

Einführung in die Informatik II Umgebungen für die Entwicklung von größeren Systemen

Prof. Bernd Brügge, Ph.D
Institut für Informatik
Technische Universität München

Sommersemester 2001

23. - 25. Juli 2001

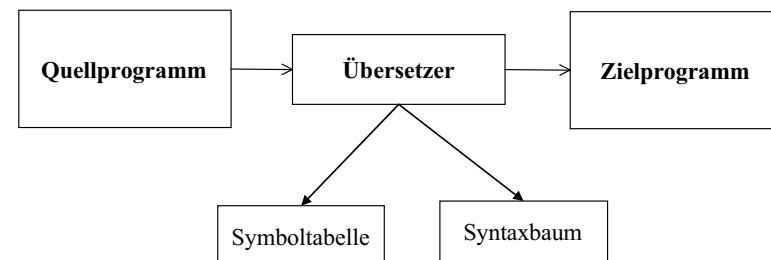
Überblick über die Vorlesung

- ❖ Interaktive Entwicklungsumgebungen
- ❖ Beispiel: Entwurf eines interaktiven Spiels
 - ◆ Threads und Synchronisation von Methodenaufrufen (kurz)
 - ◆ Entwicklung der graphischen Oberfläche eines Applets mit einem Paletteneditor
 - ◆ Das Beispiel stammt aus dem Buch:
S. Gilbert, B. McCarthy:
"Object-oriented Design in Java",
Mitchell Waite Series, 1998
Kapitel 9 (pp. 279-289):
Implementing Class Relationships: Inheritance and Interfaces

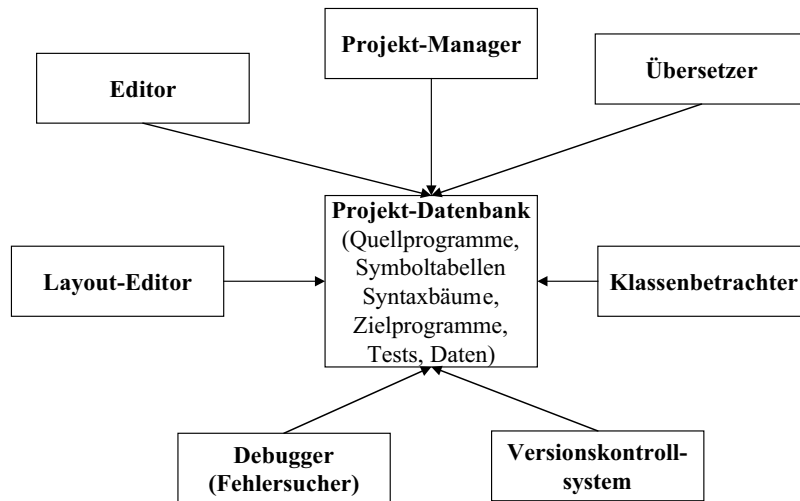
Entwicklungsumgebungen

- ❖ **Compiler** helfen beim Übersetzen eines Quellprogramm in ein Zielprogramm
- ❖ **Interaktive Entwicklungsumgebungen** sind wesentlich mächtiger.
 - ◆ Sie helfen beim Erstellen, Verwalten, Übersetzen und Testen, sowie bei der Versionskontrolle von Quell- und Zielprogrammen.
- ❖ Beispiele von Entwicklungsumgebungen: CodeWarrior, JBuilder.
- ❖ Viele Hilfsmittel zur Navigation: Beispiele
 - ◆ Zugriff auf alle im Programm definierten Bezeichner
 - ◆ Visualisierung der Klassen-Hierarchie: Klassen-Betrachter
- ❖ Werkzeuge für die Entwicklung von Applets ohne Codierung der interaktiven Komponenten:
 - ◆ Paletteneditor zur Kombination graphischer Komponenten

Architektur eines (einfachen) Compilers



Architektur von Entwicklungsumgebungen



Verwaltung von Quellprogrammen

- ❖ Zur Verwaltung aller Quellprogramme gibt es verschiedene Möglichkeiten.
- ❖ Makefile (Unix):
 - ◆ Auflistung aller Quellprogramme und deren Compilationsabhängigkeiten
 - ◆ Veränderung eines Quellprogramms induziert Übersetzung aller davon abhängigen Quellprogramme
- ❖ Projekt-Manager (Code-Warrior, JBuilder):
 - ◆ Visuelle Darstellung der Quellprogramme, Übersetzungsstatus, Formulierung von Untergruppen für bestimmte Tests und Versionen, Versionskontrolle.

Projekt-Manager (Code-Warrior)

Sicht aller Quell-Programme

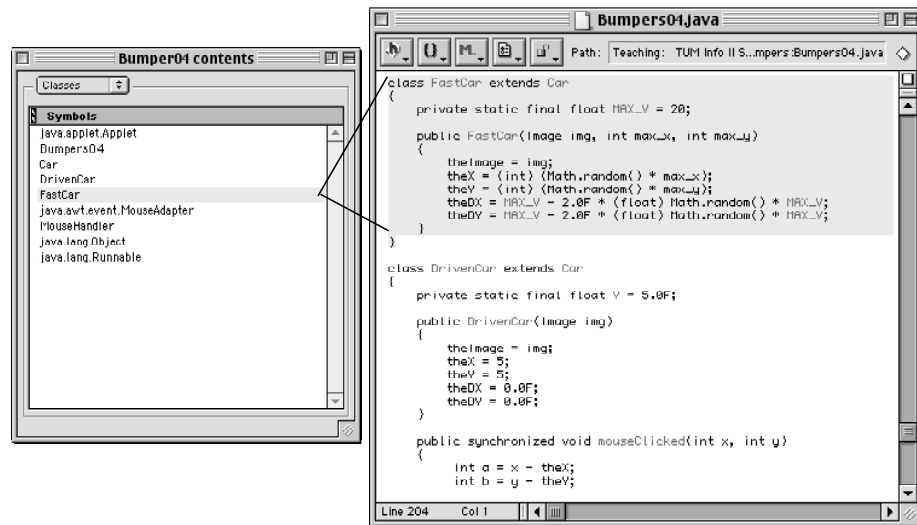
Sicht aller Versionen (targets)

