Chapter 4

Software Engineering

Prof. Mirco Tribastone, Ph.D.

20 December 2012

Aims

- Learn the derivation a static and dynamic design model from an object oriented design.
- Understand different alternatives for the realisation of state diagrams.
- Understand the principles of systems architectures.
- Develop graphical user interfaces (GUIs).
- Link the application to a relational database.
- Learn about principles of distributed systems (if time permits).

Starting point

Static and dynamic model in the object oriented design process

Aim

Model of the system implementation (describing *how* all the required functionalities are implemented)

Challenges

- Refinement of a design model through integration of static and dynamic models. Leads to *object design*.
- Integration into the system environment by design of user interfaces, interfaces to data bases, network wrappers, etc.
- Final design engineering of the system architecture

4.1 Object Design

The static model is extended and revised by using dynamic models developed during the analysis phase.

Role of the object design

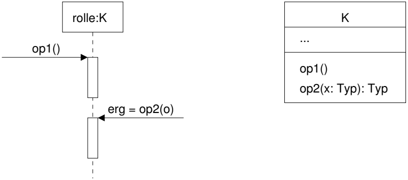
1. Add operations
2. Add directions in associativity
3. Add access rights
4. Resolve multiple inheritance
5. Improve reusability of classes

4.1.1 Add operations

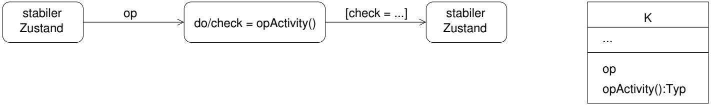
Procedure

Let K be a class of the object model of the analysis phase.

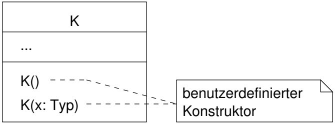
- Add an operation for each message sent to an object of K.



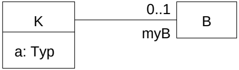
- Add an operation for each call event in the state diagrams and for each (local) operation that is called in an activity state (if necessary also for the activities in activity diagrams)

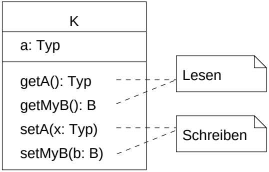


- Add user-defined constructors for non-abstract classes

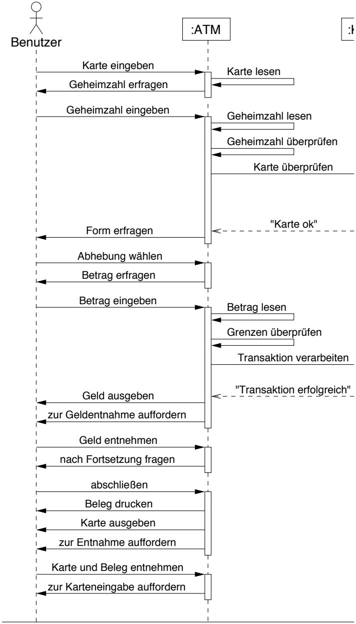


- Add necessary access operations for attributes and roles (read and/or write operations)





Example ATM:



Benutzer

:ATM

:k

Karte eingeben

Karte lesen

Geheimzahl erfragen

Geheimzahl eingeben

Geheimzahl lesen

Geheimzahl überprüfen

Karte überprüfen

"Karte ok"

Form erfragen

Abhebung wählen

Betrag erfragen

Betrag eingeben

Betrag lesen

Grenzen überprüfen

Transaktion verarbeiten

"Transaktion erfolgreich"

Geld ausgeben

zur Geldentnahme auffordern

Geld entnehmen

nach Fortsetzung fragen

abschließen

Beleg drucken

Karte ausgeben

zur Entnahme auffordern

Karte und Beleg entnehmen

zur Karteneingabe auffordern

ATM

geldvorrat: Real
grenzen: Real

ATM()

karteEin
abbruch
geheimzahlEin
abhebungWählen
betragEin
geldEntnehmen
abschliessen
karteBelegEntnehmen

karteneingabeAuffordernActivity()
karteEinActivity(): String
geheimzahlEinActivity(): String
abhebungActivity()
betragEinActivity(): String
geldEntnehmenActivity()
abschlussActivity()

karteLesen(): String
geheimzahlÜberprüfen(): String
grenzenÜberprüfen(): String

Konsortium

name: String

karteÜberprüfen(): String
transaktionVerarbeiten(): String
blzÜberprüfen(): String

Bank

blz: Integer
name: String

bankKarteÜberprüfen(): String
bankTransaktionVerarbeiten(): String
kartennrÜberprüfen(): String
kontoAktualisieren(): String

- Describe algorithms of the operations

  Input: Interaction diagrams, if applicable also activity diagrams of the analysis phase

  Possible presentation of the algorithms:

  - Detailed activity diagrams
    (detailed interaction diagrams where appropriate)
  - Pseudo code
    (for instance based on Java or by using an "Action Language")

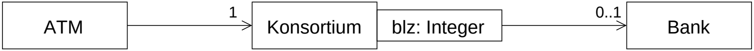
  Note:

  During the development of the algorithms the object model is revised. If necessary derived (redundant) associations for a direct access to other objects are added.

4.1.2 Add association directions

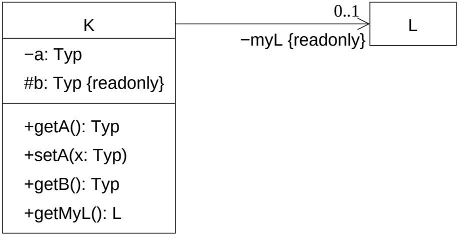
- Analyze in which direction(s) the associations (for sending messages or calling operations) are used.
- If an association is only used in one direction, then add an arrow correspondingly.

ATM Example:



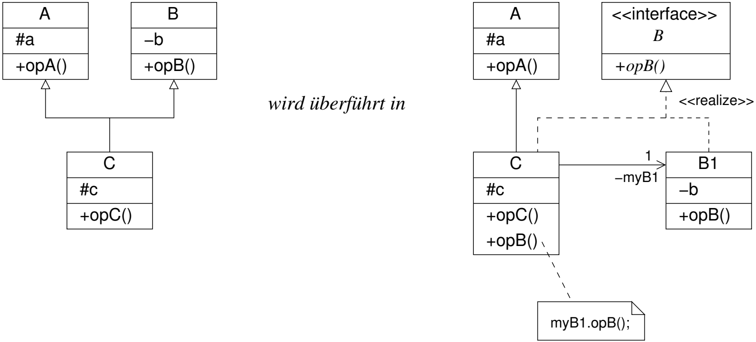
4.1.3 Add access rights

Add the access rights for attributes, role names and operations. (Note that attributes and role names should not be public!)



4.1.4 Resolve multiple inheritance

- Is necessary if the target language does not support multiple inheritance for classes (like Java).
- Resolving multiple inheritance is possible by introducing interfaces.



A
#a
+opA()

B
–b
+opB()

C
#c
+opC()

wird überführt in

A
#a
+opA()

<<interface>>
B
+opB()

<<realize>>

C
#c
+opC()
+opB()

1
–myB1

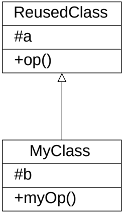
B1
–b
+opB()

myB1.opB();

4.1.5 Reuse of classes

- It is often advisable to reuse well-tested, existing classes in the design.
- Missing functionalities of a reused class can be added in a subclass by **specialisation**.

In general:



ReusedClass

#a

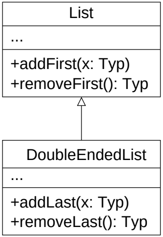
+op()

MyClass

#b

+myOp()

Example:



List

...

+addFirst(x: Typ)
+removeFirst(): Typ

DoubleEndedList

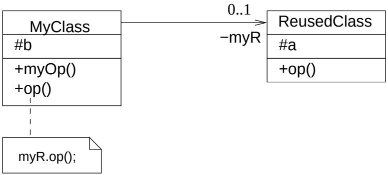
...

+addLast(x: Typ)
+removeLast(): Typ

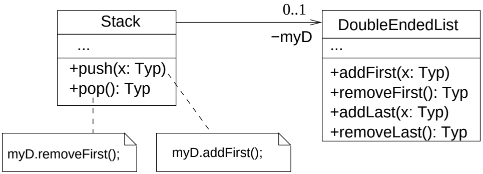
Note: If the reused class provides operations that the specialized class does not need, then the encapsulation principle might be violated. In this case, reuse by *delegation* should be preferred.

- Reuse by **delegation**:

In general:



Example:



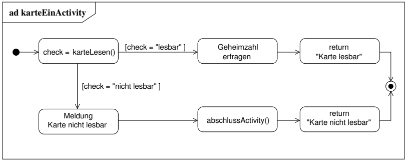
4.1.6 Object Design for ATM

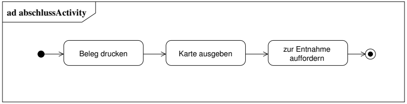
- Algorithms for the operations (identified in Section 4.1.1) are developed (in Java pseudocode).
- The algorithms are deduced by refinement of the activity diagrams of Section 3.4.
- Additionally, required constructors and access operations (*getter* and *setter* methods) are identified.
- The class diagram from the analysis phase is revised accordingly.

Algorithmen

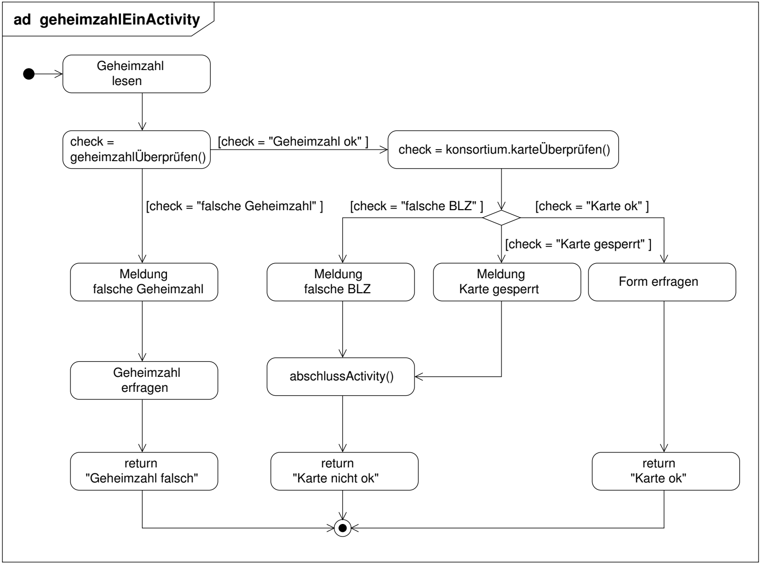
1. Operations of the class ATM

Operations that are events of the state diagram (karteEin, ..., karteBelegEntnehmen) are not considered here. They are handled later on during realization of the state diagram.





```
// Konstruktor ATM
ATM() {
  geldvorrat = 100000;
  grenzen    = 250;
  karteneingabeAuffordernActivity();
}
karteneingabeAuffordernActivity() {
  output("Karte eingeben?");
}

karteEinActivity(): String {
  String check = karteLesen();

  if (check.equals("lesbar")) {
    output("Geheimzahl?");
    return "Karte lesbar";
  }
  else if (check.equals("nicht lesbar")) {
    output("Karte nicht lesbar!");
    abschlussActivity();
    return "Karte nicht lesbar";
  }
  else return "Error";
}
```



