

GRUNDLAGEN DER EVENT-BASIERTEN PROGRAMMIERUNG

Eingaben von NutzerInnen abfangen und verwenden

Struktur und Inhalt des Kurses wurden 2012 von Markus Heckner entwickelt. Im Anschluss haben Alexander Bazo und Christian Wolff Änderungen am Material vorgenommen. Die aktuellen Folien wurden von Alexander Bazo erstellt und können unter der [MIT-Lizenz](#) verwendet werden.

AKTUELLER SEMESTERFORTSCHRITT (WOCHE 6)

Kursabschnitt	Themen		
Grundlagen	Einführung	Einfache Programme erstellen	Variablen, Klassen & Objekte
	Kontrollstrukturen & Methoden	Arrays & komplexe Schleife	
Klassenmodellierung	Grundlagen der Klassenmodellierung		
	Vererbung & Sichtbarkeit		
Interaktive Anwendungen	Event-basierten Programmierung		String- & Textverarbeitung
Datenstrukturen	Listen, Maps & die Collections		
	Speicherverwaltung		
	Umgang mit Dateien		
Software Engineering	Planhaftes Vorgehen bei der Softwareentwicklung		
	Qualitätsaspekte von Quellcode		

PINGO-QUIZ



<https://pingo.coactum.de/365849> 

DAS PROGRAMM FÜR HEUTE

- Nächste Studienleistung
- Ereignis-basierte Anwendungen: Auf Maus und Tastatur reagieren
- *Interfaces* und das *Observer*-Muster

DIE NÄCHSTE STUDIENLEISTUNG

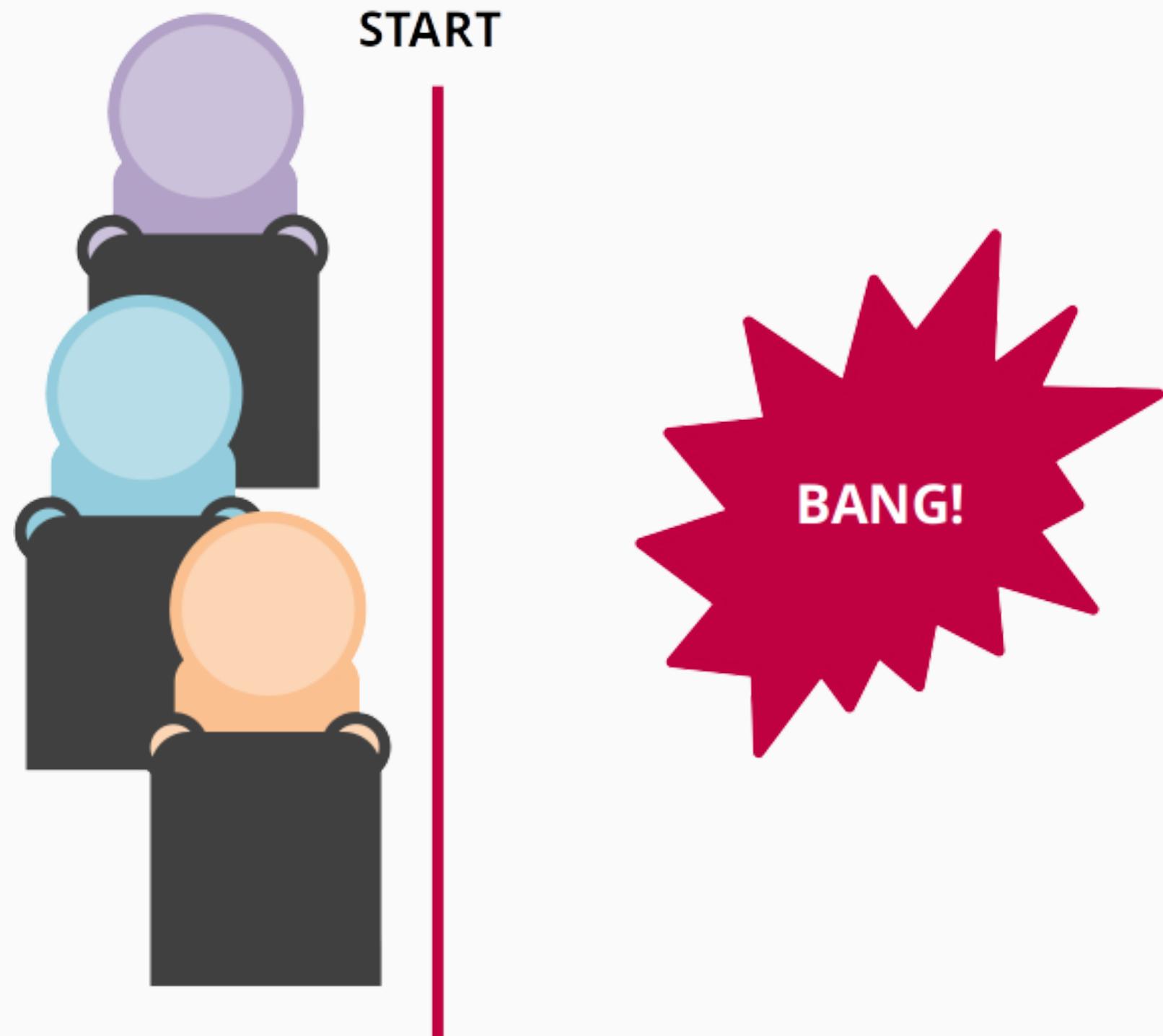
- Sie implementieren ein ersten, interaktives Spiel und setzen dabei das bekannte *Memory*-Konzept um
- Sie stellen ein Spielbrett mit mehreren Karten dar und erlauben den NutzerInnen, einzelne Karten anzuklicken
- Sie arbeiten *Event*-basiert und reagieren auf die Eingaben der NutzerInnen
- Handout und Code-Vorgabe finden Sie ab heute Abend im GRIPS-Kurs
- Abgabedatum ist der 15. Dezember

