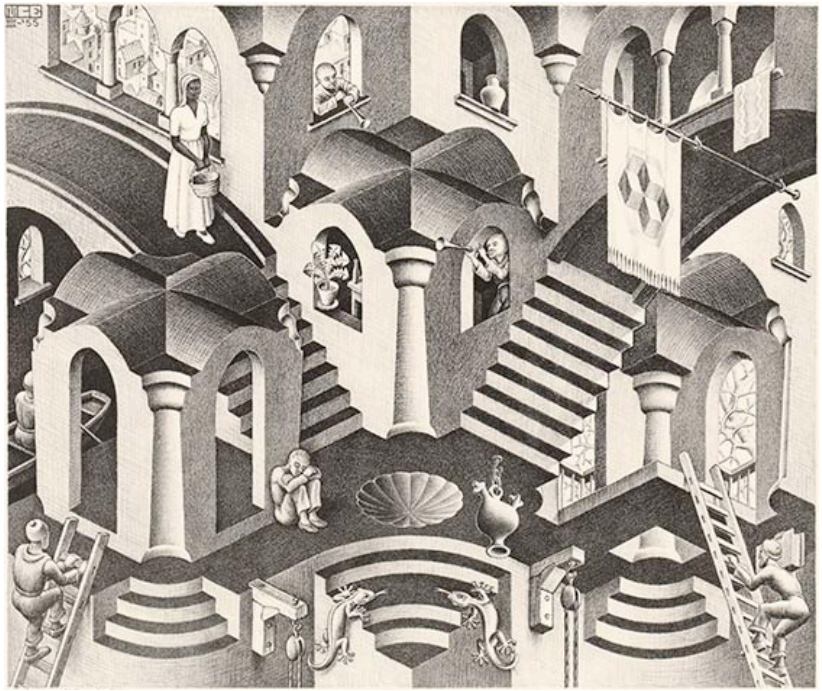


Software Architectures

Part 1: Modelling and Design Principles

1. Part 1 Learning Objectives

2. What is Architecture?

3. Describing Architecture
   - UML
   - Correctness, Modifiability, and Understandability
   - Encapsulation, abstraction, and information hiding
   - Separation of concerns and the single-responsibility principle
   - Interface segregation principle
   - Loose Coupling
   - Liskov substitution principle
   - Design by contract
   - Open-closed principle
   - Dependency inversion principle
   - Reducing dependencies

What are you about to learn?

Knowledge

- What is architecture?
- Architectural objectives in software design
- UML structural modeling elements

Skills

- Can detect violations of object oriented design principles
- Can explain all object oriented design principles using UML diagrams

The Parthenon in Athens, Greece

Some Genius Improvising

The Houses of Parliament, London

The Bauhaus in Dessau, Germany

The Taj Mahal, India

The National Congress of Brazil

The Carpenter did the Design

The Gare de Oriente in Lisbon, Portugal

The Golden Pavilion in Kyoto, Japan

Notre Dame de Paris, France

MI6 Headquarter in London

The Opera House in Sydney, Australia

Architecture in general

- A general term to describe buildings and other physical and some non-physical structures.
- The art and science of designing buildings and (some) nonbuilding structures (”Baukunst”).
- The style of design and method of construction of buildings and other physical and non-physical structures.

Architecture has to consider

- functional
- technical
- aesthetic

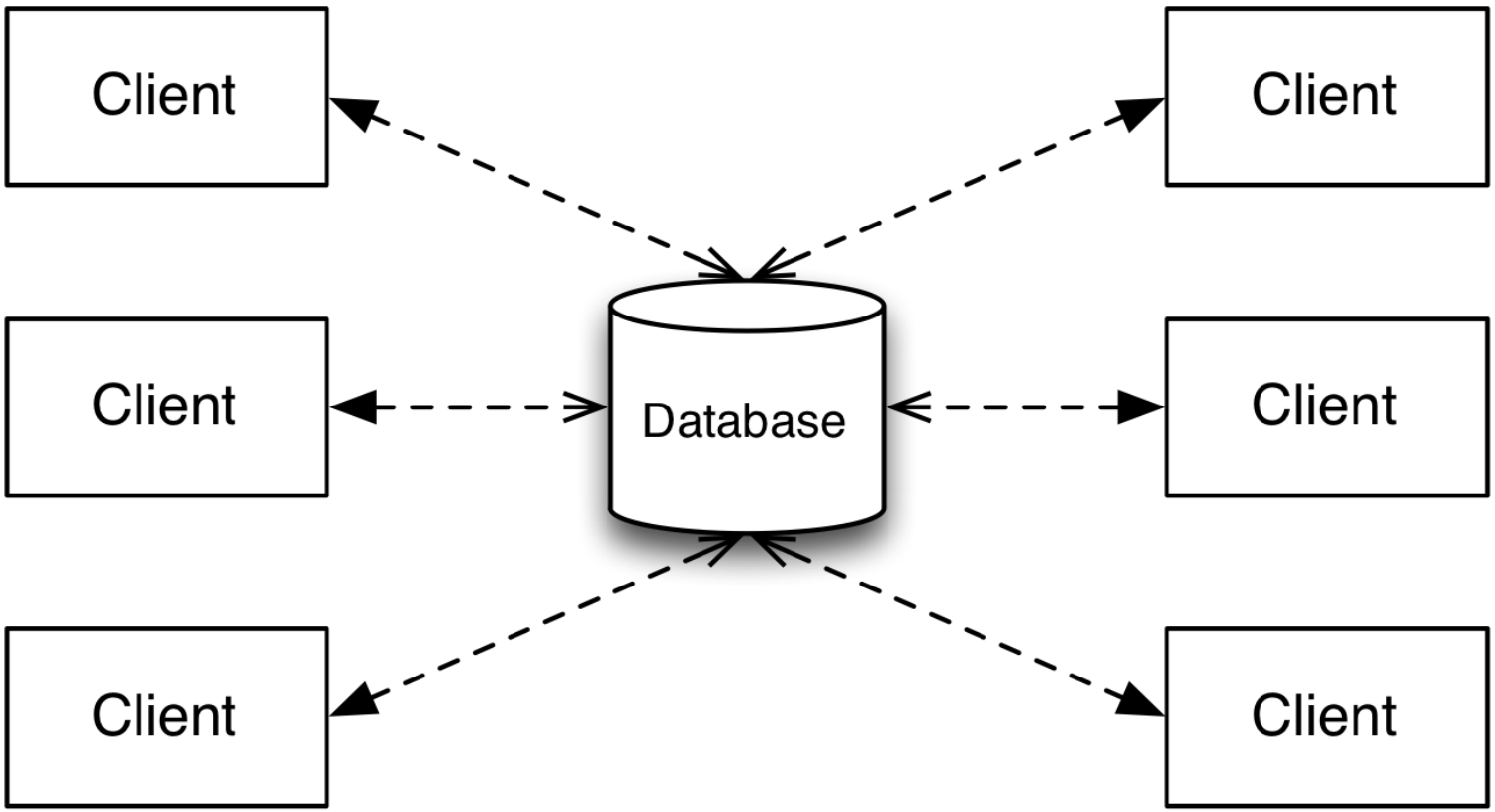
aspects in the planning, designing, and construction of its artefacts.

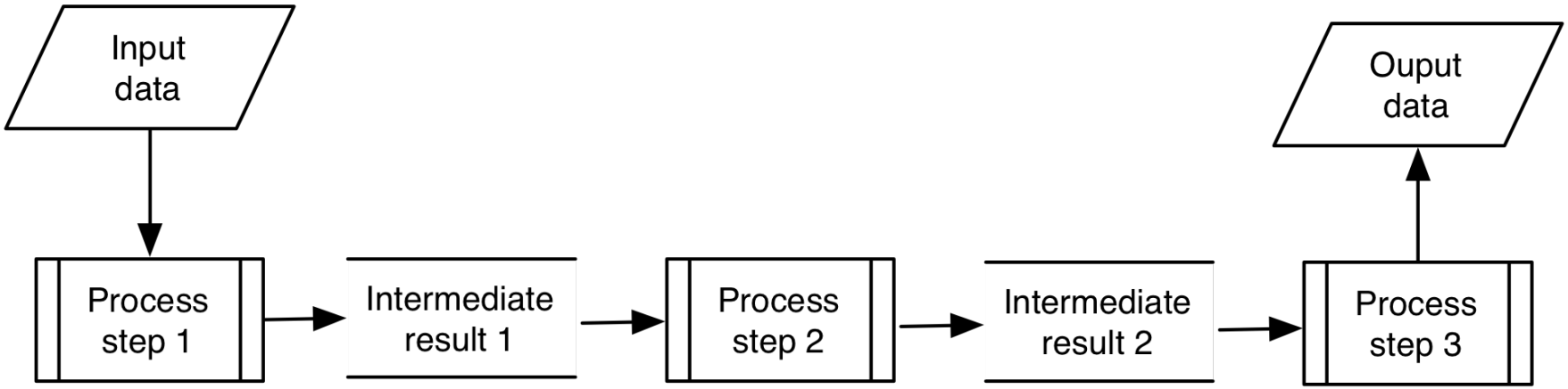
System as components

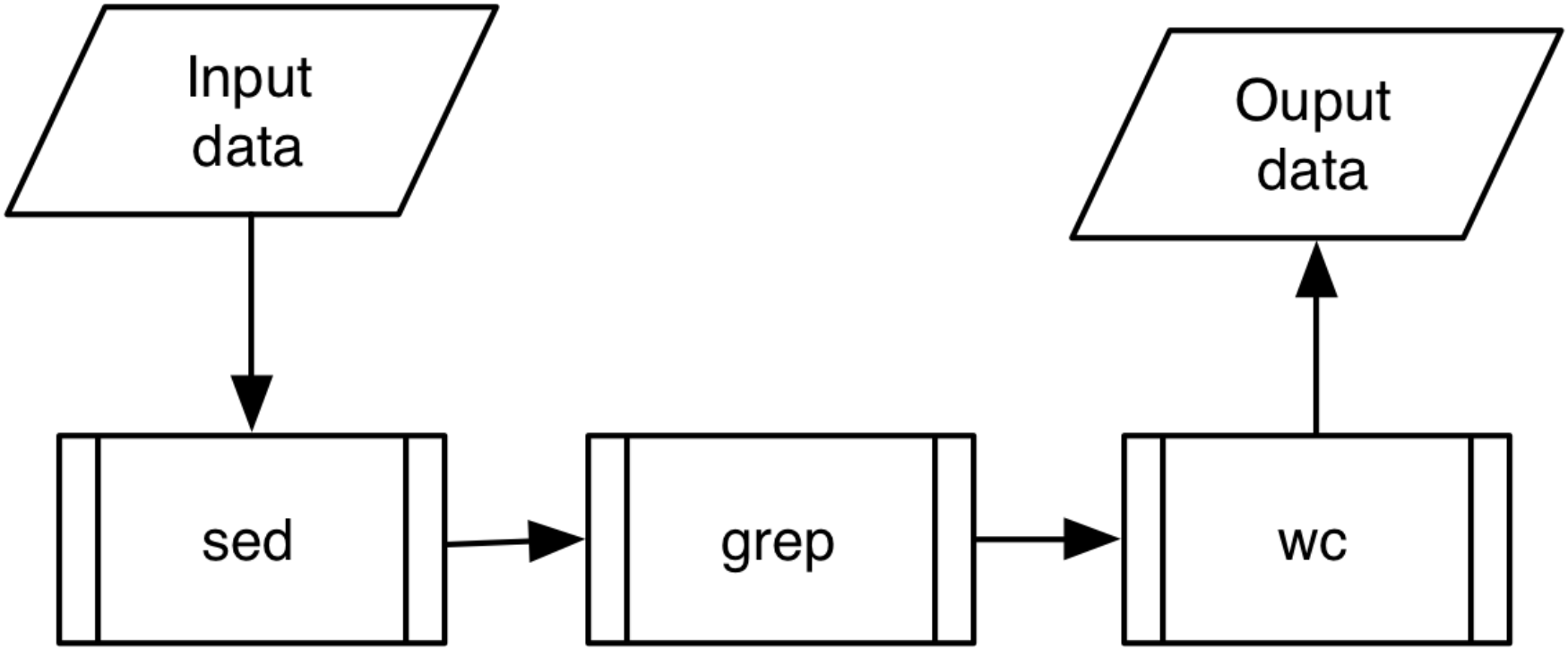
- Large systems are complex and cannot be built from one piece
- One of the oldest principles to manage complexity is ”divide et impera”
- We understand systems as a composition of interacting and related components
- Components themselves can be made up of other components
- Architecture describes what components are used and how they are organized
- Architecture describes the behaviour of components only as far as their collaboration is concerned