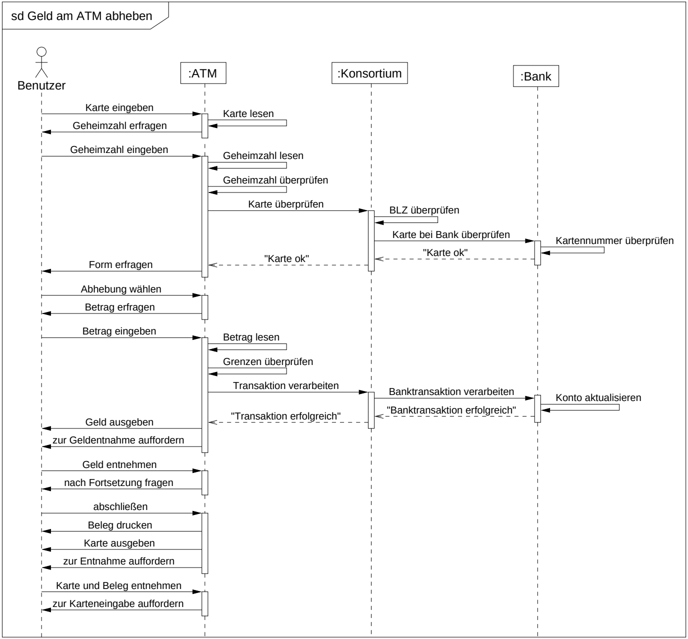
[from Section 3]

```
geheimzahlEinActivity(): String {
  Integer typedGeheimzahl = inputTypedGeheimzahl();
  String  check = geheimzahlUeberpruefen(typedGeheimzahl);

  if (check.equals("Geheimzahl ok")) {
    // neues Attribut aktKontonr
    check = konsortium.karteUeberpruefen(aktKartennr, aktBLZ, aktKontonr);
    if (check.equals("Karte ok")) {
      output("Transaktionsform?");
      return "Karte ok";
    }
    else if (check.equals("falsche BLZ")) {
      output("falsche BLZ!");
      abschlussActivity();
      return "Karte nicht ok";
    }
    else if (check.equals("Karte gesperrt")) {
      output("Karte gesperrt!");
      abschlussActivity();
      return "Karte nicht ok";
    }
    else return "Error";
  }
  else if (check.equals("falsche Geheimzahl")) {
    output("falsche Geheimzahl!");
    output("Geheimzahl?");
    return "Geheimzahl falsch";
  }
  else return "Error";
}
abhebungActivity() {
  output("Betrag?"); }
```

```
betragEinActivity(): String {
  Real betrag = inputBetrag();
  String check = grenzenUeberpruefen(betrag);
  if (check.equals("Grenzen ok")) {
    check = konsortium.transaktionVerarbeiten(aktBLZ, aktKontonr, betrag);
    if (check.equals("Transaktion erfolgreich")) {
      Aussentransaktion atrans =
        new Aussentransaktion("Abhebung", aktKartennr, betrag, aktBLZ, aktKontonr);
      addTransaktion(atrans); // neue Operation von Terminal
      geldvorrat = geldvorrat-betrag;
      output("Geld ausgeben");
      output("Geld entnehmen?");
      return "Transaktion erfolgreich";
    }
    else if (check.equals("Transaktion gescheitert")) {
      output("Transaktion gescheitert!");
      output("Transaktionsform?");
      return "Transaktion gescheitert";
    }
    else return "Error";
  }
  else if (check.equals("Grenzen ueberschritten")) {
    output("Grenzen ueberschritten!");
    output("Betrag?");
    return "Grenzen ueberschritten";
  }
  else return "Error";
}
```

```
geldEntnehmenActivity() {
  output("Fortsetzung?");
}

abschlussActivity() {
  output("Beleg drucken");
  output("Karte ausgeben");
  output("Karte und Beleg entnehmen?");
}
```

```
karteLesen(): String {
  aktKartennr   = inputKartennr();   // neues Attribut von ATM
  aktBLZ        = inputBLZ();        // neues Attribut von ATM
  aktGeheimzahl = inputGeheimzahl(); // codierte Geheimzahl, neues Attribut von ATM
  return "lesbar";
}

geheimzahlUeberpruefen(tgz: Integer): String {
  if (tgz == aktGeheimzahl) return "Geheimzahl ok";
  else return "falsche Geheimzahl";
}

grenzenUeberpruefen(b: Real): String {
  if (b <= grenzen) return "Grenzen ok";
  else return "Grenzen ueberschritten";
}
```



[Sequence diagram from Chapter 3, slide 52]

2. Operations of the class Konsortium

```
karteUeberpruefen(kartennr: Integer, blz: Integer, out kontonr: Integer): String {
  String check = blzUeberpruefen(blz);

  if (check.equals("BLZ richtig")) {
    check = banken[blz].bankKarteUeberpruefen(kartennr, kontonr);
    if (check.equals("Karte ok")) return "Karte ok";
    else if (check.equals("Karte bei Bank gesperrt")) return "Karte gesperrt";
    else return "Error";
  }
  else if (check.equals("BLZ falsch")) return "falsche Bankleitzahl";
  else return "Error";
}


transaktionVerarbeiten(blz: Integer, kontonr: Integer, b: Real): String {
  String check = banken[blz].bankTransaktionVerarbeiten(kontonr, b);

  if (check.equals("Banktransaktion erfolgreich"))
    return "Transaktion erfolgreich";
  else if (check.equals("Banktransaktion gescheitert"))
    return "Transaktion gescheitert";
  else return "Error";
}


blzUeberpruefen(blz: Integer): String {
  if (banken[blz] != null) return "BLZ richtig";
  else return "BLZ falsch";
}
```

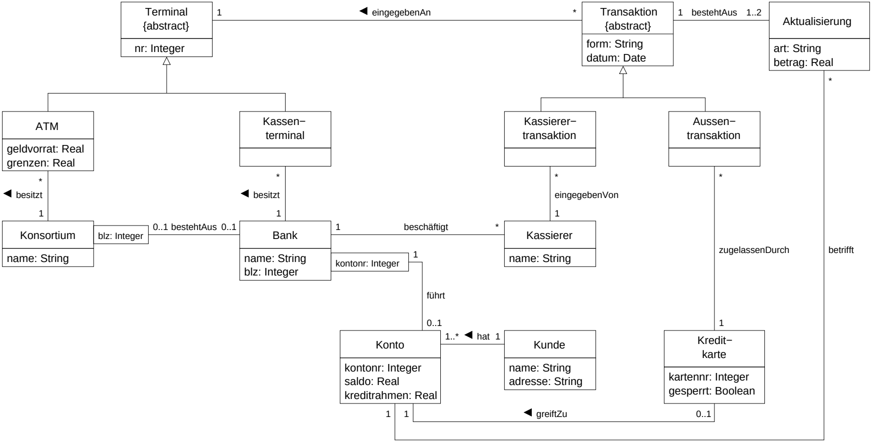
3. Operations of the class Bank

```
bankKarteUeberpruefen(kartennr: Integer, out kontonr: Integer): String {
  String check = kartennrUeberpruefen(kartennr, kontonr);

  if (check.equals("gueltig")) return "Karte ok";
  else if (check.equals("gesperrt")) return "Karte bei Bank gesperrt";
  else return "Error";
}
bankTransaktionVerarbeiten(kontonr: Integer, b: Real): String {
  String check = kontoAktualisieren(kontonr, b);

  if (check.equals("erfolgreich")) return "Banktransaktion erfolgreich";
  else if (check.equals("gescheitert")) return "Banktransaktion gescheitert";
  else return "Error";
}
kartennrUeberpruefen(kartennr: Integer, out kontonr: Integer) :String {
  // kreditkarten ist Rollenname einer neuen (abgeleiteten) qualifizierten
  // Assoziation zwischen Bank und Kreditkarte
  if (kreditkarten[kartennr] != null) {
    // getKonto() und getKontonr() werden als Zugriffsoperationen bei
    // Kreditkarte bzw. Konto gebraucht
    kontonr = kreditkarten[kartennr].getKonto().getKontonr();
    if (! kreditkarten[kartennr].getGesperrt()) return "gueltig";
    else return "gesperrt";
  } else return "Error";
}
kontoAktualisieren(kontonr: Integer, b: Real): String {
  Konto k = konten[kontonr];
  if (k.getSaldo()-b >= k.getKreditrahmen()) {
    k.abheben(b);
    return "erfolgreich";}
  else return "gescheitert";}
```

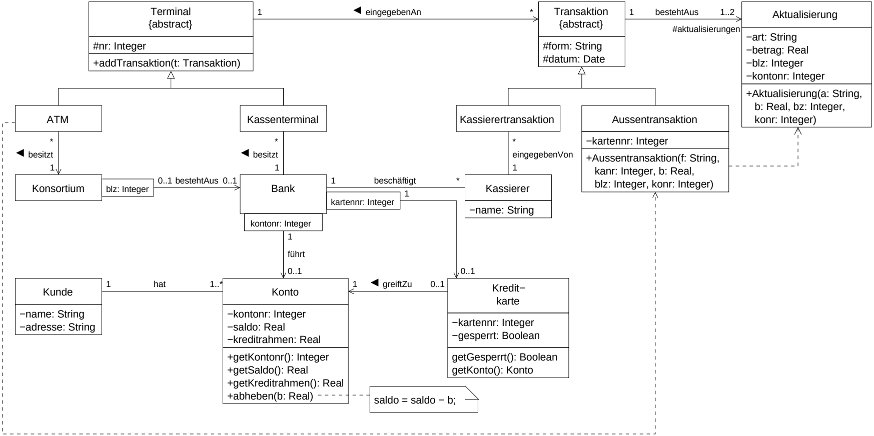
4. Constructors of “Aussentransaktion” and “Aktualisierung”

```
Aussentransaktion(f: String, kanr: Integer, b: Real, blz: Integer, konr: Integer) {
  form = f;
  datum = new date();
  // Neues Attribut kartennr von Aussentransaktion.
  // Dafuer wird die Assoziation zu Kreditkarte gestrichen.
  kartennr = kanr;

  if (form.equals("Abhebung"))
    aktualisierungen[0] = new Aktualisierung("Lastschrift", b, blz, konr);

  else if (form.equals("Einzahlung"))
    aktualisierungen[0] = new Aktualisierung("Gutschrift", b, blz, konr);
}


Aktualisierung(a: String, b: Real, bz: Integer, konr: Integer) {
  art = a;
  betrag = b;
  // blz und kontonr sind neue Attribute von Aktualisierung.
  // Dafuer wird die Assoziation zu Konto gestrichen.
  blz = bz;
  kontonr = konr;
}
```

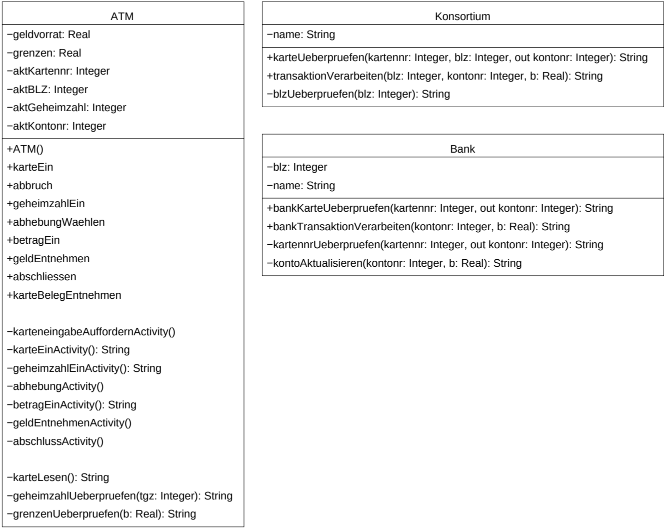
Class digram of ATM before object design (from Section 3)



Class digram of ATM after object design



Revised classes



ATM

−geldvorrat: Real
−grenzen: Real
−aktKartennr: Integer
−aktBLZ: Integer
−aktGeheimzahl: Integer
−aktKontonr: Integer

+ATM()
+karteEin
+abbruch
+geheimzahlEin
+abhebungWaehlen
+betragEin
+geldEntnehmen
+abschliessen
+karteBelegEntnehmen

−karteneingabeAuffordernActivity()
−karteEinActivity(): String
−geheimzahlEinActivity(): String
−abhebungActivity()
−betragEinActivity(): String
−geldEntnehmenActivity()
−abschlussActivity()

−karteLesen(): String
−geheimzahlUeberpruefen(tgz: Integer): String
−grenzenUeberpruefen(b: Real): String

Konsortium

−name: String

+karteUeberpruefen(kartennr: Integer, blz: Integer, out kontonr: Integer): String
+transaktionVerarbeiten(blz: Integer, kontonr: Integer, b: Real): String
−blzUeberpruefen(blz: Integer): String

Bank

−blz: Integer
−name: String

+bankKarteUeberpruefen(kartennr: Integer, out kontonr: Integer): String
+bankTransaktionVerarbeiten(kontonr: Integer, b: Real): String
−kartennrUeberpruefen(kartennr: Integer, out kontonr: Integer): String
−kontoAktualisieren(kontonr: Integer, b: Real): String

Summary of Section 4.1

- The object design is the outcome of the integration of the static and dynamic model of the analysis phase (the treatment of state diagrams is described in the next section).
- In the object design, the operations are added to the classes.
- The algorithms of the (non-trivial) operations are described
    - by (preferably complete) activity diagrams, or
    - by pseudocode deduced by refinement of the activity diagrams.
- During the formulation of algorithms for the operations, the static model is continuously revised (amongst others, for refinement of the associations, or for introducing newly found associations).
- Another important task of object design is to possibly resolve multiple inheritance and allow for class reuse.

4.2 Implementation of state diagrams

Given

State diagram of a class K.

Aim

Object design with algorithms realising the behaviour described by the state diagrams.

We consider four possibilities:

- *Procedural* implementation
- Implementation by case distinction
- Implementation by state objects
- Implementation by state machines

4.2.1 Procedural implementation

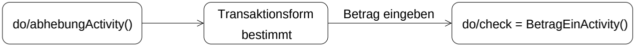
Idea

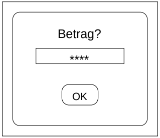
Events are implemented by *modal dialogues* (i.e., forced user inputs).

Prerequisite

Objects are located at the system *boundary*
(control objects for dialogues).

Example:





in pseudocode: `betrag = input(‘‘Betrag?");`

in Java: `String betrag = JOptionPane.showInputDialog(‘‘Betrag?");`

Approach

The whole state diagram is transformed to a procedure with

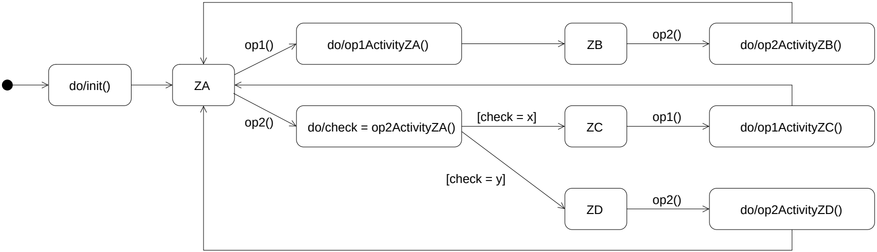
- modal dialogues for the (external) events
- conditional statements for branching
- iteration statements for cycles

Remark:

Flexible user interfaces are difficult to realise.

