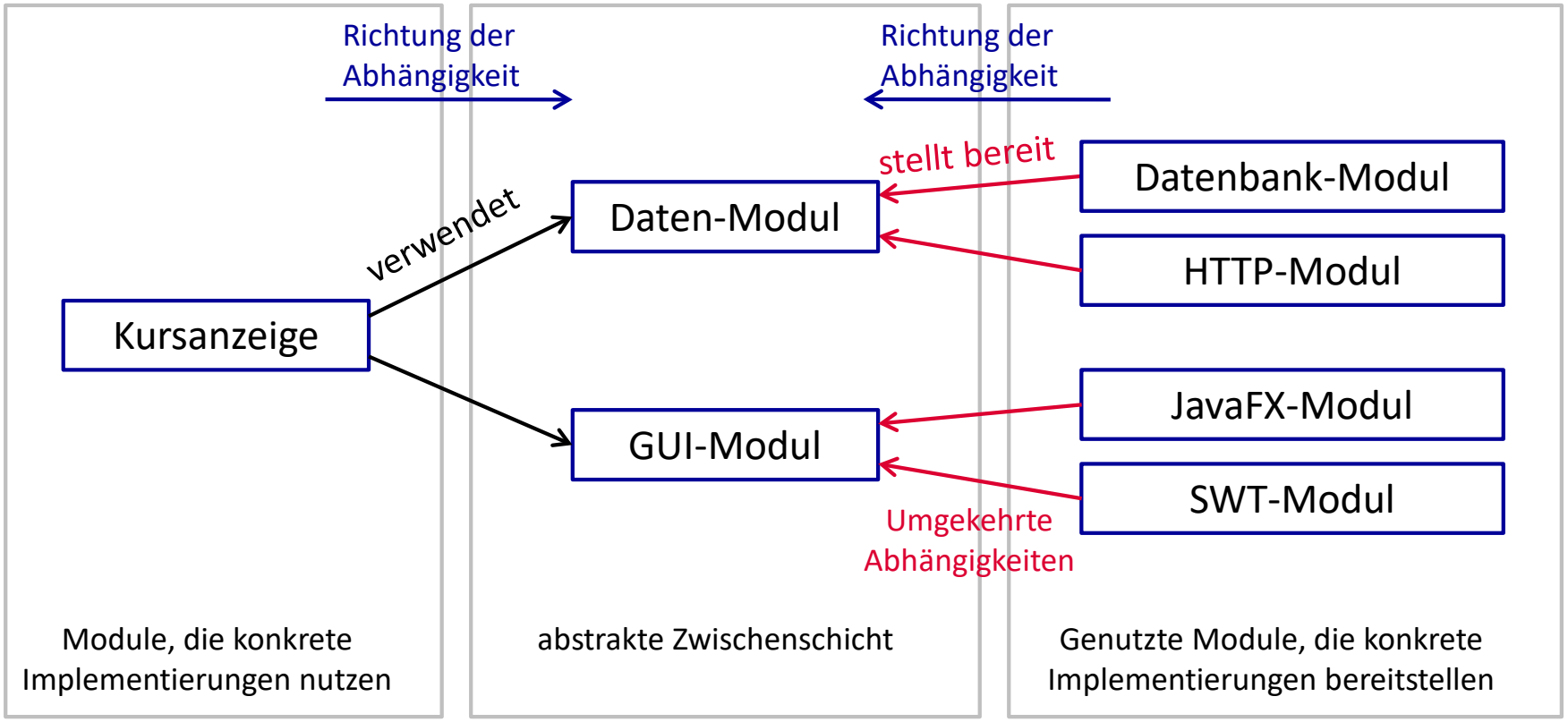
  → Diese Anforderungen können mit diesem Top/Down-Ansatz nur schwer umgesetzt werden.
- Lösung: Umkehrung der Abhängigkeiten
- **Prinzip der Umkehr der Abhängigkeiten (Dependency Inversion Principle)**:

  Der Entwurf der Module stützt sich auf Abstraktionen, nicht auf konkrete Implementierungen.

Prinzipien der Modularisierung

Prinzip 6: Umkehr der Abhängigkeiten

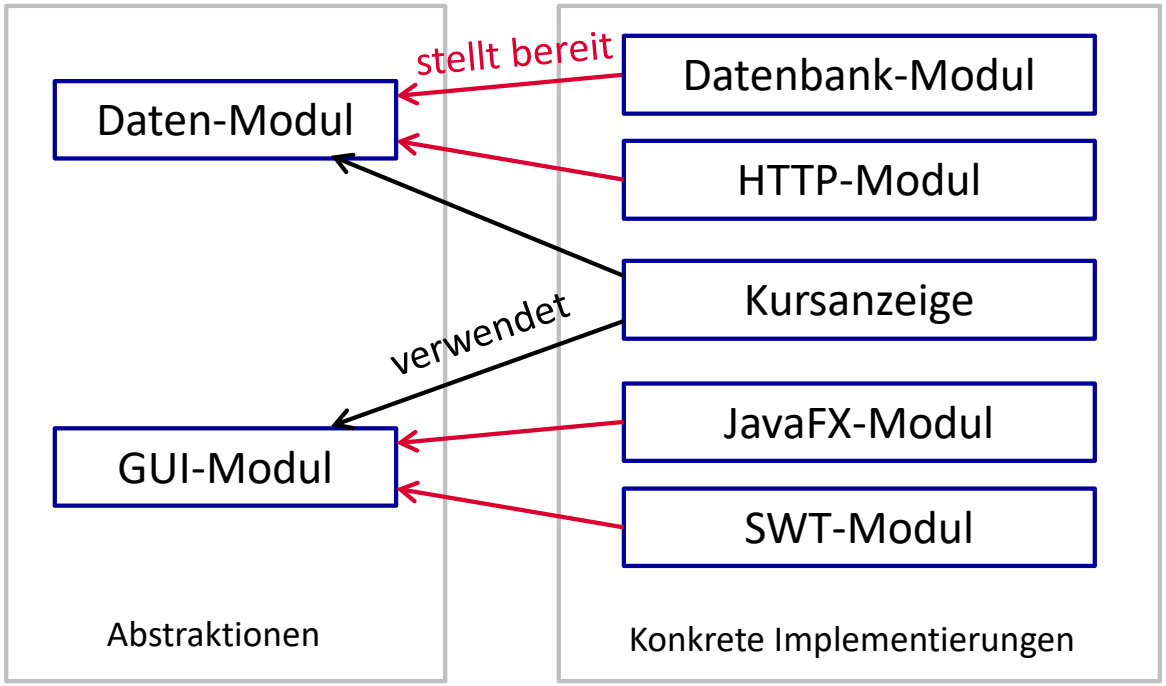
- Abstraktion von konkreten Modulen:
  - Für die HTTP- und Datenbankmodule wird ein abstraktes Datenmodul erstellt.
  - Für die Ausgabe wird ein abstraktes GUI-Modul erstellt.



Prinzipien der Modularisierung

Prinzip 6: Umkehr der Abhängigkeiten

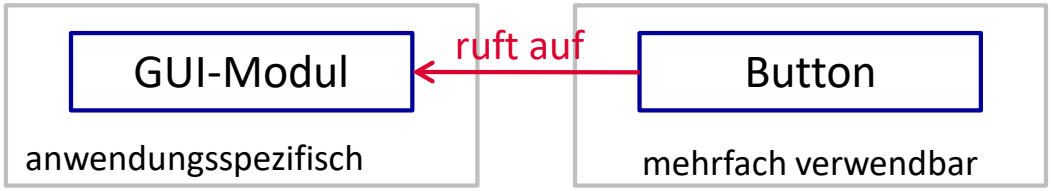
- Vorteile des Prinzips:
  - Bei Änderungen an den konkreten, genutzten Modulen sind selten Änderungen an den nutzenden Modulen erforderlich.
  - Weitere konkrete Module lassen sich leicht ergänzen und wie existierende nutzen.
- Ergebnis des Prinzips:
  Konkrete Module
  sind von abstrakten
  und nicht von anderen
  konkreten Modulen
  abhängig.



Prinzipien der Modularisierung

Prinzip 7: Umkehr des Kontrollflusses

- Prinzip 7 ist eine Anwendung des „Prinzips der Umkehr der Abhängigkeiten“
- **Umkehrung des Kontrollflusses (Inversion of Control)**:
  - Ein spezifisches Modul wird von einem mehrfach verwendbaren Modul aufgerufen.
  - Hollywood-Prinzip: „Don’t call us, we’ll call you“.
- Beispiel:
  - Die Behandlung von Ereignissen wird in einem mehrfach verwendbaren Modul bereitgestellt.
  - Das mehrfach verwendbare Modul übernimmt die Aufgabe, die anwendungsspezifischen Module aufzurufen, wenn bestimmte Ereignisse stattfinden.
  - Die spezifischen Module rufen also die mehrfach verwendbaren Module nicht auf, sie werden stattdessen von ihnen aufgerufen.



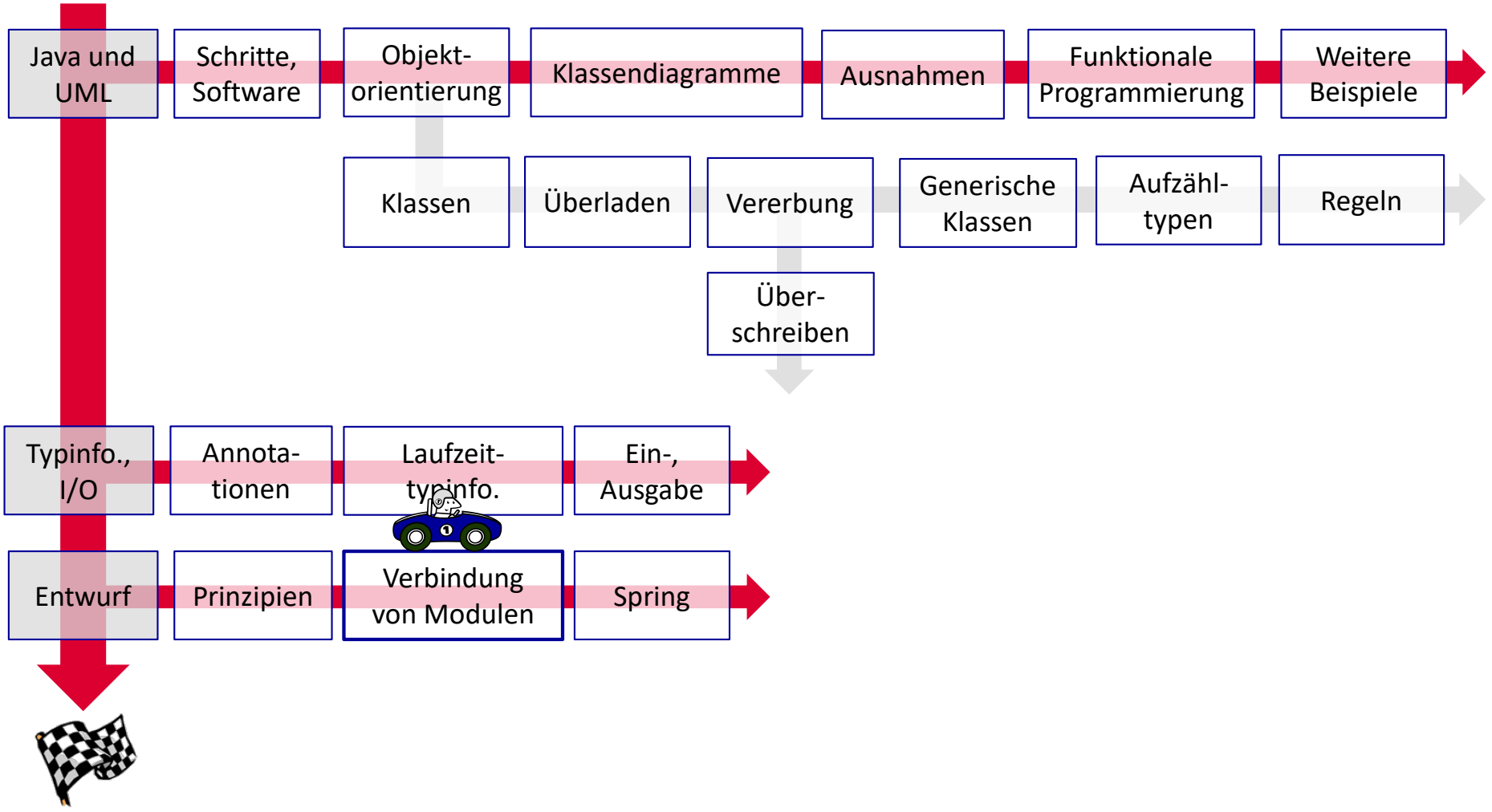
Prinzipien der Modularisierung

Prinzip 8: Mache es testbar

- Ziel des Prinzips: Konstruktion der Software, so dass sie leichter automatisch zu testen ist.
- **Unit-Tests**:
  - Ein Unit-Test ist ein Teil eines größeren Testprogramms, das die Umsetzung einer Anforderung an ein Modul überprüft.
  - Unit-Tests können automatisiert bei der Übersetzung der Software laufen, um Fehler schnell zu erkennen.

