Kapitel 15: Systemarchitektur und Packages

Prof. Dr. David Sabel

Lehr- und Forschungseinheit für Theoretische Informatik

Institut für Informatik, LMU München

WS 2018/19

Stand der Folien: 6. Februar 2019

Die Inhalte dieser Folien basieren – mit freundlicher Genehmigung – tlw. auf Folien von Prof. Dr. Rolf Hennicker aus dem WS 2017/18 und auf Folien von PD Dr. Ulrich Schöpp aus dem WS 2010/11

Ziele

- Grundprinzipien der Systemarchitektur verstehen
- Schichtenarchitekturen kennenlernen
- Modelle und Programme mit Paketen strukturieren
- Beispiel: Architektur für eine einfache Bankanwendung

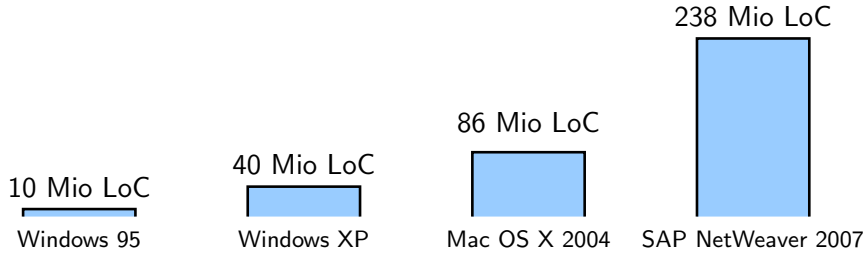
Grundprinzipien der Systemarchitektur

Große Softwaresysteme

- bestehen aus mehr als **50.000 Lines of Code (LoC)** und
- benötigen i.A. mehr als 10 Personenjahre für die Entwicklung

Sehr große Softwaresysteme

- mehr als **1 Million** LoC
- Beispiele:



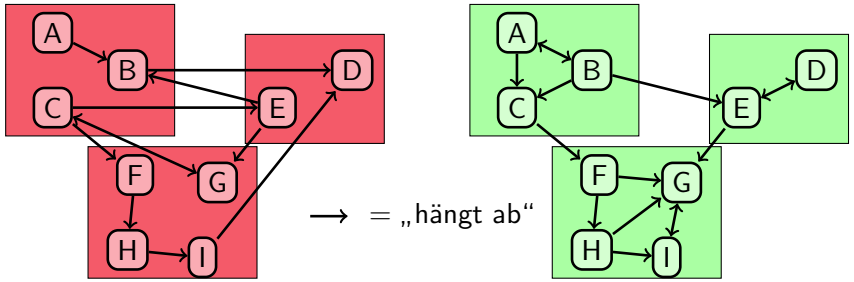
SAP NetWeaver: Plattform für Geschäftsanwendungen mit zahlreichen Komponenten

Grundprinzipien der Systemarchitektur (2)

- Große Softwaresysteme bestehen aus vielen Teilsystemen, Modulen, Komponenten
- Die **Systemarchitektur** beschreibt die **Struktur** eines Softwaresystems durch Angabe seiner
  - **Teile** und deren
  - **Verbindungen** (häufig über Schnittstellen)

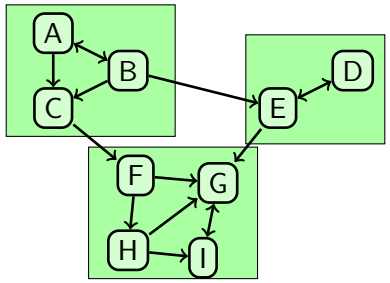
Grundregeln bei der Erstellung von Systemarchitekturen

- **Hohe Kohärenz** (high cohesion):
  Zusammenfassung logisch zusammengehörender Elememente in einem Teilsystem
- **Geringe Kopplung** (low coupling):
  Möglichst wenige Abhängigkeiten zwischen den einzelnen Teilsystemen (insbesondere keine zyklischen Abhängigkeiten)



Niedrige Kohärenz & starke Kopplung Hohe Kohärenz & geringe Kopplung

Grundregeln bei der Erstellung von Systemarchitekturen (2)