EVENT-DRIVEN PROGRAMMING

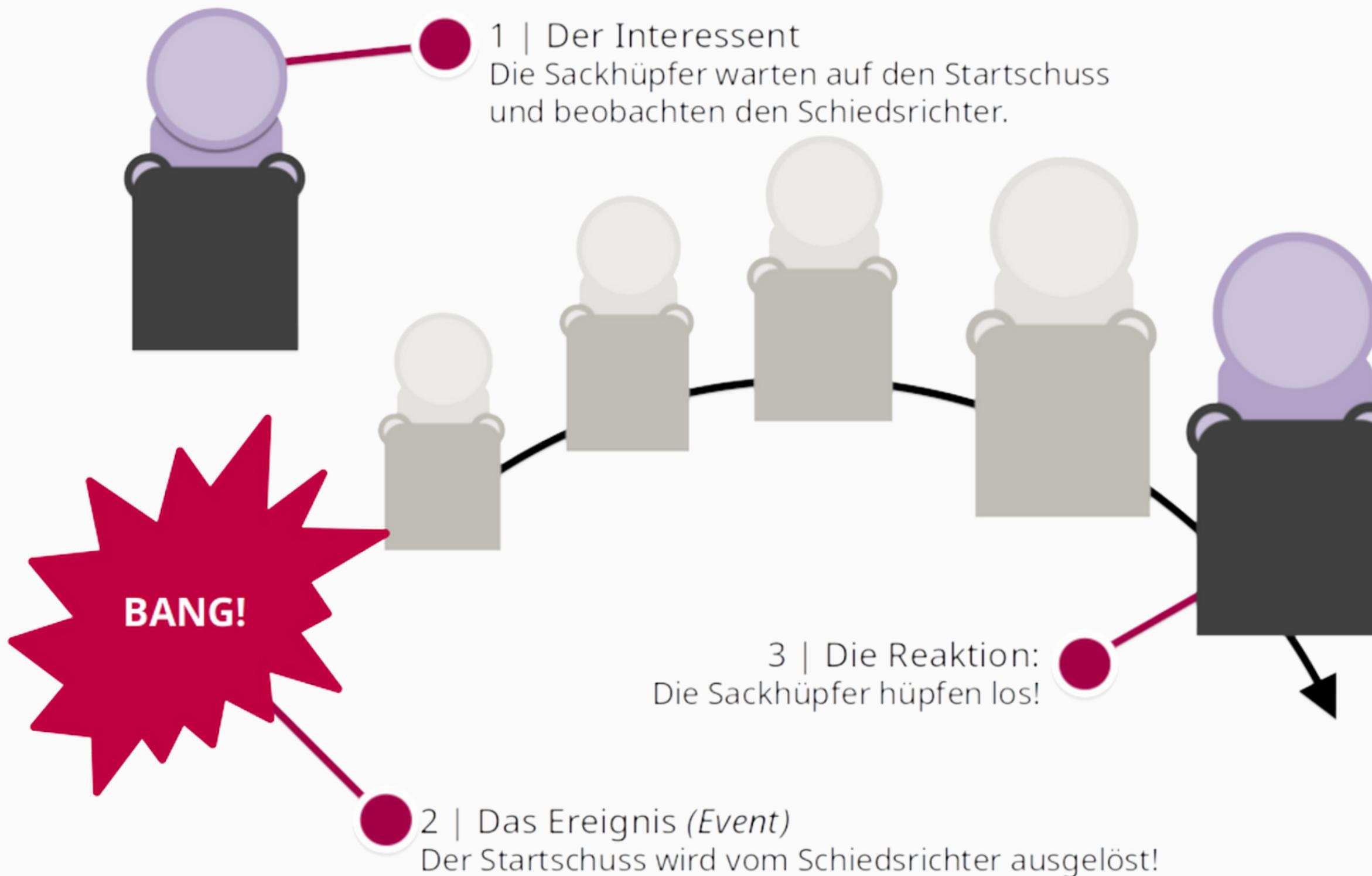
In computer programming, event-driven programming is a programming paradigm in which the flow of the program is determined by events such as user actions (mouse clicks, key presses), sensor outputs, or messages from other programs or threads. Event-driven programming is the dominant paradigm used in graphical user interfaces and other applications (e.g., JavaScript web applications) that are centered on performing certain actions in response to user input. This is also true of programming for device drivers (e.g., P in USB device driver stacks).