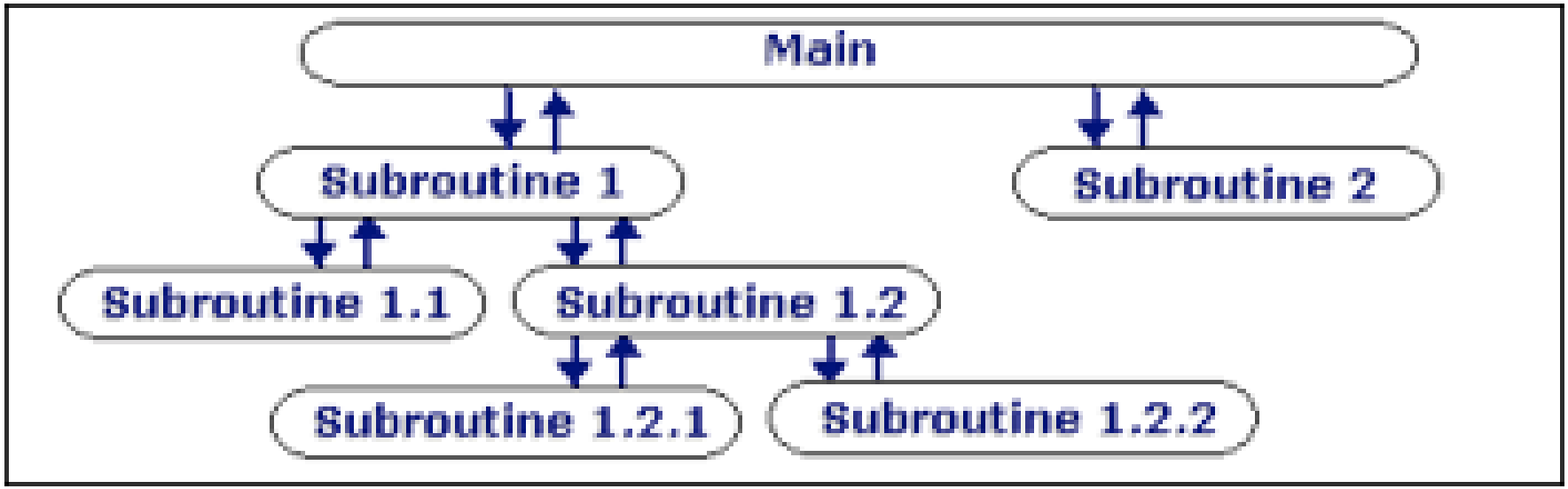
Architectural style in software system is determined by:

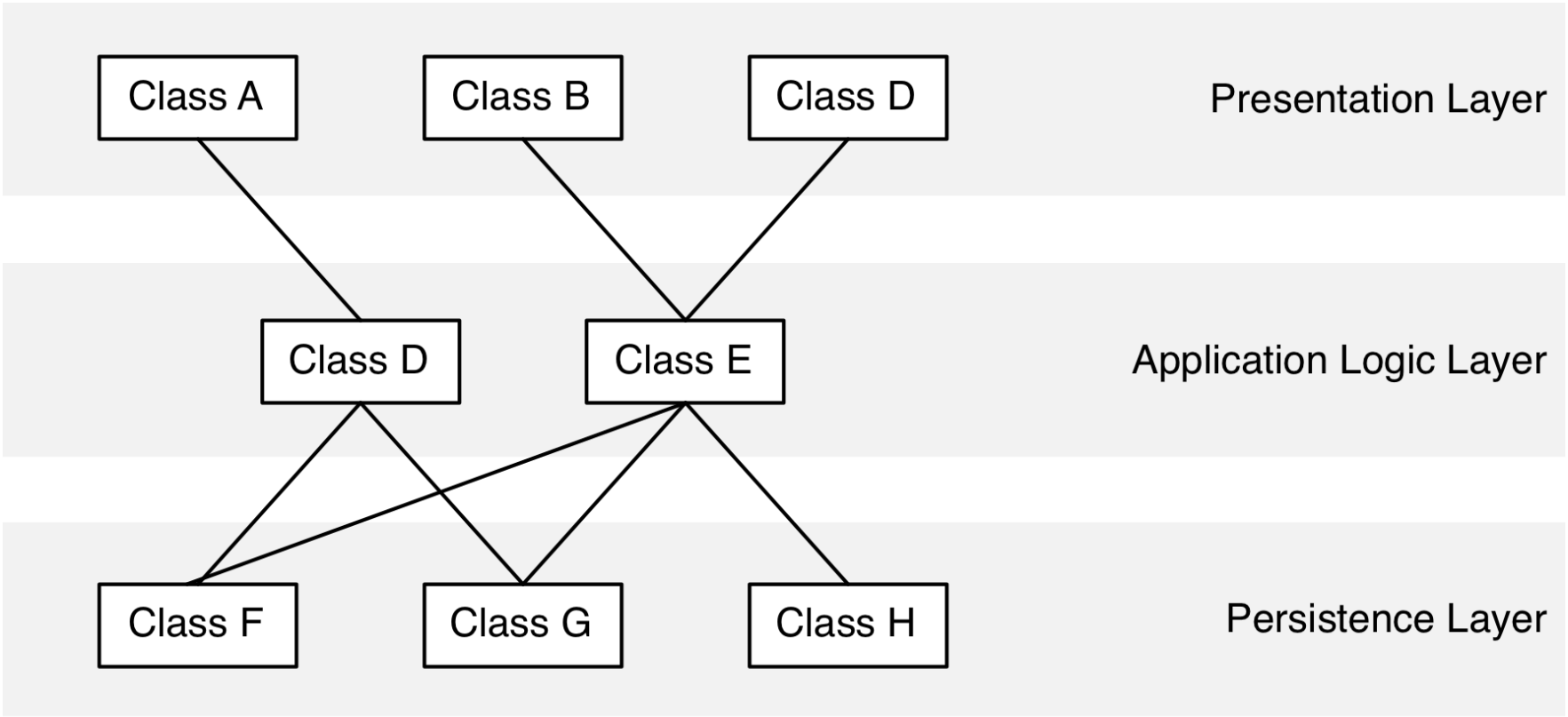
- set of component types that perform some function
- topological layout of these components indicating their relationship
- set of connectors that implement communication or coordination among components
- set of semantic constraints (for example, a filter must be stateless)

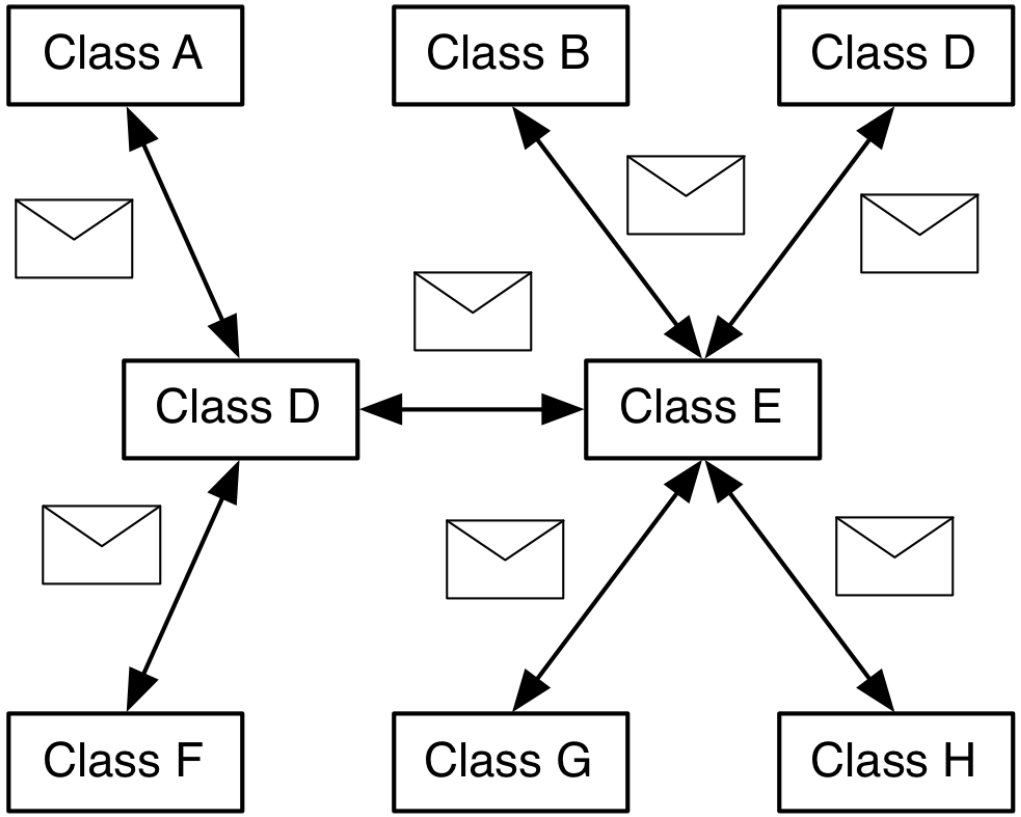


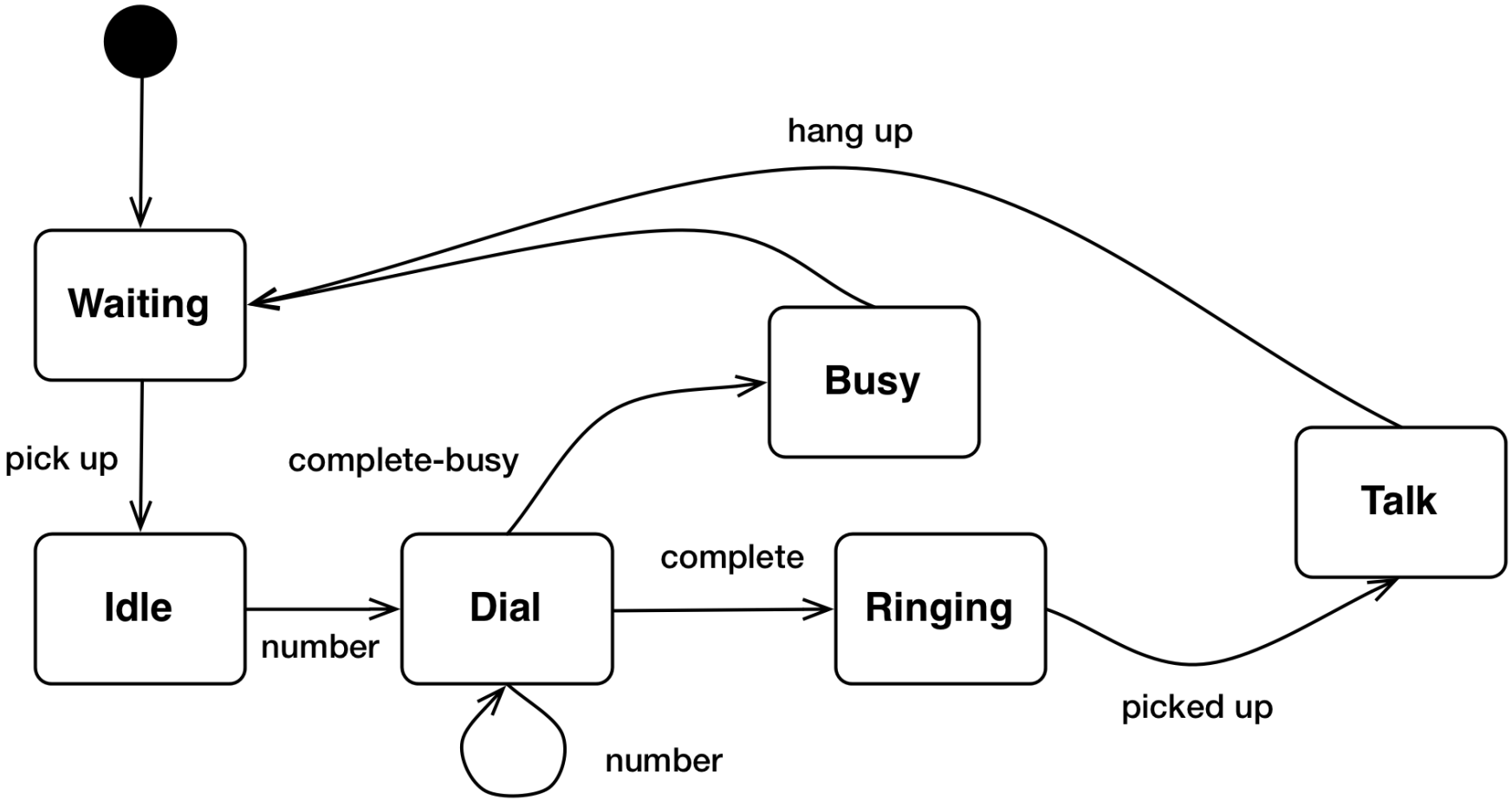












We use **models** to describe *things* in the real world.

Models are abstractions of the real world, simplifying reality.