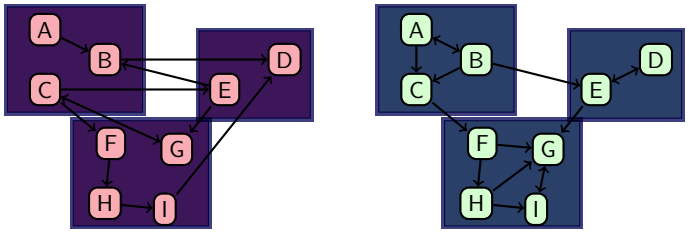
Vorteile bei geringer Kopplung:

- leichte Änderbarkeit und
- leichte Ausstauschbarkeit der einzelnen Teile

Grundregeln bei der Erstellung von Systemarchitekturen (3)

Beachte:

- Wird ein Teil T geändert, so
- müssen alle Teile, die auf T verweisen, auf etwaige nötige Änderungen geprüft werden.



Schichtenarchitekturen

- Oft baut man ein System als **Schichtenarchitektur** auf
- Die **untere** Schicht stellt Dienste für **darüberliegende** Schichten bereit
- Beispiel: Das OSI-Schichtenmodell für Netzwerkprotokolle besteht aus 7 Schichten:

Schicht 7	Anwendungsschicht (Application Layer)
Schicht 6	Darstellungsschicht (Presentation Layer)
Schicht 5	Sitzungsschicht (Session Layer)
Schicht 4	Transportschicht (Transport Layer)
Schicht 3	Vermittlungsschicht (Network Layer)
Schicht 2	Sicherungsschicht (Data Link Layer)
Schicht 1	Bitübertragungsschicht (Physical Layer)

OSI = Open Systems Interconnection

- Beispiele: Betriebssystemschichten, 3-Ebenen-Datenbankarchitektur, ...

Schichtenarchitekturen (2)

Geschlossene Architekturen

- Eine Schicht darf nur auf die direkt darunterliegende Schicht zugreifen

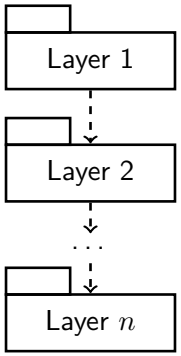
Offene Architekturen

- Eine Schicht darf auf alle darunterliegende Schichten zugreifen

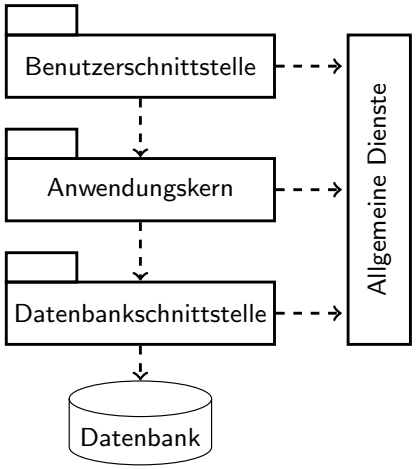
Client/Server-Systeme

- Verschiedene Schichten sind auf verschiedene Rechner verteilt.

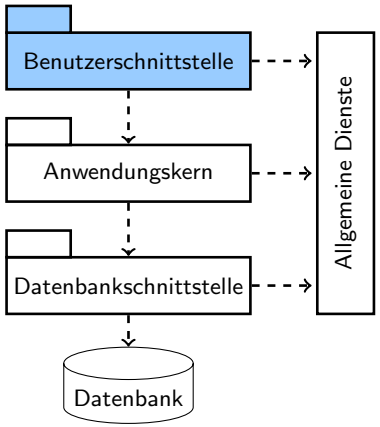
Beachte: Eine Schicht kann selbst wieder aus verschiedenen Teilsystemen bestehen



Drei-Schichten-Architektur für betriebliche Informationssysteme (1)



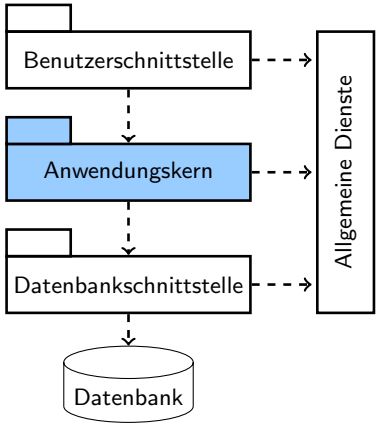
Drei-Schichten-Architektur für betriebliche Informationssysteme (2)