Quelle: Wikipedia (https://en.wikipedia.org/wiki/Event-driven_programming)

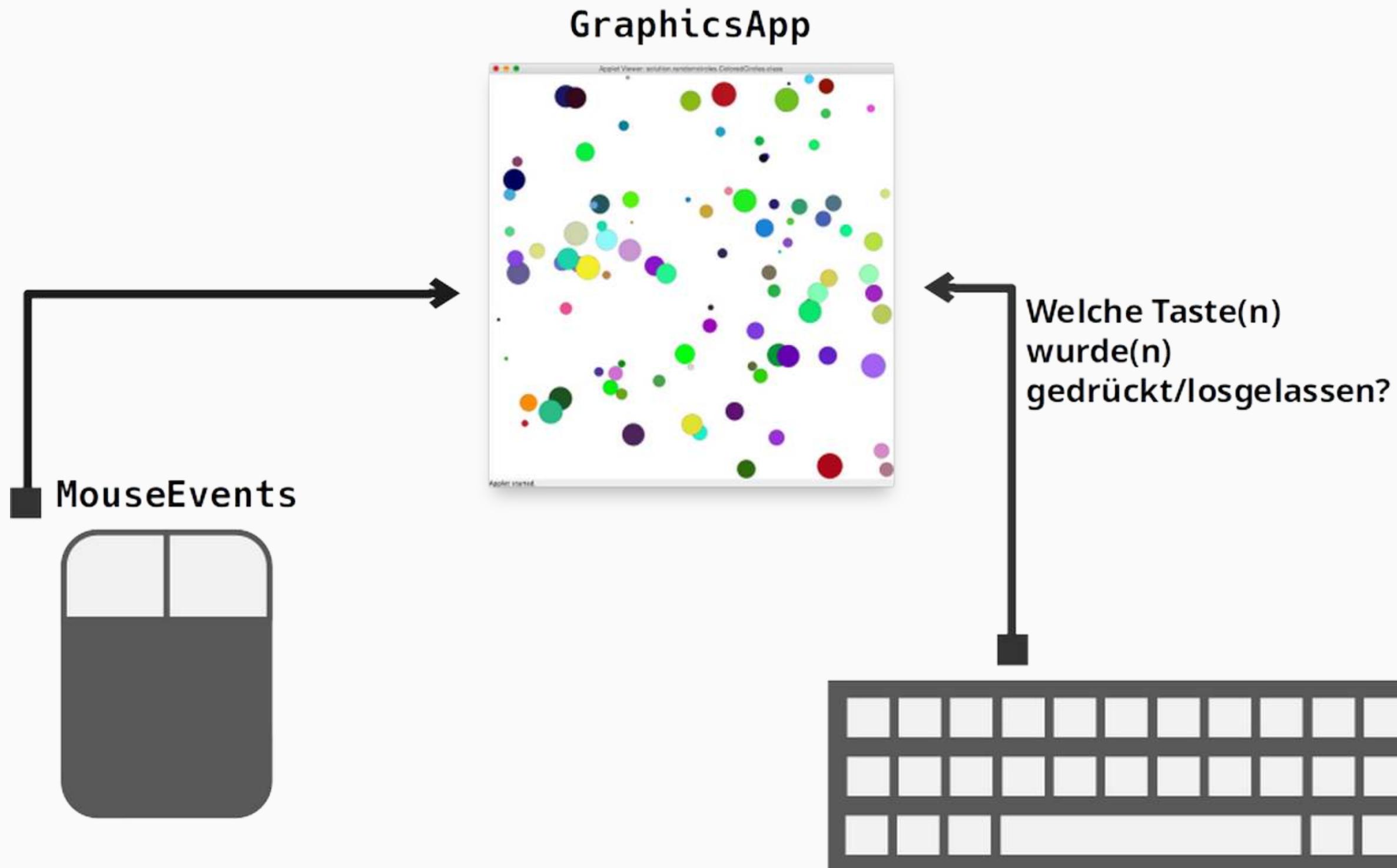
EVENTS (EREIGNISSE) IN DER REALEN WELT



EVENTS IN DER REALEN WELT: DETAILS



EVENTS IN DER GRAPHICSAPP (1/2)



EVENTS IN DER GRAPHICSAPP (2/2)

- *UI Events (User Interface Events)* werden durch z.B. die Verwendung der Tastatur oder Maus durch BenutzerInnen erzeugt.
- Events werden bei jeder Interaktion mit Maus und Tastatur ausgelöst: Damit eine sinnvolle Reaktion erfolgt, müssen die Events gezielt aber *abgefangen*, interpretiert und verarbeitet werden.
- Ein Objekt, das auf bestimmte *Events* wartet und auf diese reagiert, nennt man *Observer* oder *Listener*.