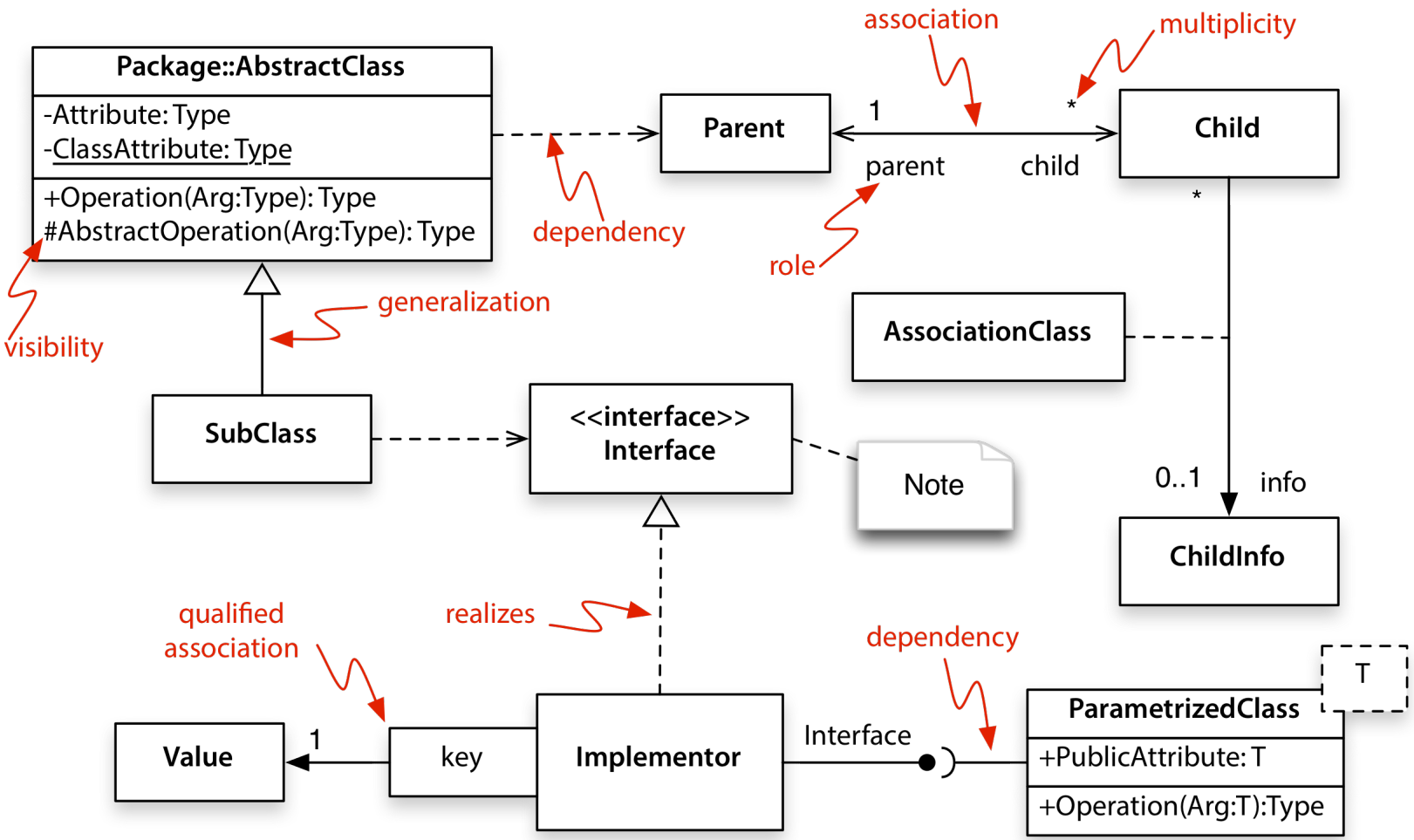
Our models focus on describing

- **Structure**
- **Behaviour**

Example

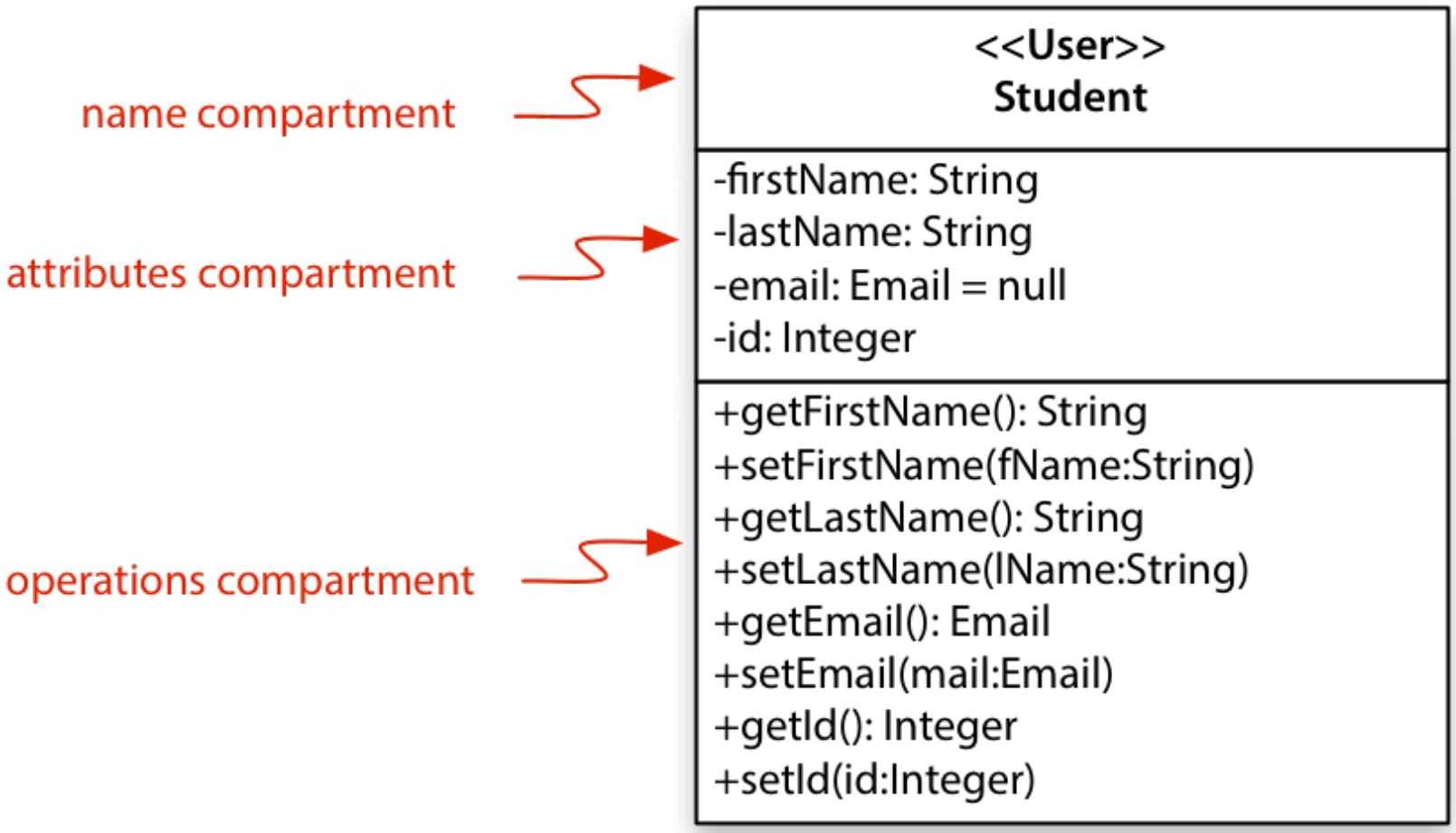
- Buildings (structure)
- Elevators (structure and behaviour)
- Buying something from an online-shop (behaviour)
- Cars (structure and behaviour)

Describing architecture we focus on structure.
For software systems we use the **Unified Modelling Language** (**UML**).



The UML offers a rich set of diagram types to describe structure and behaviour.

- **Class diagrams** ← this is what we use here
- Component diagrams
- Activity diagrams ← covered in "Real-Time Systems"
- State charts ← covered in "Real-Time Systems"
- Sequence diagrams
- Use case diagrams
- and eight more ...



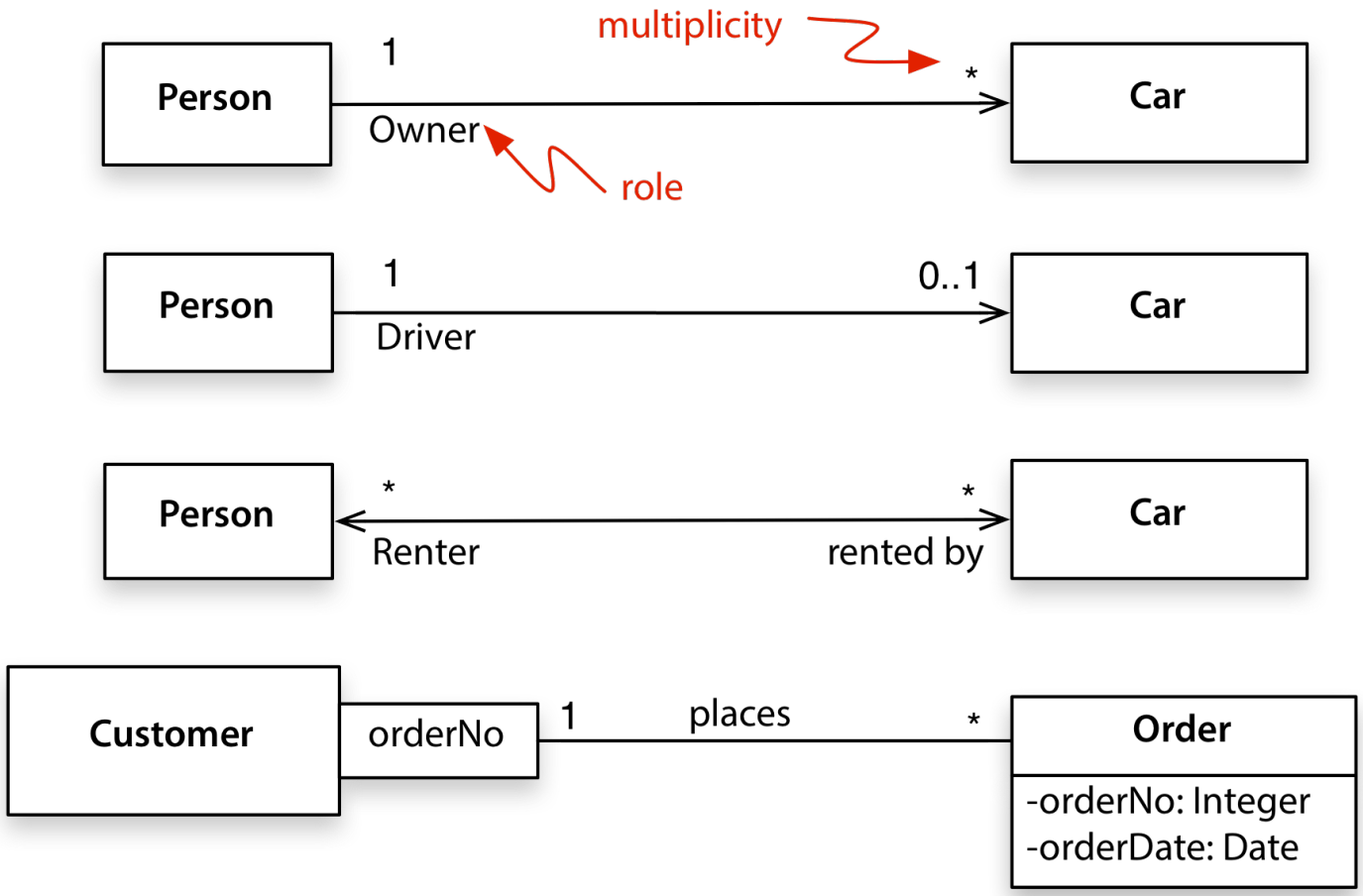
- **Attributes** describe the appearance and knowledge of a class of objects.
- **Operations** define the behavior that a class of objects can manifest.
- **Stereotypes** help you understand this type of object in the context of other classes of objects with similar roles within the system's design.
- **Properties** provide a way to track the maintenance and status of the class definition.
- **Association** is just a formal term for a type of relationship that this type of object may participate in. Associations may come in many variations, including simple, aggregate and composite, qualified, and reflexive.
- **Inheritance** allows you to organize the class definitions to simplify and facilitate their implementation.

Each attribute definition must specify what other objects are allowed to see the attribute – that is its visibility. Visibility is defined as follows:

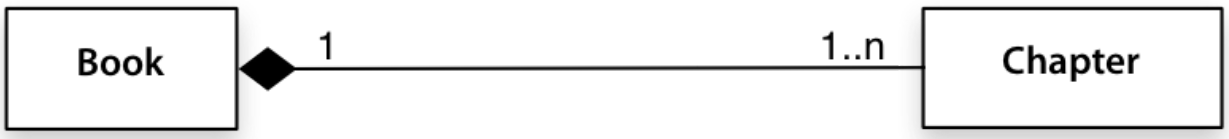
- Public (+) visibility allows access to objects of all other classes.
- Private (-) visibility limits access to within the class itself. For example, only operations of the class have access to a private attribute.
- Protected (#) visibility allows access by subclasses. In the case of generalizations (inheritance), subclasses must have access to the attributes and operations of the superclass or they cannot be inherited.
- Package (~) visibility allows access to other objects in the same package.