Benutzerschnittstelle

- Behandlung von Terminalereignissen (Maus-Klick, Tastendruck,...)
- Ein-/Ausgabe von Daten, Diaglogkontrolle

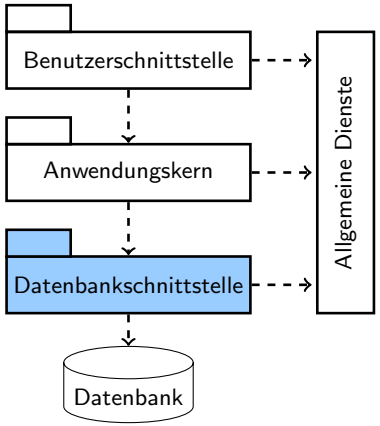
Drei-Schichten-Architektur für betriebliche Informationssysteme (3)



Anwendungskern (Fachkonzept)

- Zuständig für die Anwendungslogik (die eigentlichen Aufgaben des Problembereichs)

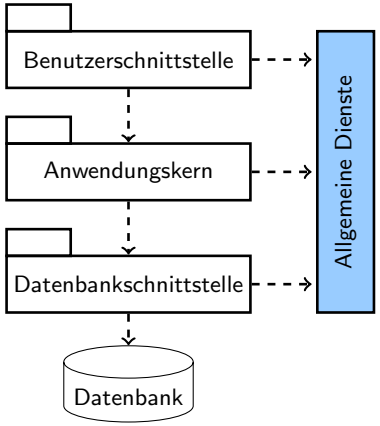
Drei-Schichten-Architektur für betriebliche Informationssysteme (4)



Datenbankschnittstelle

- Sorgt für die Speicherung von und den Zugriff auf persistente Daten der Anwendung

Drei-Schichten-Architektur für betriebliche Informationssysteme (5)



Allgemeine Dienste

- z.B. Kommunikationsdienste, Dateiverwaltung, Bibliotheken (APIs, GUI, math. Funktionen,...)

Thin vs. Thick-Client

Bei Client/Server-Architekturen (z.B. Web-Anwendungen) spricht man von einem

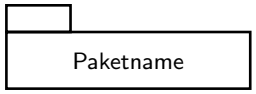
- **Thick-Client**, wenn Benutzerschnittstelle und Anwendungskern auf demselben Rechner ausgeführt werden.
- **Thin-Client**, wenn Benutzerschnittstelle und Anwendungskern auf verschiedene Rechner verteilt sind.

Pakete

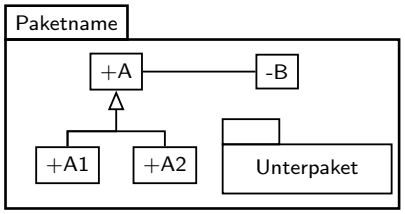
- Pakete dienen zur Gruppierung von Elementen größerer Systeme.
- Sie können sowohl in der Modellierung (UML) als auch in der Programmierung (Java) verwendet werden.

Pakete in UML-Notation

- Pakete ohne Anzeigen der Inhalte

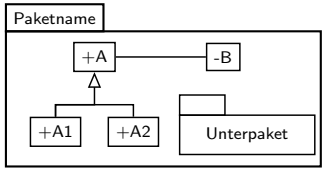


- Pakete mit Anzeigen der Inhalte



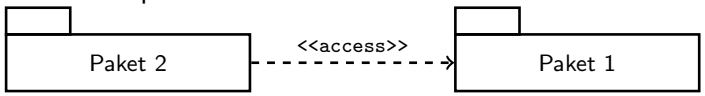
Sichtbarkeiten und Elemente von Paketen

- Klassen in Paketen sind öffentlich **oder** nur im selben Paket sichtbar
- Pakete können Unterpakete enthalten
- Die öffentlichen Elemente eines Pakets sind außerhalb des Pakets (immer) zugreifbar unter Verwendung ihres qualifizierten Namens
  z.B. **Paketname::A** in UML und **Paketname.A** in Java



