Simulation
von
digitalen Schaltungen

Inhalt:

1. Bezug zum Unterricht
2. Funktionsumfang des Simulators HASI
3. Klassenhierarchien
4. Die Erzeugung von Komponenten unter Delphi
5. Bausteine erzeugen und verschieben
6. Bausteine verdrahten
7. Bausteine arbeiten lassen
8. Aufgaben

1. Bezug zum Unterricht

Die Simulation digitaler Schaltelemente gehört zu den ältesten Beispielen der Schulinformatik – jeweils realisiert mit den aktuellen Werkzeugen. Im Zeitalter der objektorientierten Programmierung ist das Thema eines der besten Beispiele für die Anwendung von OOP-Entwurfs- und – implementationsmethoden. Die Objekthierarchien ergeben sich hier teilweise aus den benutzten visuellen Komponenten, teilweise aus den simulierten Bauteilen selbst.

Für den Unterricht bietet das Thema die Möglichkeit,

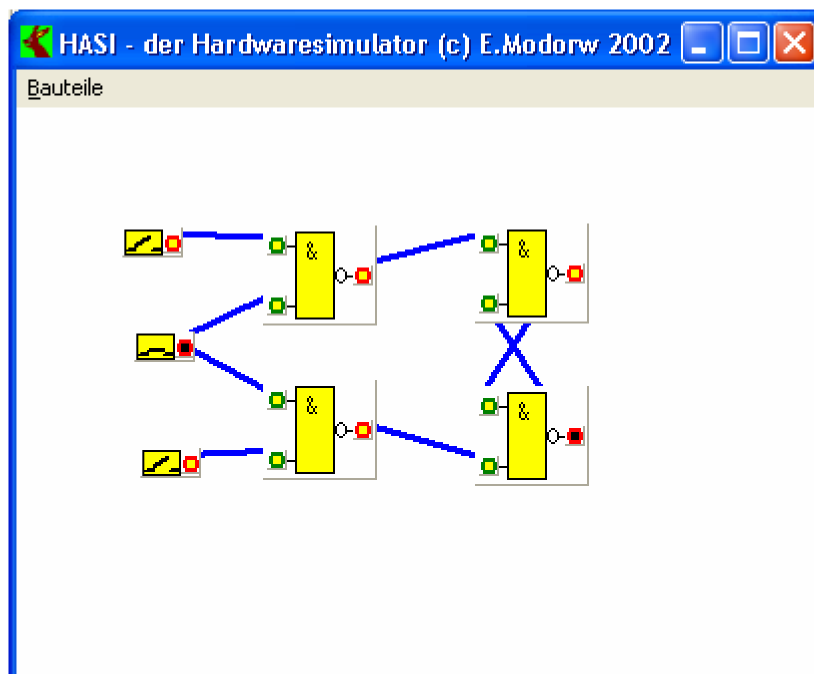
- alternativ zum Entwurf von technischen Komponenten Softwaremethoden einzusetzen und so schon sehr früh zu praktischen Arbeiten zu kommen.
- einen Anwendungsfall für Datenstrukturen zu liefern.
- ein zwar komplexes, aber immer noch übersichtliches Projekt in Angriff zu nehmen.
- die methodischen Möglichkeiten zu erweitern.

2. Funktionsumfang des Simulators

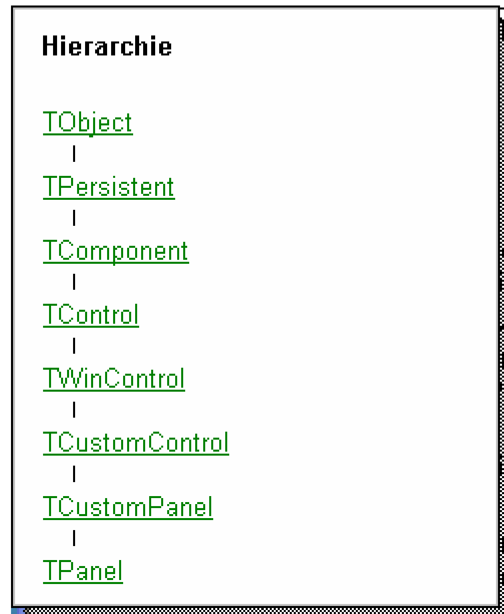
Das System soll hier in einer sehr reduzierten, aber leicht erweiterbaren Form realisiert werden. Dazu sollen

