Einführung in die Programmierung: Tutorium

Dr. Peer Kröger,
Andreas Züfle, Johannes Niedermayer

Ludwig-Maximilians-Universität München,
Institut für Informatik,
LFE Datenbanksysteme

Wintersemester 2012/2013

Abschnitt 1: Strukturierung von Java-Programmen: Packages

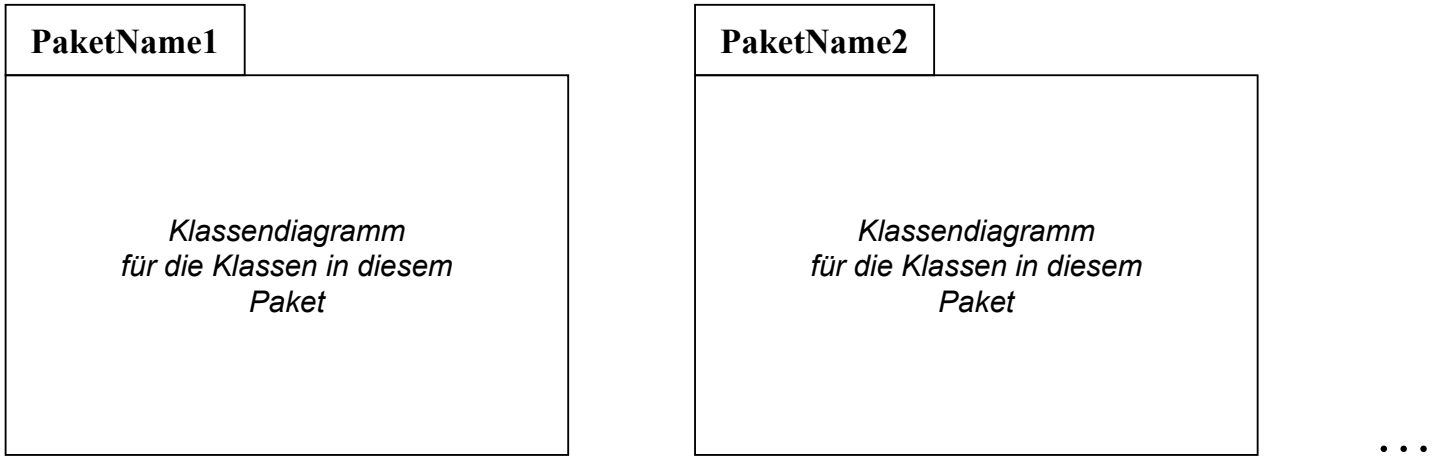
Überblick

Pakete in der Software-Entwicklung

- Bei großen Programmen entstehen viele Klassen und Schnittstellen.
- Um einen Überblick über diese Menge zu bewahren, wird ein Strukturierungskonzept benötigt, das von den Details abstrahiert und die übergeordnete Struktur verdeutlicht.
- Ein solches Strukturierungskonzept stellen die *Pakete* (*packages*) dar. Packages erlauben es, Komponenten zu größeren Einheiten zusammenzufassen.
- Die meisten Programmiersprachen bieten dieses Strukturierungskonzept (teilweise unter anderen Namen) an.

Packages in UML

- Packages gruppieren Klassen, die einen gemeinsamen Aufgabenbereich haben.
- In einem UML Klassendiagramm kann dies folgendermaßen notiert werden:



- Bemerkung: Es kann Beziehungen zwischen Klassen unterschiedlicher Pakete geben.
- Welche grundlegenden oo Modellierungsaspekte werden durch das Konzept der Pakete realisiert?

Packages in Java

- In Java können Klassen zu Packages zusammengefasst werden.
- Packages dienen in Java dazu,
  - große Gruppen von Klassen, die zu einem gemeinsamen Aufgabenbereich gehören, zu bündeln,
  - potentielle Namenskonflikte zu vermeiden,
  - Zugriffe und Sichtbarkeit zu definieren und kontrollieren,
  - eine Hierarchie von verfügbaren Komponenten aufzustellen.