Programme für Version Bumper04

Zugriff auf Bezeichner (Code-Warrior)

- Classes
- Constants
- Enums
- Functions
- Globals
- Macros
- Templates
- Typedefs

Zugriff auf Klassen-Definitionen (Code-Warrior)



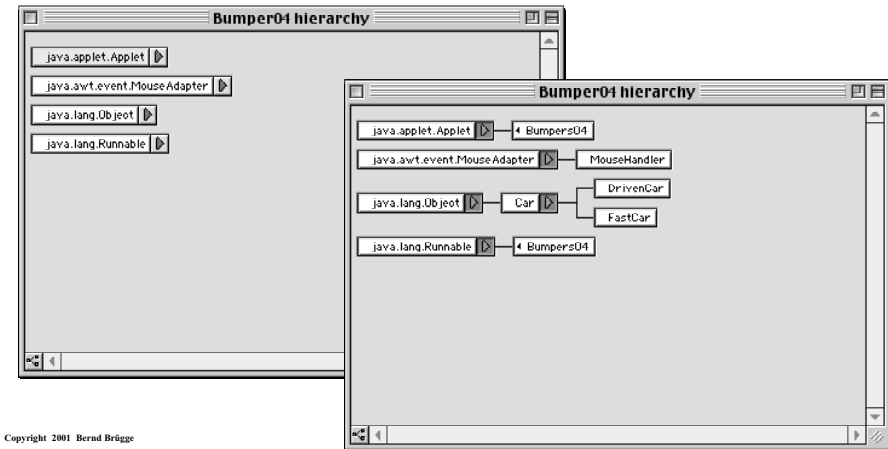
Copyright 2001 Bernd Brügge

Einführung in die Informatik II TUM Sommersemester 2001

13:52:29 9

Klassen-Betrachter (Code-Warrior)

- ❖ Der Klassen-Betrachter (class browser) erlaubt eine Visualisierung der Klassenhierarchie, wobei Teil-Hierarchien unterdrückt oder gezeigt werden können (Notation nicht in UML):



Copyright 2001 Bernd Brügge

Beispiel: Entwicklung eines interaktiven Spiels

- ❖ Auf einem rechteckigen Spielfeld fahren viele Autos. Einige sind schnell, einige sind langsamer (Die Anzahl der Autos ist beliebig, wird aber zur Compilationszeit festgelegt).
- ❖ Sie kontrollieren ein bestimmtes Auto mit der Maus.
- ❖ Die anfängliche Fahrtrichtung der Autos wird zufällig gewählt.
- ❖ Die Autos können miteinander und mit dem Spielrand kollidieren.
 - ◆ Bei einer Kollision zwischen Autos gibt es einen Gewinner und einen Verlierer. Der Verlierer bleibt stehen und scheidet aus.
 - ◆ Wenn ein Auto mit dem Spielfeldrand kollidiert, wird es abhängig von seiner Geschwindigkeit und Richtung wieder auf das Spielfeld zurückgeworfen.
- ❖ **Das Ziel:** Werfen Sie mit ihrem Auto so viele Autos wie möglich aus dem Rennen, bevor Ihr Auto aus dem Rennen geworfen wird.

Copyright 2001 Bernd Brügge

Einführung in die Informatik II TUM Sommersemester 2001

13:52:29 11

Anwendungsfälle

- ❖ Erzeugung des Spielfeldes
 - ◆ Unterlegung von Musik für das Spielfeld
- ❖ Auffüllen mit n Autos, wobei zwischen langsamen, schnellen und Benutzer-geführten Autos unterschieden werden muss
- ❖ Auswählen einer Anfangsposition für ein Auto
- ❖ Fahren eines Autos mit gleichmäßiger Geschwindigkeit
- ❖ Überprüfung von Kollisionen zwischen Autos
 - ◆ Audio-Untermalung für jede Kollision
- ❖ Überprüfung von Kollisionen von Autos mit dem Spielrand
- ❖ Mausklicks zur Änderung der Richtung eines Autos
 - ◆ Das Auto bewegt sich mit konstanter Geschwindigkeit auf die Spielfeldposition zu, an der der letzte Mausklick stattfand.

Copyright 2001 Bernd Brügge

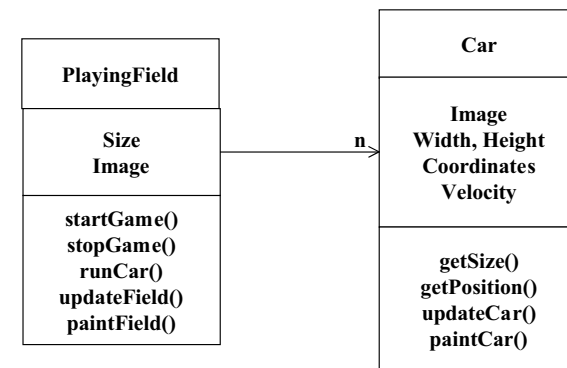
Einführung in die Informatik II TUM Sommersemester 2001

13:52:29 12

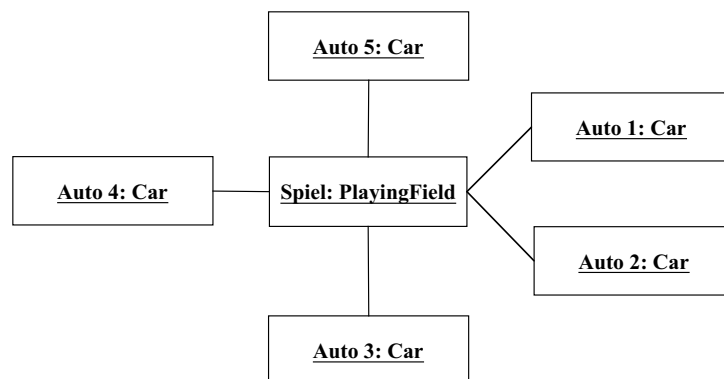
Fragen bei der Entwicklung des Spiels

- ❖ Wie stellen wir das Feld dar?
- ❖ Wie modellieren wir das Auto?
- ❖ Wie animieren wir Autos, die gleichzeitig fahren?
- ❖ Was machen wir, wenn ein Auto den Rand trifft?
- ❖ Wie berechnen wir eine Kollision von zwei oder mehr Autos?
- ❖ Wie lassen wir das Auto auf einen Mausklick reagieren?
 - ◆ Wie berechnen wir die neue Route?