- nur zwei Bauteile erzeugbar sein: NANDs und SCHALTER
- die Bauteile am Bildschirm verschiebbar und bedienbar sein.
- ein sehr spartanisches Menü erzeugt werden.
- eine halbwegs übersichtliche Objekthierarchie erzeugt werden, in die später weitere Komponenten leicht eingeklinkt werden können.
- die benötigten Leitungen nur sehr rudimentär angezeigt werden. (Die Verdrahtung der Komponenten ist ein eigenes, ziemlich komplexes Problem.)
- rein objektorientiert programmiert werden. D. h. die erzeugten Bausteine werden nicht vom Programm verwaltet, sondern arbeiten „autonom“ selbstständig vor sich hin. Die Aktionen werden vollständig ereignisgesteuert ausgelöst.

Als Programmiersprache wird hier Delphi verwandt. Auf Effizienz des Systems wird bewusst zugunsten von Klarheit verzichtet.



3. Klassenhierarchien



Unsere Digitalbausteine sollen – wie gezeigt - am Bildschirm verschiebbare Elemente sein, die auf Mausereignisse reagieren. Es bietet sich an, solche Obkjekte von der Klasse *tPanel* abzuleiten, die genau dieses kann. Deren Klassenhierarchie kann direkt von Delphi angezeigt werden.

Alle unsere Bauteile sollen also aus Panels bestehen, die um weitere Eigenschaften erweitert werden. Dabei wollen wir *Bausteine* und *Buchsen* unterscheiden:

- Bausteine enthalten weitere Elemente, z. B. Buchsen, und man kann sie verschieben.
- Buchsen enthalten keine weiteren Elemente.

Gemeinsam ist diesen Klassen, dass auf ihnen gezeichnet werden kann. Dazu versehen wir sie mit einem *Image*. Zusätzlich soll die absolute Position auf dem Fenster gespeichert werden, da diese zum Zeichnen der Leitungen benötigt wird. Diese Eigenschaften fassen wir in einer Oberklasse *tTeil* zusammen. Alle drei Klassen enthalten einen Konstruktor *init* und weitere, dann spezielle Methoden.