Packages in Java

- Jede Klasse in Java ist Bestandteil von genau einem Package.
- Ist eine Klasse nicht explizit einem Package zugeordnet, dann gehört es implizit zu einem *Default*-Package.
- Packages sind hierarchisch gegliedert, können also Unterpackages enthalten, die selbst wieder Unterpackages enthalten, usw.
- Die Package-Hierarchie wird durch Punktnotation ausgedrückt:
  `package.unterpackage1.unterpackage2. ... .Klasse`

Import von Packages

- Der vollständige Name einer Klasse besteht aus dem Klassen-Namen *und* dem Package-Namen: `packagename.KlassenName`
- Package-Namen bestehen nach Konvention immer aus Kleinbuchstaben.
- Um eine Klasse verwenden zu können, muss angegeben werden, in welchem Package sie sich befindet. Dies kann auf zwei Arten geschehen:
  1. Die Klasse wird an der entsprechenden Stelle im Programmtext über den vollen Namen angesprochen:
     ```
     java.util.Random einZufall = new java.util.Random();
     ```
  2. Am Anfang des Programms werden die gewünschten Klassen mit Hilfe einer **`import`**-Anweisung eingebunden:
     ```
     import java.util.Random;
     ...
     Random einZufall = new Random();
     ```
- Achtung: werden zwei Klassen gleichen Namens aus unterschiedlichen Packages importiert, müssen die Klassen trotz **`import`**-Anweisung mit vollem Namen aufgerufen werden!

Import von Packages

- Klassen des Default-Packages können ohne explizite **`import`**-Anweisung bzw. ohne vollen Namen verwendet werden.
- Wird in der **`import`**-Anweisung eine Klasse angegeben, wird genau diese Klasse importiert. Alle anderen Klassen des entsprechenden Packages bleiben unsichtbar.
- Will man alle Klassen eines Packages auf einmal importieren, kann man dies mit der folgenden **`import`**-Anweisung:

  ```
  import packagename.*;
  ```
- Achtung: es werden dabei wirklich nur die Klassen aus dem Package `packagename` eingebunden und *nicht* etwa auch die Klassen aus Unter-Packages von `packagename`.

Packages aus der Java-Klassenbibliothek

- Die Java-Klassenbibliothek bietet bereits eine Vielzahl von Klassen an, die alle in Packages gegliedert sind.
- Beispiele für vordefinierte Packages:

`java.io`	Ein- und Ausgabe
`java.util`	nützliche Sprach-Werkzeuge
`java.awt`	Abstract Window Toolkit
`java.lang`	Elementare Sprachunterstützung

  usw.
- Die Klassen im Package `java.lang` sind so elementar (z.B. enthält `java.lang` die Klasse `Object`, die implizite Vaterklasse aller Java-Klassen), dass sie von jeder Klasse automatisch importiert werden. Ein expliziter Import mit **`import`** `java.lang.*;` ist also *nicht* erforderlich.

Deklaration eigener Packages

- Ein eigenes Package `mypackage` wird angelegt, indem man vor eine Klassendeklaration und vor den **`import`**-Anweisungen die Anweisung **`package`** `mypackage;` platziert.
- Es können beliebig viele Klassen (jeweils aber mit unterschiedlichen Namen) mit der Anweisung **`package`** `mypackage;` im selben Package gruppiert werden.
- Um Namenskollisionen bei der Verwendung von Klassenbibliotheken unterschiedlicher Hersteller zu vermeiden, ist es Konvention, für Packages die URL-Domain der Hersteller in umgekehrter Reihenfolge zu verwenden, z.B.

  `com.sun. ...` für die Firma Sun,

  `de.lmu.ifi.dbs. ...` für den DBS-Lehrstuhl an der LMU.

Deklaration eigener Packages

- Wie bereits erwähnt, muss die Deklaration einer Klasse `X` in eine Datei `X.java` geschrieben werden.
- Darüberhinaus müssen alle Klassendeklarationen (also die entsprechenden `.java`-Dateien) eines Packages `p` in einem Verzeichnis `p` liegen.
- Beispiel:
  - Die Datei `Klasse1.java` mit der Deklaration der Klasse `package1.Klasse1` liegt im Verzeichnis `package1`.
  - Die Datei `Klasse2.java` mit der Deklaration der Klasse `package1.unterpackage1.Klasse2` liegt im Verzeichnis `package1/unterpackage1`.