Analyse



Instanzdiagramm mit 5 Autos



Einschub: Mehrfachbetrieb und Java-Threads

- ❖ **Mehrfachbetrieb (Multitasking)** ist die Technik der gleichzeitigen (*nebenläufigen*, engl. *concurrent*) Ausführung von mehreren Programmen auf einer Rechenanlage.
- ❖ Java unterstützt den Mehrfachbetrieb durch **Threads** (wörtlich übersetzt: (Kontroll-)Faden). Ein **Java-Thread** ist eine Sequenz von ausführbaren Anweisungen innerhalb eines Programms, die parallel zu anderen Sequenzen ausgeführt werden kann.
- ❖ Die Implementierung der Java Virtual Machine selbst beruht auf Threads:
 - ◆ Der sog. **Müllsammler (garbage collector)** ist z.B. ein Thread zum Sammeln von nicht mehr referenzierten Objekten, der nebenläufig zum Java-Programm abläuft.
- ❖ Auf einem sequentiellen Rechner müssen sich viele Threads ein einziges Rechenwerk teilen.
 - ◆ Scheduling ⇒ Thema der Vorlesung Betriebssysteme

Die Java-Klasse Thread

```
public class Thread extends Object implements Runnable {
    // Konstruktoren
    public Thread();
    public Thread(Runnable target);
    public Thread( String name );
    // Klassen Methoden
    public static native void sleep(long ms) throws InterruptedException;
    public static native void yield();
    // Instanz Methoden
    public final String getName();
    public final int getPriority();
    public void run();           // Methode der Java-Schnittstelle Runnable
    public final void setName();
    public final void setPriority();
    public synchronized native void start();
    public final void stop();
}
```

Java-Schnittstelle Runnable

- ❖ **Thread** implementiert die Methode **run ()** der Schnittstelle **Runnable**.

```
public abstract interface Runnable {
    public abstract void run()
}
```

- ❖ **Definition:** Jede Instanz einer Klasse, die die Schnittstelle **Runnable**, d.h. die Methode **run ()** implementiert, ist ein **lauffähiges Objekt**.
- ❖ Die Klasse **Thread** enthält die Methode **start ()**. Beim Aufruf der **start ()**-Methode eines Threads *t* aus einem Programm *p* heraus wird die **run ()**-Methode von *t* nebenläufig zu *p* ausgeführt.
 - ♦ Unterklassen von **Thread** überschreiben im allgemeinen die **run ()**-Methode von **Thread**.

Ausführung von Threads: Überschreibung von run ()

- ❖ **Beispiel:** Ein Thread, der eine gegebene Nummer 10-mal ausdrückt.
- ❖ Wir definieren **NumberThread** als Unterklasse von **Thread** und überschreiben die **run ()**-Methode:

```
public class NumberThread extends Thread {
    int num;
    int id;
    public NumberThread(int i, int n) {
        id = i;
        num = n;
    }
    public void run() {
        for (int k = 0; k < num; k++) {
            System.out.print(id);
        } //for
    } // run()
} // NumberThread
```

Exekution mehrerer Threads

- ❖ Hier erzeugen wir 5 Instanzen vom Typ **NumberThread** und starten dann jede Instanz mit **start ()**:

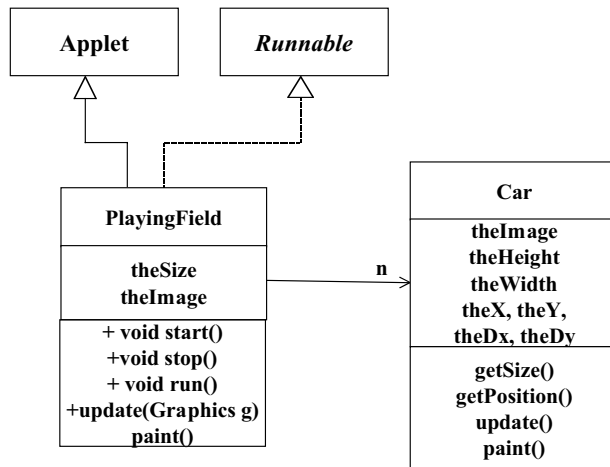
```
public class Numbers {
    public static void main(String args[]) {
        NumberThread number1, number2, number3, number4, number5;
        number1 = new NumberThread(1, 10); number1.start();
        number2 = new NumberThread(2, 10); number2.start();
        number3 = new NumberThread(3, 10); number3.start();
        number4 = new NumberThread(4, 10); number4.start();
        number5 = new NumberThread(5, 10); number5.start();
    } // main()
} // Numbers
```

Die Threads laufen in der Reihenfolge, in der sie gestartet worden sind. Bei kleinen Werten für **num** ist die Ausgabe noch sequentiell.

```
1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 3 3 3 3 3 3 3 4 4 4 4 4 4 4 5 5 5 5 5 5
```


Detaillierter Entwurf unseres Spiels

❖ Wir entwerfen jetzt **PlayingField** als lauffähiges Objekt



Implementierung des Spielfelds (Bumper01)

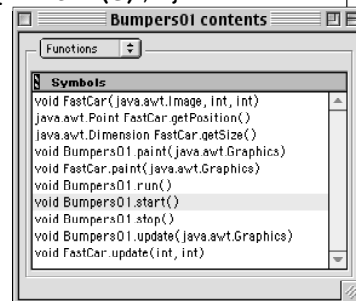
```

public class Bumper01 extends Applet implements Runnable {
    public static final int CARS = 6;
    Graphics theGraphics;
    Dimension theSize;
    Image theFieldImage;
    Thread theThread;
    boolean isRunning;
    FastCar[] theCar;
    Image theCarImage;
    AudioClip theMusic;
    public void start () {}
    public void stop () {}
    public void run () {}
    public void update () {}
    public void paint () {}
} // Bumper01
    
```