  Häufig hilfreich: Mock-Objekte
- **Mock-Objekte**:
  - Platzhalter für echte Objekte, um das Testen zu erleichtern.
  - Beispiele: Ein Mock-Objekt, dass
    - statt einer langsamen Datenbank eingesetzt wird.
    - statt indeterministischer Objekte verwendet wird.
- Konsequenz: Die Forderung nach Testbarkeit verändert das Design. Mock-Objekte sind ein Kandidat für „Umkehr der Abhängigkeiten“.

Verbindung von Modulen
Übersicht



Verbindung von Modulen
Motivation

- *Wie kann die Erstellung von Modulen von deren Verknüpfung getrennt werden?*
- *Wie lassen sich unterschiedliche Modulzusammenstellungen leicht herstellen?*
- *Ziel: Die Kopplung von Modulen erfolgt über ein separates Modul.*

Verbindung von Modulen

Java-Beans

- Die Java-SE kennt bereits den Begriff der Komponente: Bean
- Eigenschaften einer Bean (soweit diese für die Vorlesung relevant sind):
  - ein parameterloser Konstruktor
  - Getter- und Setter-Methoden für Attribute, die sich am besten an den Standard zur Namensvergabe halten.
  - Beispiel:
    - Attribut: `private int age`
    - Getter: `public int getAge()`
    - Setter: `public void setAge(int age)`
    - Name der Eigenschaft („property“): `age`
  - Ausnahme `boolean`-Attribut:
    - Attribut: `private boolean retired`
    - Getter: `public boolean isRetired()`
    - Setter: `public void setRetired(boolean retired)`
    - Name der Eigenschaft („property“): `retired`

Verbindung von Modulen

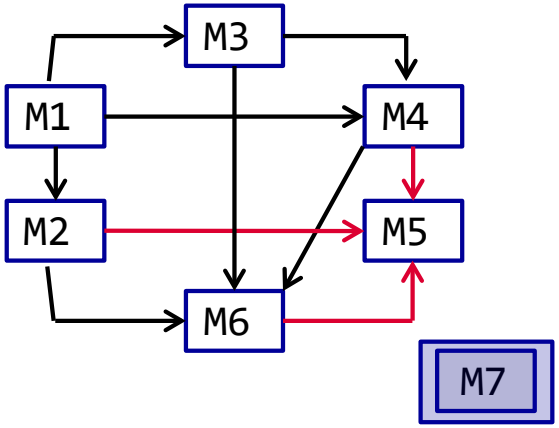
Java-Beans

- Weitere Eigenschaften, die für die Vorlesung nicht relevant sind:
  - Beans kennen Standard-Ereignisse, die bei Änderungen von Attributen ausgelöst werden (`PropertyChangeEvent`, `VetoableChangeEvent`).
  - Beans können eigene Ereignistypen definieren.
  - Beans können sich durch Bean-Info-Klassen selbst beschreiben.
  - Es gibt eine API, mit der Beans zur Laufzeit untersucht werden können (`java.beans.Introspector`).

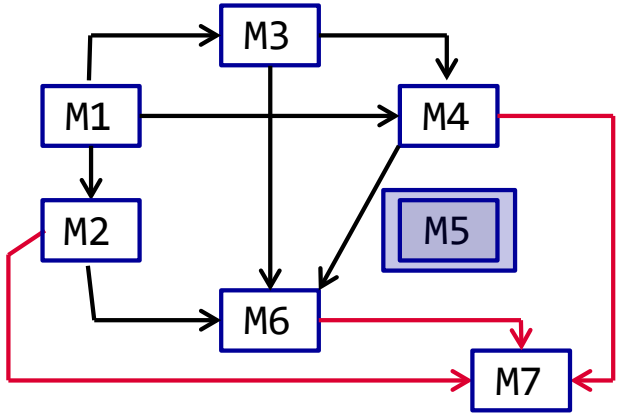
- JavaFX setzt stark auf den Bean-Ansatz, um Eigenschaften verschiedener Objekte zu koppeln (siehe auch Animation).

Verbindung von Modulen

- Fragestellungen:
  - Wie sollen Module gekoppelt (verbunden) werden? Wer macht das?
  - Soll jedes Modul seine notwendigen Abhängigkeiten selbst herstellen?
  - Problem: Je nach Kunden, auszuliefernder Version und Zustand (Test/Betrieb) der Anwendung sind unterschiedliche Kopplungen sinnvoll.



Betrieb (ohne **M7**)

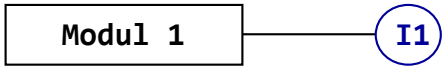


Test (ohne **M5**)

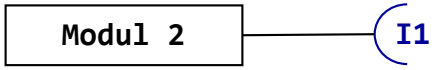
Verbindung von Modulen

- Weitere Probleme:
  - Je nach Konfiguration sollen die Module unterschiedliche Parameter erhalten (andere Datenbank, andere Benutzeranmeldung, …).
  - Im komplexen Graphen von Abhängigkeiten entsteht das Problem der Initialisierungsreihenfolge der Module.

- Lösungsidee („Trennung der Schnittstelle von der Implementierung“):
  - Angebot: Die Module geben Schnittstellen nach außen bekannt, die andere Module nutzen dürfen.

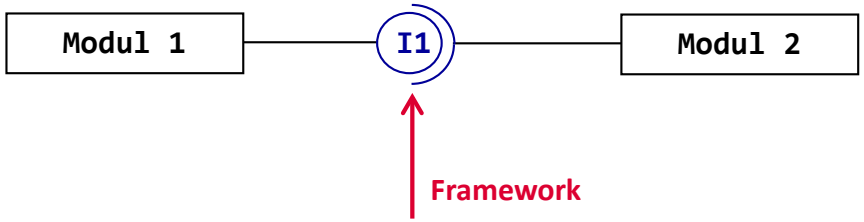


  - Nachfrage: Module definieren die Schnittstellen, die sie von anderen Modulen benötigen.



Verbindung von Modulen

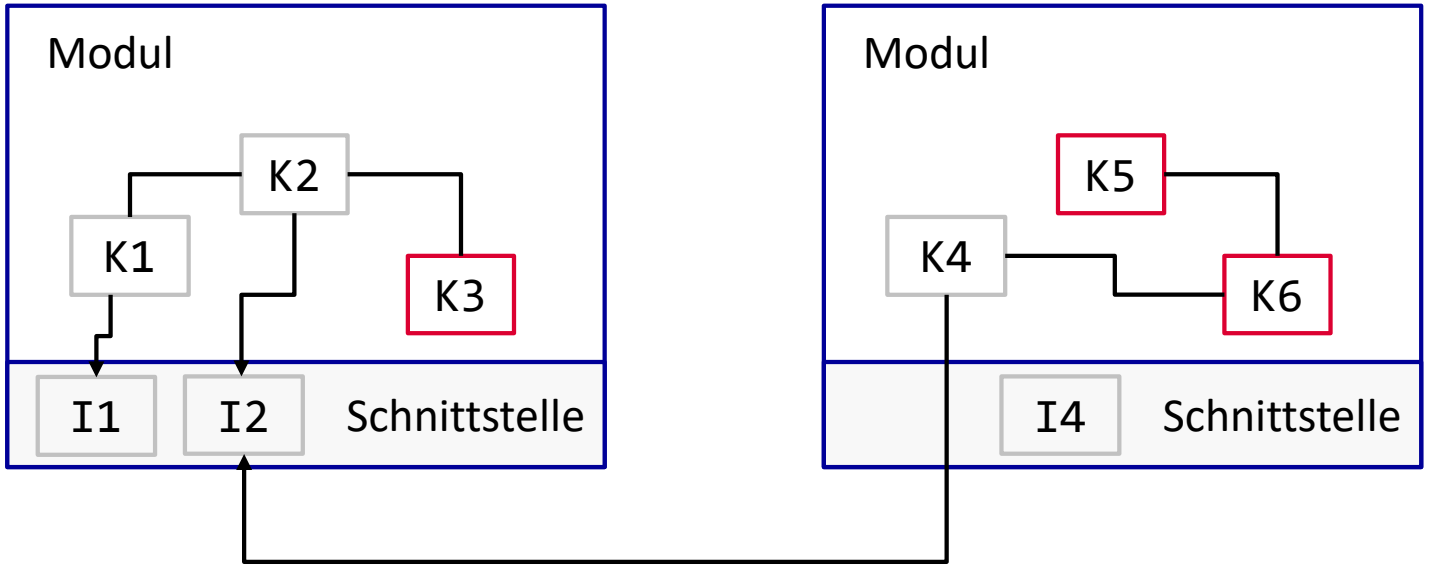
- Ein externes Framework verknüpft automatisch Angebot und Nachfrage: keine direkte Kopplung durch die Module selbst



- Anmerkung: „Schnittstelle“ muss nicht zwangsläufig ein Interface in Java sein. Es ist lediglich eine definierte öffentliche Schnittstelle, die genutzt werden darf:
  - Java-Interfaces
  - Klassen
  - abstrakte Klassen

Verbindung von Modulen

- Beispiel-Module mit öffentlichen Schnittstellen:



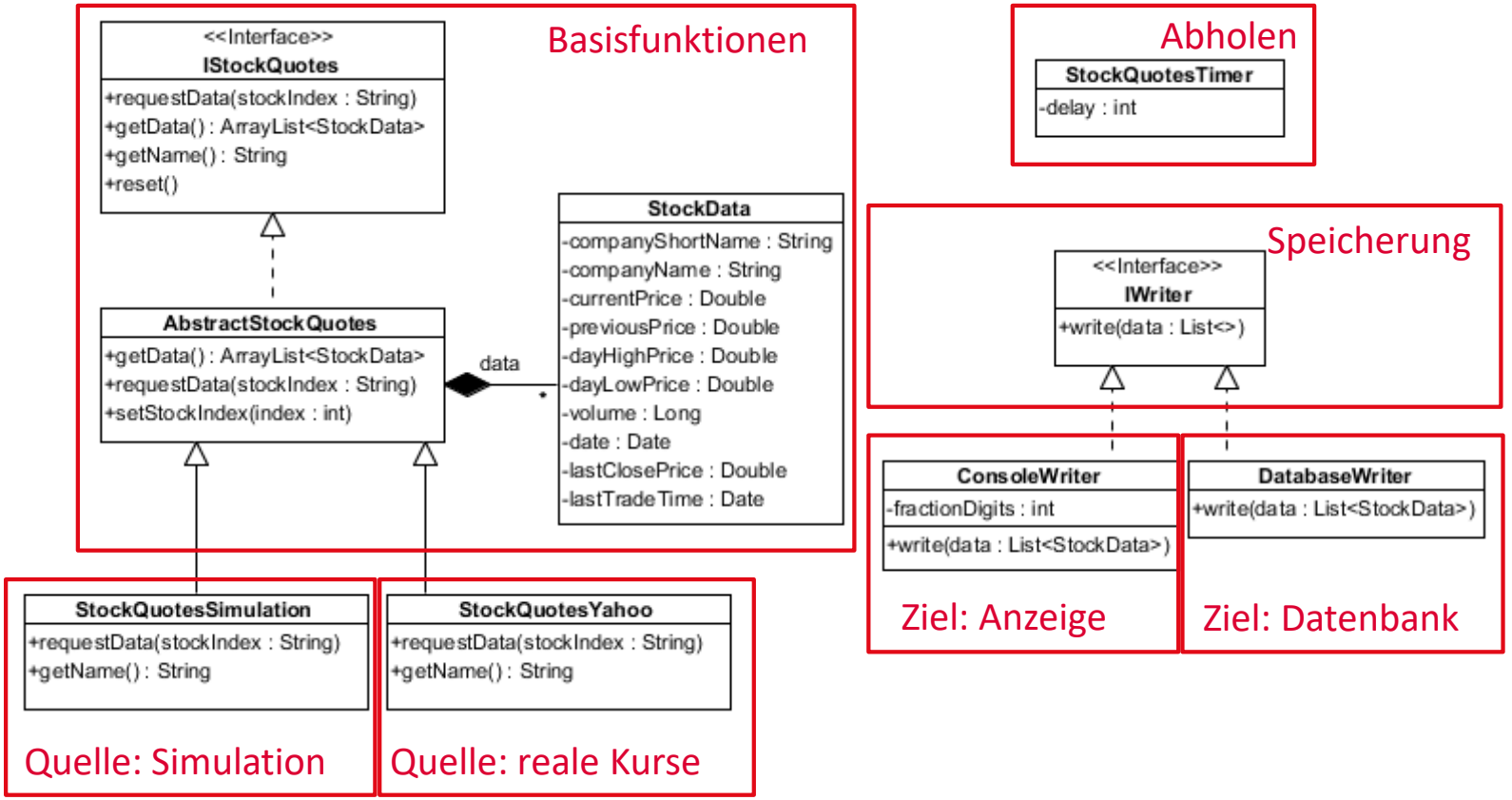
K4 verwendet **K2**,
kennt aber nur dessen
Schnittstelle **I2**

Verbindung von Modulen

- Die folgenden Einführungen verwendet ein kleines Beispiel:
  - Es können Börsenkurse von einem Yahoo-Server abgeholt werden.
  - Die Kurse können aber auch aus einer Simulation stammen.
  - Das Abholen erfolgt zeitgesteuert in einem festen Takt.
  - Die Ausgabe kann auf verschiedene Ziele erfolgen. Hier wird nur die Konsole eingesetzt.
- Hinweis: Den Schnittstellennamen ist ein „I“ vorangestellt. Das ist in Java unüblich, stammt aber aus der „Eclipse Rich Client Platform“, für die das Beispiel ursprünglich entstanden ist.

Verbindung von Modulen

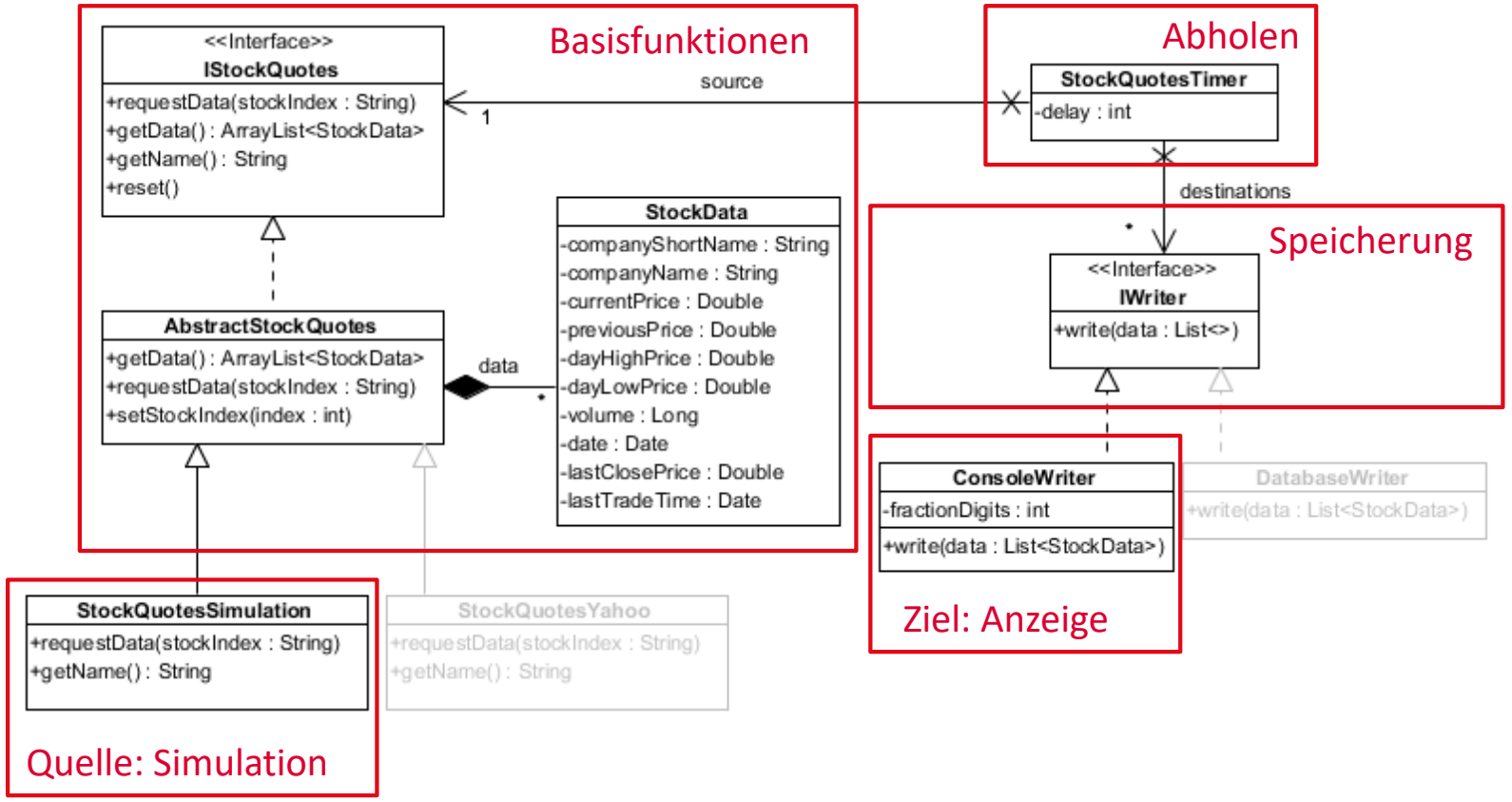
- Modulübersicht ohne Verknüpfungen (nicht alle Attribute sichtbar):



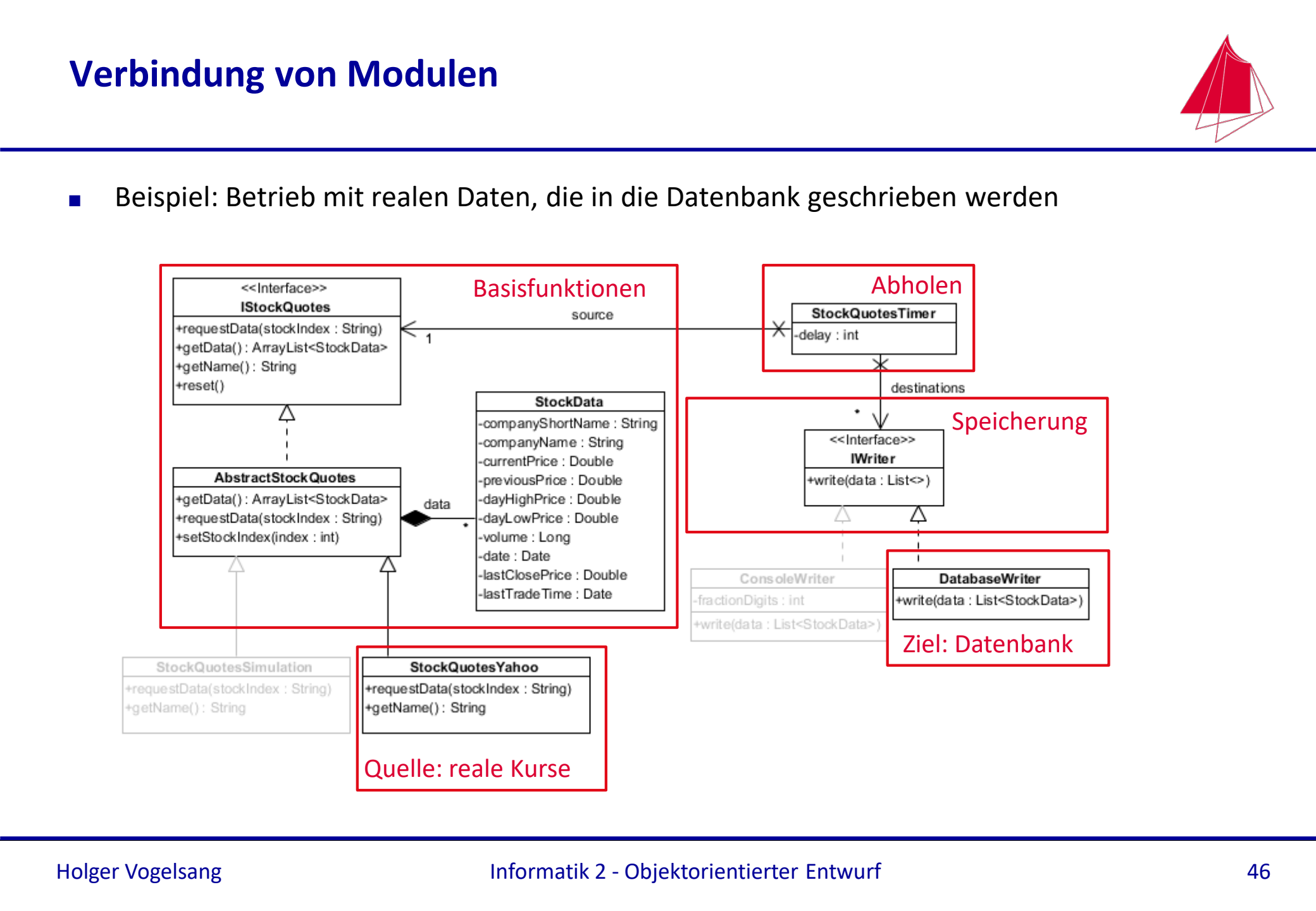
- Aufgabe: zeitgesteuertes Abholen aus Quellen und Speicherung/Anzeige in Zielen