Attribute Element Description	Attribute Element Example
Create an attribute name	company
Add the attribute data type	company:character
Add the attribute's default value, if any	company:character = spaces
Set the constraints on the attribute value. For this example, first identify the field length.	company:character = spaces {1 to 30 characters}
Next identify the types of data that can be used in the attribute. Add this information within the brackets.	company:character = spaces {1 to 30 characters including alphabetic, spaces, and punctuation characters; no special characters allowed}
Set the attribute visibility (designate private visibility with a minus (-) sign in front of the attribute).	- company:character = spaces {1 to 30 characters including alphabetic, spaces, and punctuation characters; no special characters allowed}

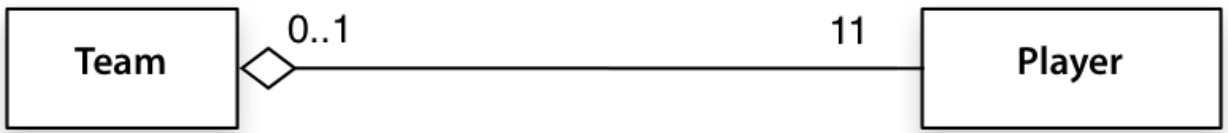
- Operation name: Required. The combination of name and parameters does need to be unique within a class.
- Arguments/parameters: Each argument requires an identifier and a data type.
- Return data type: Required for a return value, but return values are optional.
- Visibility (+, -, #, ~): Required before code generation.
- Class level operation (underlined operation declaration): Optional. In Java: static methods.
- Argument name: Required for each parameter, but parameters are optional. Any number of arguments is allowed.
- Argument data type: Required for each parameter, but parameters are optional.

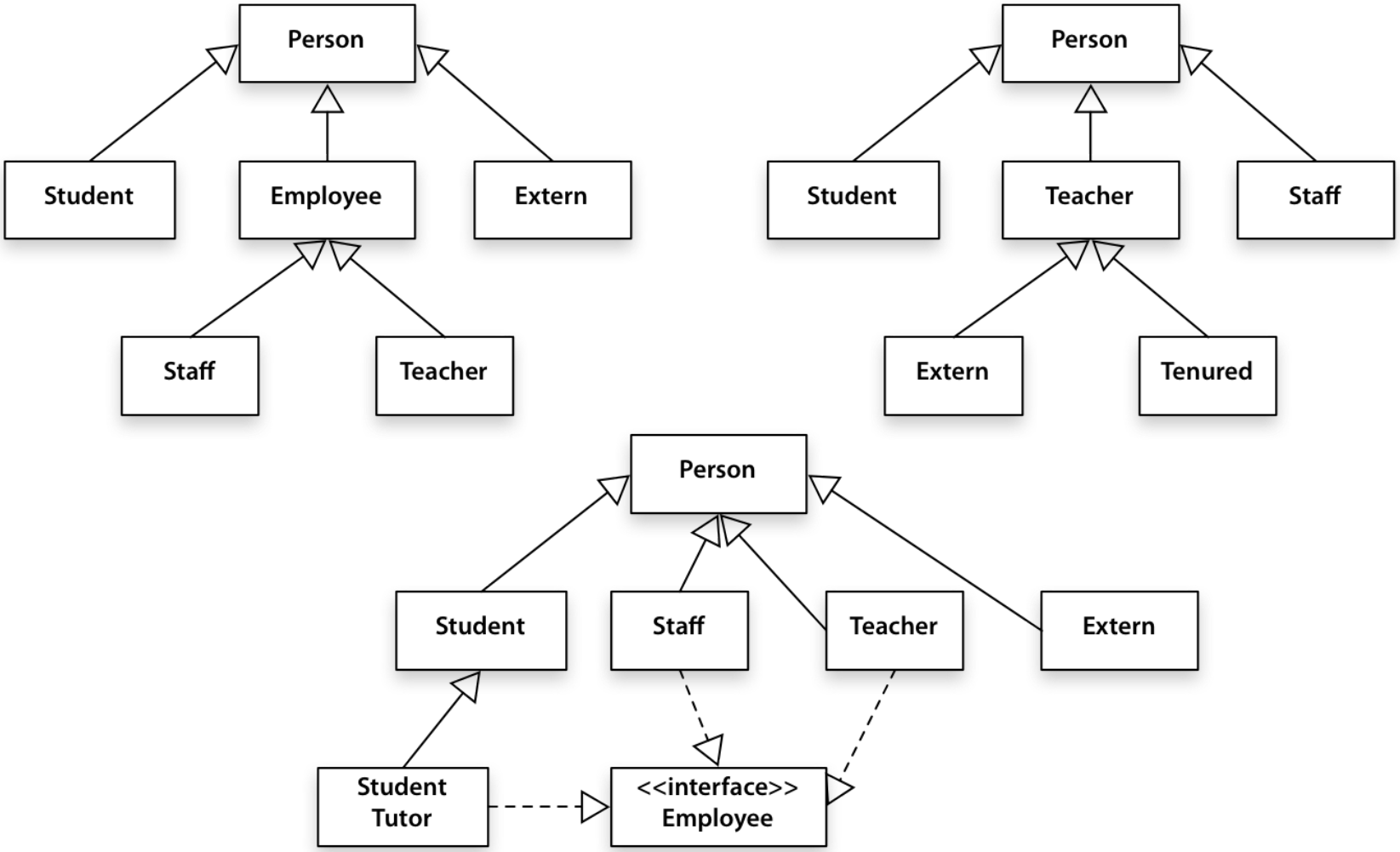


Composition



Aggregation





The system to be designed must be

- correct
- modifiable
- understandable

To that end we use object oriented design principles

Principles for Single Class Design

- Encapsulation, abstraction, and information hiding
- Separation of concerns and the single-responsibility principle
- Interface segregation principle

Principles for Design of Class Cooperation

- Loose Coupling
- Liskov substitution principle
- Design by contract
- Open-closed principle
- Dependency inversion principle

The architecture shall support correctness which implies

- Adherence to the **Design by contract** principle
- Adherence to the **Liskov substitution principle**

Both principles foster correctness of polymorphic programs

The architecture shall support later modification of the system.
This is promoted by

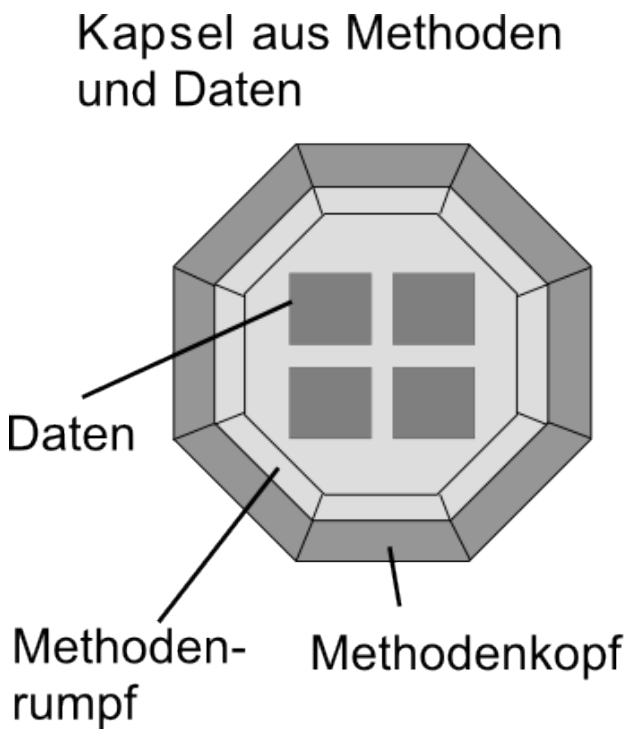
- **Interface segregation** principle
- **Loose coupling**
- Liskov substitution principle
- **Open-closed principle**
- **Dependency inversion** principle

The architecture shall make it easy to understand how the system works. This is promoted by

- **Encapsulation** and **Information hiding** principle
- **Separation of concerns** and **Single responsibility** principle

Encapsulation is one of the most important system design principles.

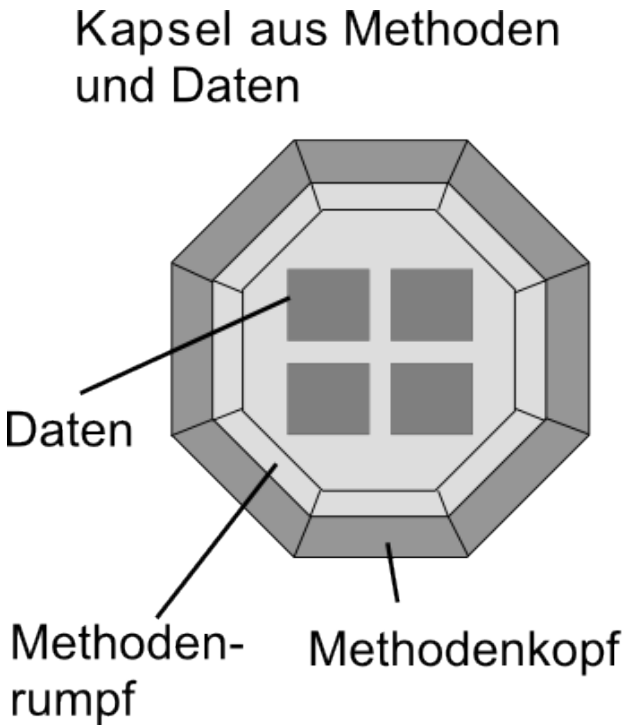
- Methods and data are contained within an object
- The type of object is called a **class**



Methodenköpfe sind die Schnittstellen des Objekts (Abstraktion).

Daten und Methoden-rümpfe sind verborgen (Information Hiding). Die Daten können nur über die Köpfe der eigenen Methoden eines Objekts manipuliert werden.

- An object exposes its behaviour solely via **interface methods**
- The method declarations define the object interface towards the outside



Methodenköpfe sind die Schnittstellen des Objekts (Abstraktion).

Daten und Methoden-rümpfe sind verborgen (Information Hiding). Die Daten können nur über die Köpfe der eigenen Methoden eines Objekts manipuliert werden.

- This concept is similar to abstraction
- The implementation of the objects behaviour is hidden
- Users of the object don't need to care about the internals
- Only the interface methods are exposed to the outside
- Internal data cannot be accessed except via interface methods

Example

- Have a look at the Java API docs for `Html2LaTeX.java`
- Have a look at the Java code for `Html2LaTeX.java`

- Separation of concerns means focussing one's attention on one specific aspect
- Helps ordering thoughts
- Helps structuring a system into a variety of parts with clear responsibility

Example

- For emergencies we have police, firemen, and ambulances
- We have a living room, a bedroom, a kitchen, and a bath

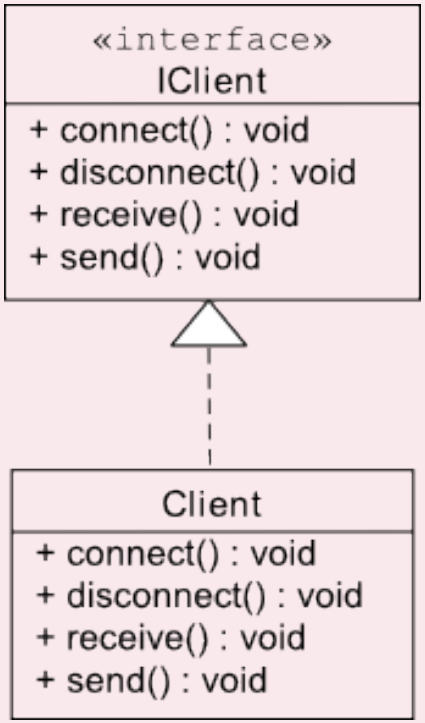
- Single responsibility: dedicate each class to a specific task
- Don't create all-round classes that take care of a variety of unrelated problems

Example

- For date related functionality we have `java.util.Date`
- For numerical computations we have `java.lang.Math`