Starten des Spieles

```

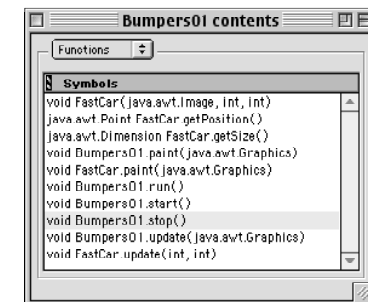
public void start () {
    try {
        theCarImage = getAppletContext().getImage(
            new URL(getDocumentBase(), "FastCar.GIF"));
        theMusic = getAppletContext().getAudioClip(
            new URL(getDocumentBase(), "Music.au"));
        theMusic.loop();
    }
    catch (Exception e) { System.out.println(e); }
    setBackground(Color.lightGray);
    theCar = new FastCar[CARS];
    for (int i = 0; i < CARS; i++) {
        theCar[i] =
            new FastCar(theCarImage,
                500, 300);
    }
    theThread = new Thread(this);
    isRunning = true;
    theThread.start();
}
    
```



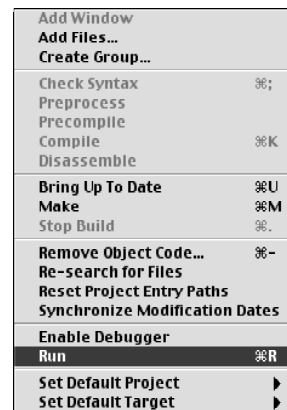
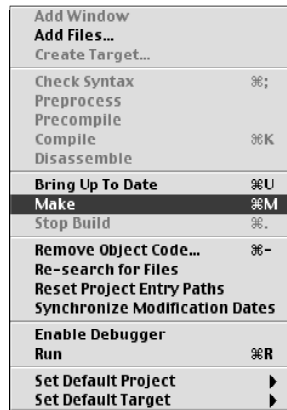
Stoppen des Spieles

```

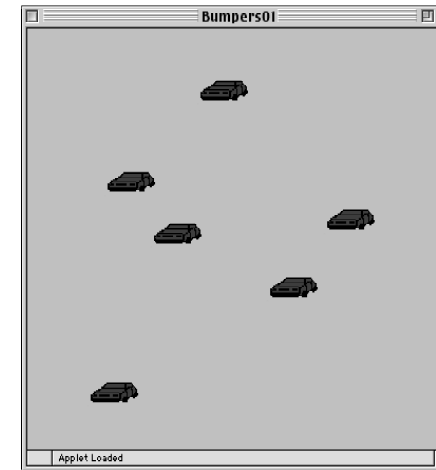
public void stop () {
    theMusic.stop();
    isRunning = false;
}
    
```



Übersetzung und Exekution des Programms



Das Spielfeld mit Autos



Bumpers01.java

Implementierung des Autos

```

class FastCar {
    Image theImage;
    int theWidth = 50;
    int theHeight = 25;
    int theX;
    int theY;
    float theDX;
    float theDY;
    static final float MAX_V = 20;
    public FastCar(Image img, int max_x, int max_y) {
        theImage = img;
        theX = (int) (Math.random() * max_x);
        theY = (int) (Math.random() * max_y);
        theDX = MAX_V - 2.0F * (float) Math.random() * MAX_V;
        theDY = MAX_V - 2.0F * (float) Math.random() * MAX_V;
    }
    public final synchronized Dimension getSize() {}
    public final synchronized Point getPosition() {}
    public synchronized void update(int max_x, int max_y) {}
    public void paint(Graphics g) {}
}
    
```



Monitore in Java

- ❖ **Definition:** Ein *Monitor* ist ein Java-Objekt, das *synchronisierte Methoden* enthält (Schlüsselwort: **synchronized**).
- ❖ Ein Monitor stellt sicher, dass nur **ein** Thread zur selben Zeit in **einer** synchronisierten Methode sein kann.
- ❖ Jeder Monitor ist mit einem Schloss (lock) versehen.
 - ◆ Wenn eine synchronisierte Methode aufgerufen wird, sperrt sie den Zugriff auf den Monitor für andere Aufrufer:
 - ◆ Während einer Sperrung kann kein anderer Thread eine synchronisierte Methode exekutieren.
 - ◆ Alle Aufrufer müssen "draußen" warten, bis die Methode das Schloss wieder aufgemacht hat.
 - ◆ Nach der Exekution des Methodenrumpfes wird das Schloss wieder aufgemacht.
- ❖ Die Klasse **Car** ist als Monitor realisiert.

Synchronisierte vs nicht-synchronisierte Methoden

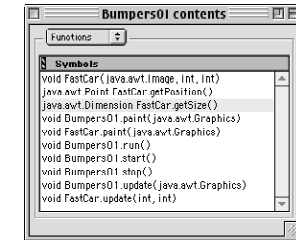
```
class Foo {
    public synchronized void blabla() {
        // Rumpf von blabla
    };

    public void foobar() {
        // Rumpf von foobar
    };
}
```



Methoden für die Klasse FastCar

```
public synchronized void
update (int max_x, int max_y) {
    theX += theDX;
    theY += theDY;
    if (theX < 0) {
        theX = 0;
        theDX = Math.abs(theDX);
    }
    if (theX + theWidth > max_x) {
        theX = max_x - theWidth;
        theDX = -1.0F *
            Math.abs(theDX);
    }
    if (theY < 0) {
        theY = 0;
        theDY = Math.abs(theDY);
    }
    if (theY + theHeight > max_y) {
        theY = max_y - theHeight;
        theDY = -1.0F *
            Math.abs(theDY);
    }
} // update
```



```
public final synchronized Dimension
getSize () {
    return new Dimension(theWidth,
                        theHeight);
}

public final synchronized Point
getPosition () {
    return new Point(theX, theY);
}

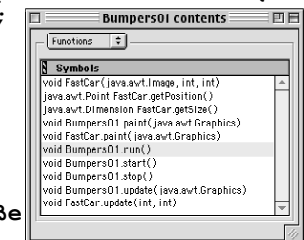
public void paint (Graphics g) {
    g.drawImage(theImage, theX, theY,
                theWidth, theHeight,
                null);
}
```