EVENTS ABFANGEN

Die interne Verarbeitung der *Events* (Abfangen der Eingabe, Erzeugen der *Events*, Weitergabe der *Events* an unsere *GraphicsApp*) wird von den *Implementors* der *GraphicsApp*-Umgebung durchgeführt. Wir müssen in unseren Anwendungen "nur noch" reagieren, d. h. die Ereignisse abfangen und in den Ablauf unserer Programme einbauen.

EVENT-BASIERTER PROGRAMMIERUNG

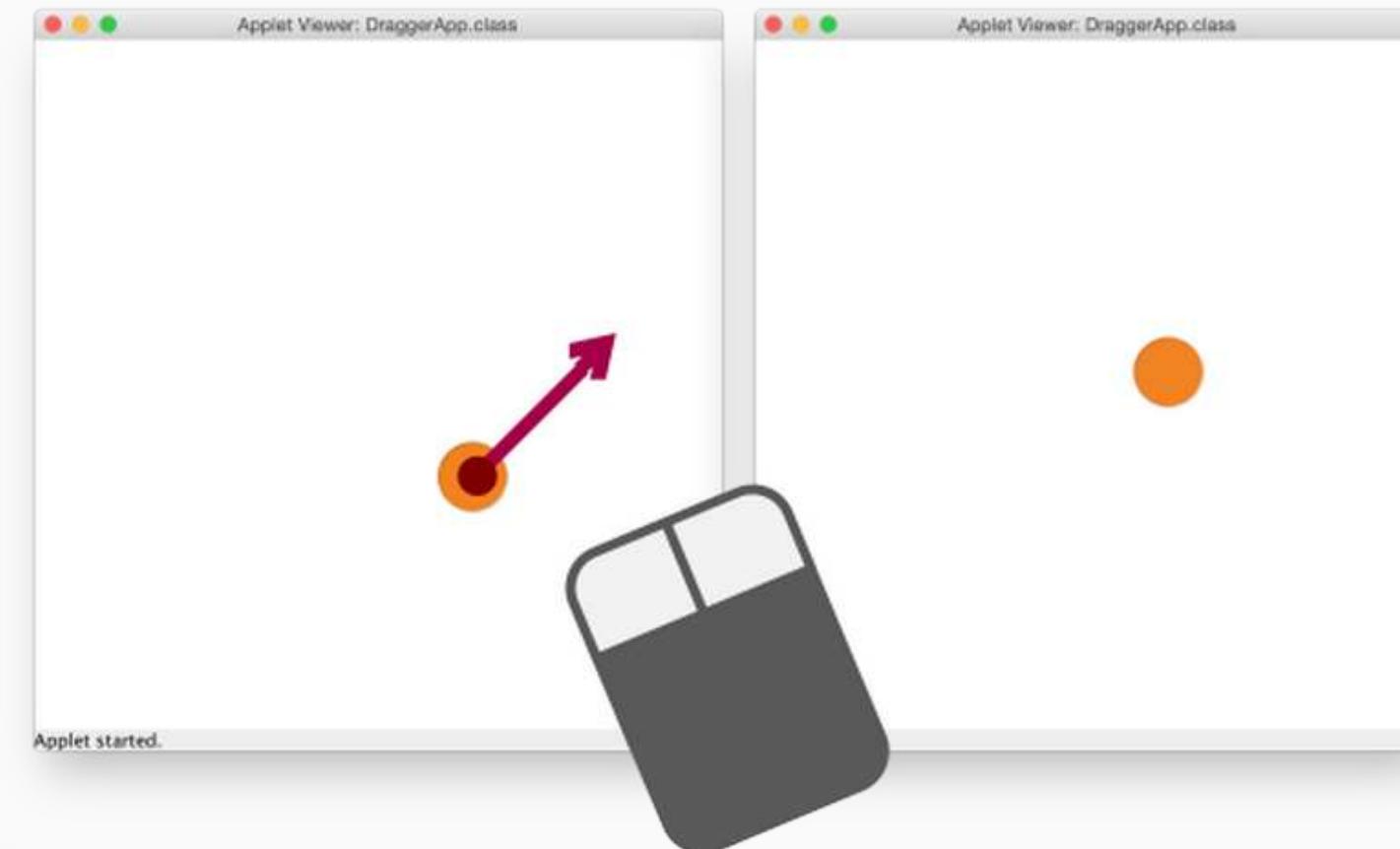
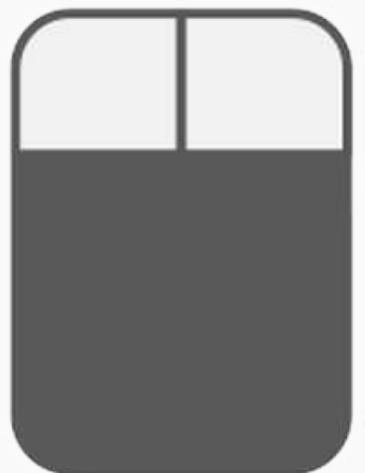
Event im Code abfangen und verarbeiten

```
@Override  
public void mouseDragged(MouseEvent e) {  
    moveEllipse(e.getX(), e.getY());  
}
```



Event und dessen Eigenschaften nutzen, um z.B. Elemente neu zu positionieren: Position der Ellipse mit neuer Mausposition aktualisieren (setPosition) und neu zeichnen (draw).