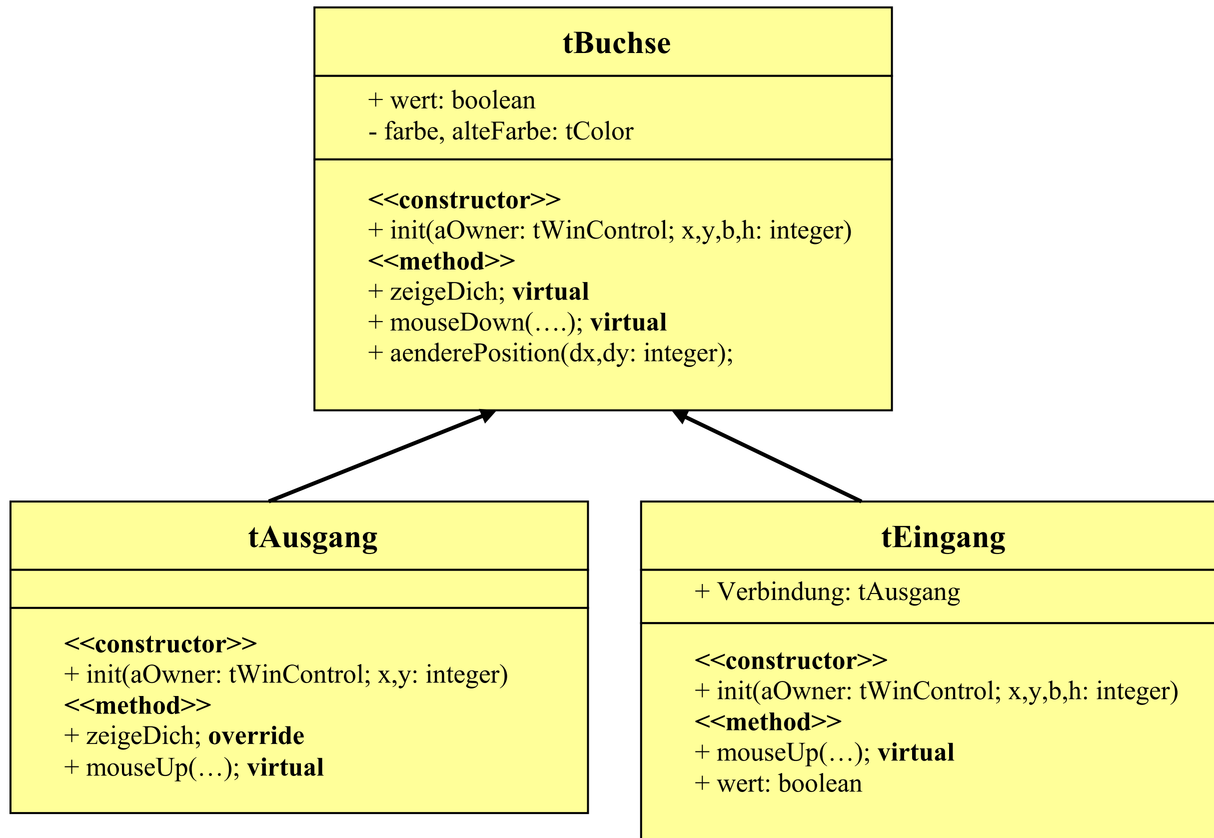
tPanel
….
….

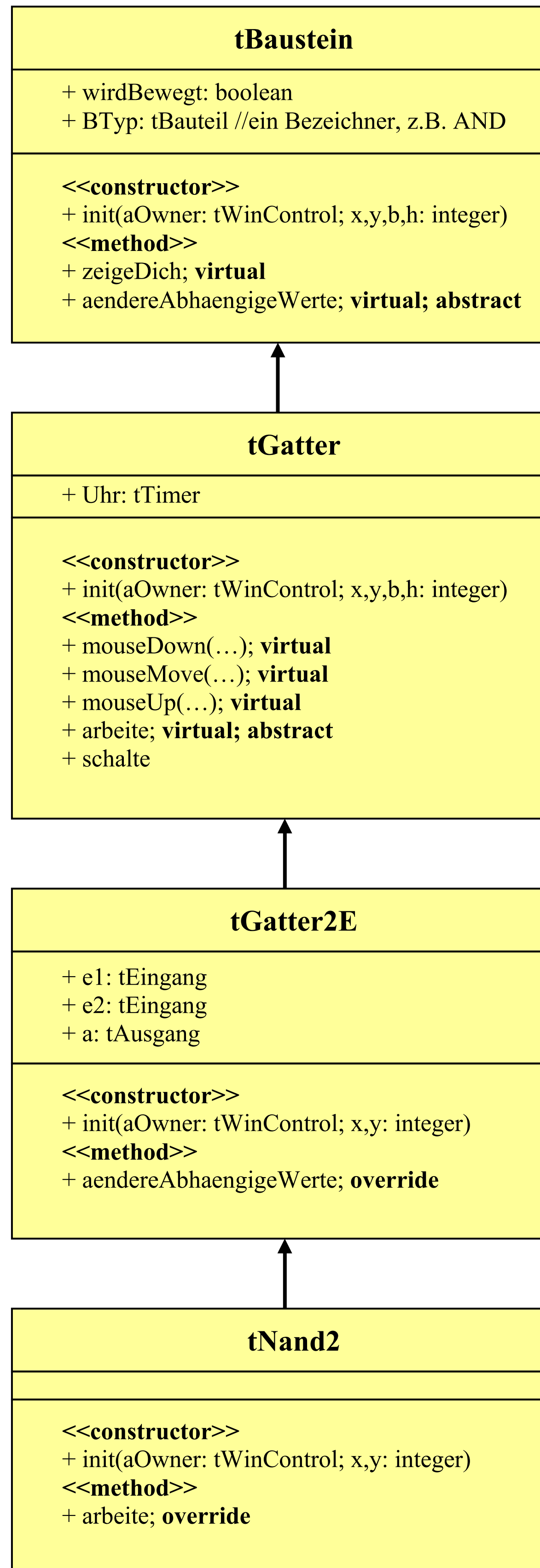
tTeil
+ xabs: integer + yabs: integer + Bild: tImage
<<constructor>> + init(aOwner: tWinControl; x,y,b,h: integer)