Animierung der Autos: Vorüberlegungen

- ❖ Nachdem das Spielfeld erschaffen worden ist und die Autos ins Spielfeld gesetzt worden sind, müssen wir sie zum Fahren bringen.
 - ♦ Wir machen dies in der **run ()**-Methode der Spielfeld-Klasse (die die Java-Schnittstelle **Runnable** implementiert).
 - ♦ Solange das Spiel noch läuft (angezeigt durch den Wert der booleschen Variablen **isRunning**), durchlaufen wir eine Endlosschleife.
- ❖ Innerhalb dieser Endlosschleife rufen wir die **update ()**-Methode für alle Autos auf, so dass sie ihre aktuelle Position zeichnen können.
 - ♦ Um das nicht zu schnell zu wiederholen, bauen wir innerhalb der Endlosschleife eine kleine Verzögerung ein.
 - ♦ Solange die Schleife ca. 20 mal pro Sekunde ausgeführt wird, ist die Animation relativ flüssig.
- ❖ Wir definieren deshalb eine Konstante **SLEEPTIME=30** und rufen die **sleep ()**-Methode vom Thread mit diesem Wert auf.

Implementierung der run ()-Methode

```
public class Bumpers01 extends Applet implements Runnable {
    public static final int SLEEPTIME = 30;
    ...
    public void run () {
        while (isRunning) {
            try {
                Thread.sleep(SLEEPTIME);
            }
            catch (InterruptedException e) {
                // alle Autos haben dieselbe Größe
                Dimension d = getSize();
                for (int i = 0; i < CARS; i++) {
                    theCar[i].update(d.width, d.height);
                }
                repaint();
            }
        }
    }
    ...
}
```



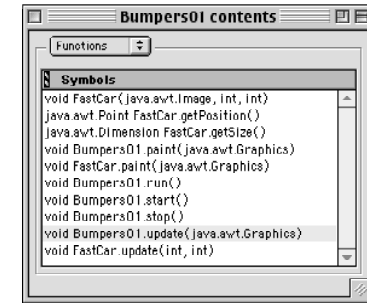
Zeichnen der aktuellen Spielsituation: Vorüberlegungen

- ❖ Das Spielfeld wird gezeichnet, indem wir die `paint()`-Methode des Applets überschreiben (`java.applet.Applet` ist eine Unterklasse von `java.awt.Component`).
- ❖ Um Flackern zu vermeiden, malen wir nicht direkt in das Spielfeld, sondern in den Bildpuffer `theImage`:
 - ♦ Um sicher zu stellen, dass die Größe von `theImage` identisch mit der Größe des Applets ist, wird notfalls ein neues Objekt vom Typ `java.awt.Image` kreiert.
 - ♦ Zur Darstellung des Applets wird einfach der Puffer-Inhalt angezeigt.
- ❖ Wie wird die Methode `paint()` aufgerufen?
 - ♦ `run()` ruft die Methode `repaint()` auf (siehe vorige Folie). Diese Methode ruft `update()` auf, die daraufhin `paint()` aufruft.
- ❖ Das eigentliche Zeichnen in `paint()` macht folgendes:
 - ♦ Löschen des Bildschirmabschnittes durch Malen dieses Abschnittes in der Hintergrundfarbe.
 - ♦ Aufforderung an jedes Auto, sich neu zu malen.

Zeichnen der aktuellen Spielsituation: Implementation

```
public final synchronized void update (Graphics g) {
    if (theSize == null || theSize.width != getSize().width ||
        theSize.height != getSize().height) {
        theSize = getSize();
        // Residenter Bildpuffer
        theImage = createImage(theSize.width, theSize.height);
        // Graphik-Kontext für den Bildpuffer
        theGraphics = theImage.getGraphics();
    }
    // Bildpuffer füllen
    paint(theGraphics);
    // Bildpuffer anzeigen
    g.drawImage(theImage, 0, 0, null);
}

public void paint (Graphics g) {
    // Lösche Bildschirm
    g.setColor(getBackground());
    g.fillRect(0, 0, getSize().width,
              getSize().height);
    // Autos zeichnen
    for (int i = 0; i < CARS; i++) {
        theCar[i].paint(g);
    }
}
```



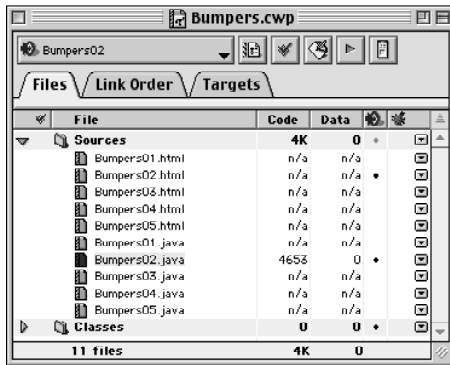
Kollisionen: Vorüberlegungen

- ❖ Wie prüfen wir Kollisionen? Wo platzieren wir den Kollisionscode?
- ❖ Wir haben zwei Möglichkeiten:
 - ♦ Beim Spielfeld oder bei den Autos selbst
- ❖ Wenn wir das Spiel erweiterbar machen wollen, dann hängt die Kollision höchstwahrscheinlich vom Typ der Autos ab (vor allem von der Größe).
- ❖ Wenn wir die Kollision vom Auto behandeln lassen wollen, dann machen wir jeden Autotyp von jedem anderen Autotyp abhängig.
 - ♦ Beispiel: Wenn wir eine neue Unterklasse "langsameres Auto" kreieren, müssen wir jeden bisherigen Autotyp modifizieren, um Kollisionen mit langsamen Autos zu implementieren.
- ❖ Die Erweiterbarkeit des Programms ist wesentlich höher, wenn wir das Spielfeld für die Behandlung von Kollisionen verantwortlich machen.
- ❖ Um "realistische" Kollisionen zu erzeugen, müssen wir einige zusätzliche Details berücksichtigen.