4.2.2 Implementation by case distinction

Consider the following state machine for class K.



Find: Realisation of op1 and op2 as well as the constructor of K.

Approach

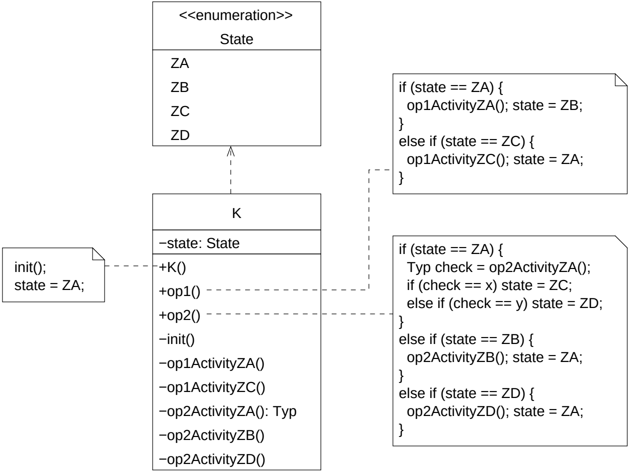
- Use an enumeration type for the representation of the (finitely many) stable states.
- Introduce an explicit state attribute for the considered class K.
- Implement state dependent operations by case distinction on the current (stable) state.

Con

Poor extensibility with respect to new states (new cases in *all* state dependent operations).

Pro

Easy extensibility with respect to new operations.



```
init();
state = ZA;
```

```
if (state == ZA) {
  op1ActivityZA(); state = ZB;
}
else if (state == ZC) {
  op1ActivityZC(); state = ZA;
}
```

```
if (state == ZA) {
  Typ check = op2ActivityZA();
  if (check == x) state = ZC;
  else if (check == y) state = ZD;
}
else if (state == ZB) {
  op2ActivityZB(); state = ZA;
}
else if (state == ZD) {
  op2ActivityZD(); state = ZA;
}
```

Remark

If type = String, replace `check == x` with `check.equals("x")`!

4.2.3 Implementation with state objects

Idea

- Each object of the class is linked to a state object representing the current state of the object.
- The call to a state-dependent operation is delegated to the state object.
- The current state object executes the desired activity.
- On a state change the new state object (of the correct subclass) is created and linked to the base object.

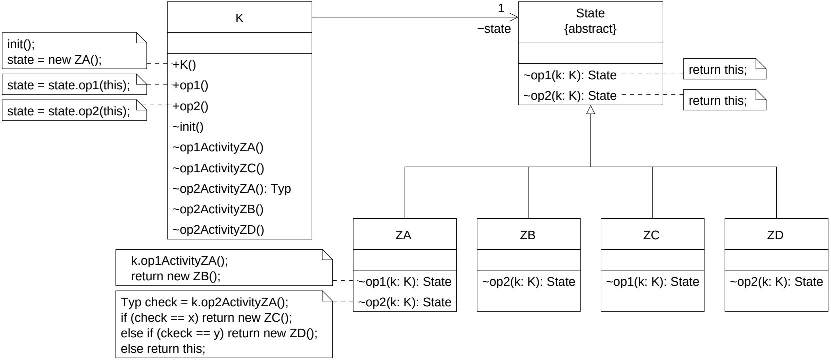
Pro

Easy extensibility with respect to new states.

Con

Poor extensibility with respect to new operations.

The above state diagram is implemented as follows:



```
init();
state = new ZA();
```

```
state = state.op1(this);
```

```
state = state.op2(this);
```

```
return this;
```

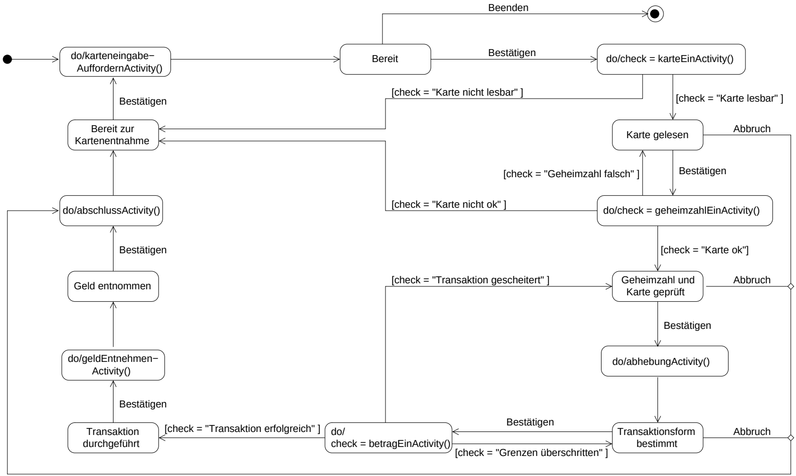
```
return this;
```

```
k.op1ActivityZA();
return new ZB();
```

```
Typ check = k.op2ActivityZA();
if (check == x) return new ZC();
else if (ckeck == y) return new ZD();
else return this;
```

Example: Implementation of the ATM class with state objects

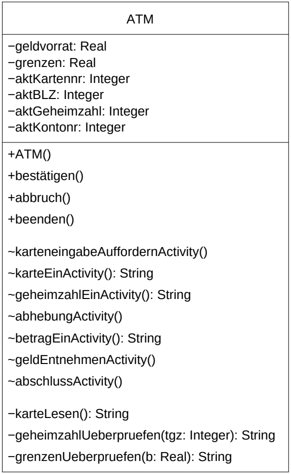
State diagram for the event-driven ATM simulation



The events “Bestätigen”, “Abbruch”and “Beenden”are caused by the activation of the corresponding buttons of the GUI.

Revised class ATM

The operations “karteEin”, “geheimzahlEin”, ..., “karteBelegEntnehmen” are replaced by the state dependent operation “bestaetigen”.



ATM

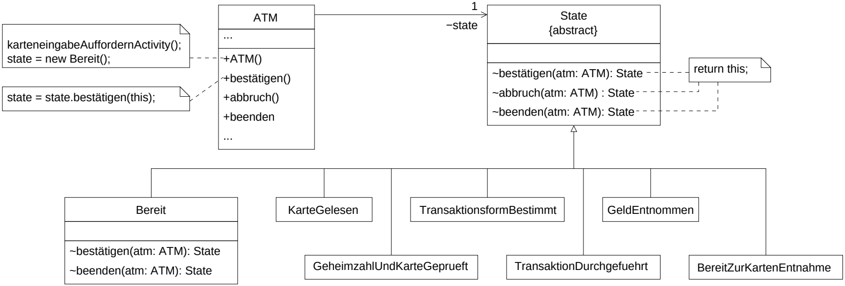
−geldvorrat: Real
−grenzen: Real
−aktKartennr: Integer
−aktBLZ: Integer
−aktGeheimzahl: Integer
−aktKontonr: Integer

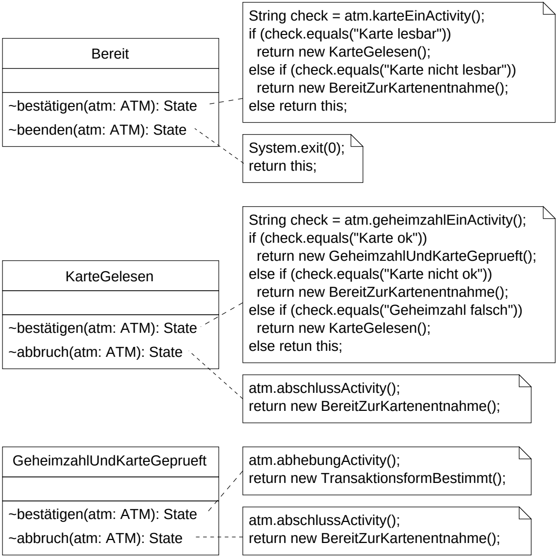
+ATM()
+bestätigen()
+abbruch()
+beenden()

~karteneingabeAuffordernActivity()
~karteEinActivity(): String
~geheimzahlEinActivity(): String
~abhebungActivity()
~betragEinActivity(): String
~geldEntnehmenActivity()
~abschlussActivity()

−karteLesen(): String
−geheimzahlUeberpruefen(tgz: Integer): String
−grenzenUeberpruefen(b: Real): String

Realisation of the state diagram of ATM simulation


karteneingabeAuffordernActivity();
state = new Bereit();
state = state.bestätigen(this);
ATM
...
+ATM()
+bestätigen()
+abbruch()
+beenden
...
1
−state
State
{abstract}
~bestätigen(atm: ATM): State
~abbruch(atm: ATM) : State
~beenden(atm: ATM): State
return this;
Bereit
~bestätigen(atm: ATM): State
~beenden(atm: ATM): State
KarteGelesen
GeheimzahlUndKarteGeprueft
TransaktionsformBestimmt
TransaktionDurchgefuehrt
GeldEntnommen
BereitZurKartenEntnahme



Bereit

~bestätigen(atm: ATM): State

~beenden(atm: ATM): State

```
String check = atm.karteEinActivity();
if (check.equals("Karte lesbar"))
  return new KarteGelesen();
else if (check.equals("Karte nicht lesbar"))
  return new BereitZurKartenentnahme();
else return this;
```

```
System.exit(0);
return this;
```

KarteGelesen

~bestätigen(atm: ATM): State

~abbruch(atm: ATM): State

```
String check = atm.geheimzahlEinActivity();
if (check.equals("Karte ok"))
  return new GeheimzahlUndKarteGeprueft();
else if (check.equals("Karte nicht ok"))
  return new BereitZurKartenentnahme();
else if (check.equals("Geheimzahl falsch"))
  return new KarteGelesen();
else retun this;
```

```
atm.abschlussActivity();
return new BereitZurKartenentnahme();
```

GeheimzahlUndKarteGeprueft

~bestätigen(atm: ATM): State

~abbruch(atm: ATM): State

```
atm.abhebungActivity();
return new TransaktionsformBestimmt();
```

```
atm.abschlussActivity();
return new BereitZurKartenentnahme();
```

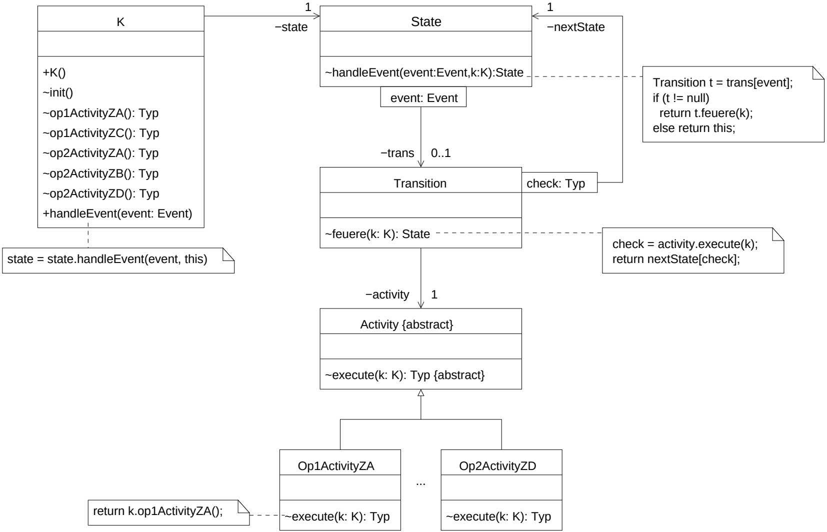
The remaining four state classes are similarly implemented.

4.2.4 Implementation with a state machine

Idea

- All entities of a state diagram (states, transitions, activities, events) are represented by objects.
- Events are interpreted by a special “Event-Handle”operation.
- The whole state diagram is represented by a (branching) object structure.

The state diagram is implemented by the following state machine:



```
state = state.handleEvent(event, this)
```

```
Transition t = trans[event];
if (t != null)
  return t.feuere(k);
else return this;
```

```
check = activity.execute(k);
return nextState[check];
```

```
return k.op1ActivityZA();
```

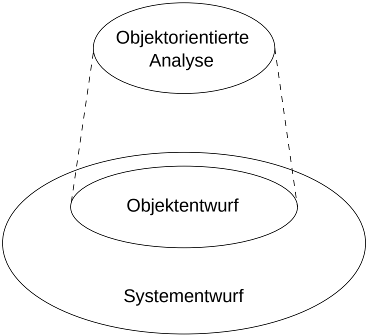
Summary of Section 4.2

- State diagrams can be systematically implemented and integrated into the object design.
- We distinguish between four alternatives for the implementation:
  - procedural implementation
  - case distinction
  - states as objects
  - state machines
- The overall object design for the ATM example consists of
  - the class diagram of Section 4.1, with the revised class AMT of Section 4.2,
  - the algorithms developed in Section 4.1,
  - the implementation of the state diagrams (Section 4.2).