Importieren von Paketen

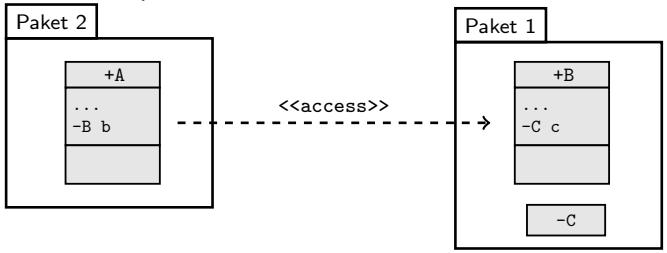
- Durch **Importieren** können die Namen von öffentlichen Elementen eines (importierten) Pakets in den Namensraum eines anderen (importierenden) Pakets übernommen werden.
  Privater Import in UML:



- Privater Import: Importierte Elemente können nicht weitergegeben werden. Ihre Sichtbarkeit wird im importierenden Paket auf „privat“ gesetzt.

Importieren von Modellelementen

- Klassen (und Interfaces) können auch einzeln aus anderen Paketen importiert werden



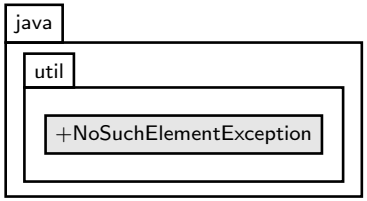
- Die öffentliche Klasse `B` (und ihre öffentlichen Elemente) sind außerhalb des Paket 1 zugreifbar, und zwar direkt mit ihrem Namen (ohne Qualifizierung), wenn die Klasse importiert wird.

Java: Verwendung vorhandener Pakete (1)

- Wir haben bereits mehrfach Standardbibliotheken in Java importiert.
- Dafür steht die `import`-Anweisung zur Verfügung.
- Beispiel:

```
import java.util.NoSuchElementException;
```

  Hierbei ist:
  - `java` ein Paket
  - `util` ein Unterpaket (von `java`)
  - `NoSuchElementException` eine Klasse des Unterpakets `util`



- Nach dem Import kann die Klasse `NoSuchElementException` verwendet werden.

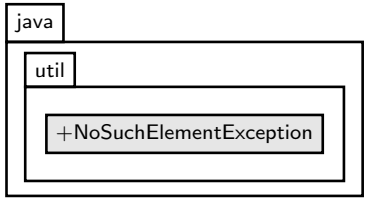
Java: Verwendung vorhandener Pakete (2)

- Verwendung von Klassen aus Paketen ist auch **ohne** import-Anweisung möglich:

  Verwende **voll-qualifizierten Namen**

- Beispiel:

```
java.util.NoSuchElementException e =
   new java.util.NoSuchElementException("Fehler...");
```



Konflikte beim Import

Annahme:

Die Pakete `meinpaket1` und `meinpaket2` exportieren jeweils eine Klasse MeineKlasse.

Importiert man beide Klassen, so führt das zum **Compilierfehler**.

```
import meinpaket1.MeineKlasse;
import meinpaket2.MeineKlasse;
```

```
error: a type with the same simple name is already defined
by the single-type-import of MeineKlasse
import meinpaket2.MeineKlasse;
```

Richtig ist in diesem Fall

Verwende die Klassen mit voll qualifizierten Namen (ohne Import-Anweisungen)

Alle Klassen und Interfaces eines Pakets importieren

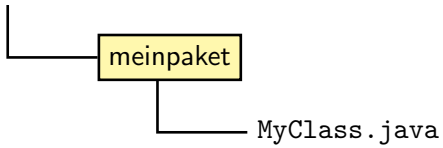
- Mit `import Paketname.*;` werden alle (sichtbaren) Klassen und Interfaces des Pakets importiert.
- Beachte: **Unterpakete** und deren Klassen und Interfaces werden dadurch **nicht** importiert.
- Diese müssen mit `import Paketname.Unterpaketname.*;` importiert werden.