Kollisionen: Vorüberlegungen (2)

- ❖ Jedes Auto repräsentieren wir durch ein sog. *umgebendes Rechteck* (bounding rectangle). Damit bezeichnen wir das kleinste Rechteck, das alle Pixel des entsprechenden Bildes vom Auto noch einschließt.
 - ♦ Wir bekommen das umgebende Rechteck durch `getSize()`.
- ❖ Wir könnten jetzt festlegen, dass eine Kollision dann vorliegt, wenn zwei gegebene umgebende Rechtecke mindestens ein Pixel gemeinsam haben (Überprüfung mit der Methode `intersects()`).
 - ♦ Das einzige Problem damit ist, dass das Bild eines Autos irregulär ist, und das umfangende Rechteck nicht ganz ausfüllt.
- ❖ Um dies zu kompensieren, reduzieren wir das umfangende Rechteck auf 3/4 der originalen Größe (mit `growth()`), und platzieren sein Zentrum leicht nach links unten (mit `translate()`).
- ❖ Letztes Problem: Wer ist der Gewinner der Kollision?
 - ♦ Das Auto mit der kleineren y-Koordinate!

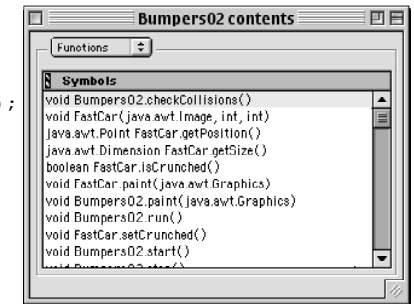
Neue Version: Bumpers02



Prüfen von Kollisionen

```
protected void checkCollisions () {
    for (int i = 0; i < CARS - 1; i++) {
        if (theCar[i].isCrunched())
            continue;
        Point p1 = theCar[i].getPosition();
        Dimension d1 = theCar[i].getSize();
        Rectangle r1 = new Rectangle(p1, d1);
        r1.translate(p1.x / 8, p1.y / 8); // offset
        r1.grow(-1 * d1.width / 4, -1 * d1.height / 4);

        for (int j = i + 1; j < CARS; j++) {
            if (theCar[j].isCrunched())
                continue;
            Point p2 = theCar[j].getPosition();
            Dimension d2 = theCar[j].getSize();
            Rectangle r2 = new Rectangle(p2, d2);
            r2.translate(p2.x / 8, p2.y / 8);
            r2.grow(-1 * d2.width / 4,
                -1 * d2.height / 4);
            if (r1.intersects(r2)) {
                theBang.play();
                if (p1.y < p2.y)
                    theCar[j].setCrunched();
                else
                    theCar[i].setCrunched();
            } // if intersects
        } // for j
    } // for i
} // checkCollisions
```

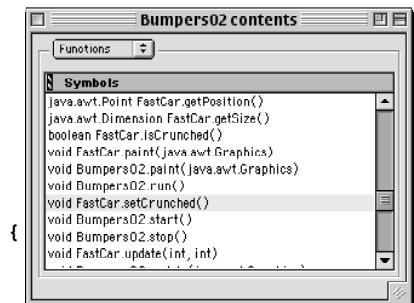


Revision der Klasse FastCar

- ❖ Die Klasse **FastCar** müssen wir noch dahingehend revidieren, dass sie jetzt mit Kollisionen leben muss:-)
- ❖ Wir führen dafür ein neues Attribut **isCrunched** ein, und natürlich die entsprechenden Zugriffsmethoden.
- ❖ Ausserdem müssen wir die **update ()**-Methode revidieren, denn ein kaputtes Auto soll sich nicht mehr bewegen dürfen.

Revision der Klasse FastCar

```
class FastCar {
    ...
    boolean isCrunched = false;
    ...
    public final void setCrunched () {
        isCrunched = true;
    }
    public final boolean isCrunched () {
        return isCrunched;
    }
    // Andere Methoden unverändert
    public synchronized void update (int max_x, int max_y) {
        if (isCrunched)
            return;
        // Der Rest von update wie bisher
    }
}
```



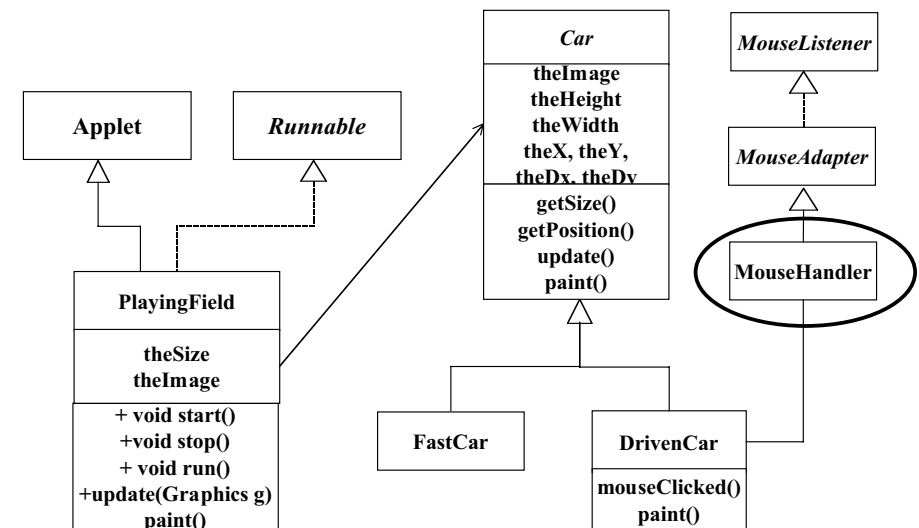
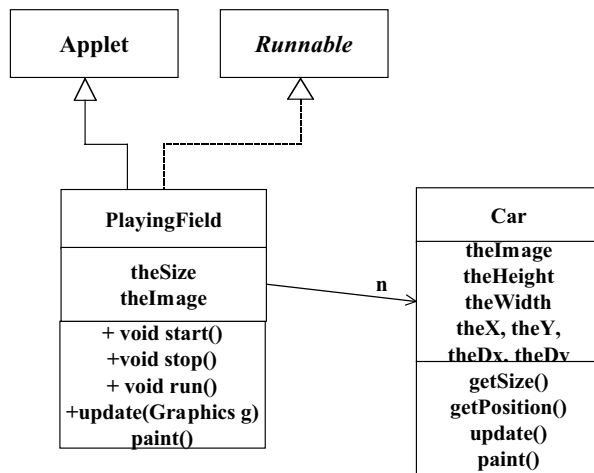
Benutzer-kontrolliertes Auto