MouseEvents



AUF EVENTS REAGIEREN (1/2)

Hinweis: Die *Event*-basierte Programmierung erfordert eine etwas andere Denkweise vom Programmierenden. Bis jetzt sind wir an die Entwicklung unserer Anwendungen mit folgendem (zeitlichen) Plan herangetreten

Bouncer: Sobald das Programm startet, beginne mit diesem Befehl. Direkt danach führe diese Aufgabe aus und dann diese und dann diese ...

GraphicsApp: Jedes Mal, wenn die draw-Methode aufgerufen wird, erledige zu erst diese Aufgabe und dann diese und dann diese ...

Wir hatten eine klare Vorstellung in welcher (zeitlichen) Reihenfolge unser Programm abläuft und was wann passiert: *Wenn der Fall eintritt, dass dieses Ereignis ausgelöst wird, dann mach zuerst das hier und dann das hier. Sobald du damit fertig bist, fahre mit dem eigentlichen Ablauf des Programms fort.*

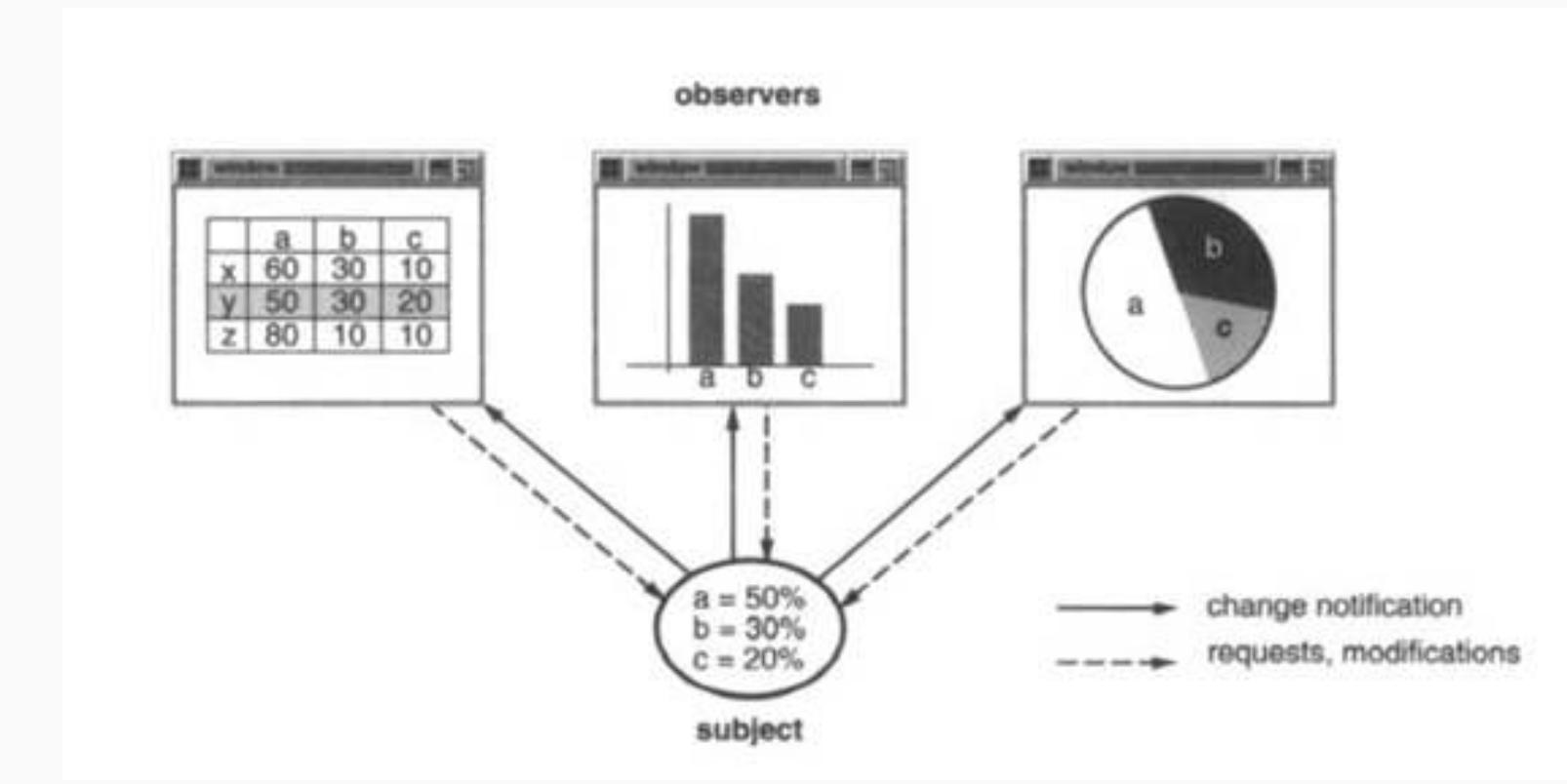
AUF EVENTS REAGIEREN (2/2)

Wenn wir *Event*-basiert programmieren, legen wir uns einen Plan zurecht, der ausgeführt werden soll, wenn ein bestimmter Zustand erreicht wird bzw. ein bestimmtes *Event* eintritt. Wann das der Fall ist, können wir in der Regel nicht genau bestimmen (Unsere BenutzerInnen können z.B. nach einer, zehn oder erst nach sechzig Sekunden auf die Maustaste drücken ...).