tBaustein
+ wirdBewegt: boolean + BTyp: tBauteil //ein Bezeichner, z.B. AND
<<constructor>> + init(aOwner: tWinControl; x,y,b,h: integer) **<<method>>** + zeigeDich; **virtual** + aendereAbhaengigeWerte; **virtual; abstract**

tBuchse
+ wert: boolean - farbe, alteFarbe: tColor
<<constructor>> + init(aOwner: tWinControl; x,y,b,h: integer) **<<method>>** + zeigeDich; **virtual** + mouseDown(….); **virtual** + aenderePosition(dx,dy: integer)

tBuchse ist die Mutterklasse für *Eingänge* und *Ausgänge*, während aus Bausteinen Gatter abgeleitet werden, die weitere Tochterklassen enthalten, z. B. Gatter mit zwei Eingängen, mit vier Eingängen, … Am Ende der Hierarchie stehen die konkreten Gatter, die im Simulator eingesetzt werden. Es ergibt sich die folgende Klassenhierarchie:





Aus Bausteinen können z. B. die Knoten (Ecken, Verzweigungen, …) von Leitungen abgeleitet werden, und die Einordnung weiterer Gatter in den Baum ist offensichtlich.

4. Die Erzeugung von Komponenten unter Delphi

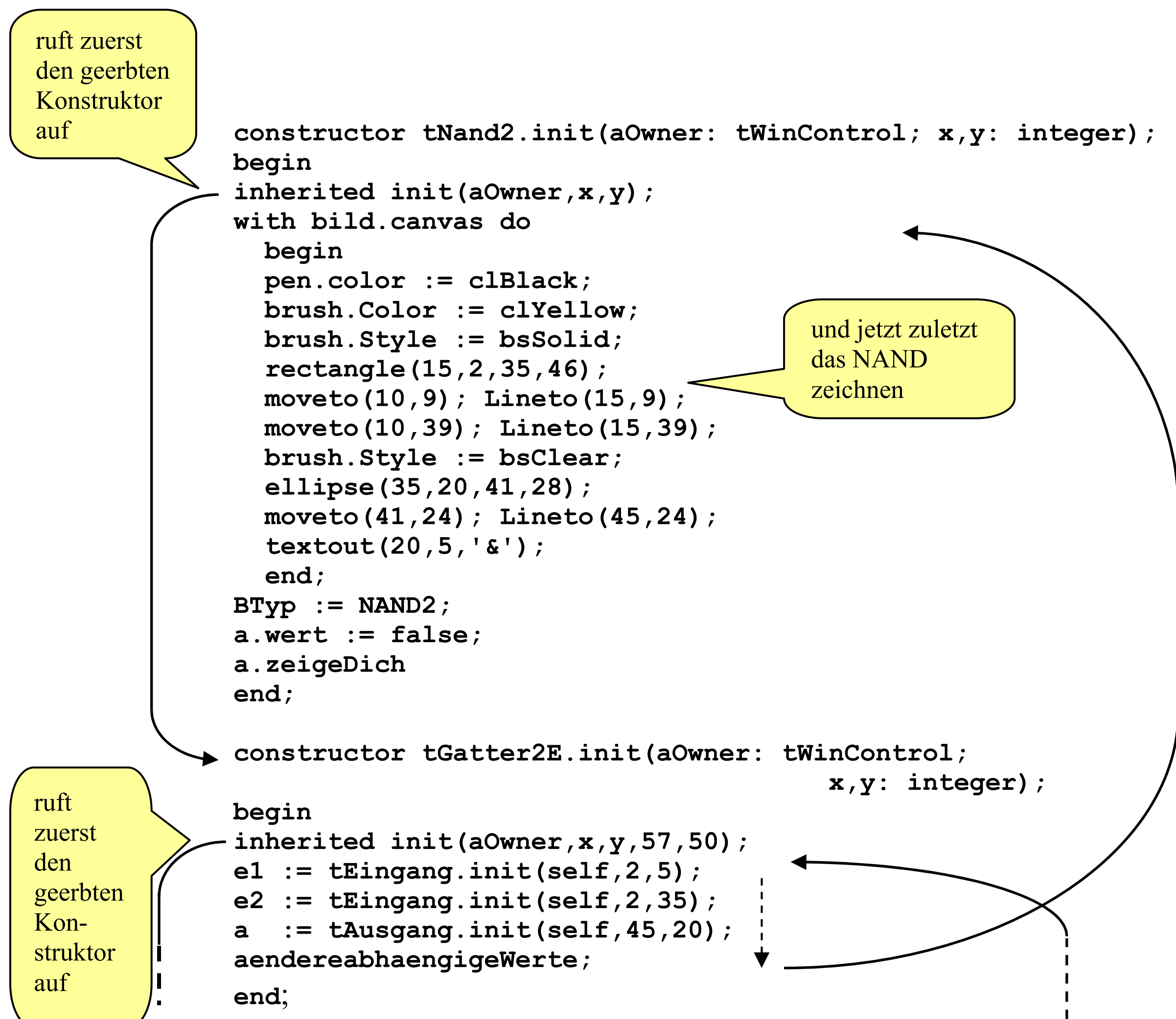
Normalerweise erzeugt man visuelle Komponenten (Buttons, …) im Objektdesigner und stattet sie im Objektinspektor mit Eigenschaften aus. Stattdessen kann man Komponenten aber auch dynamisch während des Programmlaufs erzeugen. Dazu

- vereinbaren wir eine Komponente als Variable → ergibt einen leeren Zeiger
- rufen den entsprechenden Konstruktor auf → auf dem Heap wird Platz für die Attribute des Objekts belegt
- weisen den Attributen Werte zu → das Objekt erhält z. B. die richtige Größe und Farbe
- geben den *Owner* und den *Parent* des Objekts an, die z. B. dafür verantwortlich sind, die Komponente anzuzeigen („zu zeichnen") oder sie beim Speichern zu verwalten.