Überblick

Zugriffsrechte und Sichtbarkeit

- Wir hatten bereits verschiedene Schlüsselwörter zur Spezifikation der Sichtbarkeit von Klassen und Klassen-Elementen (Attributen/Methoden) kennengelernt und teilweise erweitert.
- Erst jetzt mit dem Konzept der Packages können wir allerdings alle Möglichkeiten kennenlernen.
- Im folgenden also nocheinmal die (nun endgültige) Möglichkeit, die Sichtbarkeit von Klassen und Elementen einer Klasse zu spezifizieren.

Zugriffsrechte und Sichtbarkeit

- Klassen und Elemente mit der Sichtbarkeit **`public`** sind von allen anderen Klassen (insbesondere auch Klassen anderer Packages) sichtbar und zugreifbar.
- Klassen und Elemente mit der Sichtbarkeit **`private`** sind nur innerhalb der eigenen Klasse (also auch *nicht* innerhalb möglicher Unterklassen oder Klassen des selben Packages) sichtbar und zugreifbar.
- Klassen und Elemente mit der Sichtbarkeit **`protected`** sind innerhalb des gesamten Packages und aller Unterklassen (auch außerhalb des Packages) sichtbar und zugreifbar.
- Klassen und Elemente, deren Sichtbarkeit *nicht* durch ein entsprechendes Schlüsselwort spezifiziert ist, erhalten per Default die sogenannte *package scoped* (*friendly*) Sichtbarkeit: diese Elemente sind nur für Klassen innerhalb des selben Packages sichtbar und zugreifbar.

Zugriffsrechte und Sichtbarkeit

Spezifikation	sichtbar in			
	allen Klassen	allen Unterklassen unabh. vom Package	Klassen im selben Package	selber Klasse
`public`	√	√	√	√
`protected`		√	√	√
			√	√
`private`				√