DAS OBSERVER-MUSTER (1/3)

Beim Entwickeln unterschiedlicher Software müssen häufig die selben, grundlegenden Probleme gelöst werden. Für viele dieser Aufgaben haben sich *Best Practices* ergeben, die in der Literatur oft als *Design Patterns* oder Muster beschrieben werden (Ein *Pattern* ist dabei mehr als nur die Lösung selbst). In der GraphicsApp wird ein bestimmtes Entwurfsmuster (*Design Pattern*) verwendet, um die *Event*-Verarbeitung zu implementieren: Das *Observer Pattern*.

DAS OBSERVER-MUSTER (2/3)



Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.

Quelle: Gamma et al. Design Patterns: Elements of Reusable Object-oriented Software

DAS OBSERVER-MUSTER (3/3)

In Java wird das Informieren der konkreten *Observer* durch den Aufruf öffentlicher Methoden realisiert. Damit das funktioniert, müssen alle *Observer* über entsprechende Methoden verfügen. Es muss sichergestellt werden, dass das *Subject* (oder *Observable*) mit dem *Observer* sprechen kann. Aber: Es können sich ganz unterschiedliche Klassen als *Observer* für ein bestimmtes *Observable* interessieren!

WIE KOMMUNIZIERT EINE *OBSERVABLE*-KLASSE MIT DEN *OBSERVER*-KLASSEN, DIE SICH FÜR DIE EREIGNISSE INTERESSIEREN?

- Klassen kommunizieren untereinander über öffentliche Methoden (Vgl. Student-Klasse). Alle Empfänger brauchen passende Methoden, über die die Events weitergegeben werden können: mouseMoved, mouseDragged, mouseClicked, ...
- Wie stellen wir sicher, dass der Empfänger über diese Methoden verfügt? Ist Vererbung hier die richtige Entscheidung?
- *Interfaces* bieten in JAVA die Möglichkeit, eine Reihe an Methoden zu definieren, die eine Klasse auf jeden Fall haben muss, ohne dass die tatsächliche Implementierung der Methoden vorgegeben ist.

PROBLEME MIT VERERBUNG

Hinweis: Häufig stoßen Sie beim Modellieren Ihrer Klassen auf Gemeinsamkeiten, die sich nicht sinnvoll in einer Vererbungshierarchie abbilden lassen – vor allem bei der Integration von Events oder der Gruppierung von Objekten.

Beispiele: Ein Videospiel, mit unterschiedlichen Gegenständen, die alle auf Wind reagieren



Eine Windmühle,
deren Flügel sich bei Wind
drehen und die dadurch Mehl
produziert.



Ein Segelschiff,
dessen Segel sich bei Wind füllt
und dadurch das Schiff bewegt.



Ein *Tumbleweed*,
das sich bei Wind durch die
Wüste bewegen lässt.

Wenn wir diese Dinge als Klassen abbilden,
konkurrieren wahrscheinlich unterschiedliche
Superklassen miteinander: Die Windmühle ist ein
Gebäude, das Segelschiff ist ein Fahrzeug und
das *Tumbleweed* ist/war eine Pflanze. Die
Reaktion auf den Wind ist nur ein kleiner Teil des
Verhaltens der Klassen.

**Trotzdem wollen wir die Klassen innerhalb
unserer Anwendung hinsichtlich dieses
Verhaltens gleich behandeln!**

WEITERGABE VON EVENTS

Unser Problem: Unterschiedliche Klassen bzw. deren Instanzen interessieren sich für ein und dasselbe Ereignis. Die Weitergabe dieser Ereignis muss über die öffentliche Schnittstelle der Klassen (public-Methoden) erfolgen.

Jede der Klassen muss eine eigene Methode für die Weitergabe des Events bereitstellen. Jede Methode muss einzeln aufgerufen werden. Jedes neue Element, das ebenfalls auf Wind reagiert, muss entsprechend implementiert werden.

Lösung: Idee: Wir wollen alle relevanten Objekte, unabhängig von ihrer eigentlichen Klasse und nur im Kontext des *Wind*-Ereignis als Objekte gleichen Typs, mit gleicher öffentlicher Schnittstelle, behandeln, um alle interessierten Elemente auf die gleiche Art und Weise über das Ereignis zu informieren. Dafür setzen wir in Java *Interfaces* ein!