4.3 System design

Aims

- Embedding the object design into the system environment
- Specification of the system architecture

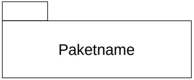


4.3.1 Packages and Components

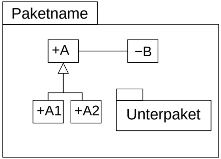
Packages serve as a structuring mechanism for models of larger systems. They gather several elements of the model into a common entity or group.

Representation of Packages in the UML

Packages without representation of contents:



Packages with representation of contents:

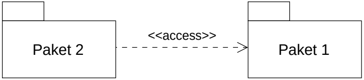


The public elements of a package are (always) accessible from outside by using *qualified names*, e.g. **Paketname::A**.

Import Relationships between Packages

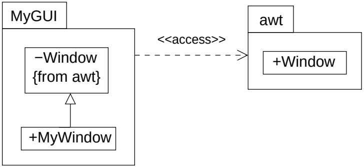
Import relationships allow to use names of public model elements of an imported package within the namespace of the importing package.

1. Import with private visibility:

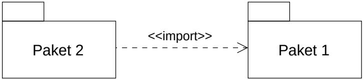


The visibility of the imported elements is set to “private”.

Example:



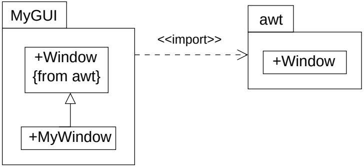
2. Import with public visibility:



The visibility of the imported elements is set to “public”.

The imported elements can be (transitively) imported by other packages.

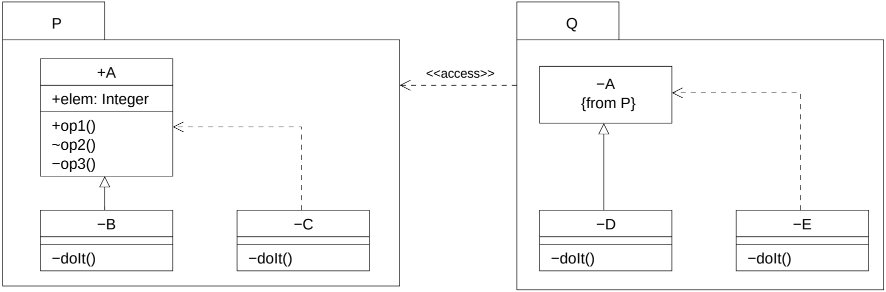
Example:



Implementation of Packages in Java

- For each package a directory with the package name is created; for each subpackage subdirectories are created.
- A class belonging to a package **P** is implemented in a file in the directory **P**.
- The file may contain at most one “**public class**”. The package name must be mentioned first: “**package P;**” or “**package P.U;**” if the class belongs to a subdirectory **U** of **P**.
- Java supports only private imports of packages and package elements: “**import P.*;**” or “**import P.U.*;**” or “**import P.Klassenname;**” or “**import P.U.Klassenname;**”

Example:



```
package P;
public class A {
  public int elem;
  public void op1() { }
  void op2() { }
  private void op3() { }
}
```

```
package P;
class B extends A {
  private void doIt {
    elem=1;
    op1();
    op2();
    ~~op3();~~
  }
}
```

```
package P;
class C {
  private void doIt {
    A a = new A();
    a.elem=1;
    a.op1();
    a.op2();
    ~~a.op3();~~
  }
}
```

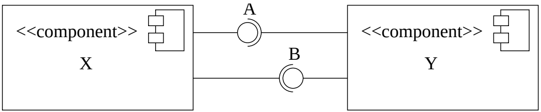
```
package Q;
import P.*;
class D extends A {
  private void doIt {
    elem=1;
    op1();
    ~~op2();~~
  }
}
```

```
package Q;
import P.A;
~~import P.B;~~
class E {
  private void doIt {
    A a = new A();
    a.elem=1;
    a.op1();
    ~~a.op2();~~
    ~~B b = new B();~~
  }
}
```

Components

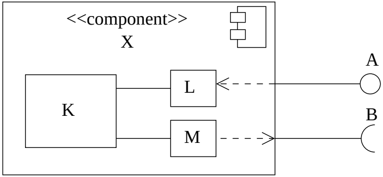
A component is a modular part of the system which encapsulates a complex (internal) structure and communicates with the environment via interfaces.

External view of components



The interface A is provided by X (*provided interface* of X) and is used or required by Y (*required interface* of Y).

Internal view of components



Remarks

- Components may have *ports* that group together several provided and required interfaces. The behaviour of ports and interfaces can be specified by UML *protocol state machines*.
- For the realization of a component we distinguish between *indirect* und *direct implementation*.s In the indirect implementation, the internal view is described via class diagrams (see above); in the direct implementation, the internal view is described via component diagrams.

4.3.2 Basics: System Architecture

The system architecture describes the whole structure of the software system with subsystems and relationships between them (by using interfaces).

Remarks

- A first draft of the system architecture is often made in the beginning of the system development.
- Subsystems are presented with packages and components (with interfaces).

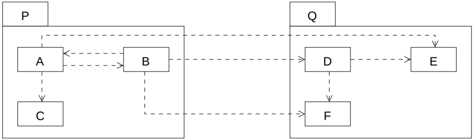
Basic rules

- *High coherence*
  Logical aggregation of associated parts of a system in a subsystem.
- *Low coupling*
  Few dependencies between the individual subsystems.

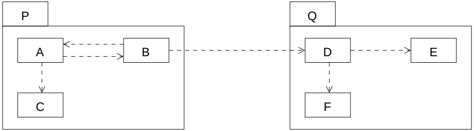
Advantage: Easy changeability and exchange of individual parts.

Example:

Subsystem with high coupling



Subsystem with low coupling



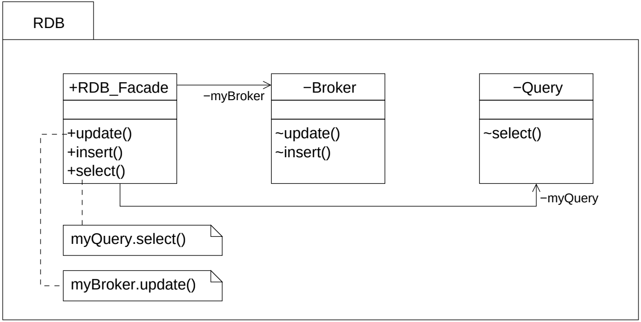
Note:

Whenever a part T is changed then all other parts having a dependency on T must be checked and adapted if necessary.

Facade classes

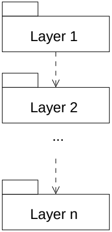
- Used to achieve low coupling.
- Group together the functionality of several classes of a subsystem and delegate calls to the appropriate objects.

Example:



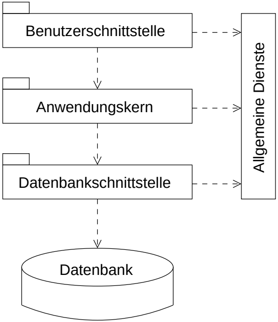
Layered Architectures

In many systems one can find layered architectures in which each layer provides services for the next layer(s) above it.
e.g. OSI model for network protocols, operation system layers, ...



- In "closed" architectures a layer may only access the directly subjacent layer; otherwise it is an "open" architecture.
- Systems in which different layers are distributed on different nodes are called client/server systems.
- A layer may again consist of different subsystems.

4.3.3 Three-Layer Architecture for information systems



Remark

In client/server architectures (e.g. web applications) we talk of

- a “thick client” if user interface and the core application is executed on the same node,
- a “thin client” if user interface and the core application are distributed on different nodes.

User Interface

- Handling of terminal events (mouse clicks, keystrokes, ...)
- Input/output of data
- Dialog control

Core application

- Responsible for the business logic (the essential responsibilities of the problem area)
- Arising from the object design

DB interface

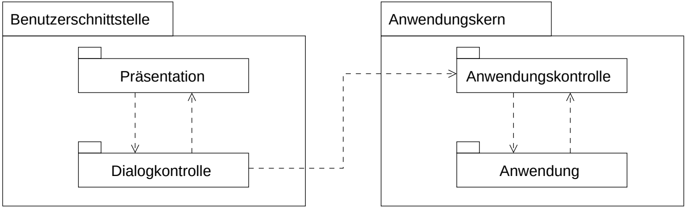
Stores and provides access to persistent data of the application.

Services

e.g. communication services, file management, libraries (APIs, GUI, DB, math. functions, ...)

Control objects

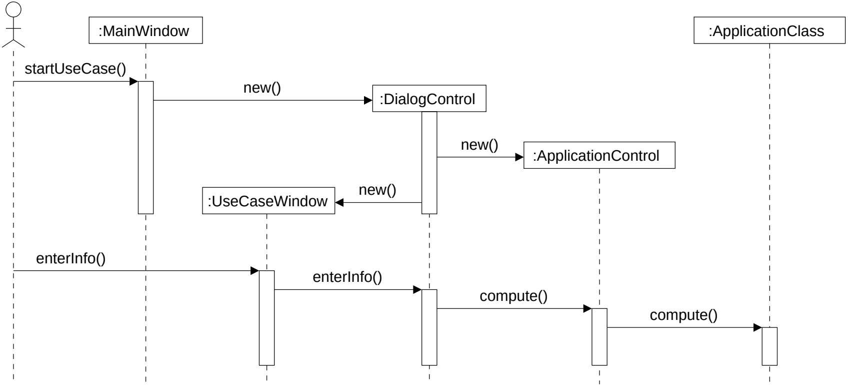
Often, specific objects are used for controlling user dialogs (e.g. management of different frames) or for controlling the responsibilities of the core application (e.g. one control object for each use case).



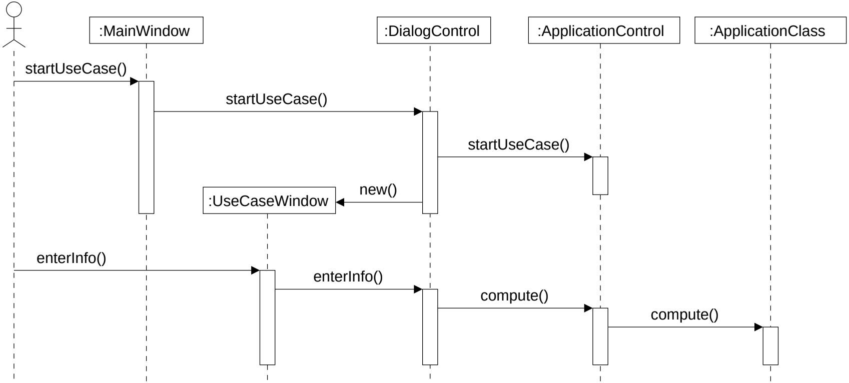
Remark

- Control objects often have an interesting behaviour which can be specified with a state diagram (e.g. ATM).
- The use of control objects supports low coupling.

Typical Interaction Pattern with Control Objects



Variant: Typical Interaction Pattern with Control Objects



4.3.4 Communication between User Interface and Core Application

Visibility Rule

The core application does *NOT* know the user interface
("Model/View Separation")!

Advantage

(Ex-)Changing the user interface does not affect the code of the core application.

Note: GUIs are often changed!

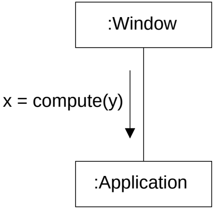
Problem

How to display in the GUI the information provided by the computation of the core application?

Possible Solutions

- Information “to be shown”can (sometimes) be passed as return value of operations.

  Example:

  Note:
  This approach does not support that the core application wants to trigger a change of the data displayed in the GUI (e.g. weather station issues a storm warning).

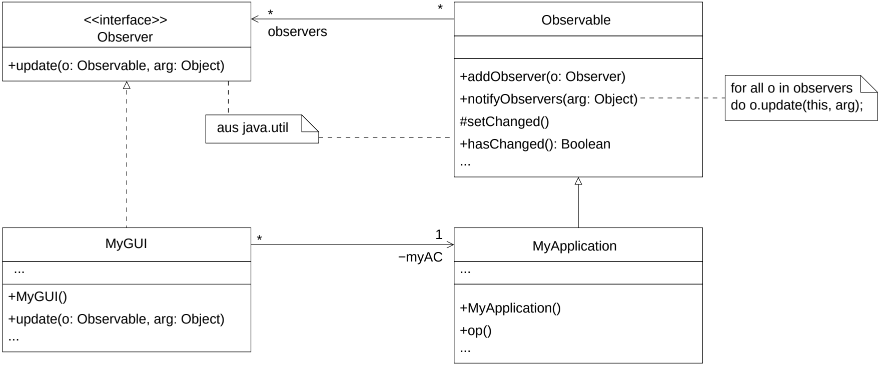
- Indirect communication: Event-Manager or Observer

Indirect Communication by Using Observers

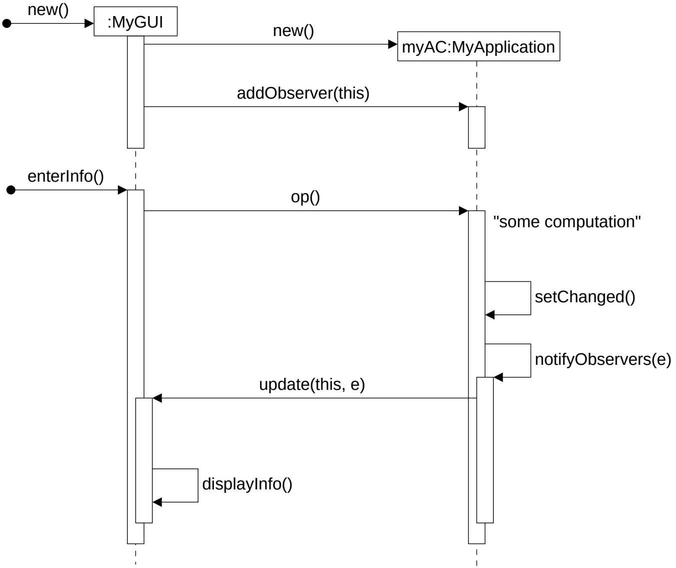
Idea

- GUI objects register themselves at the core application as observer (*addObserver*).
- If the core application wants to issue an event, it notifies all its observers (*notifyObservers*) which react accordingly (*update*).
- Each concrete observer implements the interface *Observer*.
- The core application only knows (at programming time) the observer interface. Concrete observer are dynamically linked at operating time.