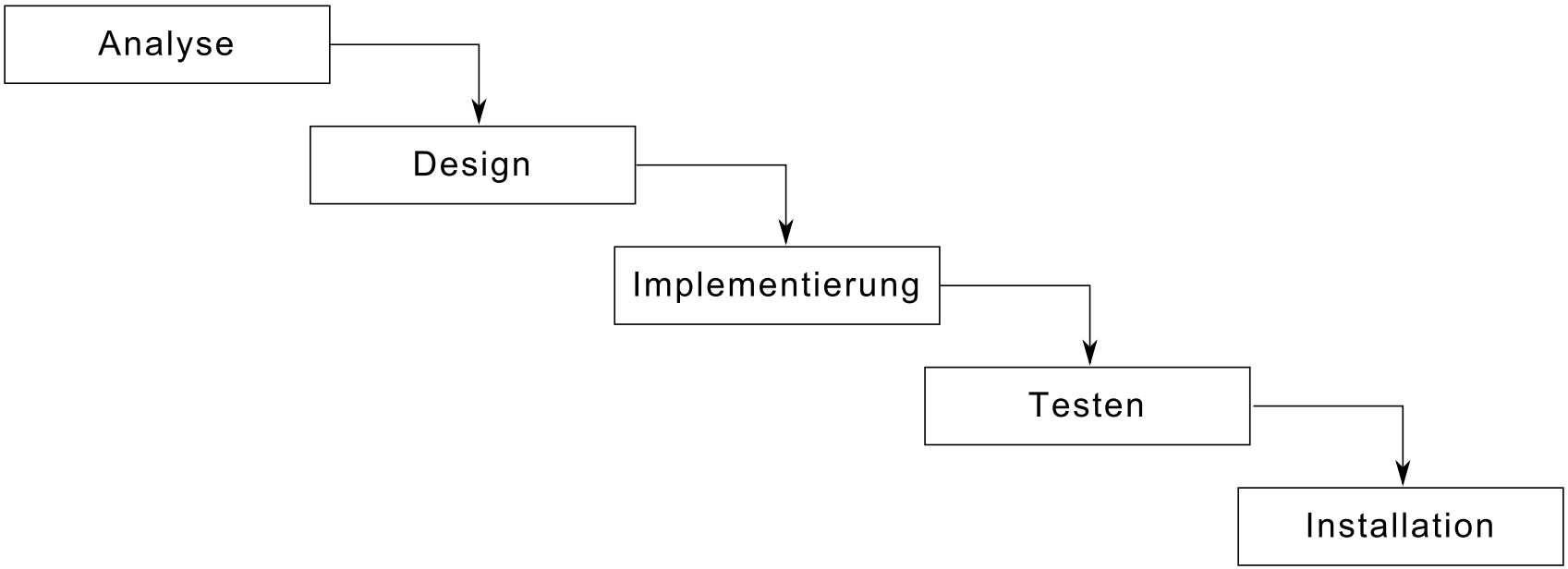
Abschnitt 2: Objektorientiertes Design

2. Objektorientiertes Design

Software-Entwicklung

- Zur Software-Entwicklung existiert eine Vielfalt von Vorgehensweisen und Modellen.
- Hier betrachten wir nur das sogenannte *Wasserfallmodell*, das den Prozess der Software-Entwicklung in fünf Phasen aufgliedert.

Das Wasserfallmodell



- Klare Trennung der verschiedenen Phasen.
- Schwierigkeiten in einer Phase verzögern das Gesamtprojekt.
- Streng sequentielles Vorgehen ist in der Praxis kaum möglich.

Wasserfallmodell: Anforderungsanalyse (requirements analysis)

- Analyse des Problembereichs
- Festlegung der (funktionalen und nicht funktionalen) Anforderungen an das Programm:
  Was soll das Programm/System leisten?
- Festlegung von organisatorischen Richtlinien (Aufwandsabschätzung, Terminplanung...) und Rahmenbedingungen (z.B. vorhandene Soft- und Hardware)
- Skizze der Systemarchitektur

Ergebnis der Anforderungsanalyse ist ein Anforderungskatalog (Anforderungsspezifikation).

Wasserfallmodell: Entwurf (design)

- Beschreibt die Art und Weise, in der die gestellten Aufgaben gelöst werden sollen:
  Wie lösen wir das Problem?
- Festlegung der Systemarchitektur.
- Entwurf der einzelnen Systemkomponenten (Wahl von Datenrepräsentationen und Algorithmen).
- *Objektorientierter* Entwurf: Festlegung der Klassen und Methoden.

Ergebnis der Entwurfsphase ist eine Entwurfsbeschreibung (z.B. mittels eines UML-Diagramms).

Wasserfallmodell: Entwurf (design)

Identifizierung von Klassen, Methoden und Assoziationen:

- Kandidaten für Klassen sind:
  - Personen bzw. Rollen (z.B. Student, Angestellter, Kunde, ...)
  - Organisationen (z.B. Firma, Abteilung, Uni, ...)
  - Gegenstände (z.B. Artikel, Flugzeug, Immobilien, ...)
  - begriffliche Konzepte (z.B. Bestellung, Vertrag, Vorlesung, ...)
- Möglichkeit: Durchsuchen des Anforderungskatalogs nach Substantiven, die Mengen bezeichnen
- Heuristik zum Auffinden von Methoden: Suche nach Verben
- Kandidaten für Assoziationen sind physische oder logische Verbindungen mit einer bestimmten Dauer, wie
  - konzeptionelle Verbindungen (arbeitet für, ist Kunde von, ...)
  - Besitz (hat, ist Teil von, ...)
  - (häufige) Zusammenarbeit von Objekten zur Lösung einer Aufgabe

Wasserfallmodell: Entwurf (design)

Beispiel: Identifizierung von Klassen, Methoden und Assoziationen

- Geg: Programm, welches Rechnungen für Kunden ausdruckt. Die Rechnungen sollen die Einzelposten sowie jeweils den Gesamtpreis enthalten.
- Klassen: Rechnung, Kunden, Einzelposten
- Methoden: ausdrucken, Gesamtpreis berechnen
- Assoziationen: Rechnungen *enthalten* Einzelposten, Rechnungen *für* Kunden

Wasserfallmodell: Entwurf (design)

Entwurf-Optimierung:

- Je weniger Beziehungen zwischen Klassen, desto unabhängiger kann man sie implementieren (hilft bei der Verteilung der Arbeit).
- Mögliche Schritte zur Optimierung:
  1. Reduziere Beziehungen
  2. Fasse gemeinsame Aspekte von Klassen zu Oberklassen zusammen
  3. Finde isolierte Inseln im UML-Diagramm
     $\rightarrow$ Arbeit organisieren, aufteilen

Wasserfallmodell: Implementierung, Test und Installation

- Implementierung: Codierung des Entwurfs in einer Programmiersprache, ggf. unter Wiederverwendung vorhandener Komponenten.
- Test:
  - Test der einzelnen Komponenten
  - Schrittweises Zusammenfügen einzelner Komponenten mit jeweiligem Integrationstest
  - Systemtest
  - Abnahmetest (mit “echten” Daten)
- Installation (und Wartung):
  - Installation des Systems
  - Fehlerbeseitigung nach Inbetriebnahme
  - Änderung und Erweiterung des Systems

Abschnitt 3: Ein Grundprinzip der Software-Architektur

3. Ein Grundprinzip der Software-Architektur

Modelle und Anwendungen

- Unsere Programme waren bisher oft eine Mischung aus Modellen und Anwendungen.
- Ein Modell stellt einen Ausschnitt der Welt in vereinfachter, schematisierter Form dar (z.B. die Klasse `Konto`).
- Eine Anwendung verwendet ein Modell um z.B. etwas zu berechnen, oft abhängig von Eingaben des Benutzers (z.B. eine Geschäftsabwicklung, bei der Konten angelegt und ihr Inhalt verändert wird).

Modelle und Anwendungen

- Als Anwendungen haben wir bisher `main`-Methoden kennengelernt.
- Modelle wurden bei uns bisher auf der Konsole dargestellt (`System.out.print`-Ausgaben).
- Interaktion war durch Eingabe auf die Konsole möglich (z.B. durch Start-Parameter).
- Die Konsole stellt damit eine bedeutende Schnittstelle zum Benutzer dar (Command-Line-Interface, CLI).
- Jede Klasse, die eine `main`-Methode zu Verfügung stellt, ist damit eine Anwendung (Applikation).
- Ein einziges Modell kann in sehr unterschiedlichen Anwendungen verwendet werden.

Verschiedene Arten von Anwendungen

- Eine Anwendung kann auf verschiedene Arten mit einem Benutzer interagieren.
- Neben CLI ist die bedeutenste Schnittstelle die GUI (Graphical User Interface).
- GUIs können eigenständige Programme sein oder auch auf Interaktionsmöglichkeiten über Web-Oberflächen – z.B. als Applets oder als Java-Server-Pages (JSP) – basieren.
- Wir werden im Softwareentwicklungspraktikum GUIs als sehr zentrale und Applets als java-typische, sehr schlanke Anwendungen kennenlernen.