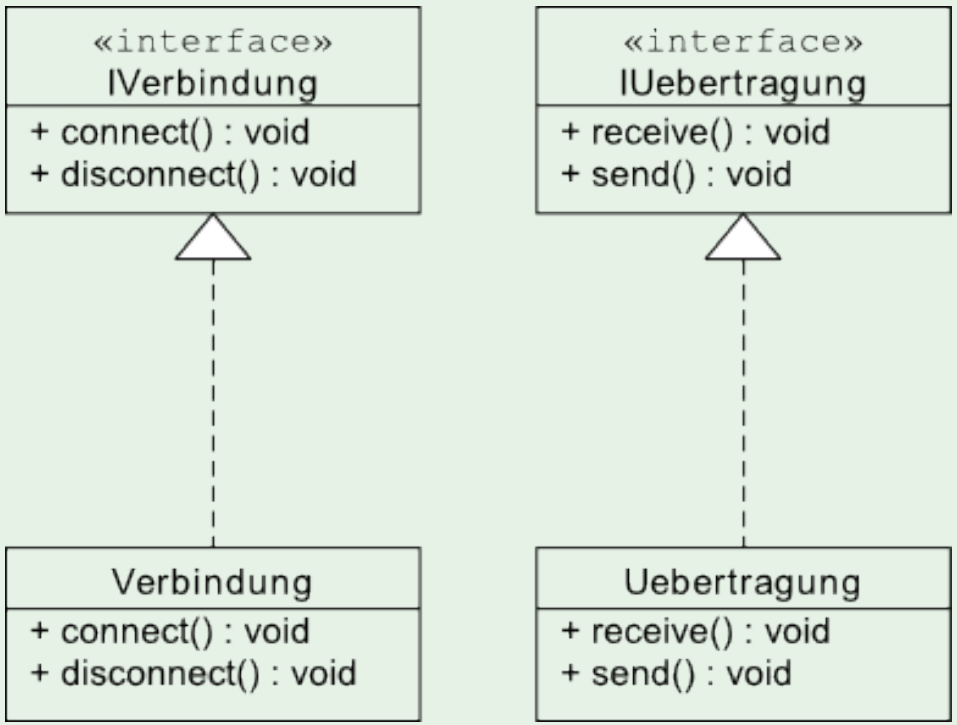
- This is a bad example mixing connectivity and message transport

Bad example



- This is a good example with dedicated classes

Example



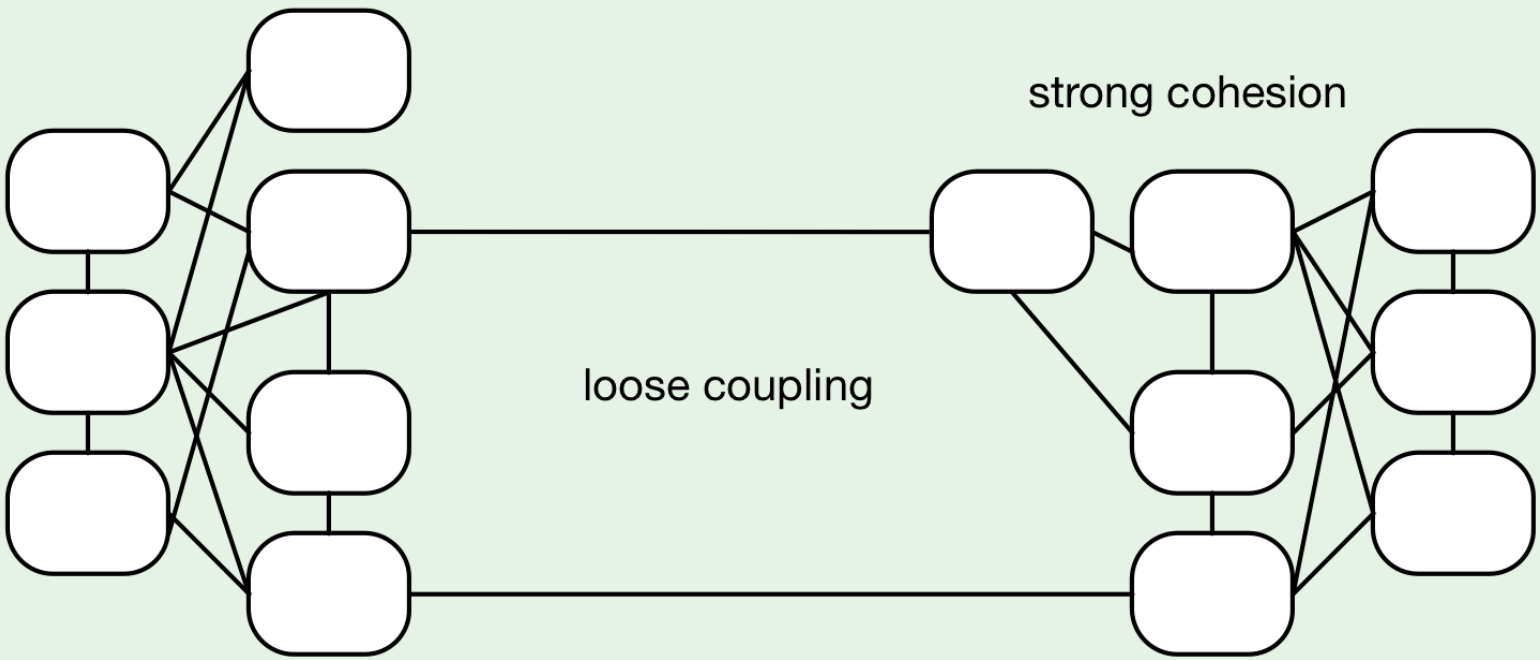
- Interface segregation: split up large interface into smaller ones
- Clients shall not depend on interfaces they don't need
- Same idea as single responsibility for classes

Example

- Have a look at class `com.aurel.track.util.ImageServerAction`

- Loose coupling: few dependencies between different system parts
- Strong cohesion: within each system part strong dependencies

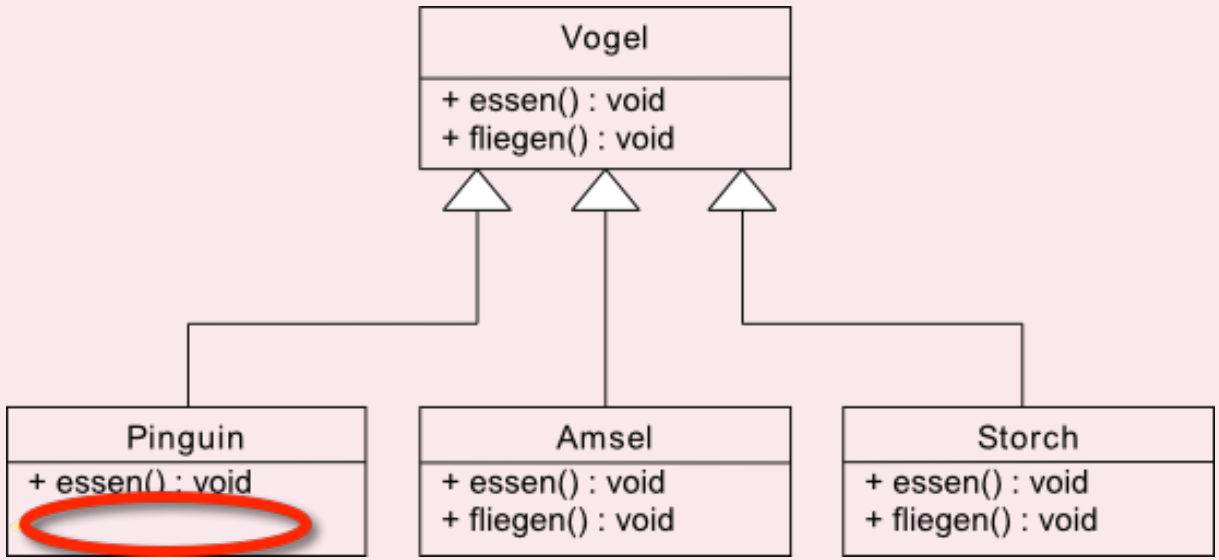
Example



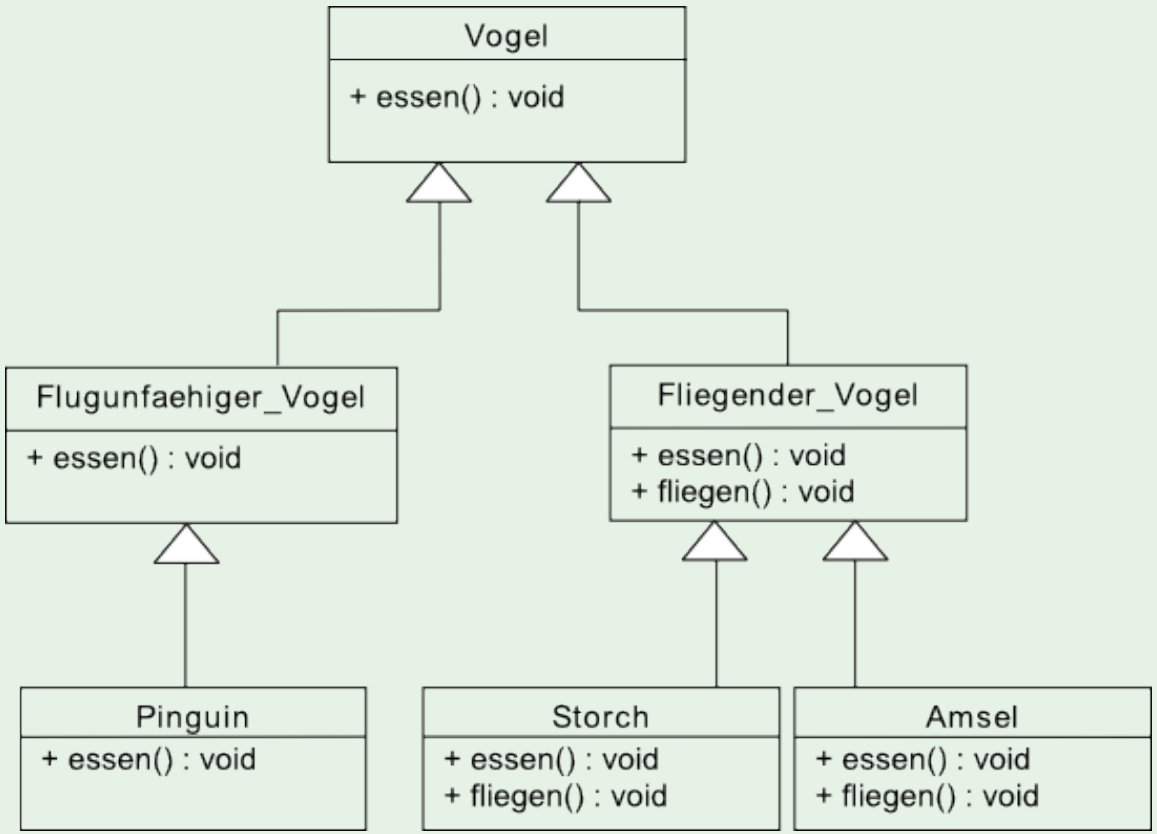
The Liskov substitution principle

Let Φ(x) be a property provable about objects x of type T. Then Φ(y) should be true for objects y of type S where S is a subtype of T.

Bad example



Example



- Design by contract prescribes that interface specifications for software components are considered like ”contracts”
- The caller or ”buyer” must abide by the contract just as the called class or ”seller”
- Contracts are specified in terms of **assertions**

Example

What all can go wrong here?

```
1   public String timeStampLog(String message) {
2         message = new Date().toString() + ": " + message.trim() ;
3         return message;
4   }
```

Assertions can be of the following types:

- **Preconditions**
- **Postconditions**
- **Invariants**

- The contract of a method comprises the pre- and postconditions of that method
- The contract of a class comprises the contracts of its methods plus the invariants of the class

Assertions can be of the following types:

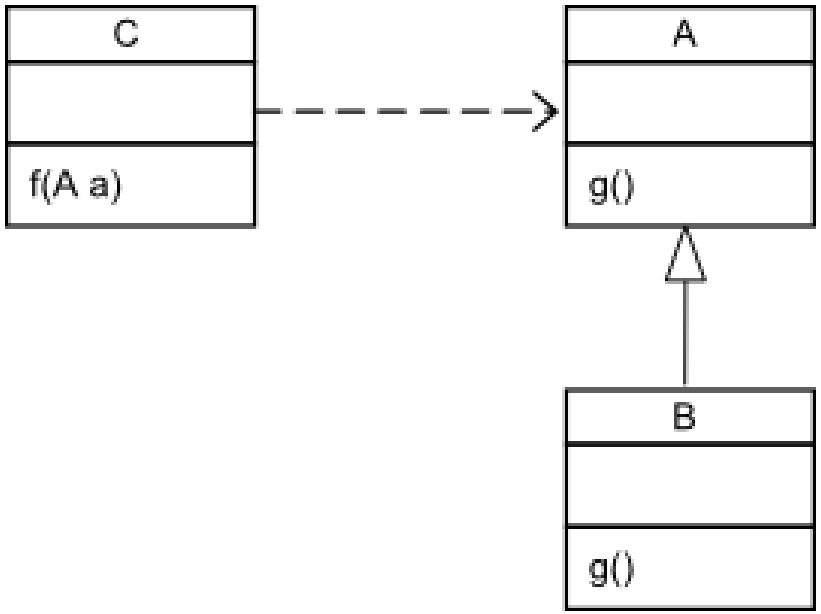
- Preconditions of the called must be checked by the caller prior to the call
- Postconditions must be checked by the called prior to responding
- Invariants are assertions which must be adhered to by all objects of a class