- ❖ Das Ziel des Spiels ist, dass wir durch geschickte Mausbewegungen unser Auto so mit anderen Autos kollidieren lassen, dass wir das Spiel gewinnen und nicht von anderen Autos überfahren werden..
- ❖ Wir definieren jetzt eine neue Klasse Benutzer-kontrolliertes Auto, dessen Richtung wir mit der Maus kontrollieren können.
 - ◆ Wenn der Benutzer die Maustaste drückt, dann bewegt sich das benutzer-kontrollierte Auto in Richtung des Punkts, an dem sich der Mauszeiger zum Zeitpunkt des Mausklicks befindet.
- ❖ Für Mausereignisse gibt es in Java das Ereignis **MouseEvent** und den Ereignisempfänger **MouseListener**
- ❖ Für die neue Klasse Benutzer-kontrolliertes Auto definieren wir einen Ereignisempfänger **MouseHandler**, der eine Unterklasse von **MouseAdapter** ist, einer trivialen Implementation von **MouseListener** (d.h. eine Implementation mit leeren Rümpfen für alle öffentlichen Methoden von **MouseListener**).

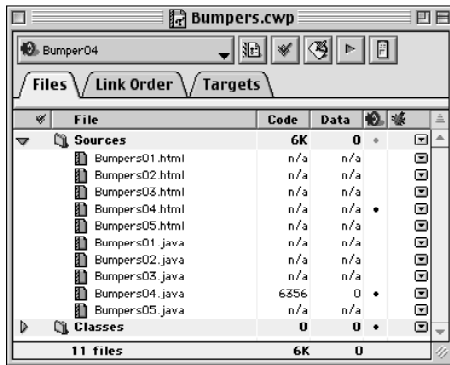
Die Klasse MouseAdapter

```
public abstract class MouseAdapter extends Object
implements MouseListener{
    public void mouseClicked (MouseEvent e) {}
    public void mouseEntered (MouseEvent e) {}
    public void mouseExited (MouseEvent e) {}
    public void mousePressed (MouseEvent e) {}
    public void mouseReleased (MouseEvent e) {}
}
```

Revision des Modells: Einfügung einer neuen Autoklasse



Neue Versionen: Bumpers03, Bumpers04

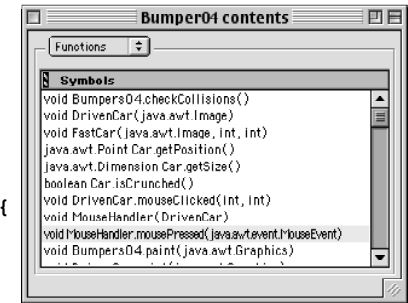


Implementierung von MouseHandler

```
class MouseHandler
  extends MouseAdapter {
  DrivenCar theDrivenCar;

  public MouseHandler (DrivenCar dc) {
    theDrivenCar = dc;
  }

  public void mousePressed(MouseEvent e) {
    theDrivenCar.mouseClicked(e.getX(), e.getY());
  }
}
```



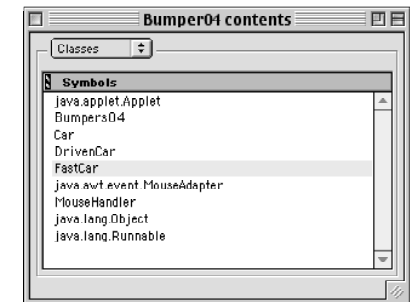
Implementierung von Car

- ❖ Wir implementieren die abstrakte Klasse **Car**, indem wir alle Elemente nehmen, die in der bisherigen konkreten Klasse **FastCar** definiert waren, mit 2 Ausnahmen.
 - ◆ **Car** hat keinen speziellen Konstruktor.
 - ◆ **getSize()** und **getPosition()** werden als **final** definiert. Es ist unwahrscheinlich, dass Unterklassen diese Methoden überschreiben.
- ❖ **FastCar** und **DrivenCar** werden als Unterklassen von **Car** implementiert.

Abstrakte Klasse Car

```
abstract class Car {
  // Code der bisherigen
  // Klasse FastCar
  // (siehe vorherige Folie)
}

class FastCar extends Car {
  private static final float MAX_V = 20;
  public FastCar(Image img, int max_x, int max_y) {
    theImage = img;
    theX = (int) (Math.random() * max_x);
    theY = (int) (Math.random() * max_y);
    theDX = MAX_V - 2.0F * (float) Math.random() * MAX_V;
    theDY = MAX_V - 2.0F * (float) Math.random() * MAX_V;
  }
}
```



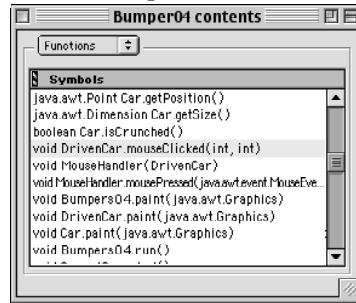
Bewegung des Benutzer-kontrollierten Autos