Model of the Java-Realization of Observers



Typical Interaction between an Observer and an Observable



Summary of Section 4.3

- A fundamental task of system design is the definition of the system's architecture (software architecture).
- The system architecture describes the overall structure of the software system, by defining subsystems (components) and relationships between the subsystems.
- Important basic rules are high cohesion and low coupling.
- Layered architectures are very common.
- The Three-Layer-Architecture of operational information systems consist of the layers "user interface", "core application" und "database interface".
- The visibility rule requires that the application core does not know the user interface. (crucial for exchangeability of the GUI!)
- The visibility rule can be implemented, e.g., by indirect communication using observers.

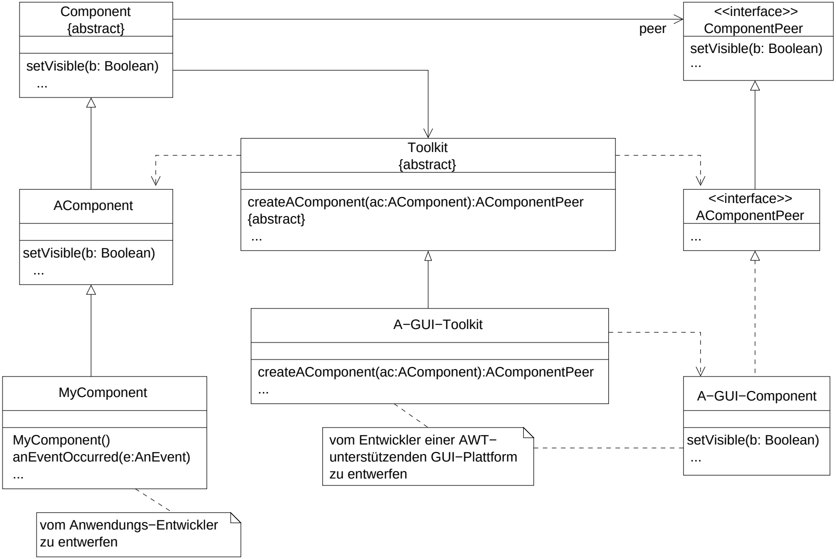
4.4 Design of Graphical User Interfaces

- GUIs ("graphical user interface") are driven by events:
  The user triggers an event (with the mouse, keyboard, ...) which is received by the GUI, with an immediate reaction of the GUI.
- *GUI toolkits* often support the easy creation of GUIs.
- A GUI toolkit provides ready-made widgets for the interaction with the user (e.g. window, button, checkbox, ...).
- Custom-made GUI widgets can be defined by specialization of given classes (reuse).
- Abstract toolkits allow to write *platform independent* code for GUIs.
  Important characteristic for Java programs:
  - Swing/AWT ("abstract window toolkit"),
  - SWT ("standard widget toolkit").

AWT (Abstract Window Toolkit) und Swing

- AWT and Swing provide a class library for the implementation of GUIs for Java programs
  (Packages `java.awt, java.awt.event, javax.swing`)
- *Basic concept:* platform-independent creation of GUIs
- *History:*
  - AWT 1.0
  - AWT 1.1 (new event handling with “listeners”)
  - Swing (complements AWT and replaces most of the AWT components with “lightweight”components)

1. Basic concepts of AWT



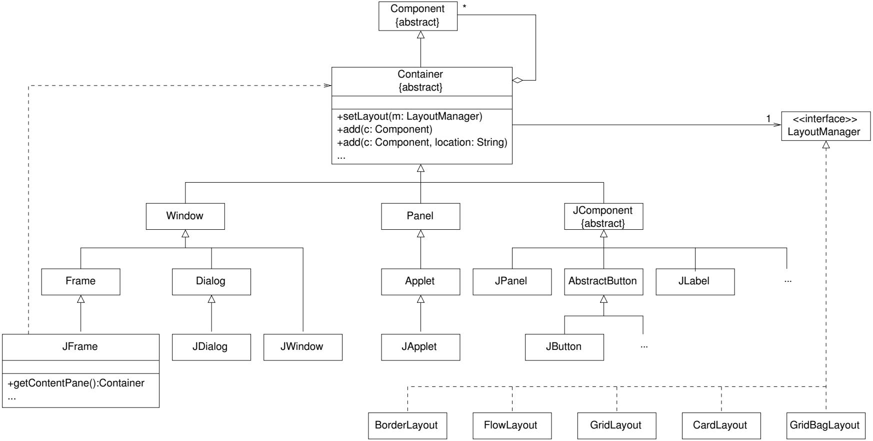
vom Entwickler einer AWT−unterstützenden GUI−Plattform zu entwerfen

vom Anwendungs−Entwickler zu entwerfen

Basic concepts of AWT (Summary)

- A toolkit translates AWT components to the corresponding GUI components of a specific platform ("native components").
- The platform-specific GUI components have to implement a corresponding peer interface of AWT. The peer interface describes the requirements of the GUI component
- If a specific GUI platform supports AWT, then a corresponding GUI toolkit must be implemented. The abstract class Toolkit (of AWT) describes the requirements on the GUI toolkit.
- The application developer must design the (plattform-independent) GUI components of the application (mostly Swing components).

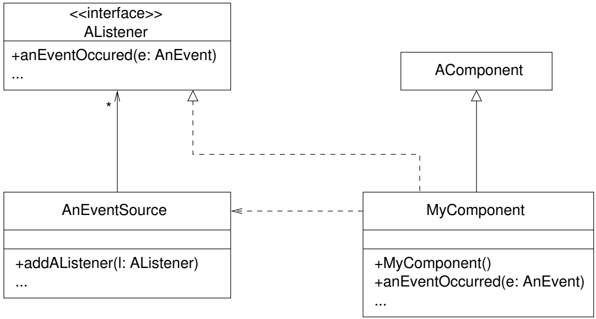
2. Hierarchy of Swing/AWT components



Component Hierarchy (Summary)

- All components starting with “J”(and some others) belong to Swing.
- Swing destinguishes between:
  - Heavyweight widgets (JFrame, JDialog, JWindow, JApplet)
  - Lightweight widgets (all specializations of JComponent)
- Heavyweight widgets are translated (like AWT widgets) to native widgets of a concrete GUI platform.
- Heavyweight widgets have a container (access with “getContentPane”) in which the lightweight widgets are painted.
- Each container possesses a layout manager.
- New components/widgets are added (with “add”) to the container (according to the configured layout manager).

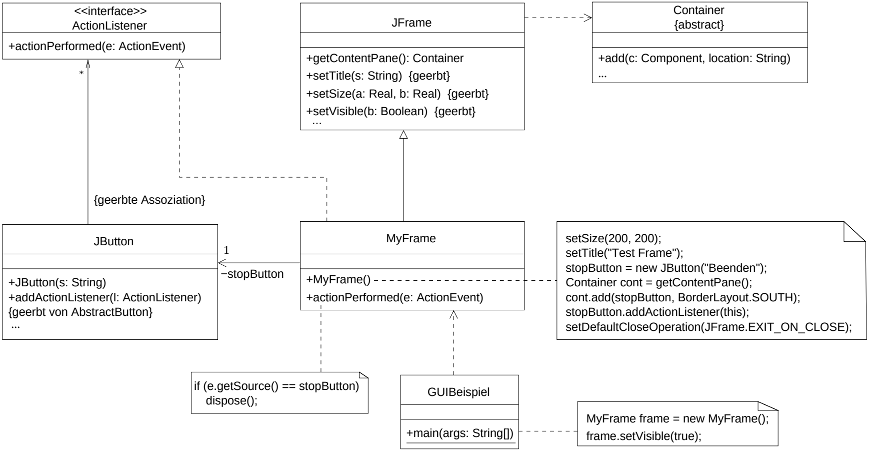
3. Event Handling in AWT

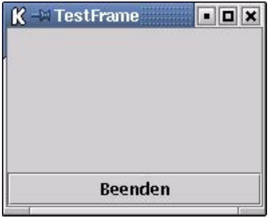

<<interface>>
AListener
+anEventOccured(e: AnEvent)
...
AComponent
*
AnEventSource
+addAListener(l: AListener)
...
MyComponent
+MyComponent()
+anEventOccurred(e: AnEvent)
...

Event Handling (Summary)

- In AWT/Swing we distinguish between several event classes: KeyEvent, MouseEvent, ActionEvent, WindowEvent, ...
- When a component is interested in the events of a specific type, then it must:
  1. register as “listener” (addAListener) in the component in which such an event can occur (AnEventSource),
  2. implement the operation (anEventOccured) of the corresponding listener interface (AListener), which is called on the occurrence of such an event.
- listener interfaces are, e.g., KeyListener, MouseListener, ActionListener, WindowListener.
- Operations of the listener interfaces are, e.g.,
  actionPerformed (of ActionListener),
  windowClosing (of WindowListener).

4. Modelling GUIs: A simple example

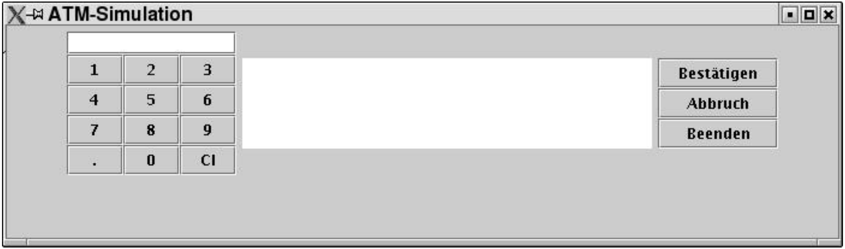

<<interface>>
ActionListener
+actionPerformed(e: ActionEvent)
JFrame
+getContentPane(): Container
+setTitle(s: String) {geerbt}
+setSize(a: Real, b: Real) {geerbt}
+setVisible(b: Boolean) {geerbt}
...
Container
{abstract}
+add(c: Component, location: String)
...
*
{geerbte Assoziation}
JButton
+JButton(s: String)
+addActionListener(l: ActionListener)
{geerbt von AbstractButton}
...
1
−stopButton
MyFrame
+MyFrame()
+actionPerformed(e: ActionEvent)
setSize(200, 200);
setTitle("Test Frame");
stopButton = new JButton("Beenden");
Container cont = getContentPane();
cont.add(stopButton, BorderLayout.SOUTH);
stopButton.addActionListener(this);
setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
if (e.getSource() == stopButton)
dispose();
GUIBeispiel
+main(args: String[])
MyFrame frame = new MyFrame();
frame.setVisible(true);



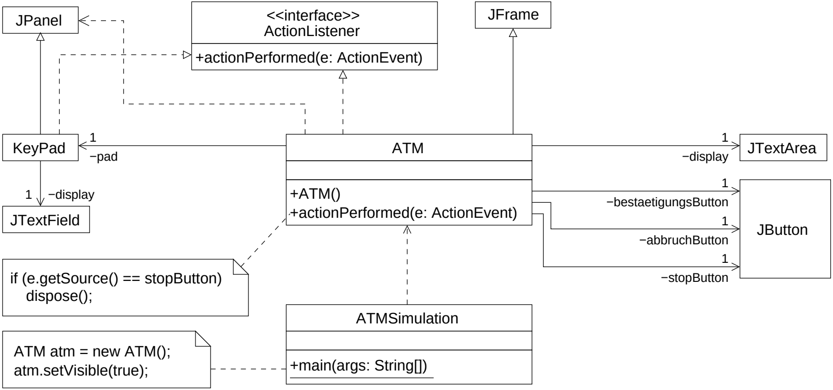
```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

class GUIBeispiel {

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

```java
class MyFrame extends JFrame implements ActionListener {

  private JButton stopButton;

  public MyFrame() {
    setSize(200,200);
    setTitle("TestFrame");
    stopButton = new JButton("Beenden");

    Container cont = getContentPane();
    cont.add(stopButton, BorderLayout.SOUTH);

    stopButton.addActionListener(this);

   //Damit mit dem Schliessen des Fensters auch das Programm beendet wird
    setDefaultCloseOperation(JFrame.DISPOSE_ON_CLOSE);
  }
  public void actionPerformed(ActionEvent e) {
    if (e.getSource() == stopButton)
      dispose();
  }
}
```

5. User interface for the ATM Simulation



GUI Model for the ATM Simulation