Trennung von Modell und Benutzer-Interaktion

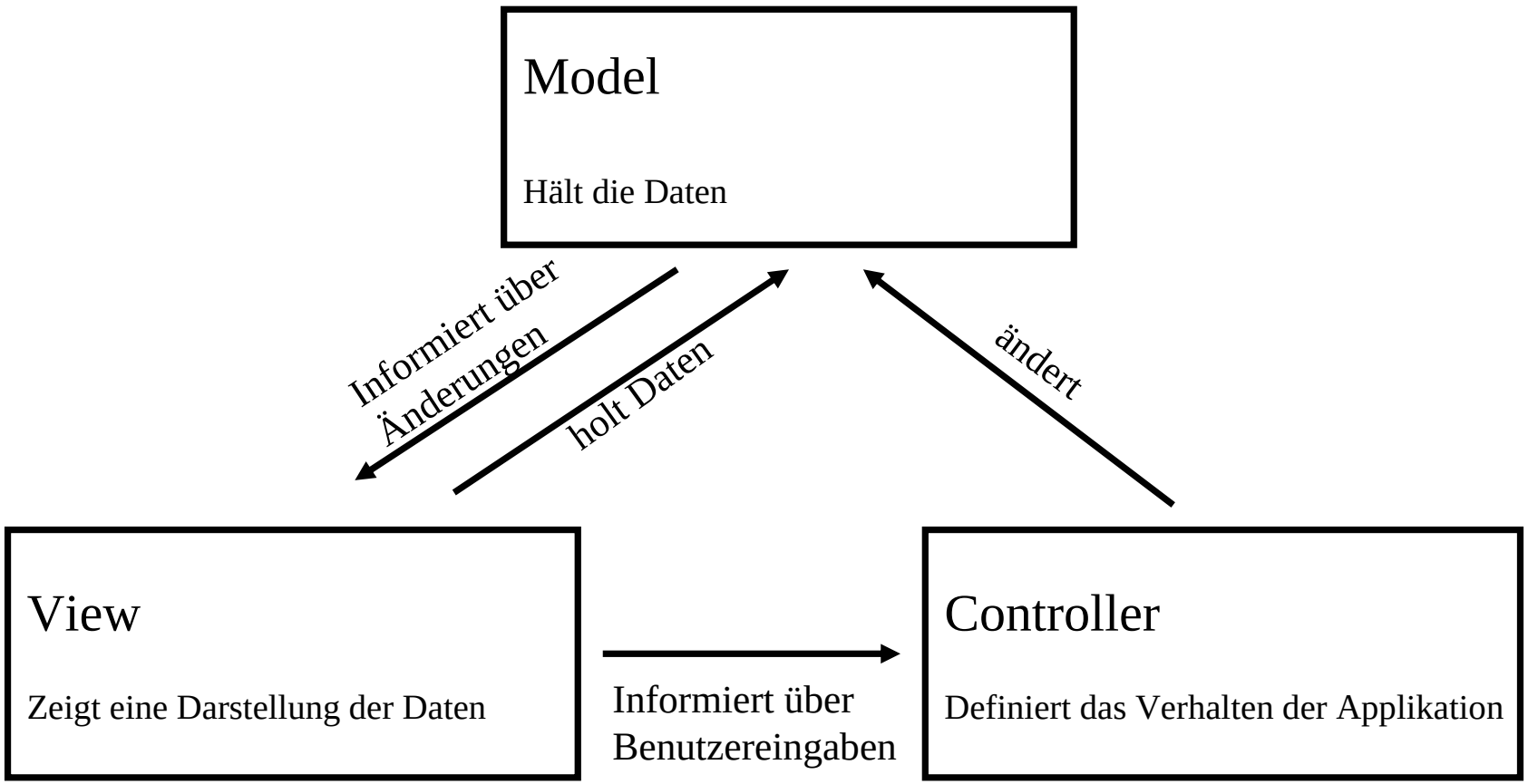
- Es ist eine wichtige Design-Hilfe für gute Programme, die Modellierung eines Ausschnitts der Wirklichkeit (das *Modell*) von den Möglichkeiten der Benutzer-Interaktion getrennt zu halten.
- In der Software-Entwicklung hat man für diese grundlegende Trennung das *Model-View-Controller* (*MVC*) Konzept geprägt.

Model-View-Controller

- Drei getrennte Programm-Komponenten sind für
  - das Modell (*Model*),
  - die Darstellung des Modells (*View*) und
  - die Beeinflussung des Modells (*Controller*)