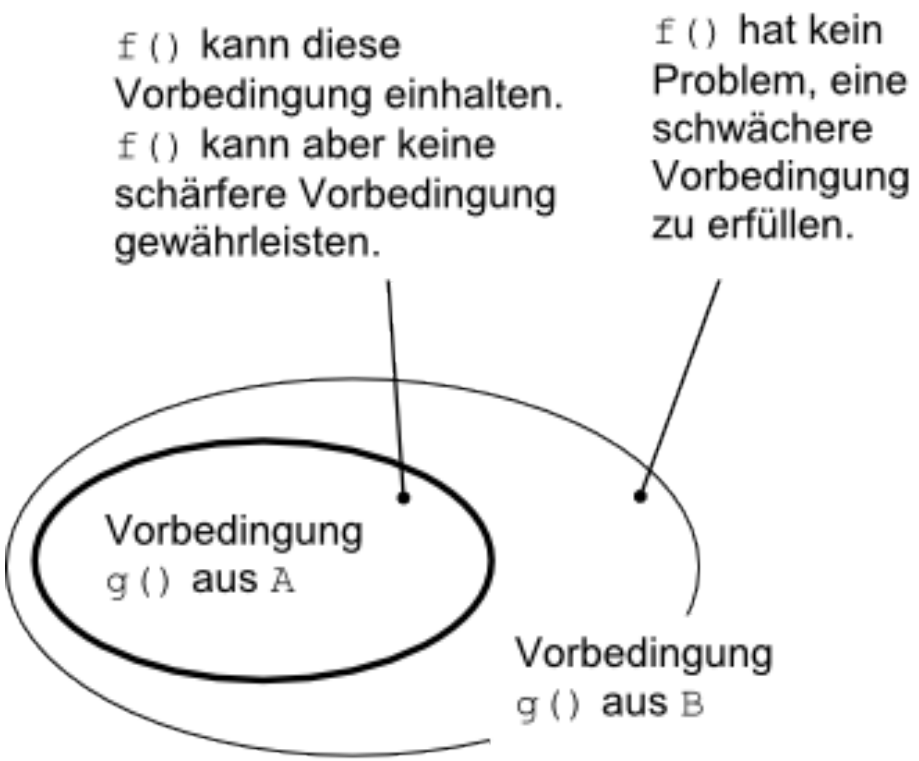
Example

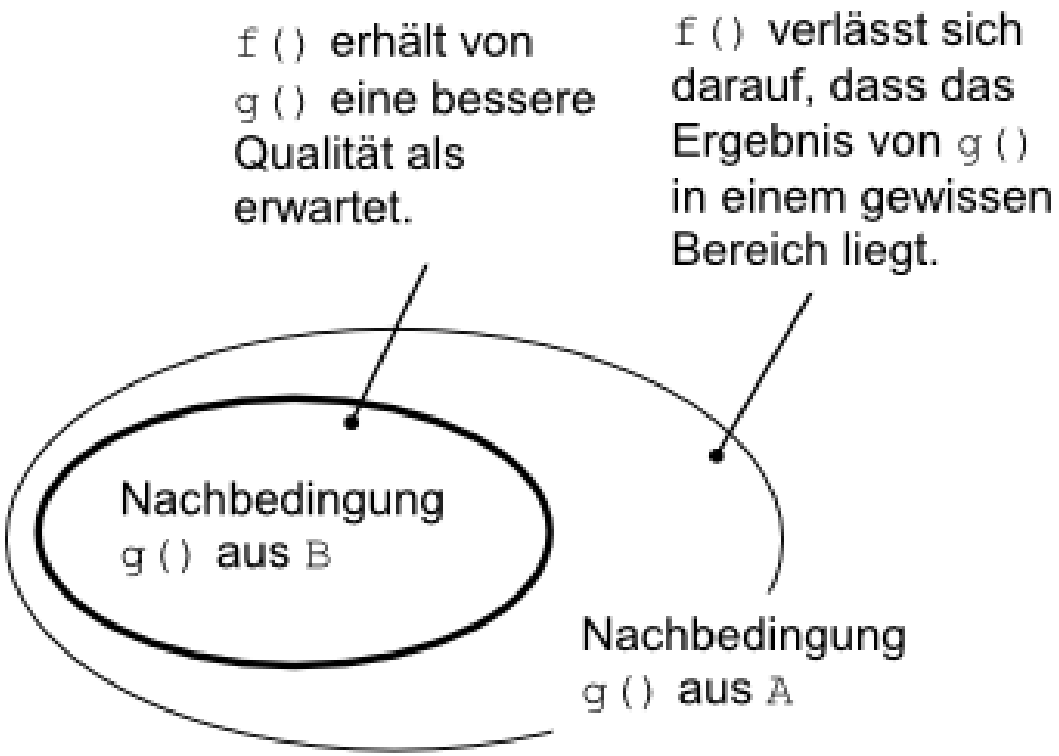
```
1  abstract class Stack {     // invariant is number of elements on
2                             // the stack is within [0,stackSize]
3      void push (Object o); // precondition is that the number of
4                             // elements on the stack <= stackSize-1
5      Object pop(); // postcondition is that the number of elements
6                    // on the stack is one less than before the call
7  }
```

When overriding, contracts need to be kept (here overriding `g()`)

- `g()` in class `B` must not tighten any preconditions
- `g()` in class `B` may loosen its preconditions
- `g()` in class `B` must not loosen any postconditions
- `g()` in class `B` may tighten its postconditions

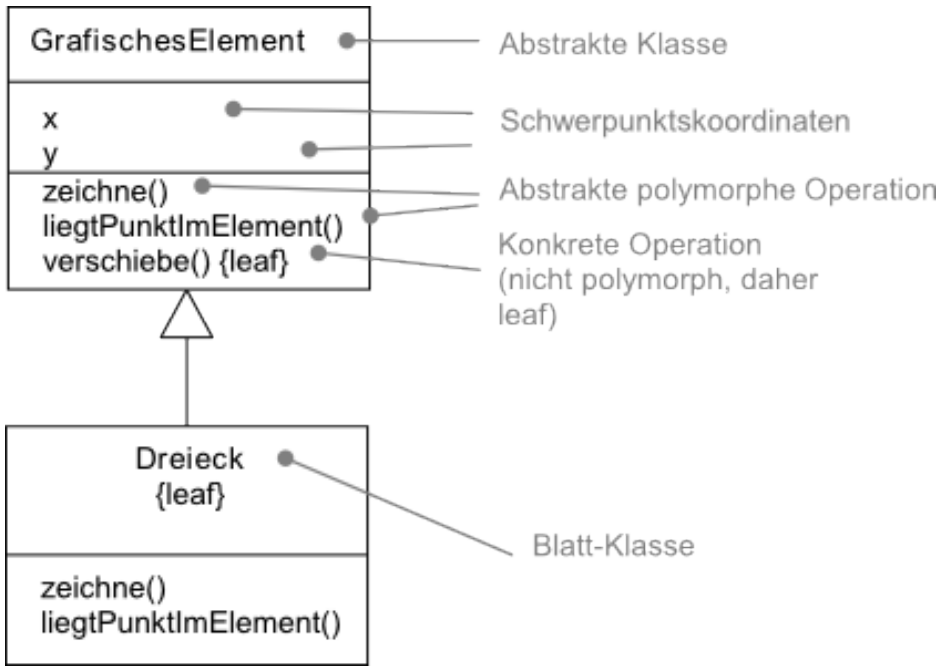




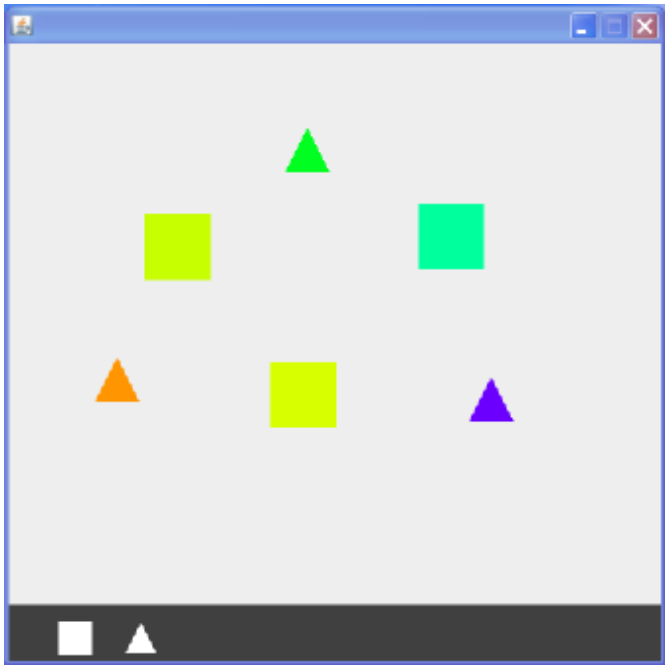


Open-closed principle means

- Modules shall be *open* in a sense that they are extensible
- Modules shall be *closed* in a sense that their source code shall not be altered
- How can we extend something without changing it?
  Answer: By inheritance



We will now look at some real code for a graphical editor



```java
// Datei: GrafischesElement.java
import java.awt.Graphics;

public abstract class GrafischesElement
{
    // Schwerpunktskoordinaten
    protected int x, y;

    // hier koennen weitere Eigenschaften, wie bspw. Fuellfarbe
    // definiert werden.

    // Konstruktor mit Schwerpunktskoordinaten
    public GrafischesElement(int x, int y)
    {
        this.x = x;
        this.y = y;
    }
```

```
19      // Methode zur Verschiebung des Schwerpunktes; durch final als
20      // leaf-Methode deklariert.
21      final public void verschiebe(int x, int y)
22      {
23          this.x += x;
24          this.y += y;
25      }
26
27      final public int getX()
28      {
29          return x;
30      }
31
32      final public int getY()
33      {
34          return y;
35      }
```

```
37      // Methodenkopf der Darstellungsroutine; Graphics ist eine
38      // Klasse, welche das Auftragen von Grafiken auf Komponenten
39      // erlaubt. In der zeichne()-Methode wird anschliessend das
40      // darzustellende Element auf das Graphics-Objekt aufgetragen.
41      public abstract void zeichne(Graphics g);
42
43      // Methode zum Ueberpruefen, ob Punkt im Element liegt.
44      public abstract boolean liegtPunktImElement (int x, int y);
45  }
```

Open-closed principle: Viereck.java
Lines 1–20


```
1  // Datei: Viereck.java
2  import java.awt.Graphics;
3  import java.awt.Rectangle;
4
5  public class Viereck extends GrafischesElement
6  {
7     int laenge, hoehe;
8     Rectangle viereck;
9
10    public Viereck(int x, int y, int laenge, int hoehe)
11    {
12       super(x, y);
13       this.laenge = laenge;
14       this.hoehe = hoehe;
15    }
16
17    // Implementierte zeichne()-Methode der Basisklasse
18    public void zeichne(Graphics g)
19    {
20       // Viereck erstellen
```