Verbindung von Modulen

- Beispiel: Test mit simulierten Daten, die auf der Konsole ausgegeben werden



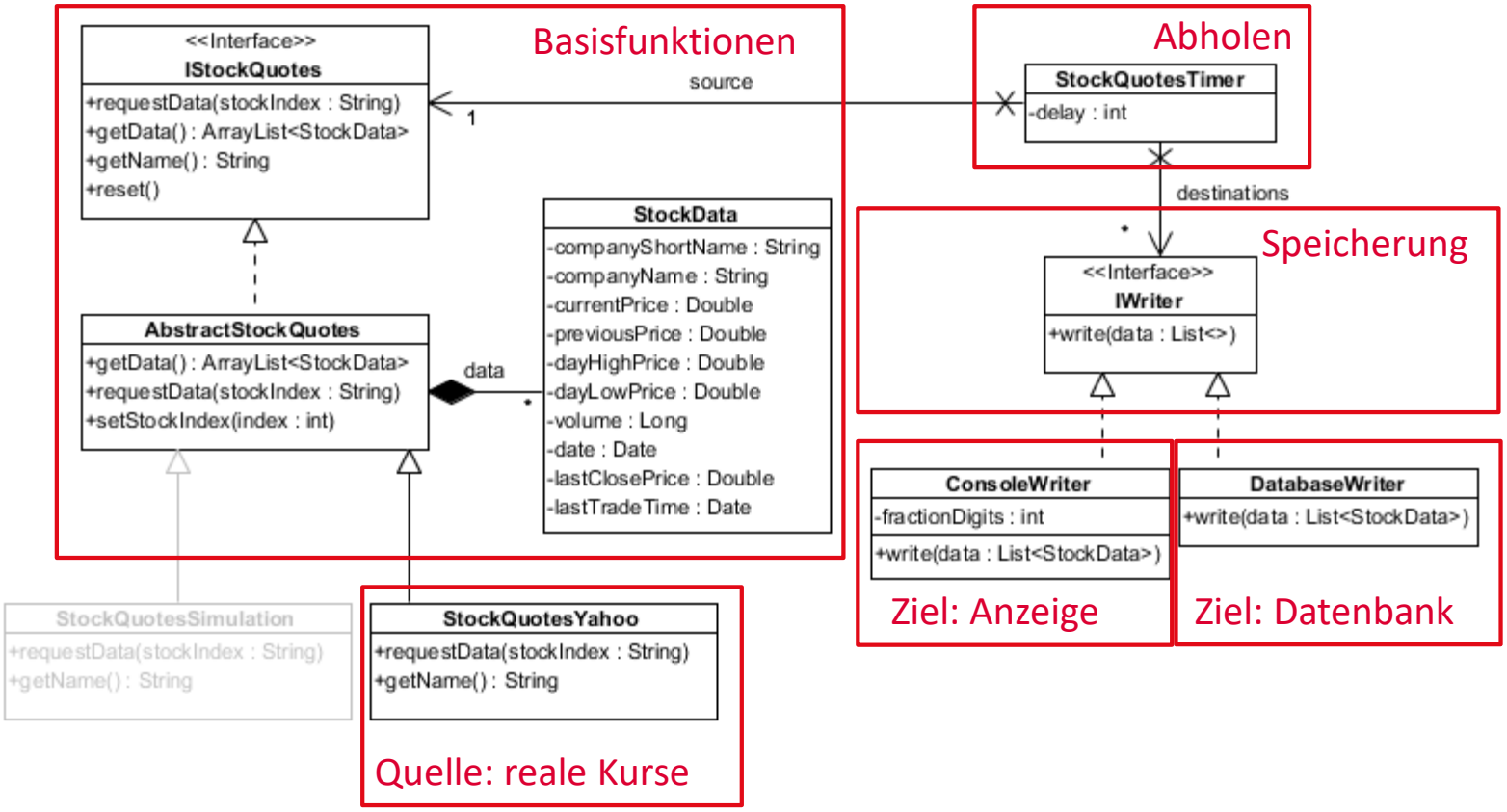
Verbindung von Modulen



- Beispiel: Betrieb mit realen Daten, die in die Datenbank geschrieben werden

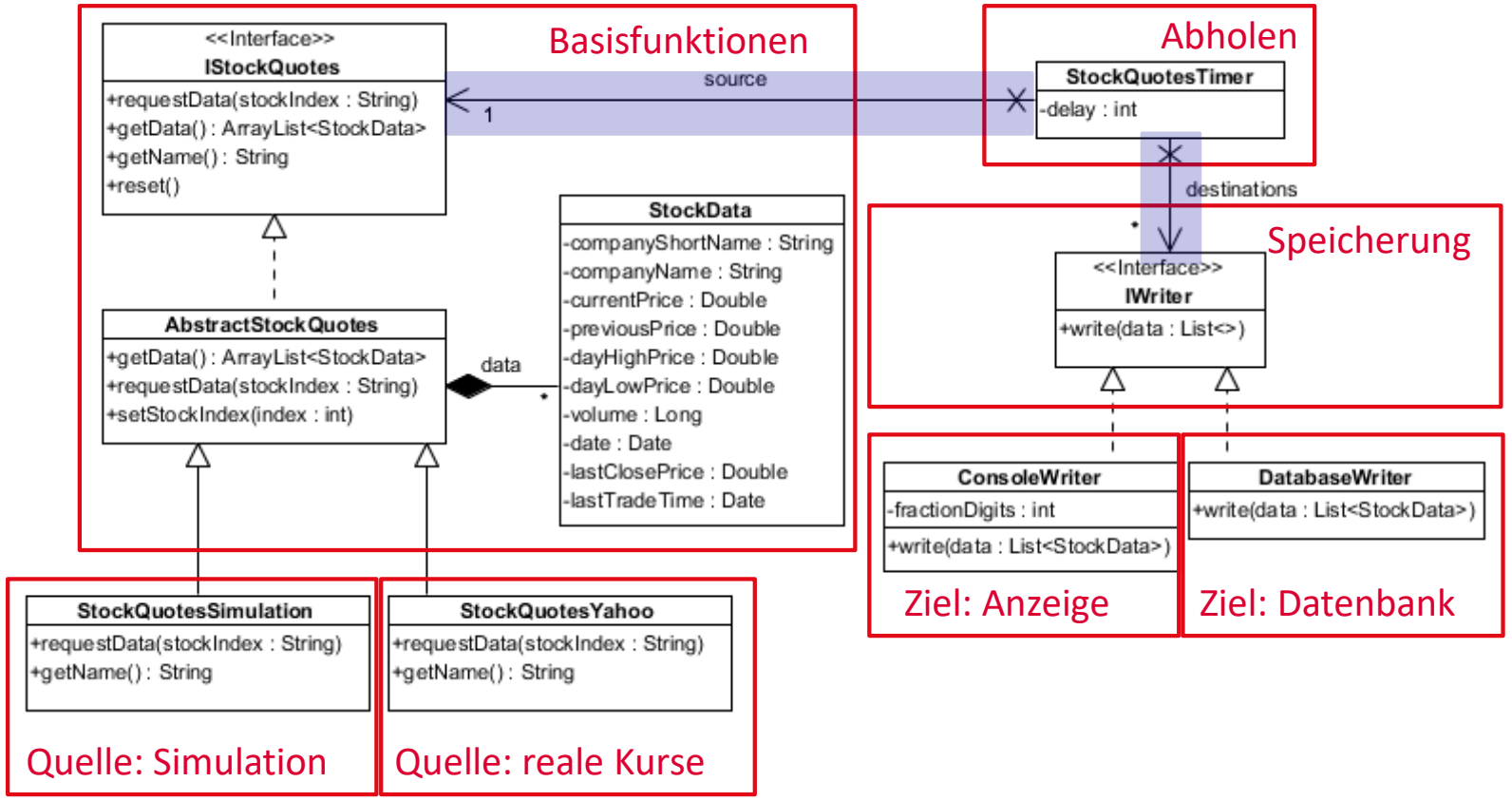
Verbindung von Modulen

- Beispiel: Betrieb mit realen Daten, die in die Datenbank geschrieben und ausgegeben werden



Verbindung von Modulen

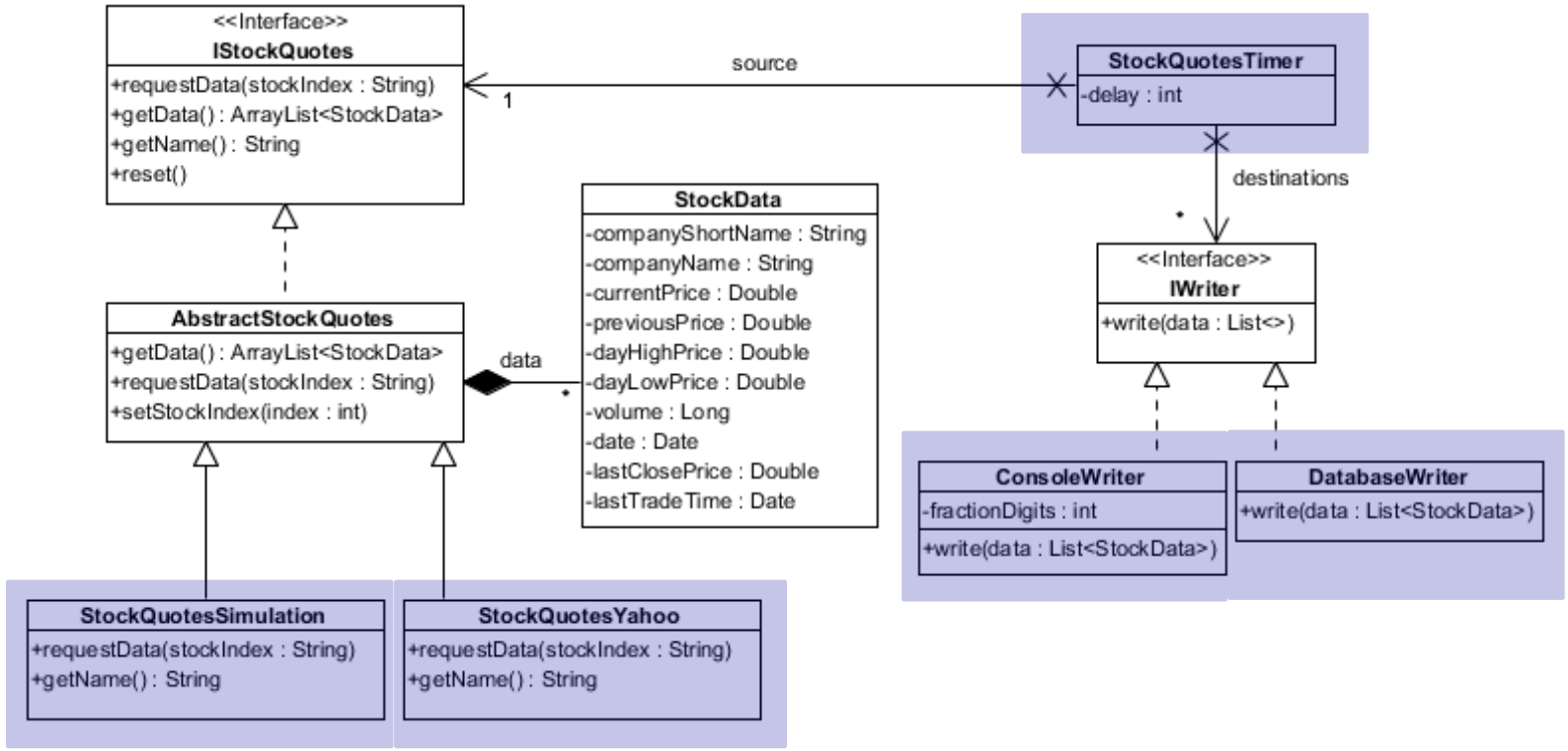
- Das Ziel ist es also, mindestens die veränderlichen Modulverknüpfungen separat zu konfigurieren:



Verbindung von Modulen

Angebot

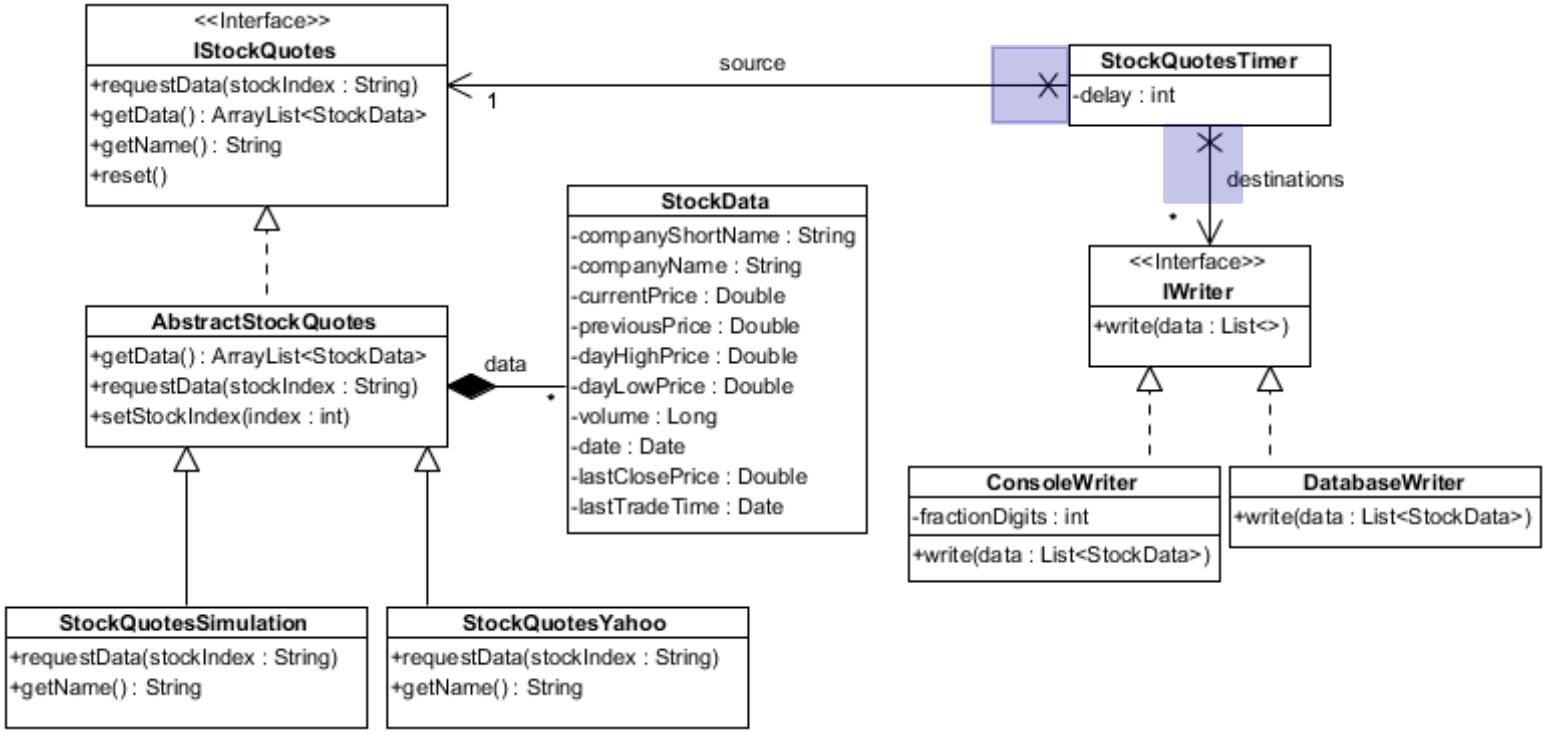
Angebot: Es muss erkennbar sein, welche Klassen als Dienste anderen zur Verfügung stehen.



Verbindung von Modulen

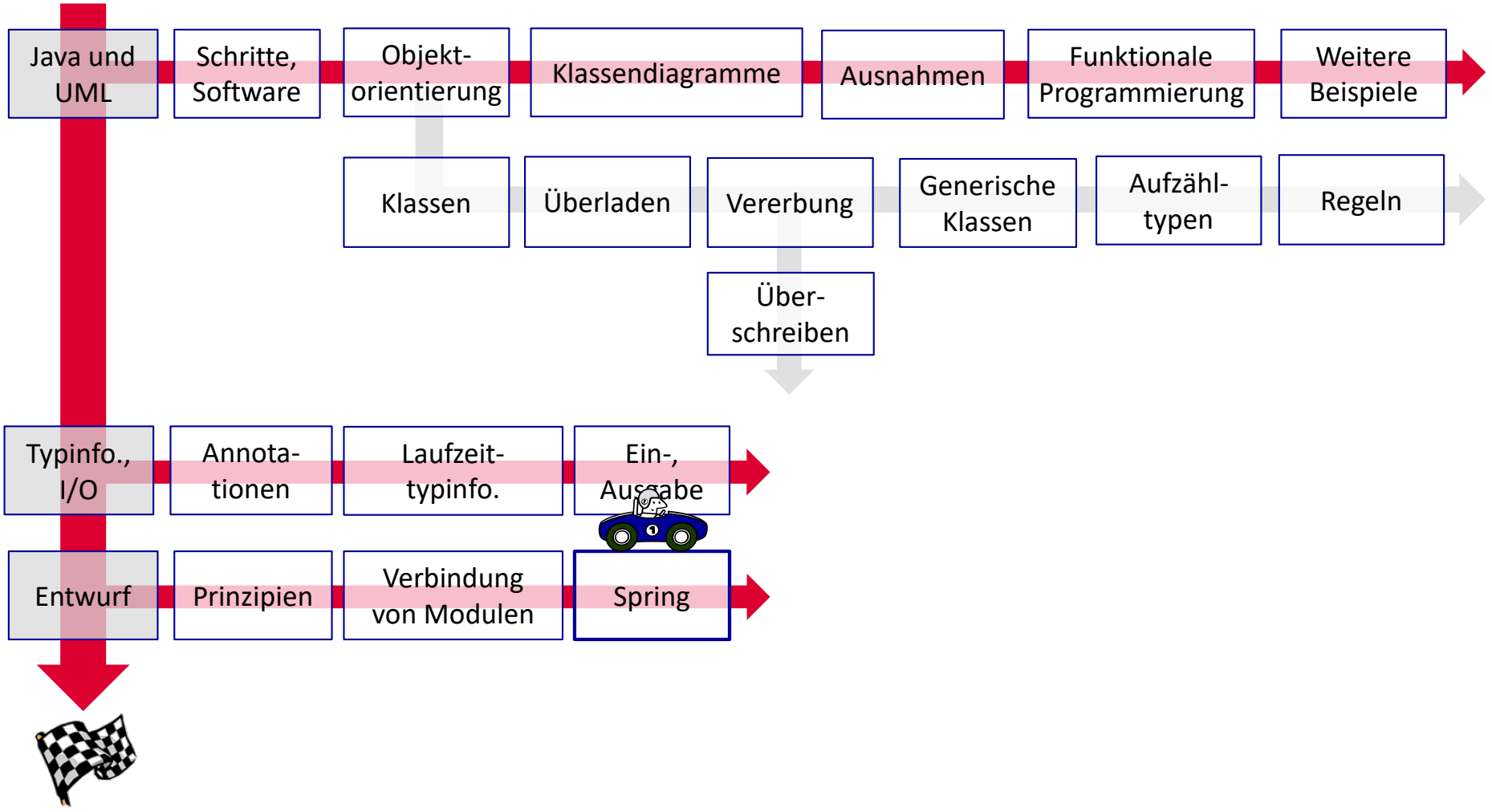
Nachfrage

Nachfrage: Die Klassen, die Dienste in Anspruch nehmen wollen, müssen ihren Bedarf kennzeichnen.



Modularisierung mit Spring

Übersicht

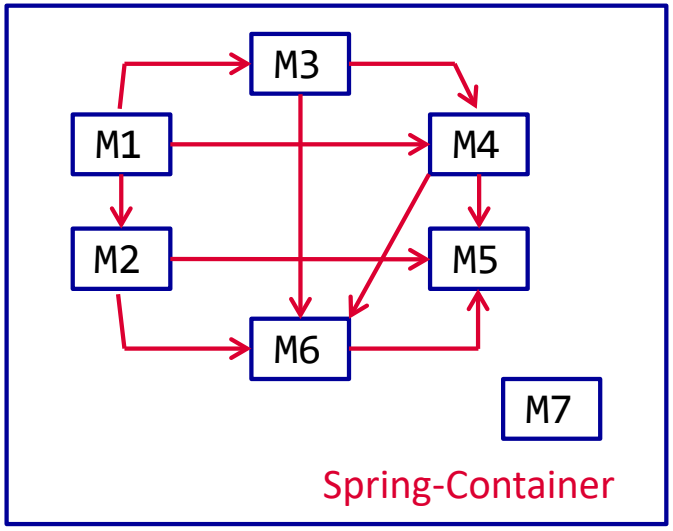


Modularisierung mit Spring

Motivation

- *Wie verbindet Spring einzelne Module?*
- *Wie können externe Abhängigkeiten oder Konfigurationen den Modulen übergeben werden?*

- Spring kennt nicht den Begriff des Moduls. Es
  - baut intern einen Abhängigkeitsgraphen zwischen Objekten auf
  - und initialisiert sie in der richtigen Reihenfolge.
  - Es können auch Konfigurationswerte z.B. aus Dateien eingetragen werden.
- Die Lebenszyklen der von Spring verknüpften Klassen verwaltet Spring: kein manuelles Erzeugen mit **new** irgendwo im Code!



Die Module **M1** bis **M7** werden von Spring erzeugt und verknüpft (rote Linien). Innerhalb der Module dürfen aber mit **new** eigene Objekte erzeugt werden.

Modularisierung mit Spring

Dependency Injection

- Die Abhängigkeiten werden „von außen“ (durch das Spring-Framework) injiziert → „Dependency Injection“ (auch kurz „DI“).
- Es gibt drei Möglichkeiten, Klassen und deren Abhängigkeiten zu beschreiben (können auch gemischt verwendet werden):
  - XML-Datei
  - Annotationen im Quelltext
  - Java-Code
- Die von Spring verwalteten Objekte besitzen auch einen Lebenszyklus, der hier nicht betrachtet wird.
- Spring Boot ist ein Projekt von Spring, um sehr schnell und einfach mit Spring-Anwendungen loslegen zu können: http://projects.spring.io/spring-boot/
- Es gibt viele Erweiterungen für Spring (werden hier nur ansatzweise verwendet).
- Sehr einfach kann eine Anwendung mit Spring Boot unter http://start.spring.io/ konfiguriert werden (ist zu kompliziert für diese Vorlesung).

Modularisierung mit Spring

Dependency Injection

- Das folgende Beispiel soll die Modularisierung anhand eines kleinen Servers zeigen.
- Notwendig ist der folgende Ausschnitt aus der Maven-Konfiguration (Versionsnummer eventuell aktualisieren):

```
<parent>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-parent</artifactId>
  <version>1.4.3.RELEASE</version>
</parent>
<dependencies>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
  </dependency>
</dependencies>
```

Modularisierung mit Spring

Dependency Injection: XML-Konfiguration

- Die Konfigurationsdatei beinhaltet:
  - Angebote: Namen und Klassen der zu erzeugenden Komponenten. Die Komponenten werden Beans genannt.
  - Gültigkeitsbereiche (spielt hier keine Rolle, wichtig z.B. für Server-Anwendungen)
  - Initialwerte für die Beans
  - Nachfragen: Verbindungen der Beans untereinander
- Die Klassen sind reine Java-Klassen (POJOs = Plain Old Java Objects).
- Die Konfigurationsdatei kann einen beliebigen Namen besitzen.

- Das vollständige Beispiel finden Sie im Projekt **`Spring StockQuotes with XML`**.