INTERFACES IN JAVA

1. Interfaces werden wie Klassen in Dateien mit der Endung .java definiert. Das *Keyword* lautet `interface` (Vgl.: `class` für Klassen)
2. Interfaces bestehen aus einer Liste von öffentlichen Konstanten und Methodensignaturen (!).
3. Bei der Definition einer Klasse können über das `implements`-Schlüsselwort ein oder mehrere Interfaces angeben werden: Die Klasse *implementiert* dann diese Interfaces.
4. Implementiert eine Klasse ein Interface, müssen innerhalb der Klasse alle im Interface enthaltende Methoden mit der korrekten Signatur und einem frei wählbaren Rumpf definiert werden.
5. Implementiert eine Klasse ein Interface, können Instanzen der Klasse als Instanzen der Interface behandelt werden. D.h., im Kontext der Instanz kann, alternativ zu eigentlichen Klasse, auch das Interface als Datentyp genutzt werden.

BEISPIEL: CLEANABLE (1/2)

Das Interface Cleanable gibt vor, über welche öffentlichen Methoden alle Dinge, die *putzbar* sind, angesprochen werden sollen.

```
1 public interface Cleanable {  
2     public void clean();  
3 }
```

```
1 public class Window implements Cleanable {  
2     public void open() {  
3     }  
4     public void clean() {  
5         // clean window  
6     }  
7 }  
8 public class Cat implements Cleanable {  
9     public void drive() {  
10    }  
11    public void clean() {  
12        // clean tooth  
13    }  
14 }
```

BEISPIEL: CLEANABLE (2/2)

Die Klassen Window und Car haben unterschiedliche Aufgaben, beide repräsentieren aber *putzbare* Dinge. Beide implementieren das Interface, um im Kontext des Putzens gleich angesprochen werden zu können. Was genau beim Putzen passiert, unterscheidet sich aber bei Window und Car (Unterschied zwischen Signatur und Rumpf der clean-Methode beachten):

```
1 Cleanable[] cleanables = new Cleanable[2];
2 cleanables[0] = new Window();
3 cleanables[1] = new Tooth();
4 for(int i = 0; i < cleanables.length; i++) {
5     cleanables[i].clean();
6 }
```

BEISPIEL: EATABLE

```
1 public interface Eatable {  
2     public void eat();  
3 }  
4  
5 public class Apple implements Eatable {  
6     public void eat() {  
7         // Specific behaviour  
8     }  
9 }
```

Hinweis: Interfaces werden, wie Klassen, in separaten Dateien definiert. Klassen, die ein Interface implementieren (`implements`), müssen zwangsläufig alle Methoden implementieren, die das Interface vorgibt (Vertrag). Dabei wird nur die Signatur vorgeschrieben, die eigentliche Implementierung bleibt der Klasse überlassen. Eine Klasse kann mehrere Interfaces implementieren.

WARUM WIR INTERFACES VERWENDEN MÜSSEN

- Oft verfügen Klassen über ähnliche Aufgaben oder Schnittstellen, ohne dass diese (sinnvoll) in einer Vererbungshierarchie abgebildet werden können.
- Klassen können aber nur von einer Superklasse abgeleitet werden. Über Interfaces lässt sich die Vererbungshierarchie *aufbrechen*.
- Interfaces definieren eine minimale öffentliche Schnittstelle, über die andere Komponenten auf die implementierenden Klassen zugreifen können.
- Diese gleiche Schnittstelle ermöglicht es Objekte unterschiedlicher Klassen einheitlich zu behandeln, da ihre öffentliche Schnittstelle (teilweise) gleich ist.

INTERFACES IM JAVA JDK

- Die Schnittstelle `Cloneable` (Paket `java.lang`) signalisiert, dass von Objekten einer Klasse, die `Cloneable` implementiert, Kopien erstellt werden können,
- Die Schnittstelle `Serializable` (Paket `java.io`) signalisiert, dass Objekte einer Klasse, die `Serializable` implementiert, serialisiert, d. h. persistent "gespeichert" werden können (z. B. in einer Datei)
- Die Implementierung der Schnittstelle `Comparable` (Paket `java.lang`) macht Objekte durch die Implementierung der `compareTo`-Methode mit anderen Objekten gleichen Typs vergleichbar (Anwendung bei Suchen und Sortieren, z. B. über die Hilfsklasse `java.util.Arrays`).

INTERFACES IN DER GRAPHICSAPP

SCALABLE

Line, Image, Ellipse und Rect implementieren das Scalable-Interface. Alle Objekte sind dadurch über die scale-Methode skalierbar. Die eigentliche Implementierung ist jeweils unterschiedlich, die öffentliche Interaktion ist aber die gleiche.

RESIZABLE

Ellipse, Image und Rect implementieren das Resizable-Interface. Die Größe von allen Objekten kann über die Methode setSize

UI-EVENTS IN DER GRAPHCISAPP

Die Superklasse GraphicsApp implementiert bereits die Interfaces GraphicsAppKeyListener und GraphicsAppMouseListener und wird in den entsprechenden Methoden über Eingabeereignisse informiert. Wir können diese Methoden überschreiben um in unseren Anwendungen individuell auf *Events* von Maus und Tastatur reagieren:

```
1 @Override
2 public void onKeyPressed(KeyEvent event) {
3     super.onKeyPressed(event);
4 }
5
6 @Override
7 public void onMouseClicked(MouseEvent event) {
8     updateCounter();
9 }
```

Die Parameter der Methoden beinhalten Informationen über das Ereignis:
Tasten, Position, ...