Open-closed principle: Viereck.java
Lines 21–40


```
21          viereck = new Rectangle(this.x - (int) (0.5 * laenge),
22                                  this.y - (int) (0.5 * hoehe), laenge, hoehe);
23
24          // Viereck zeichnen
25          g.fillRect(viereck.x, viereck.y, viereck.width,
26                     viereck.height);
27      }
28
29      public boolean liegtPunktImElement(int x, int y)
30      {
31          return viereck.contains(x,y);
32      }
33  }
```

```
1  // Datei: Dreieck.java
2  import java.awt.Graphics;
3  import java.awt.Polygon;
4
5  final public class Dreieck extends GrafischesElement
6  {
7     int seitenlaenge;
8     Polygon dreieck;
9
10    public Dreieck(int x, int y, int seitenlaenge)
11    {
12       super(x, y);
13       this.seitenlaenge = seitenlaenge;
14    }
15
16    public Polygon getDreieck(int seitenlaenge)
17    {
18       // Ein Dreieck wird als Polygon betrachtet:
19       // Zuerst werden die Koordinaten der Eckpunkte berechnet und
20       // dann ein Polygon erzeugt. Das Polygon wird als Ergebnis
```

```
21          // zurueckgegeben.
22          int xArr[]={getX()-(seitenlaenge/2),
23                      getX()+(seitenlaenge/2),getX()};
24          int yArr[]={getY()+(seitenlaenge/2),getY()+(seitenlaenge/2),
25                      getY()-(seitenlaenge/2)};
26          // Neues Polygon zurueckgeben
27          return new Polygon(xArr, yArr, 3);
28      }
29
30      public void zeichne(Graphics g)
31      {
32          dreieck = getDreieck(seitenlaenge);
33          g.fillPolygon(dreieck);
34      }
35
36      public boolean liegtPunktImElement (int x, int y)
37      {
38          return dreieck.contains(x,y);
39      }
40  }
```

```
1  // Datei: Editor.java
2  import java.awt.Color;
3  import java.awt.Graphics;
4  import java.awt.event.MouseAdapter;
5  import java.awt.event.MouseEvent;
6  import java.util.ArrayList;
7  import javax.swing.JFrame;
8  import javax.swing.JPanel;
9
10 public class Editor extends JPanel
11 {
12    // Hoehe und Weite des Editors. Ausserdem die Hoehe der
13    // Auswahlleiste
14    final static int hoehe = 500, weite = 500;
15    final static int auswahlLeisteHoehe = 75;
16
17    // Liste fr Auswahlelemente
18    final static ArrayList<GrafischesElement> auswahlElement = new
19                                          ArrayList<GrafischesElement>();
20    // Liste fr eingezeichnete Elemente
```

```
21
22      ArrayList<GrafischesElement> gezeichneteElemente = new
23                                              ArrayList<GrafischesElement>();
24
25      // Referenz auf das  nchste zu zeichnende Objekt
26      GrafischesElement naechstesElement = null;
27
28      static public void main(String[] args)
29      {
30          // Fenster anfordern
31          JFrame frame = new JFrame();
32          // Groesse setzen
33          frame.setSize(weite, hoehe);
34          // Nicht vergroesser- oder verkleinerbar
35          frame.setResizable(false);
36          // wenn Schliessen-Button gedrueckt wird, soll Programm
37          // beendet werden
38          frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
39
40          // Editor instanziieren und in Fenster einbetten
41          frame.add(new Editor());
```

```
41
42
43          // Fenster sichtbar machen
44          frame.setVisible(true);
45      }
46
47
48      public Editor() {
49          super();
50          // Initialisierung der Leistenelemente
51          auswahlElement.add(new Viereck(50, hoehe - 50, 25, 25));
52          auswahlElement.add(new Dreieck(100, hoehe - 50, 25));
53
54          // Maus-Listener und Programmlogik hinzufgen
55          addMouseListener(new MouseAdapter()
56          {
57              // Maus wurde geklickt ...
58              public void mousePressed(MouseEvent e)
59              {
60
61                  // wenn in Leiste ...
```

```
61
62              if (e.getY() >= (hoehe - auswahlLeisteHoehe))
63              {
64                  // pruefe, ob LeistenElement gedrueckt wurde
65                  for (GrafischesElement g : auswahlElement)
66                  {
67                      // LeistenElement wurde gedrueckt
68                      if (g.liegtPunktImElement(e.getX(), e.getY()))
69                      {
70                          // sichere dieses Element temporaer
71                          naechstesElement = g;
72                          break;
73                      }
74                  }
75              }
76
77              // wenn in Zeichenflaeche ...
78              else {
79
80                  // pruefe, ob ein bestehendes Objekt
81                  // auf der Zeichenflaeche gedrueckt wurde ...
```

```
81
82                  for (GrafischesElement g : gezeichneteElemente)
83                  {
84                      // wenn ja ...
85                      if (g.liegtPunktImElement(e.getX(), e.getY()))
86                      {
87                          // bereite Verschiebung vor.
88                          naechstesElement = g;
89                          return;
90                      }
91                  }
92
93                  // Es wurde kein bestehendes Objekt auf der Zeichen-
94                  // flaeche gedrueckt. Pruefe ob vorher ein
                        LeistenElement
95                  // ausgewaehlt wurde und nun eingezeichnet werden soll
96                  if (naechstesElement instanceof Viereck)
97                      gezeichneteElemente.add(new Viereck(e.getX(),
98                                                  e.getY(), 50, 50));
99                  else if (naechstesElement instanceof Dreieck)
100                     gezeichneteElemente.add(new Dreieck(e.getX(),
101                                                         e.getY(),35));
```

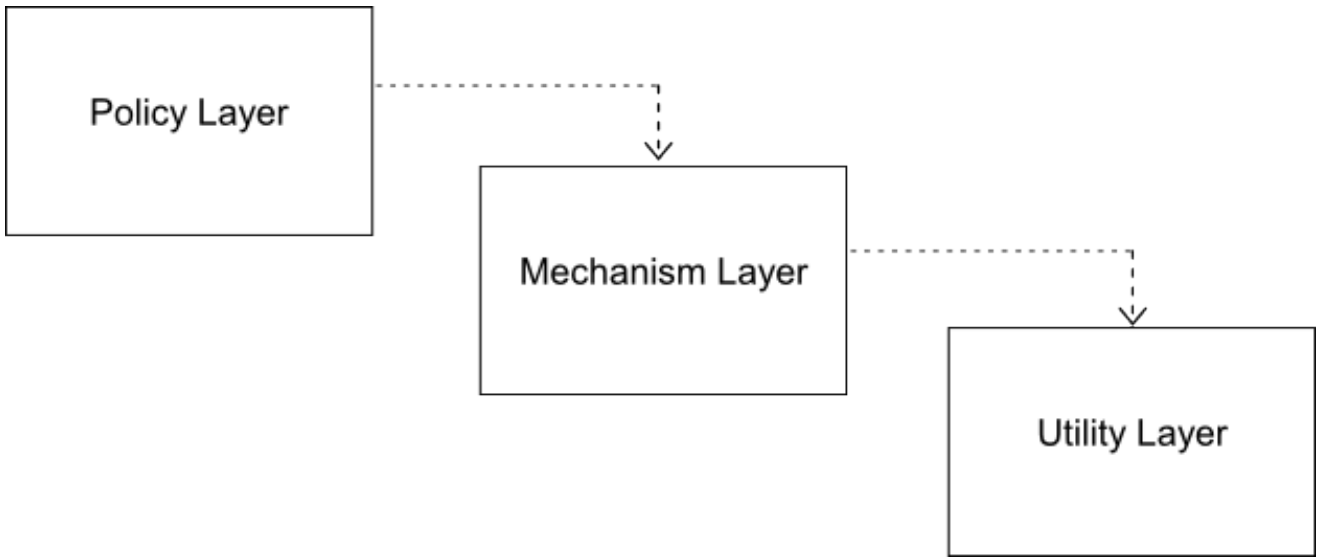
```
101
102             // Ausgewaehltes Leistenelement wurde eingezeichnet
103             // setze naechstesElement zurueck
104             naechstesElement = null;
105
106             // Neuzeichnen der Darstellung
107             repaint();
108         }
109     }
110
111     // Maus wurde losgelassen
112     public void mouseReleased(MouseEvent e) {
113         // wenn naechstesElement nicht null und kein
114         // Auswahlobjekt ist, soll ein Objekt verschoben werden
115         if ((naechstesElement != null) &&
116             (auswahlElement.indexOf(naechstesElement) < 0))
117         {
118             naechstesElement.verschiebe(-1 *
119                     (naechstesElement.getX() - e.getX()),
120                     (naechstesElement.getY() - e.getY()) * -1);
121             naechstesElement = null;
```

```
121
122             }
123
124             repaint();
125         }
126     });
127 }
128
129 // Diese Funktion wird durch repaint() aufgerufen und zeichnet
130 // die Oberflaeche. Nur mit dieser Methode kann die zeichne()-
131 // Funktion von graphischen Elementen aufgerufen werden.
132 @Override
133 protected void paintComponent(Graphics g)
134 {
135     // Komponenten zeichnen
136     super.paintComponent(g);
137
138     // zeichne Elemente auf die  Flche
139     for (GrafischesElement ge : gezeichneteElemente)
140     {
141         // Farbe nach Position setzen ...
```

```
141
142             g.setColor(Color.getHSBColor((float) 1.0 / (ge.getX() /
143                                                 ((float) ge.getY())), 1, 1));
144             // und einzeichnen
145             ge.zeichne(g);
146         }
147
148         // Auswahlleiste zeichnen ...
149         g.setColor(Color.DARK_GRAY);
150         g.fillRect(0, hoehe - auswahlLeisteHoehe, weite, hoehe);
151
152         // ... mit zugehoerigen Elementen
153         g.setColor(Color.WHITE);
154         for (GrafischesElement ge : auswahlElement)
155         {
156             ge.zeichne(g);
157         }
158     }
159
160 }
```

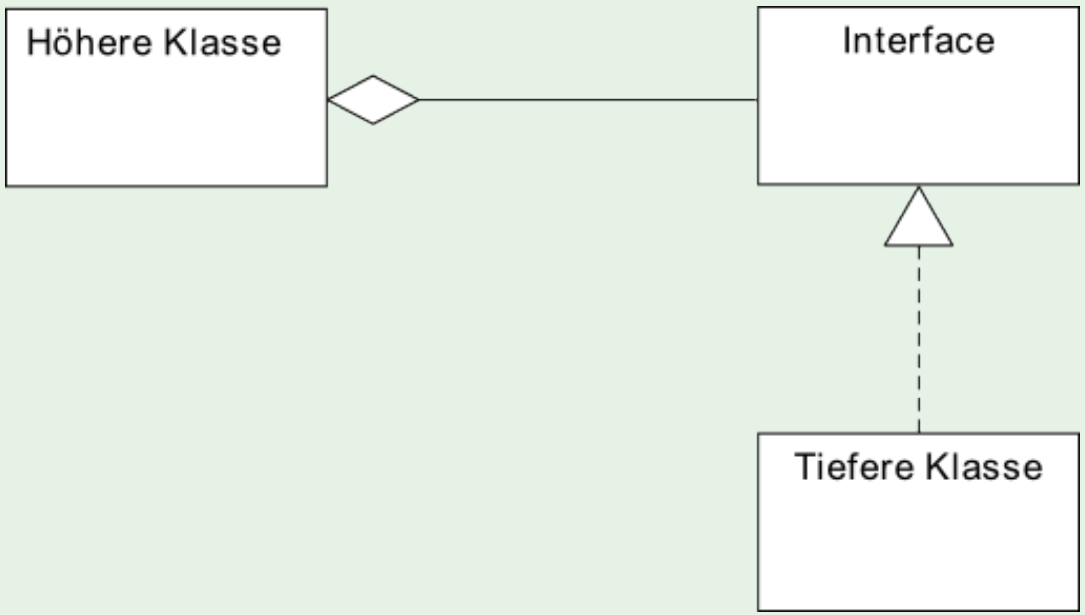
Dependency inversion

High-level modules should not depend on low-level modules. Both should depend on abstractions.
Abstractions should not depend on details. Details should depend on abstractions.



Example

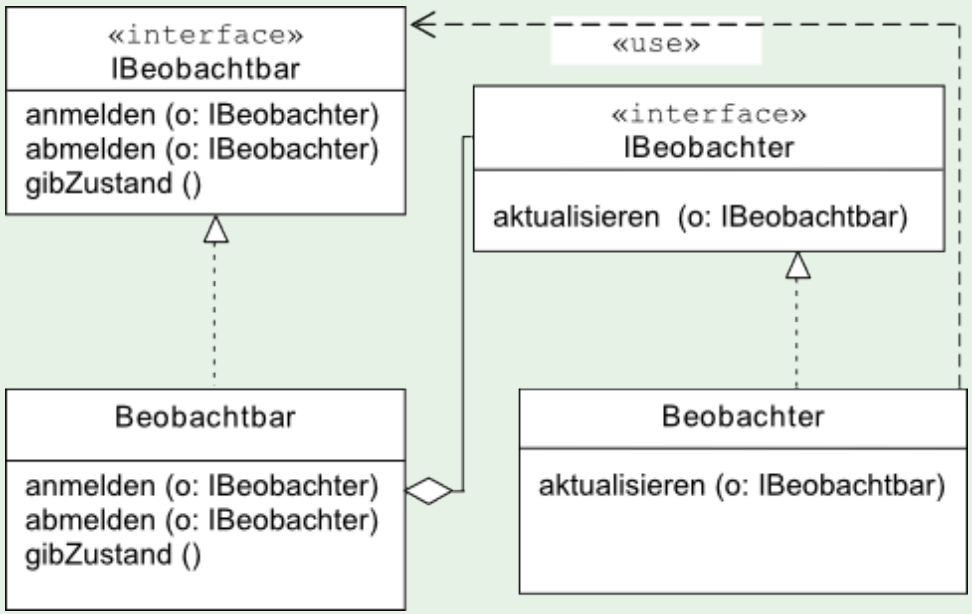
- High-level class aggregates (uses) interface
- Low-level class implements interface



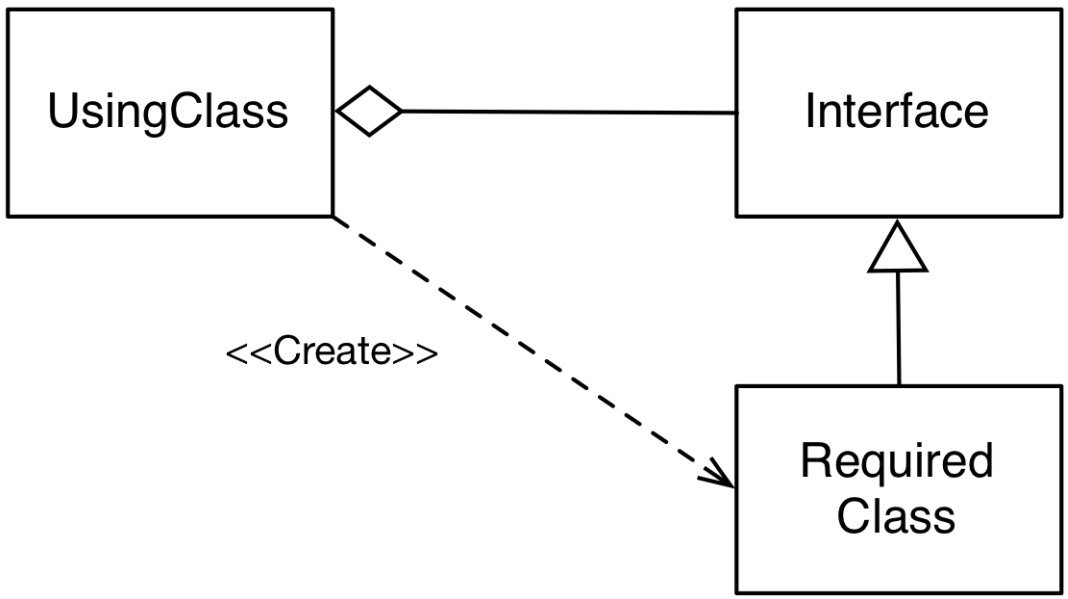
- Don't call us, we'll call you
- Switch from polling to event-driven software

Example

Observer and observable with callback



- Even though there is no direct dependency from `UsingClass` to `RequiredClass` there is a ”Create” dependency
- It would be better to transfer responsibility for creating `RequiredClass` to a special instance



Dependency lookup

Dependency look-up means that an object looks up its association with another object during runtime by looking for this object at pre-agreed places.