Eigene Pakete deklarieren

- Die Zugehörigkeit einer Quelldatei zu einem Paket `meinpaket` wird durch die Zeile

```
package meinpaket;
```

  gekennzeichnet.
- Die Zeile muss **vor** den `import`-Anweisungen stehen.
- Die Quelldatei muss im Verzeichnis `meinpaket` im Dateisystem abgelegt werden.
- Name der Quelldatei: `MyClass.java`, wenn sie die Klasse `MyClass` als öffentliche Klasse definiert (mit `public` gekennzeichnet).
- Pro `.java`-Datei darf nur eine Klasse als `public` definiert werden.

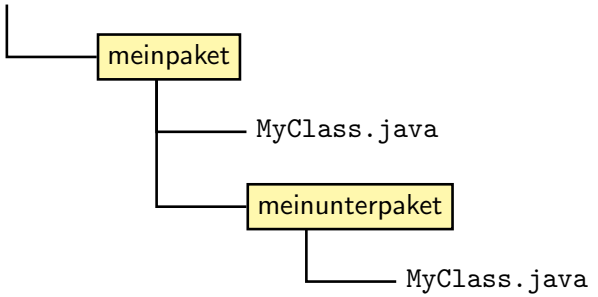


Eigene Pakete deklarieren (2)

- Unterpakete werden durch die Punkt-Notation deklariert, z.B.

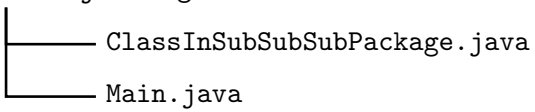
```
package meinpaket.meinunterpaket;
```

  deklariert `meinunterpaket` als Unterpaket von `meinpaket`
- Die Quelldateien des Unterpakets müssen im Verzeichnis `meinunterpaket` liegen, welches selbst im Verzeichnis `paket` liegt.



Beispiel: Falsche Verzeichnisstruktur

Annahme: Die Dateien `ClassInSubSubSubPackage.java` und `Main.java` liegen im **selben** Verzeichnis im Dateisystem.



Datei `Main.java`

```
import top.sub.sub.subpackage.*;
public class Main {
 public static void main(String[] args)
 { ClassInSubSubSubPackage test =
     new ClassInSubSubSubPackage();
 }
}
```

Datei `ClassInSubSubSubPackage.java`

```
package top.sub.sub.subpackage;
public class ClassInSubSubSubPackage {}
```

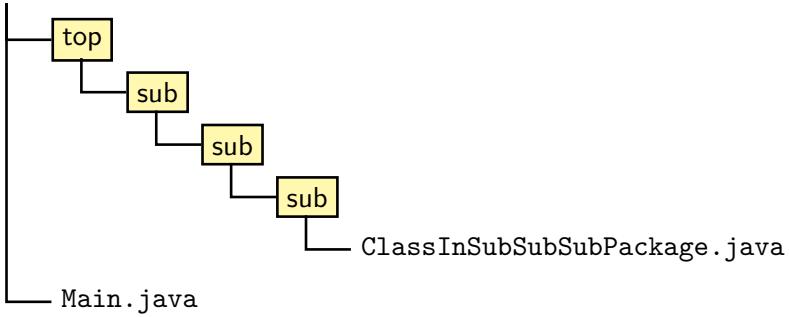
Kompilieren ergibt einen **Fehler**:

```
Main.java:1: error: package top.sub.sub.subpackage
does not exist
import top.sub.sub.subpackage.*;
^
```


Beispiel mit richtiger Verzeichnisstruktur

Die Datei

- `ClassInSubSubSubPackage.java` liegt im Verzeichnis `top/sub/sub/sub/` und
- `Main.java` liegt im aktuellen Verzeichnis, das `top` enthält



Nun sind die Konventionen eingehalten und die Kompilierung ist erfolgreich.

Sichtbarkeiten: Klassenebene

- Auf Klassenebene kennt Java nur zwei Sichtbarkeiten:
  - Öffentlich (Klasse ist mit `public class ...` definiert.)
  - „Package-private“ (die Klasse ist ohne `public` definiert).
- Öffentliche Klassen sind überall sichtbar
- Package-private Klassen können nur innerhalb desselben Pakets verwendet werden.