Modularisierung mit Spring

Dependency Injection: XML-Konfiguration

- Präambel der Konfigurationsdatei **`applicationContext.xml`** für das Beispiel:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xmlns:context="http://www.springframework.org/schema/context"
      xmlns:p="http://www.springframework.org/schema/p"
      xmlns:aop="http://www.springframework.org/schema/aop"
      xsi:schemaLocation="http://www.springframework.org/schema/beans
      http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
      http://www.springframework.org/schema/context
      http://www.springframework.org/schema/context/spring-context-3.0.xsd
      http://www.springframework.org/schema/aop
      http://www.springframework.org/schema/aop/spring-aop-3.0.xsd">
```

- Es sind weitere Namensräume vorhanden, die hier keine Rolle spielen.

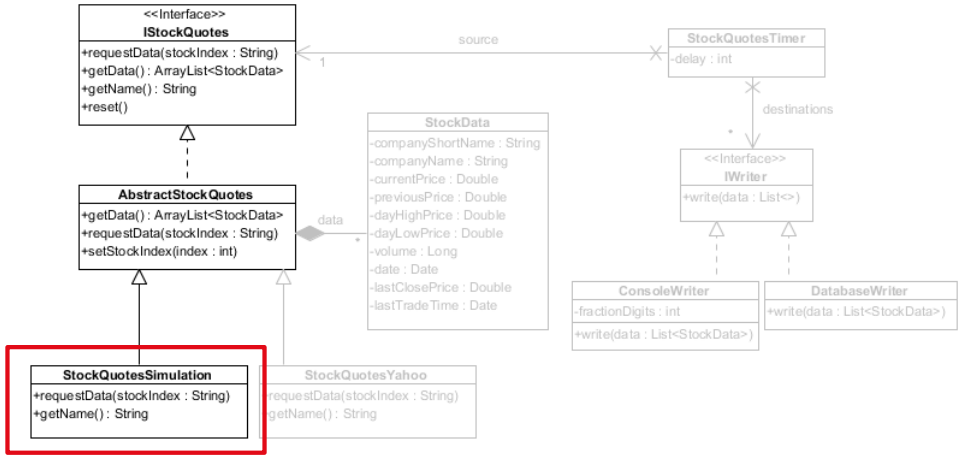
Modularisierung mit Spring

Dependency Injection: XML-Konfiguration

- Definition von Modulen (Beans):

```
<bean id="stockQuotesSimulation"
      class="de.hska.iwii.i2.simulation.StockQuotesSimulation"/>
```

  - **id** ist der logische Name, unter dem die Bean angesprochen wird (= Modulname). Normalerweise wird der Klassenname mit kleinem Anfangsbuchstaben verwendet.
  - **class** ist die Klasse, die diese Bean implementiert.
  - Beim Erzeugen wird der Standardkonstruktor verwendet.
  - Es wird nur ein einziges Bean-Objekt erzeugt.



Modularisierung mit Spring

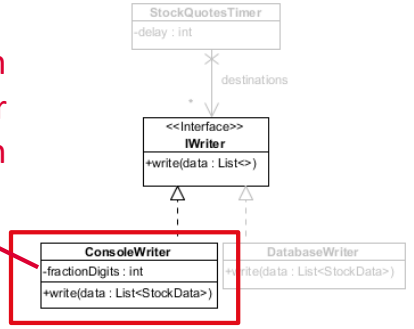
Dependency Injection: XML-Konfiguration

- Die Klasse zur Konsolenausgabe soll im Konstruktor die Anzahl Nachkommastellen, mit denen die Zahlen formatiert werden, übergeben bekommen. Quellcodefragment der Klasse:

```
public class ConsoleWriter implements IWriter {
  public ConsoleWriter(int fractionDigits) {
    // ...
  }

  @Override
  public void write(List<StockData> values) {
    // ...
  }
}
```

Attribut wird dem Konstruktor übergeben



- Die Konstruktorargumente können als einzelne **`constructor-arg`**-Tags in der Bean-Spezifikation angegeben werden:

```
<bean id="consoleWriter" class="de.hska.iwii.i2.output.ConsoleWriter">
  <constructor-arg value="2"/>
</bean>
```

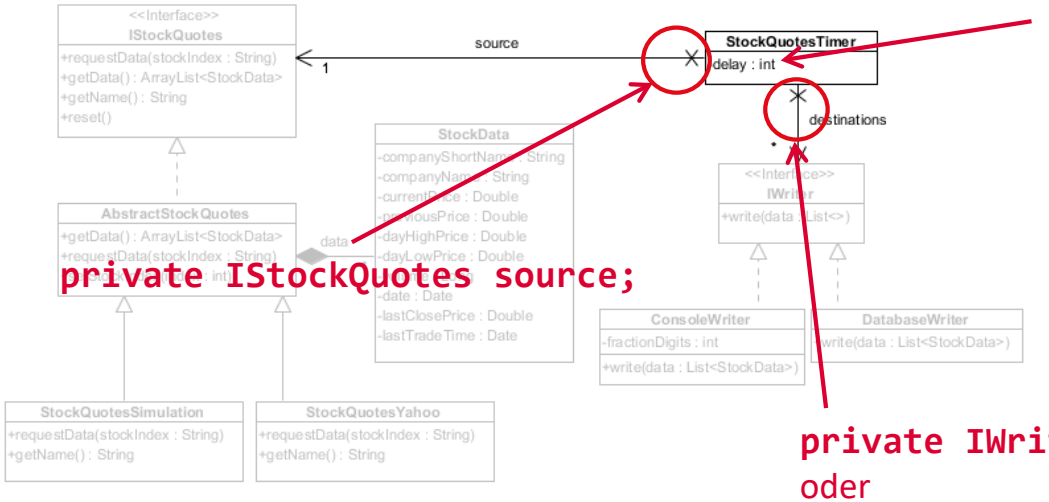
- Der Wert **`"2"`** wird automatisch in einen **`int`**-Wert konvertiert.

Modularisierung mit Spring

Dependency Injection: XML-Konfiguration

- Der Timer ist etwas komplizierter aufgebaut: Er bekommt
  - die Zeitabstände zwischen zwei Abholvorgängen,
  - die Referenz auf die Datenquelle
  - und die Referenzen auf die Datenziele

  per XML-Konfiguration übergeben.



Modularisierung mit Spring

Dependency Injection: XML-Konfiguration

- Quellcodefragment der Klasse:

```
public class StockQuotesTimer {
  // Zeitverzögerung zwischen zwei Abholvorgängen in msec
  private int delay;

  // Aktuell verwendete Datenquelle
  private IStockQuotes source; // eine Referenz

  // Alle Datenziele
  private IWriter[] destinations; // viele Referenzen

  public void setDelay(int delay) {
    this.delay = delay;
  }

  public void setSource(IStockQuotes source) {
    this.source = source;
  }
  // Oder List<IWriter> destinations
  public void setDestinations(IWriter[] destinations) {
    this.destinations = destinations;
  }
```

Modularisierung mit Spring

Dependency Injection: XML-Konfiguration

- Bean-Definition:

```
<bean id="stockQuotesTimer"
      class="de.hska.iwii.i2.stockquotestimer.StockQuotesTimer"
  p:delay="1000">
  <property name="destinations">
    <list>
      <ref bean="consoleWriter"/>
    </list>
  </property>
  <property name="source" ref="stockQuotesSimulation"/>
</bean>
```

- Das Objekt wird mit dem Standardkonstruktor erzeugt.
- **`p:delay`**: Die Eigenschaft („Property“) **`delay`** wird durch den Aufruf von **`setDelay`** eingetragen.
- **`p:delay`** ist eine Kurzform für **`<property name="delay" value="1000"/>`**.

- Die Referenz auf das Bean **`consoleWriter`** wird mit der Methode **`setDestinations`** übergeben.
  - **`<list>`**: Es wird eine Liste von Daten übergeben.
  - Die Liste hat hier nur einen Eintrag. **`ref`** verweist auf die einzige Bean, die übergeben werden soll.
- Die Referenz auf das Bean **`stockQuotesSimulation`** wird mit der Methode **`setSource`** eingetragen.

- Anmerkung zu **`<list>`**: Es können prinzipiell auch Hashtabellen übergeben werden, wenn eine Methode **`Map`**-Objekte als Parameter erwartet:

```
<map>
  <entry key="key1" value-ref="bean1"/>
  <entry key="key2" value-ref="bean2"/>
</map>
```

- Der Schlüssel darf auch eine referenzierte Bean sein: **`key-ref="bean1"`**
- Der Wert kann ein fest verdrahteter Wert sein: **`value="1000"`**
- Statt einer **`<map>`** kann auch ein **`<set>`** verwendet werden (Syntax wie **`<list>`**).

Modularisierung mit Spring

Dependency Injection: XML-Konfiguration

- Jetzt muss die Konfiguration noch gelesen werden, hier in **`main`**:

```
public class Starter {
  public static void main(String[] args) throws IOException {
    ApplicationContext ctx =
                  new ClassPathXmlApplicationContext("applicationContext.xml");
    StockQuotesTimer timer = ctx.getBean(StockQuotesTimer.class);
    timer.setStockQuotesIndex("DJI");
    timer.start();
  }
}
```

- **`ClassPathXmlApplicationContext`**: Aufgaben:
  - Lädt die übergebene XML-Datei aus dem Klassenpfad der Anwendung,
  - erzeugt die erforderlichen Objekte bei Bedarf und
  - stellt deren Beziehungen her.
- **`ctx.getBean(`*`Klasse`*`)`**: Erzeugt das Timer-Objekt. Dadurch werden die abhängigen Objekte auch erzeugt und verknüpft.
- Die Timer-Methoden sind selbst implementiert. Sie wählen einen Börsenindex aus und starten das zeitgesteuerte Abholen.

Modularisierung mit Spring

Dependency Injection: Annotationen

- Die Annotationen:
  - kennzeichnen die zu erzeugenden Komponenten und geben ihnen optional einen alternativen Namen.
  - besitzen Gültigkeitsbereiche (spielen auch hier keine Rolle)
  - bieten Initialwerte für die Beans
  - verbinden Beans untereinander
- Die Klassen sind keine reinen Java-Klassen (POJOs) mehr.
- Annotationen dürfen mit XML-Konfigurationsdateien gemischt eingesetzt werden.
- Es werden unterschiedliche Annotationen unterstützt:
  - Spring-eigene
  - JSR-330: In diesem Standard sind Annotationen definiert, die aber nicht den vollen Umfang der Spring-Annotationen abdecken. Vorteil: Unabhängigkeit von Spring, auch andere DI-Frameworks können verwendet werden.
- Das vollständige Beispiel finden Sie im Projekt **`Spring StockQuotes with Annotations`**.

Modularisierung mit Spring

Dependency Injection: Annotationen

- Definition von Modulen (Beans) durch die Annotation:

```
@Component
public class StockQuotesSimulation extends AbstractStockQuotes {
```

  - **`@Component`** markiert diese Klasse, damit Spring dessen Objekte als Anbieter per „Dependency Injection“ anderen Komponenten übergeben kann. Der Name der Komponente entspricht dem Klassennamen mit kleinem Anfangsbuchstaben (hier **`stockQuotesSimulation`**).
  - Ein abweichender Namen kann mit **`@Qualifier("name")`** angegeben werden.
  - Beim Erzeugen wird der Standardkonstruktor verwendet.
  - Es wird nur ein einziges Bean-Objekt je Gültigkeitsbereich erzeugt.

Modularisierung mit Spring

Dependency Injection: Annotationen

- Wie sieht es mit der Übergabe von externen Konfigurationsparametern an einen Konstruktor oder eine Methode aus?