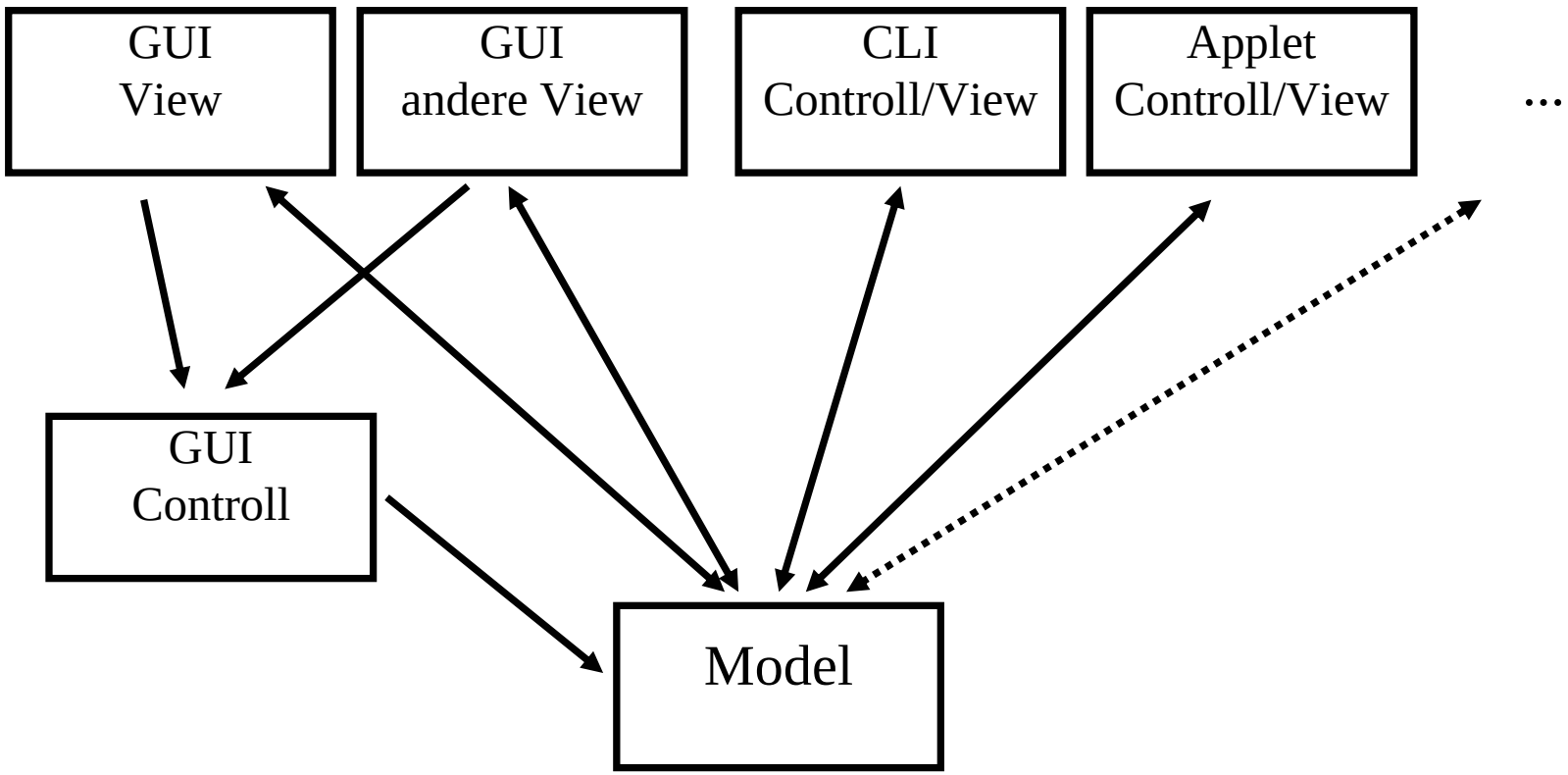
  zuständig.
- *View* und *Controller* hängen meist enger zusammen, da beide für eine bestimmte Art der Applikation (CLI, GUI, Applet, JSP, ...) bestimmt sind.
- Aber auch innerhalb des Anwendungsszenarios GUI ist es sinnvoll, View und Controller getrennt zu halten. Das Design der Darstellung ("Look-And-Feel") kann dadurch leicht ausgetauscht werden, ohne die Kontroll-Ebene zu beeinflussen.

Model-View-Controller



Model-View-Controller

Das Modell steht dem View-Controller-Paar eher unabhängig gegenüber und könnte auch von einem ganz anderen View-Controller-Paar verwendet werden.



Applikationen

- In unseren bisherigen Programmen haben wir implizit eine CLI-View-Controll verwendet.
- Um das MVC-Prinzip bei einfachen Aufgaben mit CLI einzuhalten, achtet man z.B. darauf, dass die Programmlogik (Methoden, die etwas berechnen) getrennt ist von der View (Methoden, die etwas ausgeben). Auf diese Weise könnte die Programmlogik auch von anderen View-Controller-Paaren verwendet werden.