- ❖ Das Benutzer-kontrollierte Auto wird an einer festen Stelle im Feld platziert. Dort bleibt es bewegungslos, bis ein Mausklick stattfindet.
- ❖ Jedesmal, wenn das Benutzer-kontrollierte Auto einen Mausklick empfängt, berechnen wir eine neue Geschwindigkeit (**theDX**, **theDY**), damit sich das Auto in Richtung des Mausklicks bewegt:

```
public synchronized void mouseClicked (int x, int y) {
    int a = x - theX; int b = y - theY;
    double theta = Math.atan2(a, b);
    theDX = V * (float) Math.sin(theta);
    theDY = V * (float) Math.cos(theta);
}
```

- ❖ Die **paint()**-Methode in der Superklasse wird nur dann aufgerufen, wenn das benutzer-kontrollierte Auto noch fährt:

```
public void paint (Graphics g) {
    if (!isCrunched()) super.paint(g);
}
```



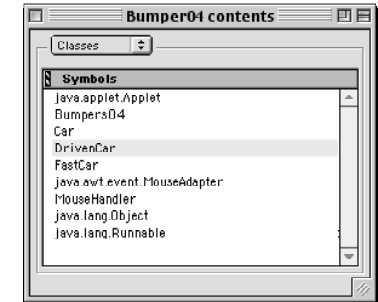
Neue Unterklasse DrivenCar

```
class DrivenCar extends Car {
    private static final float V = 5.0F;

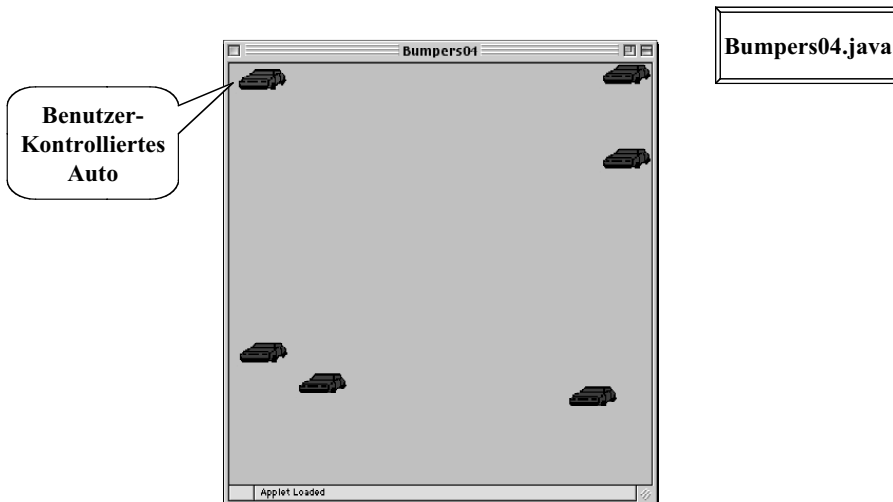
    public DrivenCar (Image img) {
        theImage = img;
        theX = 5;
        theY = 5;
        theDX = 0.0F;
        theDY = 0.0F;
    }

    public synchronized void mouseClicked (int x, int y) {
        double theta = Math.atan2(x - theX, y - theY);
        theDX = V * (float) Math.sin(theta);
        theDY = V * (float) Math.cos(theta);
    }

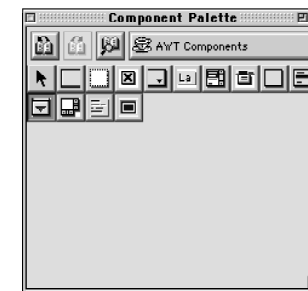
    public void paint (Graphics g) {
        if (!isCrunched()) super.paint(g);
    }
}
```



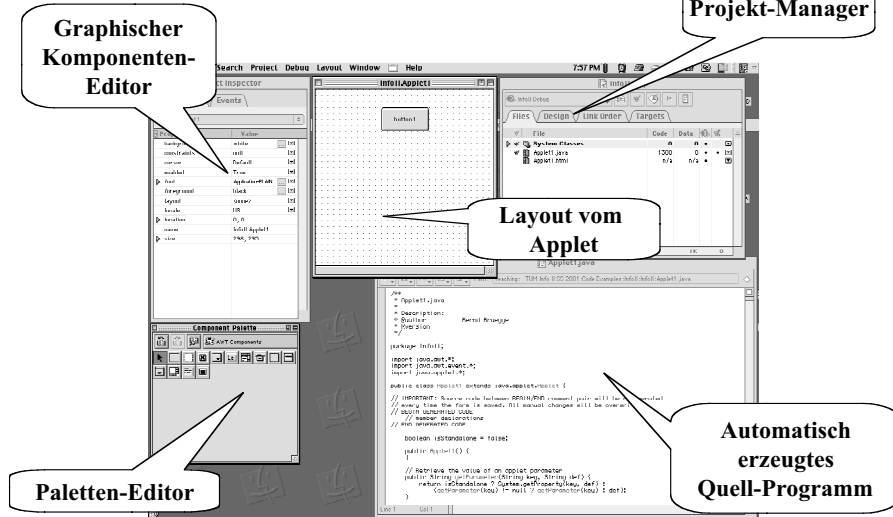
Spiel mit benutzer-kontrolliertem Auto



Paletten-Editor für graphische Komponenten



Paletten-Basierte Entwicklung von Applets



Zusammenfassung

- ❖ Gezeigt: Entwicklung eines Systems mit einer interaktiven Entwicklungsumgebung
- ❖ **Interaktive Entwicklungsumgebungen** vs Compiler
 - ◆ Vorteil: Integration vieler Werkzeuge
 - ◆ Nachteil: Erhöhter Lernaufwand
- ❖ **Mehrfachbetrieb**: Gleichzeitige Ausführung von mehreren Programmen auf einer Rechenanlage.
- ❖ Java unterstützt Mehrfachbetrieb durch Threads, eine Sequenz von ausführbaren Anweisungen innerhalb eines Programm, die parallel zu anderen Anweisungssequenzen ablaufen kann.
- ❖ Synchronisieren von Thread-Methoden garantiert exklusiven Zugriff auf ein Objekt (Monitor-Konzept).