```
@Component
public class ConsoleWriter implements IWriter {
  private int fractionDigits;
  public ConsoleWriter(int fractionDigits) { /* ... */ }
  // ...
}
```

Nicht mit Annotationen → XML-Konfiguration oder Expression Language verwenden.

- Attribute können aber direkt initialisiert werden:

```
@Component
public class ConsoleWriter implements IWriter {
  @Value("2")
  private int fractionDigits;
  public ConsoleWriter() { /* ... */ }
  // ...
}
```

- Ziemlich sinnlos → mit Expression Language aber doch sehr mächtig (kommt wirklich bald).

Modularisierung mit Spring

Dependency Injection: Annotationen

- Das Beispiel mit **`@Value`** klappt dummerweise nicht immer so wie erwartet:

```
@Component
public class ConsoleWriter implements IWriter {
  @Value("2")
  private int fractionDigits;

  public ConsoleWriter() {
    System.out.println(fractionDigits); // Ausgabe 0!!
  }
  // ...
}
```

- Was passiert hier?
  - Es wird erst das Objekt mit dem Standardkonstruktor erzeugt.
  - Dann werden per „Dependency Injection“ die Abhängigkeiten und auch die Werte eingetragen. **`fractionDigits`** hat also erst nach der Erzeugung den Wert **`2`**.
- Probleme:
  - Wann ist das Objekt denn nun fertig initialisiert?
  - Wie kann der Entwickler auf die abgeschlossene Initialisierung reagieren?

- Der Entwickler kann eine Methode mit **`@PostConstruct`** annotieren. Sie wird nach der Initialisierung aufgerufen → „ersetzt“ den Konstruktor.

```
@Component
public class ConsoleWriter implements IWriter {
  @Value("2")
  private int fractionDigits;

  @PostConstruct
  private void init() {
    System.out.println(fractionDigits); // Ausgabe 2
  }
  // ...
}
```

- Ablauf mit Spring:
  1. Das Objekt wird erzeugt, Konstruktor aufgerufen.
  2. Die Abhängigkeiten und Werte werden eingetragen.
  3. Die mit **`@PostConstruct`** annotierte Methode wird aufgerufen.

Modularisierung mit Spring

Dependency Injection: Annotationen

- Auflösen der Modulabhängigkeiten:
  - anhand der Klasse der Abhängigkeit
  - anhand des Namens der abhängigen Komponente
  - anhand von Konstruktorparametertypen (soll hier nicht gezeigt werden)

```
public class StockQuotesTimer {
  // Zeitverzögerung zwischen zwei Abholvorgängen in msec
  @Value("2000")
  private int delay;

  // Aktuell verwendete Datenquelle
  @Autowired
  private IStockQuotes source; // eine Referenz

  // Alle Datenziele
  @Autowired
  private IWriter[] destinations; // viele Referenzen
```

- Die Setter-Methoden sind nicht mehr erforderlich.
- Mit **`@Autowired`** werden hier alle Objekte übergeben, die die Schnittstellen **`IStockQuotes`** bzw. **`IWriter`** implementieren (Auflösung anhand der Klasse).

Modularisierung mit Spring

Dependency Injection: Annotationen

- Was passiert, wenn es mehrere Quellen (Simulation und Yahoo) gibt?
  - Ausnahme durch Spring (der Timer erwartet nur eine Quelle)
  - Es kann eine Quelle direkt durch Angabe ihres Namens ausgewählt werden (Auflösung anhand des Namens). Hier wird direkt die Simulation gewählt:

```
@Autowired
@Qualifier("stockQuotesSimulation")
private IStockQuotes source;
```

  - Nachteil: Die Abhängigkeit kann schlecht „von außen“ gesteuert werden.
- Was passiert, wenn es keine Quelle oder kein Ziel gibt?
  - Ausnahme
  - Wenn das aber vorkommen darf, dann kann die Abhängigkeit als optional markiert werden:

```
@Autowired(required=false)
private IStockQuotes source;
```

- Die automatische „Verdrahtung“ mit **`@Autowired`** gibt es übrigens auch in der XML-Konfiguration → siehe Spring-Dokumentation.

Modularisierung mit Spring

Dependency Injection: Annotationen

- Statt direkt Attribute zu „injizieren“, funktioniert das auch über Setter-Methoden und Konstruktoren:
  - Setter:

```
@Autowired
public void setCustomer(Customer cust) {
```

    Die Methode wird automatisch nach der Erzeugung aufgerufen.
  - Konstruktor:

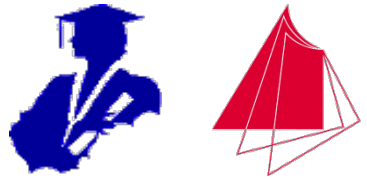
```
@Autowired
public Shop(Customer cust) {
```

- Vorteile dieser Ansätze: Es sind weitere Aktionen im Setter bzw. dem Konstruktor möglich.
- Hinweis zum Standard JSR-330:
  - Statt **`@Autowired`** wird **`@Inject`** verwendet.
  - Statt **`@Qualifier`** wird **`@Named`** verwendet.
  - Statt **`@Component`** wird **`@Named`** (mit oder ohne Namensangabe) verwendet.
- Die Annotationen sind ähnlich, aber nicht exakt gleich.

- Jetzt muss die Konfiguration noch gelesen werden, hier in **`main`**:

```
public class Starter {
  public static void main(String[] args) throws IOException {
    String package = Starter.class.getPackage().getName();
    ApplicationContext ctx =
                new AnnotationConfigApplicationContext(package);
    StockQuotesTimer timer = ctx.getBean(StockQuotesTimer.class);
    timer.setStockQuotesIndex("DJI");
    timer.start();
  }
}
```

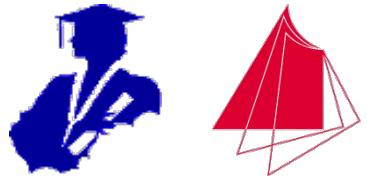
- **`AnnotationConfigApplicationContext`**: Aufgaben:
  - untersucht die Klassen aus dem übergebenen Paket sowie seinen Unterpaketen auf Annotationen
  - erzeugt die erforderlichen Objekte bei Bedarf
  - und stellt deren Beziehungen her.



- Nachteil der Annotation-Variante ist die fehlende Trennung der Anliegen:
  - Annotationen gehören zur Infrastruktur.
  - Die Klassen implementieren die Anwendungslogik und sollten als reine Java-Klassen (POJOs) keinen Bezug zur Infrastruktur aufweisen.
- Die Java-basierte Konfiguration umgeht das Problem, indem eine oder mehrere Konfigurations-Klassen die Beans erzeugen. Am Beispiel:

```
@Configuration
public class StockQuotesConfiguration {
  @Bean
  public StockQuotesTimer timer() {
    StockQuotesTimer timer = new StockQuotesTimer(stockQuotesSimulation());
    timer.addDestination(consoleWriter());
    return timer;
  }

  @Bean
  public IStockQuotes stockQuotesSimulation() {
    return new StockQuotesSimulation();
  }
```



```
  @Bean
  public IWriter consoleWriter() {
    return new ConsoleWriter(2);
  }
}
```

- Spring stellt auch bei mehrfachen Aufrufen der Bean-Methoden sicher, dass immer nur ein Objekt erzeugt wird. Wie soll das funktionieren?
  - Spring fügt zur Laufzeit Code in den Bytecode ein, der die Aufrufe abfängt, um zu kontrollieren, wie oft ein Objekt erzeugt wird → mit AOP möglich (Umleitung von Methodenaufrufen zur Laufzeit).
- Annotationen:
  - **`@Configuration`** markiert eine Klasse, die Beans erzeugen kann.
  - **`@Bean`** kennzeichnet die Methoden, die von Spring aufgerufen werden, um Beans zurückzugeben. Der Methodenname wird als Bean-Name verwendet.
- Alle anderen Klassen sind normale POJOs (siehe XML-Beispiel).
- Die Hauptklasse **`Starter`** zum Start unterscheidet sich nicht von der aus dem Annotations-Beispiel.

Modularisierung mit Spring

Dependency Injection: Vergleich der Ansätze

- Vergleich der der Konfigurationsansätze:
  - XML-Datei
    - + sehr flexibel und mächtig, Konfiguration außerhalb des Programms
    - - XML-Hölle (kein Bezug zum Quelltext erkennbar, viel Code, Änderungen im Quelltext müssen in der XML-Datei nachgezogen werden)
  - Annotationen:
    - + sehr kompakte Schreibweise, Verdrahtung ist direkt im Quelltext erkennbar
    - - Vermischung verschiedener Anliegen in einer Klasse
    - - Die Verwendung von Konfigurationsdateien muss trotzdem separat erfolgen.
  - Java-basiert:
    - + saubere Trennung der Anliegen, Konfiguration mit „normalen“ Java-Sprachmitteln
    - - Anhand des Quelltextes ist nicht mehr ersichtlich, welche Klassen irgendwo verwendet werden.
    - - Die Verwendung von Konfigurationsdateien muss trotzdem separat erfolgen.

Modularisierung mit Spring

Expression Language

- Expression Language:
  - Die Eigenschaften in der XML-Datei oder die **`@Value`**-Annotation können nicht nur einfache Werte aufnehmen. Sie erlauben recht komplexe Ausdrücke in einer „Expression Language“.
  - Verwendung:

    `"#{bean.property}"`: Der Ausdruck ist ein String, der immer mit **`#{`** und **`}`** eingefasst wird.
- Beispiele:
  - **`"#{bean.property}"`**: **`bean`** ist die ID (der Name) einer Bean, **`property`** der Name eines Attributs der Bean, das mit dem Getter **`getProperty`** gelesen wird.
  - **`"#{bean1.sum * bean2.counter}"`**: berechnet das Produkt der beiden Bean-Attribute
  - **`"#{bean.firstname + " " + bean.lastname}"`**: erzeugt einen String, bestehend aus Vor- und Nachnamen (getrennt durch ein Leerzeichen).
  - **`"#{bean.method()}"`**: Das Ergebnis ist der Wert, den die Methode **`method`** zurückgibt.

- `"#{(bean.property > 0 && !bean1.total > 0) ? 42 : 66}"`: bedingter Ausdruck. Achtung in einer XML-Datei so nicht möglich. Daher können auch Operatorennamen verwendet werden:

  `"#{(bean.property gt 0 and not bean1.total gt 0) ? 42 : 66}"`

- Der Operation `T()` erlaubt den Zugriff auf Typen (Klassen). Das ist für statische Attribute und Methoden erforderlich:
  - `"#{T(java.lang.Math).PI"`: liest das statische Attribut `PI` der Klasse `Math` aus.
  - `"#{T(java.lang.Math).random()"`: ruft die statische Methode `random()` der Klasse `Math` auf.
- Weitere komplexe Ausdrücke, auch in Zusammenhang mit Collection-Klassen sind möglich → siehe Spring-Dokumentation.

Modularisierung mit Spring

Einfacher REST-Server als Beispiel

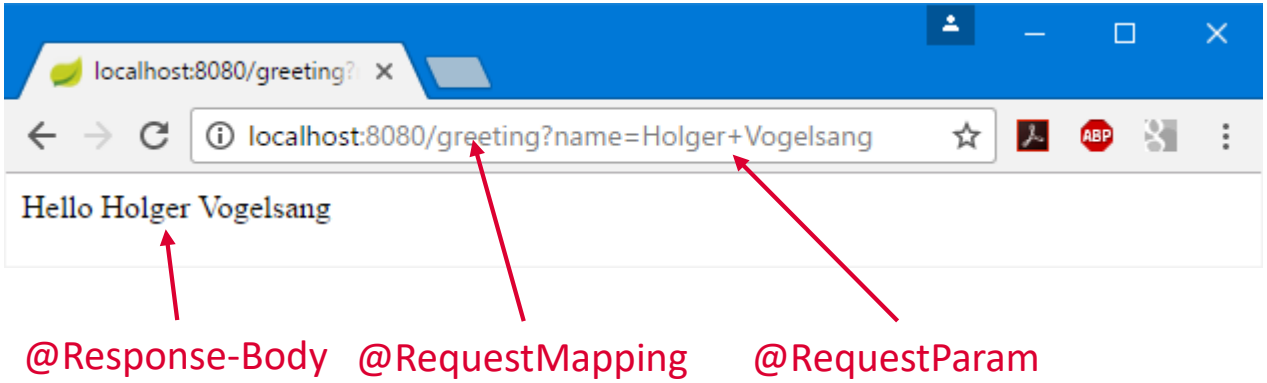
- Hauptklasse, sollte in der Paket-Hierarchie über allen anderen Klassen stehen:

```
@Controller
@SpringBootApplication
public class GreetingController {
  @RequestMapping("/greeting") // Unter der URL erreichbar
  @ResponseBody String getGreeting(@RequestParam String name) {
    return "Hello " + name;   // Rückgabe
  }
  // Start des Servers, sucht alle Controller.
  public static void main(String[] args) {
    SpringApplication.run(GreetingController.class, args);
  }
}
```

- Annotationen:
  - **`@Controller`**: Controller, der über REST-Aufrufe erreichbar ist.
  - **`@SpringBootApplication`**:
    - Kennzeichnet die Boot-Anwendung
    - Teilt Spring mit, weitere Controller zu suchen und Dependency Injection durchzuführen

- **`@RequestMapping`**: URL, unter der der Aufruf erreichbar sein soll.
- **`@RequestParam`**: Parameter, der dem Aufruf übergeben wird.
- **`@ResponeBody`**: Kennzeichnet den Rückgabetyp für den REST-Aufruf, wird automatisch in JSON umgewandelt.

- Weitere Controller können (natürlich ohne main) direkt zum Projekt hinzugefügt werden.
- Aufruf:



Modularisierung mit Spring

Börsenkurse als REST-Server-Beispiel

- Beispiel, siehe Projekt „Spring-StockQuotes-REST-Server“
- Einziger Controller, andere Klassen werden aus dem Projekt „ Spring StockQuotes with Annotations“ übernommen und deren Beans hier injiziert.

```
@Controller
public class StockQuotesController {
  @Autowired
  @Qualifier("stockQuotesSimulation")
  private IStockQuotes sourceSimulation;

  @RequestMapping("/stockquotes")
  @ResponseBody public ResponseEntity<ArrayList<StockData>> getData(String index) {
    try {
      sourceSimulation.requestData(index);
      return new ResponseEntity<>(sourceSimulation.getData(), HttpStatus.OK);
    } catch (IOException e) {
      return new ResponseEntity<>(HttpStatus.INTERNAL_SERVER_ERROR);
    }
  }
```

```
// Zweiter REST-Aufruf, um alle Indizes zu ermitteln.
  @RequestMapping("/indexes")
  @ResponseBody public String[] getStockQuoteIndicesOfCurrentService() {
    return IStockQuotes.ALL_STOCK_INDEXES;
  }
}
```

- Hauptklasse für den Start:

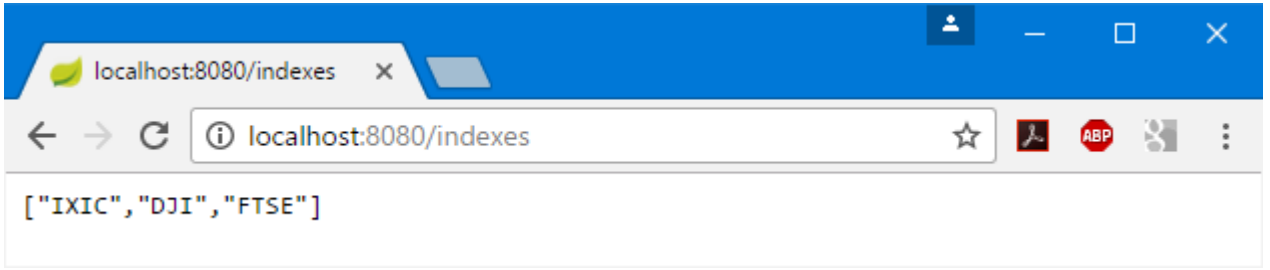
```
@SpringBootApplication
public class Starter {
  public static void main(String[] args ) {
    SpringApplication.run(Starter.class, args);
  }
}
```

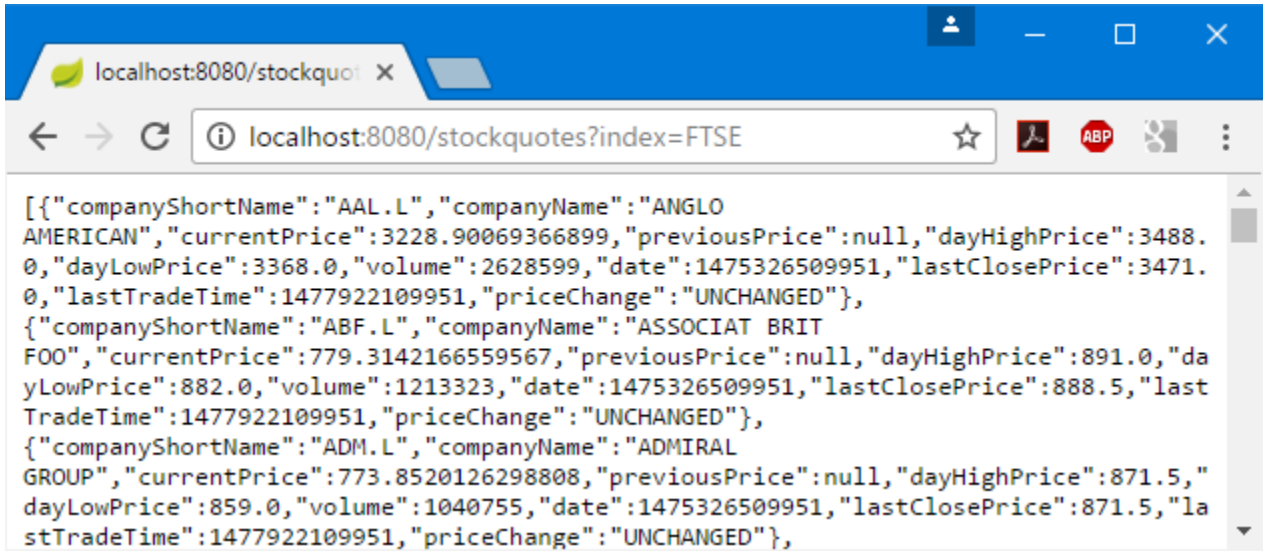
- Weitere Typen:
  - **`ResponseEntity`**: kapselt den Rückgabewert inkl. HTTP-Status-Code, der Standard-Wert ist OK (200).
  - **`HttpStatus`**: Aufzählwert mit den möglichen HTTP-Status-Codes.

Modularisierung mit Spring

Börsenkurse als REST-Server-Beispiel

- Aufrufe:





Ergebnisse im JSON-Format

Modularisierung mit Spring

Börsenkurse als REST-Server-Beispiel

- Die Angabe der Übergabewerte als Request-Parameter ist optisch unschön.
- Es geht auch direkt als Teil der URL:

```
@Controller
public class StockQuotesController {
  @Autowired
  @Qualifier("stockQuotesSimulation")
  private IStockQuotes sourceSimulation;

  @RequestMapping("/stockquotes/{index}")
  @ResponseBody public ResponseEntity<ArrayList<StockData>>
    getData(@PathVariable String index) {
    try {
      sourceSimulation.requestData(index);
      return new ResponseEntity<>(sourceSimulation.getData(), HttpStatus.OK);
    } catch (IOException e) {
    return new ResponseEntity<>(HttpStatus.INTERNAL_SERVER_ERROR);
    }
  }
```

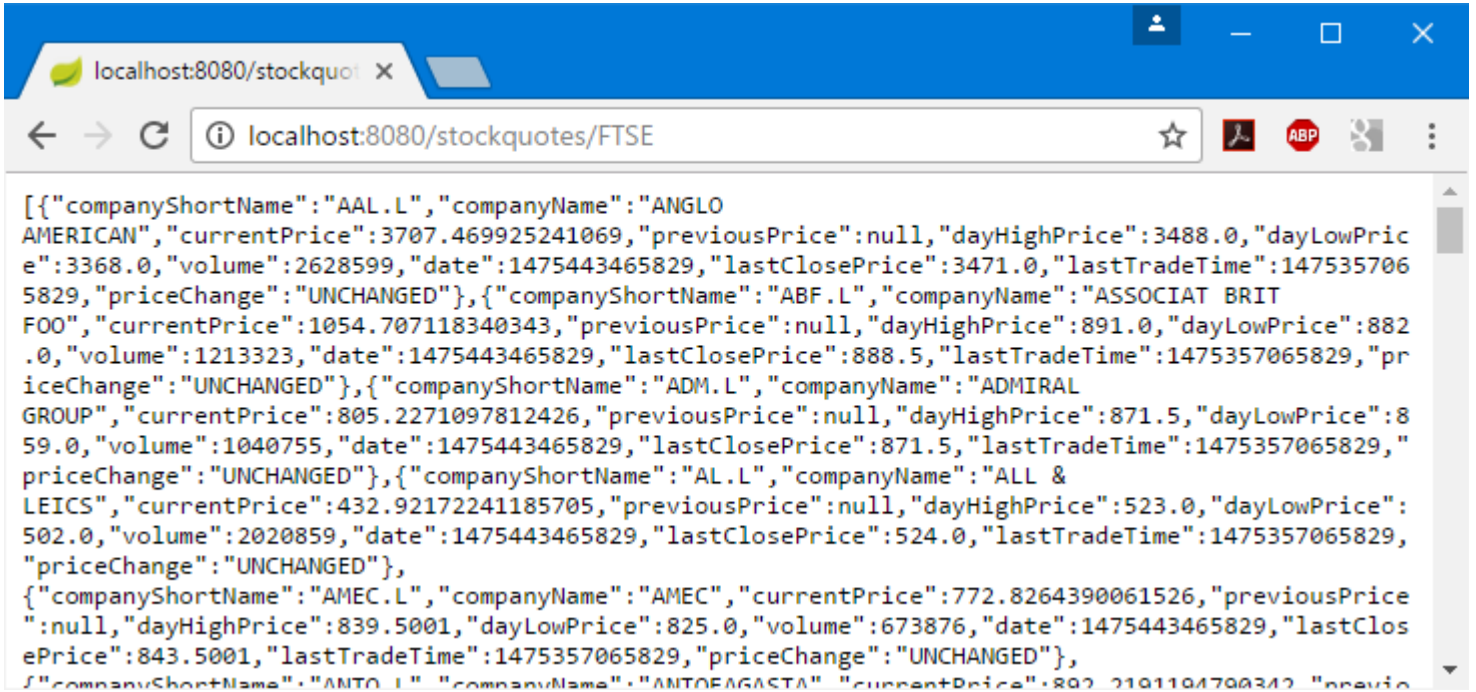
Variable in der URL (im Pfad)

- Erlaubter Aufruf z.B.: **http://localhost:8080/stockquotes/FTSE**

Modularisierung mit Spring

Börsenkurse als REST-Server-Beispiel

- Beispiel:



Modularisierung mit Spring

Fazit

- Zusammenfassung:
  - Spring wird sehr häufig für Server-Anwendungen verwendet.
  - Mit JSF (Java Server Faces) können in der XHTML-Seite z.B. direkt Spring-Beans mit der Expression Language angesprochen werden.
  - Spring bietet viele Module, die für eigene Anwendungen nur noch konfiguriert werden müssen (Authentifizierung, Autorisierung, Datenbankanbindung, …).
  - Bei REST-Aufrufen gibt es noch viel mehr Möglichkeiten
    - Vergabe von Basis-URLS je Controller
    - weitere Konfigurationen
    - Rückgabe anderer MIME-Tyen (PDF, JPEG, ...)

Ende