```
if (e.getSource() == stopButton)
   dispose();
```

```
ATM atm = new ATM();
atm.setVisible(true);
```

```
// Vorlaeufige Version der ATM-Simulation zur Erstellung des GUI-Prototypen

import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

class ATMSimulation {
   public static void main(String[] args) {
      ATM atm = new ATM();
      atm.setVisible(true);
    }
}

class ATM extends JFrame implements ActionListener {

   //Referenzattribute fuer GUI
   private KeyPad pad;
   private JTextArea display;
   private JButton bestaetigungsButton;
   private JButton abbruchButton;
   private JButton stopButton;
public ATM() {
   //Konstruktion der GUI
   setSize(700, 200);
   setTitle("ATM-Simulation" );
   pad = new KeyPad();
   display = new JTextArea(5, 31);
   bestaetigungsButton = new JButton("Bestaetigen");
   abbruchButton = new JButton("Abbruch");
   stopButton = new JButton("Beenden");

   JPanel buttonPanel = new JPanel();
   buttonPanel.setLayout(new GridLayout(3, 1));
   buttonPanel.add(bestaetigungsButton);
   buttonPanel.add(abbruchButton);
   buttonPanel.add(stopButton);

   Container cont = getContentPane();
   cont.setLayout(new FlowLayout());
   cont.add(pad);
   cont.add(display);
   cont.add(buttonPanel);

   //ATM als ActionListener zu allen Buttons hinzufuegen
   bestaetigungsButton.addActionListener(this);
   abbruchButton.addActionListener(this);
   stopButton.addActionListener(this);
   setDefaultCloseOperation(JFrame.DISPOSE_ON_CLOSE);
}
public void actionPerformed(ActionEvent e) {
   if (e.getSource() == stopButton) dispose();
}}
```

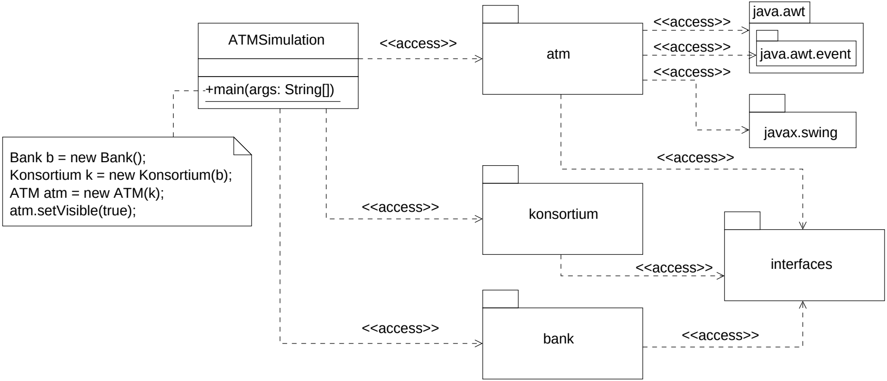
```
//angelehnt an "C. Horstmann: Computing Concepts with Java2 Essentials" (S. 601)

class KeyPad extends JPanel implements ActionListener {

   private JTextField display;

   public KeyPad() {
     setLayout(new BorderLayout());
     display = new JTextField();
     add(display, BorderLayout.NORTH);

     JPanel buttonPanel = new JPanel();
     buttonPanel.setLayout(new GridLayout(4, 3));
     addButton(buttonPanel, "1");
     addButton(buttonPanel, "2");
     addButton(buttonPanel, "3");
     addButton(buttonPanel, "4");
     addButton(buttonPanel, "5");
     addButton(buttonPanel, "6");
     addButton(buttonPanel, "7");
     addButton(buttonPanel, "8");
     addButton(buttonPanel, "9");
     addButton(buttonPanel, ".");
     addButton(buttonPanel, "0");
     addButton(buttonPanel, "Cl");
     add(buttonPanel, BorderLayout.CENTER);
   }
   private void addButton(JPanel buttonPanel, String label) {
     JButton button = new JButton(label);
     buttonPanel.add(button);
     button.addActionListener(this);
   }
   public void actionPerformed (ActionEvent e) {
     JButton source = (JButton)e.getSource();
     String label = source.getText();
     if (label.equals("Cl"))
       clear();
     else
       display.setText(display.getText()+label);
   }
   public double getValue() {
     return Double.parseDouble(display.getText());
   }
   public void clear() {
     display.setText("");
   }
}
```

Summary of Section 4.4

- Programming of graphical user interfaces can often be simplified by using GUI toolkits.
- Application specific GUI elements can be defined by specialization of given GUI classes.
- Swing/AWT provides a class library for the platform independent implementation of GUIs for Java programs.
- Integral tasks during the realization of a GUI is
  - the static construction of the GUI comopnents
  - the programming of the event handling, by implementation of the corresponding listener interfaces.

4.5 Implementation of the ATM simulation

1. System architecture: Representation with packages

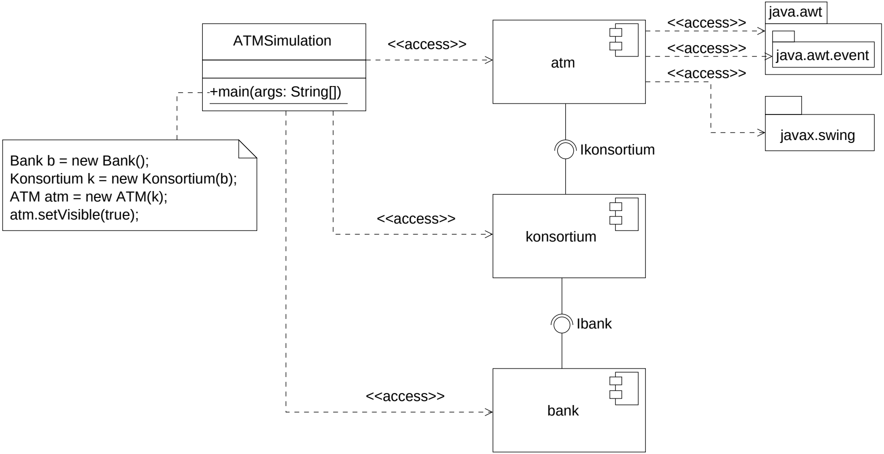


```
Bank b = new Bank();
Konsortium k = new Konsortium(b);
ATM atm = new ATM(k);
atm.setVisible(true);
```

Remark:

“atm” represents the user interface, “konsortium” und “bank” constitute the core application.

System architecture: Representation with components

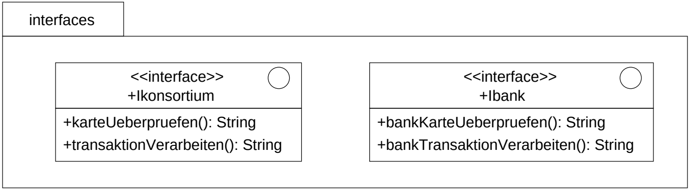


```
Bank b = new Bank();
Konsortium k = new Konsortium(b);
ATM atm = new ATM(k);
atm.setVisible(true);
```

Simplification (in the following):

- When the credit card is read, no card number or BLZ is entered, but just the PIN stored on the card.
- Consortium and bank are realized by a simplified implementation. The formal parameters of the methods are omitted.
- The ATM does not store any transactions (contrary to the design).

2. The package "interfaces"


interfaces
<<interface>>
+Ikonsortium
+karteUeberpruefen(): String
+transaktionVerarbeiten(): String
<<interface>>
+Ibank
+bankKarteUeberpruefen(): String
+bankTransaktionVerarbeiten(): String

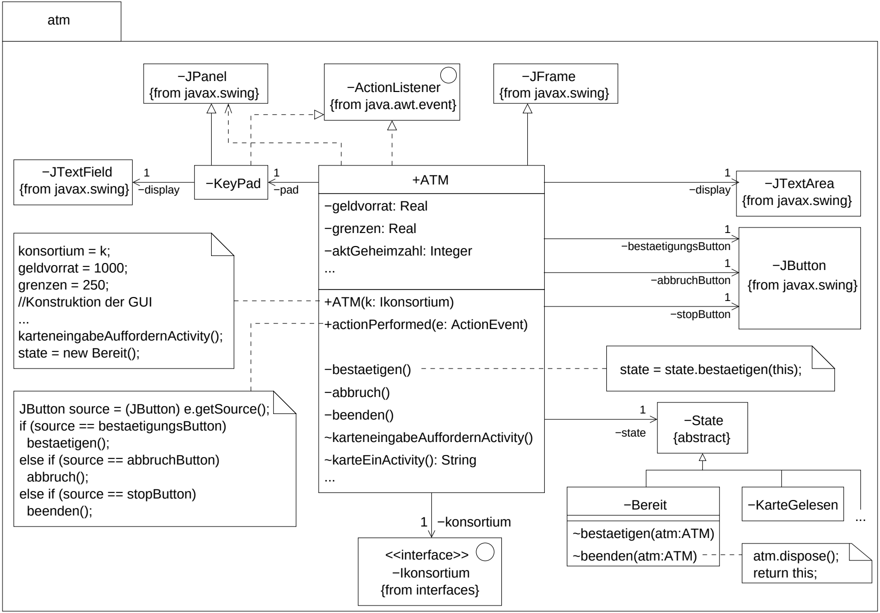
3. The subsystem “atm”

Based on

1. GUI of the ATM simulation (cf. Section 4.4)
2. Pseudo code of the state-independent operations of the ATM (cf. Section 4.1.6)
3. Realization of the state diagram of the event-based ATM simulation (cf. Section 4.2.3).

Note:

When integrating 1. and 3. the event handling mechanism of AWT must be taken into consideration by specifying a proper implementation of “actionPerformed” which calls the correct operation of the class ATM once a button is pushed.



atm

−JPanel
{from javax.swing}

−ActionListener
{from java.awt.event}

−JFrame
{from javax.swing}

−JTextField
{from javax.swing}

1
−display

−KeyPad

1
−pad

+ATM

−geldvorrat: Real
−grenzen: Real
−aktGeheimzahl: Integer
...

+ATM(k: Ikonsortium)
+actionPerformed(e: ActionEvent)

−bestaetigen()
−abbruch()
−beenden()
~karteneingabeAuffordernActivity()
~karteEinActivity(): String
...

1
−display

−JTextArea
{from javax.swing}

1
−bestaetigungsButton

1
−abbruchButton

1
−stopButton

−JButton
{from javax.swing}

```
konsortium = k;
geldvorrat = 1000;
grenzen = 250;
//Konstruktion der GUI
...
karteneingabeAuffordernActivity();
state = new Bereit();
```

```
JButton source = (JButton) e.getSource();
if (source == bestaetigungsButton)
  bestaetigen();
else if (source == abbruchButton)
  abbruch();
else if (source == stopButton)
  beenden();
```

```
state = state.bestaetigen(this);
```

1
−state

−State
{abstract}

−Bereit

~bestaetigen(atm:ATM)
~beenden(atm:ATM)

−KarteGelesen

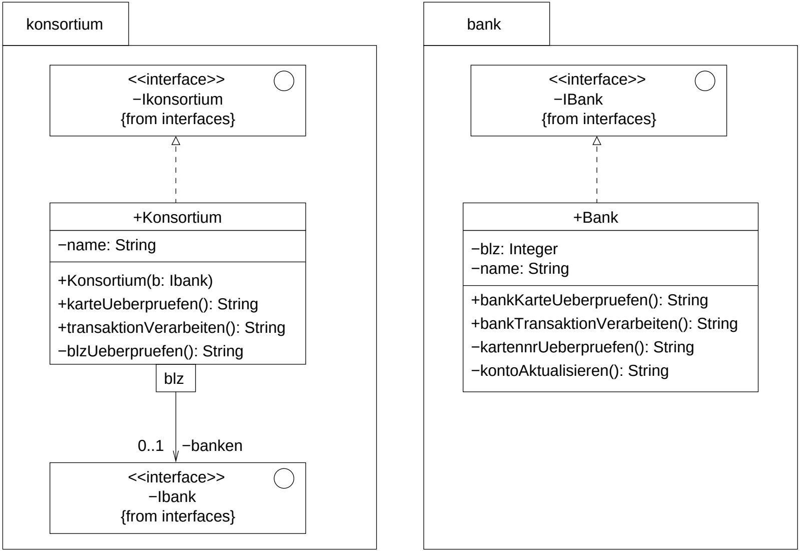
...