Beispiel: Sichtbarkeit auf Klassenebene

Datei: `paket1/PublicClass1.java`

```
package paket1;
public class PublicClass1 {
 PublicClass1() {
   PackagePrivateClass1 m = new PackagePrivateClass1();
 }
}
```

Datei: `paket1/PackagePrivateClass1.java`

```
package paket1;
class PackagePrivateClass1 {
 PackagePrivateClass1() {}
}
```

Kompilierung

```
javac paket1/PublicClass1.java
```

ist erfolgreich (da `PublicClass1` auf `PackagePrivateClass1` zugreifen darf, da beide in Paket `paket1` sind)

Beispiel: Sichtbarkeit auf Klassenebene (2)

Datei: `paket2/PublicClass2.java`

```
package paket2;
import paket1.PublicClass1;
import paket1.PackagePrivateClass1;
public class PublicClass2 {
 PublicClass2() {}
}
```

Kompilierung schlägt fehl:

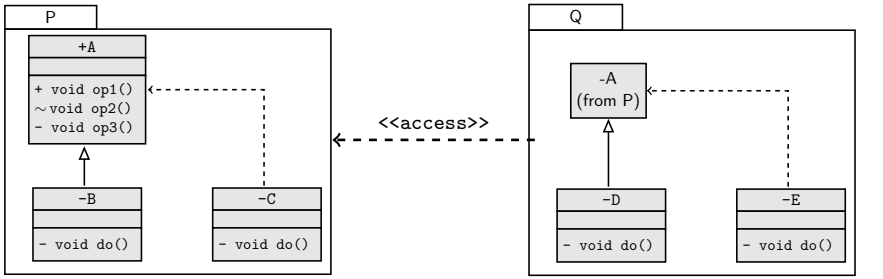
```
>javac paket2/PublicClass2.java
paket2\PublicClass2.java:3: error: PackagePrivateClass1 is not
public in paket1; cannot be accessed from outside package
import paket1.PackagePrivateClass1;
               ^
```


Sichtbarkeiten auf Attribut- und Methodenebene

Modifier	Zugriff von			
	selber Klasse	selbes Paket	Unterklasse	irgendwo
public	erlaubt	erlaubt	erlaubt	erlaubt
protected	erlaubt	erlaubt	erlaubt	verboten
ohne	erlaubt	erlaubt	verboten	verboten
private	erlaubt	verboten	verboten	verboten

- Ohne Modifier nennt man auch „package-private“
- In UML-Notation werden „package-private“ Methoden durch voranstehendes Symbol $\sim$ gekennzeichnet

Pakete in UML und Java: Beispiel




```
package P;
public class A {
 public void op() {}
 void op2() {}
 private void op3() {}}
```

```
package P;
class C {
 private void do() {
  A a = new A();
  a.op1();
  a.op2();
  a.op3();} }
```

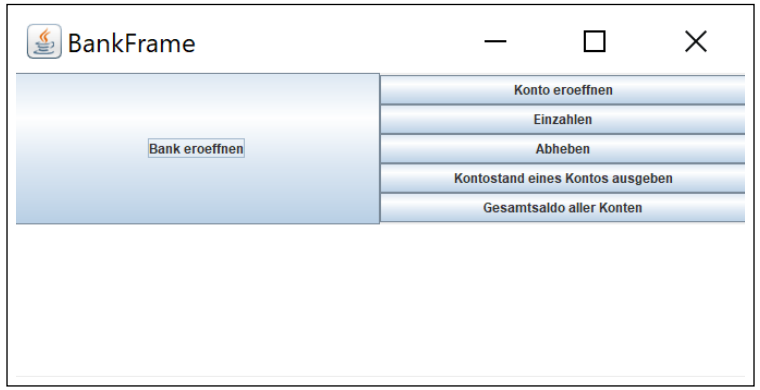
```
package Q;
import P.*;
class D extends A {
 private void do() {
  this.op1();
  this.op2();} }
```

```
package Q;
import P.*;
class E {
 private void do() {
  A a = new A();
  a.op1();
  a.op2();
  B a = new B();} }
```