MAUS-EREIGNISSE

Über den Parameter vom Typ MouseEvent (bzw. den spezialisierten Klassen, z.B. MouseClickedEvent) kann in den Interface-Methoden auf die Eigenschaften des Events zugegriffen werden:

```
1 // mit: MouseEvent e  
2 int cursorX = e.getX();  
3 int cursorY = e.getY();  
4 int mouseButton = e.getButton();  
5 // ...
```

Hinweis: Sowohl die Mouse- als auch die Key-Methoden werden bei jedem Ereignis aufgerufen. Jeder Tastendruck und jede Mausbewegung, die innerhalb Ihrer Anwendung ausgeführt wird landet als Parameter in der entsprechenden Methoden. Für jedes Ereignis wird die jeweilige Methode einmal aufgerufen.

TASTATUR-EREIGNISSE

Über den Parameter vom Typ KeyEvent kann in den Interface-Methoden auf die Eigenschaften des Events zugegriffen werden:

```
1 // mit: KeyEvent e  
2 int keyCode = e.getKeyCode();  
3 // ...
```

Taste	Numerischer Code	Konstante
Pfeiltaste (oben)	38	GraphicsAppKeyEvent.VK_UP
Pfeiltaste (oben)	40	GraphicsAppKeyEvent.VK_DOWN
Leertaste	32	GraphicsAppKeyEvent.VK_SPACE
R	82	GraphicsAppKeyEvent.VK_R
G	71	GraphicsAppKeyEvent.VK_G
B	66	GraphicsAppKeyEvent.VK_B

IM CODE AUF EVENTS REAGIEREN (1/2)

```
1 public void onMouseMoved(MouseMovedEvent event) {  
2     // What happens when mouse is moved  
3     moveEllipse(e.getX(), e.getY());  
4 }
```

Hinweis: In den Methoden kann auf die Parameter des Events zugegriffen werden (Was ist passiert? Wo ist es passiert?). Diese Informationen werden in der Regel an eine andere Methode Ihrer *GraphicsApp* weitergegeben und dort verarbeitet (Das Objekt, das sich für die Informationen interessiert, ist auch für deren Verarbeitung verantwortlich).

IM CODE AUF EVENTS REAGIEREN (2/2)

```
1 public void onKeyPressed(KeyEvent event) {  
2     switch (event.getKeyCode()) {  
3         case KeyEvent.VK_0:  
4             // Key ,0‘ pressed  
5             break;  
6         case KeyEvent.VK_1:  
7             // Key ,1‘ pressed  
8             break;  
9         default:  
10            break;  
11    }  
12 }
```

Hinweis: Über die switch-Konstruktion können unterschiedliche Fälle (=Tasten) abgefragt werden; dazu nutzen wir die Konstanten aus der Event-Klasse. Je nach Taste kann ein anderes Verhalten (=Methode) ausgelöst werden.

ÜBERSICHT ÜBER VERWENDBARE EVENTS

Ereignis	Methode	Event-Objekt	Eigenschaften
Mausklick	onMouseClicked	MouseClickedEvent	x- und y-Koordinaten und gedrückte Maustaste
Mausbewegung	onMouseMoved	MouseMovedEvent	x- und y-Koordinaten
Taste wird gedrückt	onKeyPresses	KeyPressedEvent	Numerische ID der Taste und deren "Inhalt" als char
Taste wird losgelassen	onKeyReleased	KeyReleasedEvent	Numerische ID der Taste und deren "Inhalt" als char

ZUSAMMENFASSUNG

- Interfaces definieren verpflichtende Methodensignaturen für alle Klassen, die das Interface implementieren (vgl. Listener in der GraphicsApp)
- Eventbasierte Anwendungen erlauben es Anwendungen, (asynchron) auf (Nutzer-)Eingaben zu reagieren
- Für Listener in der GraphicsApp: Die jeweiligen Interface-Methoden werden bei Eintritt eines Ereignisses aufgerufen, aus dem übergebenen Event-Objekt lassen sich Details zum Ereignis auslesen (Mausposition, Code der gedrückten Taste, ...)

VIELEN DANK FÜR IHRE AUFMERKSAMKEIT. WENN SIE MÖCHTEN, SEHEN WIR UNS IM ANSCHLUSS IN DER ZENTRALÜBUNG!

Fragen oder Probleme? In allgemeinen Angelegenheiten und bei Fragen zur Vorlesung wenden Sie sich bitte an Alexander Bazo (alexander.bazo@ur.de). Bei organisatorischen Fragen zu den Studienleistungen und Übungsgruppen schreiben Sie bitte Florin Schwappach (florin.schwappach@ur.de). Bei inhaltlichen Fragen zu den Übungen, Beispielen und Studienleistungen schreiben Sie uns unter mioop@mailman.uni-regensburg.de.

QUELLEN

Eric S. Roberts, *The art and science of Java: an introduction to computer science, New international Edition*, 1. Ausgabe, Pearson, Harlow, UK, 2014.