```
atm.dispose();
return this;
```

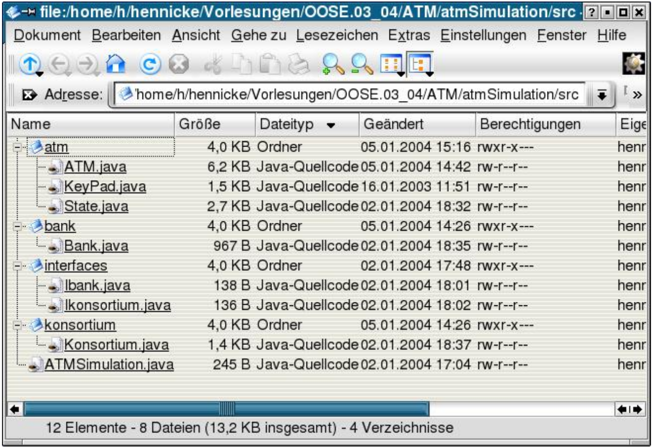
1 −konsortium

<<interface>>
−Ikonsortium
{from interfaces}

4. The subsystems “konsortium”and “bank”



5. Java-Implementation of the ATM-Simulation



home/h/hennicke/Vorlesungen/OOSE.03_04/ATM/atmSimulation/src

Name	Größe	Dateityp	Geändert	Berechtigungen	Eige
atm	4,0 KB	Ordner	05.01.2004 15:16	rwxr-x---	henr
ATM.java	6,2 KB	Java-Quellcode	05.01.2004 14:42	rw-r--r--	henr
KeyPad.java	1,5 KB	Java-Quellcode	16.01.2003 11:51	rw-r--r--	henr
State.java	2,7 KB	Java-Quellcode	02.01.2004 18:32	rw-r--r--	henr
bank	4,0 KB	Ordner	05.01.2004 14:26	rwxr-x---	henr
Bank.java	967 B	Java-Quellcode	02.01.2004 18:35	rw-r--r--	henr
interfaces	4,0 KB	Ordner	02.01.2004 17:48	rwxr-x---	henr
Ibank.java	138 B	Java-Quellcode	02.01.2004 18:01	rw-r--r--	henr
Ikonsortium.java	136 B	Java-Quellcode	02.01.2004 18:02	rw-r--r--	henr
konsortium	4,0 KB	Ordner	05.01.2004 14:26	rwxr-x---	henr
Konsortium.java	1,4 KB	Java-Quellcode	02.01.2004 18:37	rw-r--r--	henr
ATMSimulation.java	245 B	Java-Quellcode	02.01.2004 17:04	rw-r--r--	henr

12 Elemente - 8 Dateien (13,2 KB insgesamt) - 4 Verzeichnisse

Note:

The directory structure conforms to the system architecture.

Class ATM Simulation

```
import atm.*;
import konsortium.*;
import bank.*;

class ATMSimulation {
    public static void main(String[] args) {
       Bank b = new Bank();
       Konsortium k = new Konsortium(b);
       ATM atm = new ATM(k);
       atm.setVisible(true);
    }
}
```

Interface Ikonsortium

```
package interfaces;
public interface Ikonsortium {
   public String karteUeberpruefen();
   public String transaktionVerarbeiten();
}
```

Interface Ibank

```
package interfaces;
public interface Ibank {
   public String bankKarteUeberpruefen();
   public String bankTransaktionVerarbeiten();
}
```

Class ATM

```
package atm;

import interfaces.Ikonsortium;
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

/* Implementierung der Klasse ATM mit integrierter GUI */

public class ATM extends JFrame implements ActionListener {

   //ATM-Attribute
   private double geldvorrat;
   private double grenzen;
   private int aktGeheimzahl;
   private int aktKartennr; //wird in der Simulation nicht verwendet
   private int aktBLZ;      //wird in der Simulation nicht verwendet
   private int aktKontonr;  //wird in der Simulation nicht verwendet

   //Referenzattribut fuer Konsortium
   private Ikonsortium konsortium;

   //Referenzattribute fuer GUI
   private KeyPad pad;
   private JTextArea display;
   private JButton bestaetigungsButton;
   private JButton abbruchButton;
   private JButton stopButton;

   //Referenzattribut fuer Zustandsobjekt
   private State state;
```

```
public ATM(Ikonsortium k) {

   //Konsortium initialisieren
   konsortium = k;

   //ATM-Attribute initialisieren
   geldvorrat = 1000;
   grenzen = 250;

   //Konstruktion der GUI
   setSize(700, 200);
   setTitle("ATM-Simulation");

   pad = new KeyPad();
   display = new JTextArea(5, 31);
   bestaetigungsButton = new JButton("Bestaetigung");
   abbruchButton = new JButton("Abbruch");
   stopButton = new JButton("Beenden");

   JPanel buttonPanel = new JPanel();
   buttonPanel.setLayout(new GridLayout(3, 1));
   buttonPanel.add(bestaetigungsButton);
   buttonPanel.add(abbruchButton);
   buttonPanel.add(stopButton);

   Container cont = getContentPane();
   cont.setLayout(new FlowLayout());
   cont.add(pad);
   cont.add(display);
   cont.add(buttonPanel);
```

```
      //ATM als ActionListener zu allen Buttons hinzufuegen
      bestaetigungsButton.addActionListener(this);
      abbruchButton.addActionListener(this);
      stopButton.addActionListener(this);
      //ordnungsgemaesse Beendigung bei Schliessen des Windows
      setDefaultCloseOperation(JFrame.DISPOSE_ON_CLOSE);

      //zur Karteneingabe auffordern
      karteneingabeAuffordernActivity();
      //Zustandsobjekt initialisieren
      state = new Bereit();
   }                                         //Ende Konstruktor
   public void actionPerformed(ActionEvent e) {
      JButton source = (JButton) e.getSource();
      if (source == bestaetigungsButton)
         bestaetigen();
      else if (source == abbruchButton)
         abbruch();
      else if (source == stopButton)
         beenden();
   }
   private void bestaetigen() {
      //Delegation an das Zustandsobjekt
      state = state.bestaetigen(this);
   }
   private void abbruch() {
      //Delegation an das Zustandsobjekt
      state = state.abbruch(this);
   }
   private void beenden() {
      //Delegation an das Zustandsobjekt
      state = state.beenden(this);
   }
```

```
//Operationen fuer die Aktivitaetszustaende des Zustandsdiagramms

void karteneingabeAuffordernActivity() {
   display.setText("<<Hauptbildschirm: Aufforderung zur Karteneingabe>>");
   display.append("\nAuf Karte gespeicherte Geheimzahl? (Eingabe bestaetigen)");
}

String karteEinActivity() {
   String check = karteLesen();
   if (check.equals("lesbar")) {
      display.setText("Karte lesbar!");
      display.append("\nGeheimzahl? (Eingabe bestaetigen)");
      return "Karte lesbar";
   }
   else if (check.equals("nicht lesbar")) {
      display.setText("Karte nicht lesbar!");
      abschlussActivity();
      return "Karte nicht lesbar";
   }
   else return "Error";
}
```

```
String geheimzahlEinActivity() {
   int typedGeheimzahl = (int)pad.getValue();
   pad.clear();
   String check = geheimzahlUeberpruefen(typedGeheimzahl);
   if (check.equals("Geheimzahl ok")) {
      check = konsortium.karteUeberpruefen();
      if (check.equals("Karte ok")) {
         display.setText("Karte ok!");
         display.append("\nTransaktionsform? (Bestaetigung = Abhebung)");
         return "Karte ok";
      }
      else if (check.equals("falsche BLZ")) {
         display.setText("Karte nicht ok: falsche BLZ!");
         abschlussActivity();
         return "Karte nicht ok";
      }
      else if (check.equals("Karte gesperrt")) {
         display.setText("Karte nicht ok: Karte gesperrt!");
         abschlussActivity();
         return "Karte nicht ok";
      }
      else return "Error";
   }
   else if (check.equals("falsche Geheimzahl")) {
      display.setText("falsche Geheimzahl!");
      display.append("\nGeheimzahl? (Eingabe bestaetigen)");
      return "Geheimzahl falsch";
   }
   else return "Error";
}
```

```
void abhebungActivity() {
   display.setText("Betrag? (Eingabe bestaetigen)");
}

String betragEinActivity() {
   double betrag = pad.getValue();
   pad.clear();
   String check = grenzenUeberpruefen(betrag);
   if (check.equals("Grenzen ok")) {
      check = konsortium.transaktionVerarbeiten();
      if (check.equals("Transaktion erfolgreich")) {
         geldvorrat = geldvorrat - betrag;
         display.setText("Transaktion erfolgreich!");
         display.append("\nGeld wird ausgegeben");
         display.append("\nGeld entnehmen? (Bestaetigung)");
         return "Transaktion erfolgreich";
      }
      else if (check.equals("Transaktion gescheitert")) {
         display.setText("Transaktion gescheitert!");
         display.append("\nTransaktionsform? (Bestaetigung = Abhebung)");
         return "Transaktion gescheitert";
      }
      else return "Error";
   }
   else if (check.equals("Grenzen ueberschritten")) {
      display.setText("Grenzen ueberschritten!");
      display.append("\nBetrag? (Eingabe bestaetigen)");
      return "Grenzen ueberschritten";
   }
   else return "Error";
   }
   void geldEntnehmenActivity() {
      display.setText("Fortsetzung? (Bestaetigung = Nein)");
   }
   void abschlussActivity() {
      pad.clear();
      display.append("\nBeleg wird gedruckt");
      display.append("\nKarte wird ausgegeben");
      display.append("\nKarte und Beleg entnehmen? (Bestaetigung)");
   }

   //private Operationen fuer Subaktivitaeten

   private String karteLesen() {
      aktGeheimzahl = (int)pad.getValue();
      pad.clear();
      return "lesbar";
   }
   private String geheimzahlUeberpruefen(int tgz) {
      if (tgz == aktGeheimzahl) return "Geheimzahl ok";
      else return "falsche Geheimzahl";
   }
   private String grenzenUeberpruefen(double b) {
      if (b <= grenzen) return "Grenzen ok";
      else return "Grenzen ueberschritten";
   }
}
```

Class State with subclasses

```
/* Das folgende Programm realisiert das Zustandsdiagramm der ATM-Simulation
   durch Zustandsobjekte*/

package atm;
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

abstract class State {
   State bestaetigen(ATM atm) {
      return this;
   }
   State abbruch(ATM atm) {
      return this;
   }
   State beenden(ATM atm) {
      return this;
   }
}
class Bereit extends State {
   State bestaetigen(ATM atm) {
      String check = atm.karteEinActivity();
      if (check.equals("Karte lesbar"))
         return new KarteGelesen();
      else if (check.equals("Karte nicht lesbar"))
         return new BereitZurKartenentnahme();
      else return this;
   }
   State beenden(ATM atm) {
      atm.dispose();
      return this; }
}
```

```
class KarteGelesen extends State {

   State bestaetigen(ATM atm) {
      String check = atm.geheimzahlEinActivity();
      if (check.equals("Karte ok"))
         return new GeheimzahlUndKarteGeprueft();
      else if (check.equals("Karte nicht ok"))
         return new BereitZurKartenentnahme();
      else if (check.equals("Geheimzahl falsch"))
         return new KarteGelesen();  //oder einfacher: return this;
      else return this;
   }

   State abbruch(ATM atm) {
      atm.abschlussActivity();
      return new BereitZurKartenentnahme();
   }
}

class GeheimzahlUndKarteGeprueft extends State {

   State bestaetigen(ATM atm) {
      atm.abhebungActivity();
      return new TransaktionsformBestimmt();
   }

   State abbruch(ATM atm) {
      atm.abschlussActivity();
      return new BereitZurKartenentnahme();
   }
}
```

```
class TransaktionsformBestimmt extends State {

   State bestaetigen(ATM atm) {
      String check = atm.betragEinActivity();
      if (check.equals("Transaktion erfolgreich"))
         return new TransaktionDurchgefuehrt();
      else if (check.equals("Transaktion gescheitert"))
         return new GeheimzahlUndKarteGeprueft();
      else if (check.equals("Grenzen ueberschritten"))
         return new TransaktionsformBestimmt(); //oder einfacher: return this;
      else return this;
   }

   State abbruch(ATM atm) {
      atm.abschlussActivity();
      return new BereitZurKartenentnahme();
   }
}

class TransaktionDurchgefuehrt extends State {
   State bestaetigen(ATM atm) {
      atm.geldEntnehmenActivity();
      return new GeldEntnommen();
   }
}

class GeldEntnommen extends State {
   State bestaetigen(ATM atm) {
      atm.abschlussActivity();
      return new BereitZurKartenentnahme();
   }
}
```

```
class BereitZurKartenentnahme extends State {
   State bestaetigen(ATM atm) {
      atm.karteneingabeAuffordernActivity();
      return new Bereit();
   }
}
```

Class Konsortium

```
package konsortium;

//beide Interfaces werden benoetigt
import interfaces.*;
import java.util.*;

public class Konsortium implements Ikonsortium {

   private String name;
   private Map<Integer,Ibank> banken = new HashMap<Integer,Ibank>();

   //Bei der Konstruktion des Konsortiums wird genau eine Bank eingefuegt
   public Konsortium(Ibank b) {
      banken.put(new Integer(101), b);
   }
   public String karteUeberpruefen() {
      String check = blzUeberpruefen();
      if (check.equals("BLZ richtig")) {
         Ibank b = (Ibank)banken.get(new Integer(101));
         check = b.bankKarteUeberpruefen();
         if (check.equals("Karte ok"))
            return "Karte ok";
         else if (check.equals("Karte bei Bank gesperrt"))
            return "Karte gesperrt";
         else return "Error";
      }
      else if (check.equals("BLZ falsch"))
         return "falsche Bankleitzahl";
      else return "Error";
   }

   public String transaktionVerarbeiten() {
      Ibank b = (Ibank)banken.get(new Integer(101));
      String check = b.bankTransaktionVerarbeiten();
      if (check.equals("Banktransaktion erfolgreich"))
         return "Transaktion erfolgreich";
      else if (check.equals("Banktransaktion gescheitert"))
         return "Transaktion gescheitert";
      else return "Error";
   }