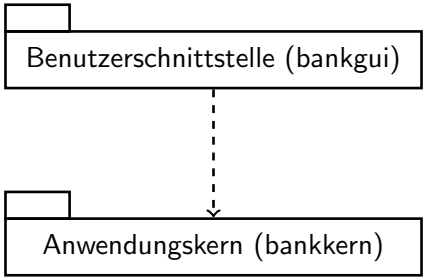
```
package P;
class B extends A {
 private void do() {
  this.op1();
  this.op2();
  this.op3();} }
```


Beispiel: Banksystem

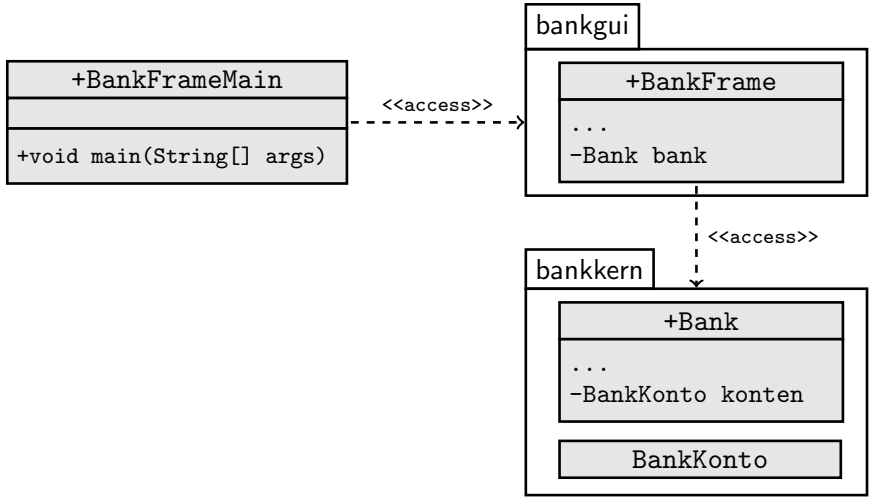
Wir restrukturieren das Banksystem aus Zentralübung 10.



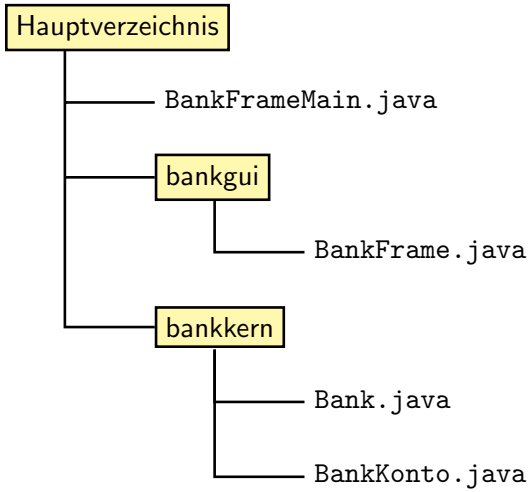
Banksystem als 2-Schichtenarchitektur (1)



Banksystem als 2-Schichtenarchitektur (2)



Verzeichnisstruktur und Quelltextdateien



Java-Klasse BankFrameMain

```
//Import aller oeffentlichen Klassen des Pakets bankgui
import bankgui.*;

public class BankFrameMain {

  public static void main(String[] args) {
    BankFrame frame = new BankFrame();
    frame.setVisible(true);
  }
}
```


Java-Klasse BankFrame

```
// Zugehoerigkeit zum Paket bankgui
package bankgui;

// Import der oeffentlichen Klassen des Pakets bankkern
import bankkern.*;
import java.awt.Container;
...

public class BankFrame extends JFrame implements ActionListener {
...
  private Bank bank;
...
```


Java-Klasse Bank

```
// Zugehoerigkeit zum Paket bankkern
package bankkern;
public class Bank {
  ...
  private BankKonto[] konten;
  ...
}
```

Java-Klasse BankKonto

```
// Zugehoerigkeit zum Paket bankkern
package bankkern;

public class BankKonto {
  ...
}
```


Zusammenfassung

- Systemarchitektur: Struktur des Softwaresystems (Teile und Verbindungen)
- Hohe Kohärenz und Geringe Kopplung
- Schichtenarchitekturen
- Pakete: UML und Java