Possible implementations are

- Objects are kept in a *central class* that can be accessed by all objects
- Objects register themselves in a *registry* where anybody looking for an object can ask for it
- Objects are kept in a *container* that is filled with objects during system initialization or runtime

Dependency look-up with registry
IDatenquelle.java


```
1 // Datei: IDatenquelle.java
2 import java.awt.Point;
3 import java.util.ArrayList;
4
5 public interface IDatenquelle {
6
7     public ArrayList<Point> getDatenreihe();
8 }
```

Dependency look-up with registry
DatenquelleImp.java


```
1  // Datei: DatenquelleImpl.java
2  import java.awt.Point;
3  import java.util.ArrayList;
4
5  public class DatenquelleImpl implements IDatenquelle {
6
7      @Override
8      public ArrayList<Point> getDatenreihe() {
9          ArrayList<Point> datenreihe = new ArrayList<Point>();
10         int anzPunkte = (int) (Math.random() * 30) + 2;
11         int startPunkt = -1 * (int) (Math.random() * 30);
12
13         for (int x = startPunkt; x < anzPunkte; x++) {
14             datenreihe.add(new Point(x, (int) (Math.random() * 20)
15                                      - (int) (Math.random() * 20)));
16         }
17
18         return datenreihe;
19     }
20 }
```

Dependency look-up with registry
Registratur.java lines 1-18


```
1  // Datei: Registratur.java
2  import java.util.HashMap;
3  /*
4   *  Die Klasse Registratur kann beliebig viele Komponenten enthalten.
5   *  Jede Komponente wird durch einen String-Schluessel registriert
6   *  und kann durch diesen angefordert werden.
7   */
8
9  public class Registratur {
10
11     private HashMap<String, Object> komponenten
12                         = new HashMap<String, Object>();
13     private static Registratur registratur = null;
14
15     private Registratur() {
16     }
17
18     ;
```

Dependency look-up with registry
Registratur.java lines 19-35


```
19
20      public static Registratur getRegistratur() {
21          if (registratur == null) {
22              registratur = new Registratur();
23          }
24          return registratur;
25      }
26
27      public Object getKomponente(String komponentenBezeichnung) {
28          return komponenten.get(komponentenBezeichnung);
29      }
30
31      public void registriereKomponente(String bezeichnung,
32                                        Object komponente) {
33          komponenten.put(bezeichnung, komponente);
34      }
35  }
```

Dependency look-up with registry
Lines 1–20


```
1  import java.awt.Color;
2  import java.awt.Dimension;
3  import java.awt.Graphics;
4  import java.awt.Point;
5  import java.util.ArrayList;
6
7  import javax.swing.JFrame;
8  import javax.swing.JPanel;
9
10 public class Plotter extends JPanel {
11
12     private int plotterLength = 1200, plotterHeight = 600;
13
14     private JFrame frame = new JFrame();
15
16     private IDatenquelle datenquelle;
17
18     public Plotter() {
19         frame.setSize(plotterLength, plotterHeight);
20         frame.setResizable(false);
```

```
21          frame.setDefaultCloseOperation(
22                  JFrame.EXIT_ON_CLOSE);
23          frame.add(this);
24      }
25
26      public void plot() {
27          // Datenquelle von der Registratur anfordern
28          Registratur registratur = Registratur
29                  .getRegistratur();
30          datenquelle = (IDatenquelle) registratur
31                  .getKomponente("Datenquelle");
32
33          // Wenn keine Datenquelle in der Registratur
34          // vorhanden ist, beende den Plotvorgang
35          if (datenquelle == null)
36              return;
37          frame.setVisible(true);
38
39          // die fuer JFrame angegebene Groesse bezieht sich
40          // auf die vom Betriebssystem zugewiesene Fenster-
```

Dependency look-up with registry
Lines 41–60


```
41          // groesse. Daher wird die tatsaechliche Plotter-
42          // groesse erst nachtraeglich gesetzt.
43          Dimension tatsaechlicheGroesse = frame
44                  .getContentPane().getSize();
45          plotterLength = (int) tatsaechlicheGroesse
46                  .getWidth();
47          plotterHeight = (int) tatsaechlicheGroesse
48                  .getHeight();
49      }
50
51      @Override
52      protected void paintComponent(Graphics g) {
53          super.paintComponent(g);
54          ArrayList<Point> datenreihe = datenquelle
55                  .getDatenreihe();
56
57          // Wenn weniger als zwei Punkte in Datenreihe,
58          // verlasse Funktion
59          if (datenreihe == null || datenreihe.size() <= 1)
60              return;
```

Dependency look-up with registry
Lines 61–80


```
61
62          // Minimum und Maximum-Punkte der x- und y-Reihe
63          // bestimmen und sichern
64          Point minPunkte = getMinPunkte(datenreihe);
65          Point maxPunkte = getMaxPunkte(datenreihe);
66
67          // Falls es keine Steigung gibt, vergroessere
68          // den y-Bereich des Plotters
69          if (minPunkte.y == maxPunkte.y) {
70              minPunkte.y -= 1;
71              maxPunkte.y += 1;
72          }
73
74          // Berechne einen Offset um Ueberschneidungen der
75          // Axenbeschriftung
76          // zu verhindern.
77          int xOffset = (int) (plotterLength * 0.05) / 2;
78          int yOffset = (int) (plotterHeight * 0.05) / 2;
79
80          // Berechne Pixel pro Einheit, um die Position der
```

Dependency look-up with registry
Lines 81–100


```
81          // Datenpunkte bestimmen zu koennen
82          Point pixelProEinheit = new Point(
83                  (int) (plotterLength
84                          / (maxPunkte.x - minPunkte.x)),
85                  (int) (plotterHeight
86                          / (maxPunkte.y - minPunkte.y)));
87
88          for (int i = 0; i < datenreihe.size(); i++) {
89              // Berechne Positionkoordinaten des aktuellen
90              // Punktes auf der Plotteroberflaeche
91              Point aktuellerPunkt = new Point(
92                      (int) ((datenreihe.get(i).x
93                              - minPunkte.x)
94                              * pixelProEinheit.x * 0.95)
95                              + xOffset,
96                      (int) (plotterHeight
97                              - (datenreihe.get(i).y
98                                      - minPunkte.y)
99                                      * pixelProEinheit.y
100                                     * 0.95)
```

Dependency look-up with registry
Lines 101–120


```
101                         - yOffset);
102
103             // Linie zwischen Punkt und nachfolgenden Punkt
104             // einzeichnen
105             if (i < (datenreihe.size() - 1)) {
106                 g.setColor(Color.RED);
107                 g.drawLine(aktuellerPunkt.x,
108                         aktuellerPunkt.y,
109                         (int) ((datenreihe.get(i + 1).x
110                                 - minPunkte.x)
111                                 * pixelProEinheit.x * 0.95)
112                                 + xOffset,
113                         (int) (plotterHeight
114                                 - (datenreihe.get(i + 1).y
115                                         - minPunkte.y)
116                                         * pixelProEinheit.y
117                                         * 0.95)
118                                 - yOffset);
119             }
```

Dependency look-up with registry
Lines 121–140


```
121             // Schwarz als Beschriftungsfarbe auswaehlen
122             g.setColor(Color.BLACK);
123
124             // Beschriftung der y-Achsenpunkte einzeichnen
125             g.drawLine(0, aktuellerPunkt.y, 3,
126                     aktuellerPunkt.y);
127             g.drawString("" + datenreihe.get(i).y, 6,
128                     aktuellerPunkt.y - 2);
129
130             // Beschriftung der x-Achsenpunkte einzeichnen
131             g.drawLine(aktuellerPunkt.x, plotterHeight,
132                     aktuellerPunkt.x, plotterHeight - 4);
133             g.drawString("" + datenreihe.get(i).x,
134                     aktuellerPunkt.x + 3,
135                     plotterHeight - 4);
136
137             // Punkt markieren
138             g.drawOval(aktuellerPunkt.x - 3,
139                     aktuellerPunkt.y - 3, 6, 6);
140             ;
```

Dependency look-up with registry
Lines 141–160


```
141         }
142     }
143
144     Point getMaxPunkte(ArrayList<Point> datenreihe) {
145         Point maxPunkte = new Point(Integer.MIN_VALUE,
146                 Integer.MIN_VALUE);
147         for (Point datenpunkt : datenreihe) {
148             if (datenpunkt.x > maxPunkte.x)
149                 maxPunkte.x = datenpunkt.x;
150             if (datenpunkt.y > maxPunkte.y)
151                 maxPunkte.y = datenpunkt.y;
152         }
153         return maxPunkte;
154     }
155
156     Point getMinPunkte(ArrayList<Point> datenreihe) {
157         Point minPunkte = new Point(Integer.MAX_VALUE,
158                 Integer.MAX_VALUE);
159         for (Point datenpunkt : datenreihe) {
160             if (datenpunkt.x < minPunkte.x)
```

Dependency look-up with registry
Lines 161–180


```
161                     minPunkte.x = datenpunkt.x;
162                 if (datenpunkt.y < minPunkte.y)
163                     minPunkte.y = datenpunkt.y;
164             }
165             return minPunkte;
166         }
167
168 }
```

Dependency look-up with registry
TestPlotter.java


```
1  // Datei: TestPlotter.java
2  public class TestPlotter {
3
4      static public void main(String[] args) {
5          // Bekomme Referenz auf die Registratur
6          Registratur registry = Registratur.getRegistratur();
7
8          // Registriere eine Datenquelle als Komponente
9          registry.registriereKomponente("Datenquelle", new
               DatenquelleImpl());
10
11         // plotten ...
12         Plotter plotter = new Plotter();
13         plotter.plot();
14     }
15 }
```

Dependency injection

Dependency injection means that associations between objects are created by an injector during runtime. An object does not actively look for its dependencies; rather they are injected.

- Constructor injection: dependencies are passed as parameters to the constructor
- Setter injection: dependencies are being added during runtime via set methods
- Interface injection: dependencies are being added via interfaces

Example

Dependency injection

```
1  public class MyDao {
2
3    protected IDataSource dataSource =
4    new DataSourceImpl("driver", "url", "user", "password");
5
6    //data access methods...
7    public Person readPerson(int primaryKey) {...}
8
9  }
```

Example

Dependency injection

```
1  public class MyDao {
2
3    protected IDataSource dataSource = null;
4
5    public MyDao(String driver, String url, String user, String
         password){
6      this.dataSource = new DataSourceImpl(driver, url, user,
           password);
7    }
8
9
10   //data access methods...
11   public Person readPerson(int primaryKey) {...}
12
13 }
```

Example

Dependency injection

```
1   public class MyDao {
2
3     protected IDataSource dataSource = null;
4
5     public MyDao(IDataSource dataSource){
6       this.dataSource = dataSource;
7     }
8
9
10    //data access methods...
11    public Person readPerson(int primaryKey) {...}
12
13  }
```

Example

Dependency injection

```
1   public class MyBizComponent{
2     public void changePersonStatus(Person person, String status){
3
4         MyDao dao = new MyDao(
5               new DataSourceImpl("driver", "url", "user", "password"));
6
7         Person person = dao.readPerson(person.getId());
8         person.setStatus(status);
9         dao.update(person);
10    }
11  }
```

Example

Dependency injection

```
1   public class MyBizComponent{
2
3     protected MyDao dao = null;
4
5     public MyBizComponent(MyDao dao){
6        this.dao = dao;
7     }
8
9
10    public void changePersonStatus(Person person, String status){
11       Person person = dao.readPerson(person.getId());
12       person.setStatus(status);
13       dao.update(person);
14    }
15  }
```

Points to remember