   //Dummy-Implementierung
   private String blzUeberpruefen() {
      return "BLZ richtig";
//zum Testen      return "BLZ falsch";
   }
}
```

Class Bank

```
package bank;
import interfaces.Ibank;

public class Bank implements Ibank {
   private int blz;
   private String name;

   public String bankKarteUeberpruefen() {
      String check = kartennrUeberpruefen();
      if (check.equals("gueltig")) return "Karte ok";
      else if (check.equals("gesperrt")) return "Karte bei Bank gesperrt";
      else return "Error";
   }
   public String bankTransaktionVerarbeiten() {
      String check = kontoAktualisieren();
      if (check.equals("erfolgreich")) return "Banktransaktion erfolgreich";
      else if (check.equals("gescheitert")) return "Banktransaktion gescheitert";
      else return "Error";
   }
/* Die folgenden Operationen haben lediglich Dummy-Implementierungen
   zum Testen der ATM Simulation */
   private String kartennrUeberpruefen() {
      return "gueltig";
//      return "gesperrt";
   }
   private String kontoAktualisieren() {
      return "erfolgreich";
//      return "gescheitert";
   }
}
```

Summary of Section 4.5

- The system architecture of the ATM simulation is based on three subsystems (atm, konsortium and bank), which are connected via interfaces.
- The classes Konsortium and Bank are implemented in a simplified manner.
- The subsystem “atm” comprises the GUI and the realization of the state diagram of the ATM simulation.

4.6 Connection to a Database

Aim

Storing of *persistent* objects, i.e. objects of the core application which are needed permanently (e.g. clients, accounts, flights, books, ...).

Options

- *Object-oriented database systems:*
  Support associations, inheritance and operations. The ODMG standard comprises: ODL (object definition language) for schema declarations, OQL (object query language) and support for C++, Java and Smalltalk.
- *Relational database systems:*
  Do *not* support associations and inheritance. Thus an explicit interface between core application and relational database system is needed.
- *Object-relational database systems:*
  Encapsulates a relational database system with a object-oriented wrapper.

4.6.1 Mapping of an Object Model to Tables

Prerequisite

For each entitiy class A a primary key aKey is introduced.

Mapping of Classes

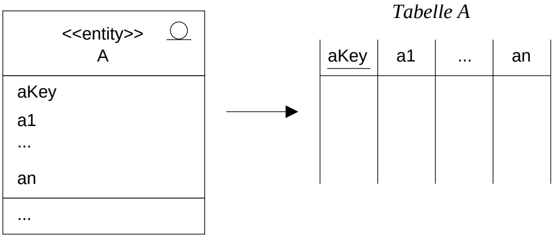


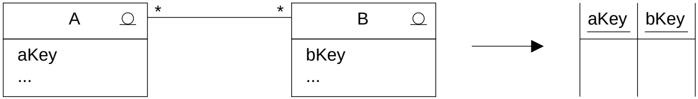
Tabelle A

aKey	a1	...	an

Mapping of Associations

Multiplicity * - *

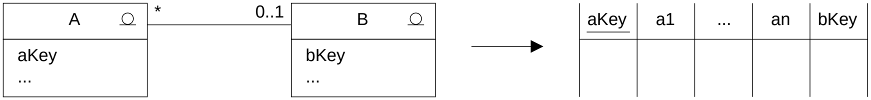
Use a separate table with primary keys of A and B.



aKey	bKey

Multiplicity * - 0..1

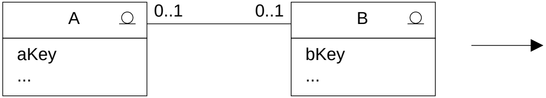
Include primary key of B as foreign key in the table for A.



aKey	a1	...	an	bKey

Multiplicity 0..1 - 0..1

Include primary key of B in the table for A or include primary key of A in the table for B.

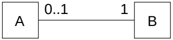


aKey	a1	...	an	bKey

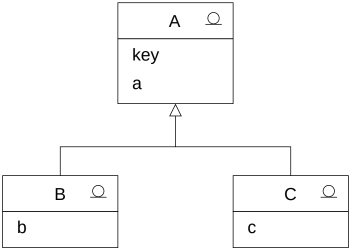
oder

bKey	b1	...	bm	aKey

Remark

In  the primary key of B needs to be included as foreign key in the table for A.

Mapping of Inheritance



Variant I: One table for each class

Tabelle A

<u>key</u>	a

Tabelle B

<u>key</u>	b

Tabelle C

<u>key</u>	c

Disadvantage:

For queries or request changes concerning objects of a subclass, it may be necessary to access entries in multiple tables (e.g. when selecting all attributes of all B-objects).

Variant II: Tables of subclasses contain inherited attributes

Tabelle A

key	a

Tabelle B

key	a	b

Tabelle C

key	a	c

If A is abstract, then one table for each subclass suffices (table A is omitted).

Disadvantage:
When changing the superclass, it is necessary to change the tables of all subclasses.

Variant III: One table for the superclass and all subclasses

Tabelle ABC

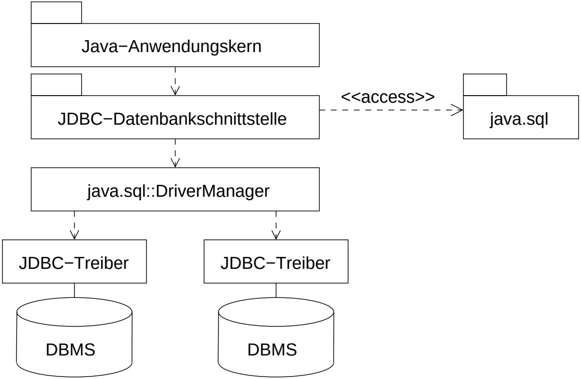
key	a	b	c	Typ

Disadvantage:
"Null" values must be inserted whenever the attribute is not relevant for an object.

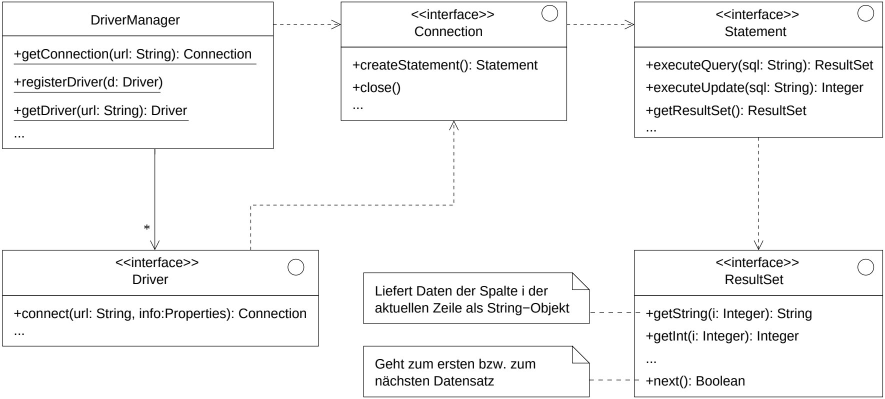
4.6.2 Database Connectivity with JDBC

- JDBC (Java Database Connectivity) provides an SQL interface for Java programs.
- JDBC is independent of a concrete database system. The access to a concrete database system is realized by using suitable drivers.

Layers of a JDBC Application



JDBC – In a Nutshell


DriverManager
+getConnection(url: String): Connection
+registerDriver(d: Driver)
+getDriver(url: String): Driver
...
<<interface>>
Connection
+createStatement(): Statement
+close()
...
<<interface>>
Statement
+executeQuery(sql: String): ResultSet
+executeUpdate(sql: String): Integer
+getResultSet(): ResultSet
...
*
<<interface>>
Driver
+connect(url: String, info:Properties): Connection
...
<<interface>>
ResultSet
+getString(i: Integer): String
+getInt(i: Integer): Integer
...
+next(): Boolean

Liefert Daten der Spalte i der aktuellen Zeile als String−Objekt

Geht zum ersten bzw. zum nächsten Datensatz

- "Driver", "Connection", "Statement" und "ResultSet" are interfaces which are implemented by the drivers.
- The DriverManager registers drivers for particular DB systems. The call to the static operation "getConnection" establishes a connection (via appropriate drivers for the given URL) to the DB system.
- A `Connection` object can create a `Statement` object (operation "createStatement").
- A `Statement` object can issue a query to the database (operation "executeQuery").
- The summary table for the query is stored in a `ResultSet` object.
- The table of a `ResultSet` object can be accessed row by row via the operation "next".
- The fields of a row can be accessed via "get" operations, of the corresponding column type.

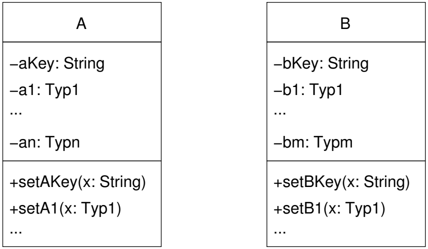
Example:

```
try {
  Class.forName("imaginary.sql.iMsqlDriver");// Treiber auch von außen setzbar
  String url = "jdbc:msql://localhost/myDB";
  Connection con = DriverManager.getConnection(url);
  Statement stmt = con.createStatement();
  ResultSet rs   = stmt.executeQuery("SELECT * FROM test");

  while (rs.next()) {
    // Zugriff auf die Spalte mit der Nummer y und dem Typ XXX
    // in der aktuellen Zeile
    XXX v = rs.getXXX(y);
  }
  con.close();
}
catch (Exception e) {
  e.printStackTrace();
}
```

4.6.3 Materialization

Given:

A class of the core application:



Tables of a relational database:

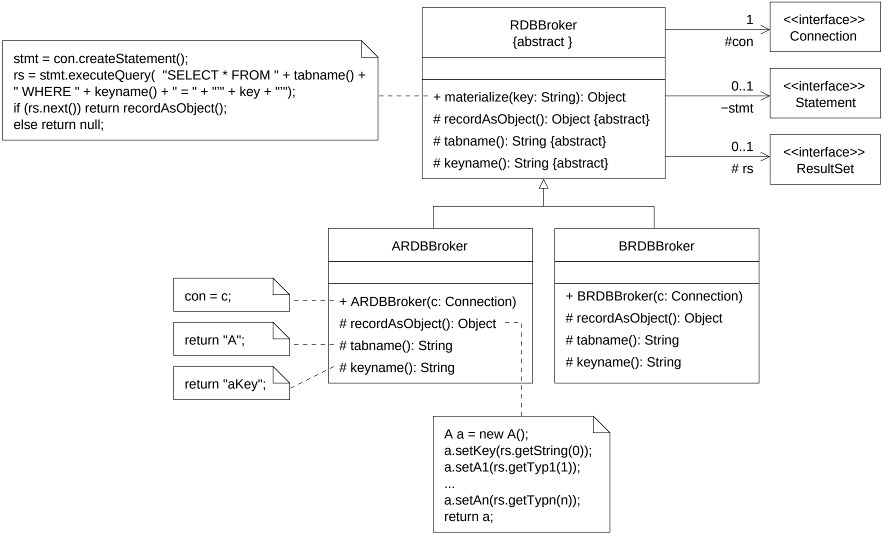
Tabelle A

aKey	a1	...	an

Tabelle B

bKey	b1	...	bm

Materialization with JDBC (retrieving object?)


RDBBroker
{abstract }
+ materialize(key: String): Object
# recordAsObject(): Object {abstract}
# tabname(): String {abstract}
# keyname(): String {abstract}
1
#con
<<interface>>
Connection
0..1
−stmt
<<interface>>
Statement
0..1
# rs
<<interface>>
ResultSet
ARDBBroker
+ ARDBBroker(c: Connection)
# recordAsObject(): Object
# tabname(): String
# keyname(): String
BRDBBroker
+ BRDBBroker(c: Connection)
# recordAsObject(): Object
# tabname(): String
# keyname(): String

```
stmt = con.createStatement();
rs = stmt.executeQuery(  "SELECT * FROM " + tabname() +
" WHERE " + keyname() + " = " + "’" + key + "’");
if (rs.next()) return recordAsObject();
else return null;
```

```
con = c;
```

```
return "A";
```

```
return "aKey";
```

```
A a = new A();
a.setKey(rs.getString(0));
a.setA1(rs.getTyp1(1));
...
a.setAn(rs.getTypn(n));
return a;
```

Example:

```
try {
  String url = "jdbc:RDB-Typ//Rechner:Port/Datenbank";
  Connection con = DriverManager.getConnection(url);
  ARDBBroker ardb = new ARDBBroker(con);
  BRDBBroker brdb = new BRDBBroker(con);
  A a = (A)ardb.materialize("xyz");
  B b = (B)brdb.materialize("uvw");
}
catch (Exception e) {
  e.printStackTrace();
}
```

Remarks

- A *cache* is often used to improve efficiency.
- Materializing objects usually loads only the needed objects *(on-demand materialization)*.
- Persistence frameworks comprise further mechanisms, e.g. for dematerialization and for transaction control.

Summary of Section 4.6

- Storing persistent objects requires the core application to be connected to a database.
- For this purpose, object-oriented, relational or object-relational databases can be used.
- When using a relational database, as a first step the object model must be mapped to tables. (In particular, the inheritance hierarchy must be adequately mapped!)
- JDBC offers a platform independent interface for the connection of Java programs to relational databases.
- Persistence frameworks are often used for the "Object-Relational-Mapping" (ORM).