Soll unsere Komponente auf Mausereignisse reagieren, dann

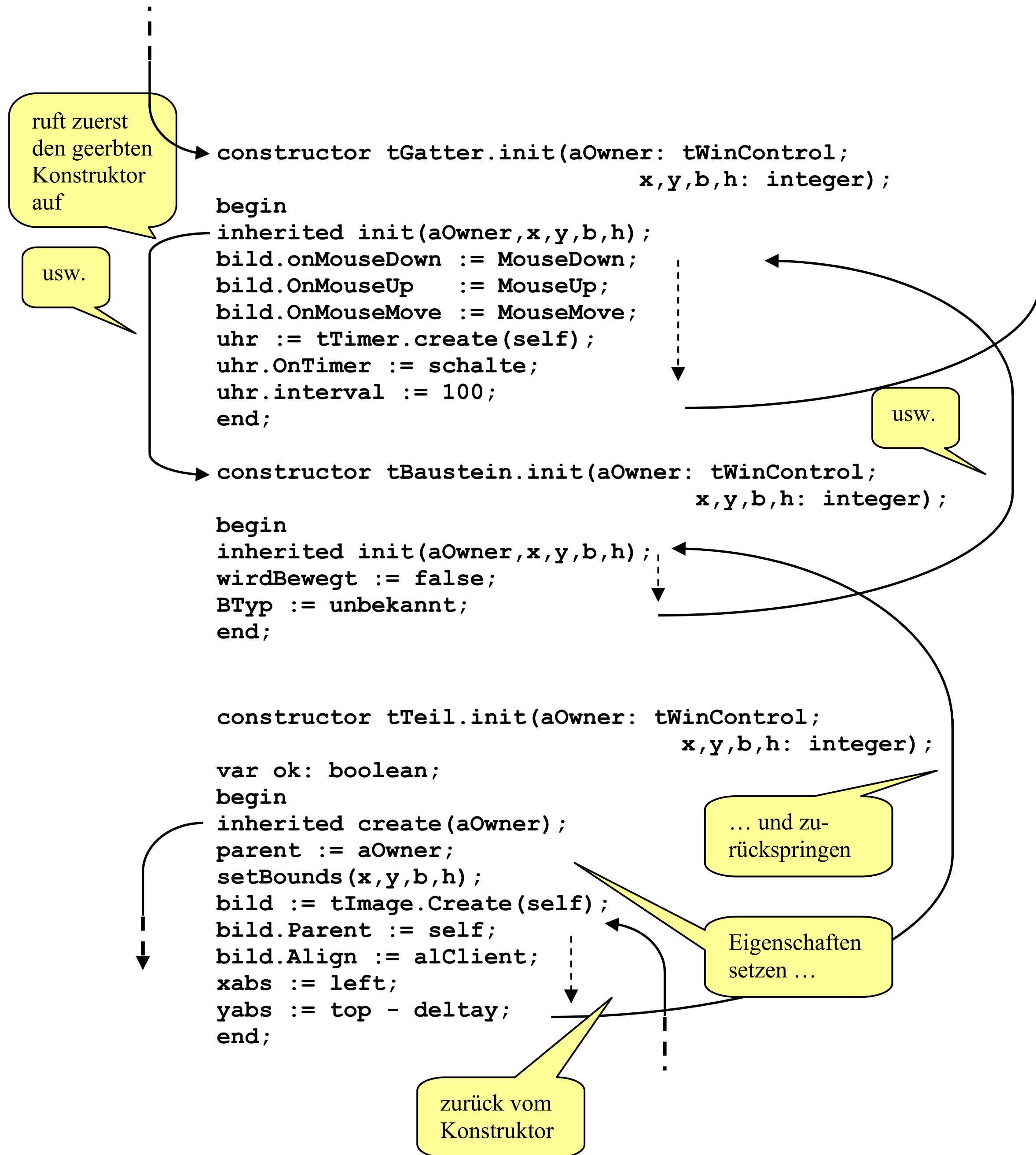
- schreiben wir Eventhandler beliebigen Namens, aber mit vorgegebener Parameterliste. (Am einfachsten kopieren wir einfach die Liste der Methoden, die von Delphi automatisch erzeugt werden.)
- weisen dem Ereignis diese Methode zu → die wird dann z. B. in die VMT eingetragen.

Als Beispiel sehen wir uns die für die Erzeugung eines Nands benötigten Konstruktoren an:




```
constructor tNand2.init(aOwner: tWinControl; x,y: integer);
begin
inherited init(aOwner,x,y);
with bild.canvas do
  begin
  pen.color := clBlack;
  brush.Color := clYellow;
  brush.Style := bsSolid;
  rectangle(15,2,35,46);
  moveto(10,9); Lineto(15,9);
  moveto(10,39); Lineto(15,39);
  brush.Style := bsClear;
  ellipse(35,20,41,28);
  moveto(41,24); Lineto(45,24);
  textout(20,5,'&');
  end;
BTyp := NAND2;
a.wert := false;
a.zeigeDich
end;

constructor tGatter2E.init(aOwner: tWinControl;
                                      x,y: integer);
begin
inherited init(aOwner,x,y,57,50);
e1 := tEingang.init(self,2,5);
e2 := tEingang.init(self,2,35);
a  := tAusgang.init(self,45,20);
aendereabhaengigeWerte;
end;
```


ruft zuerst den geerbten Konstruktor auf
usw.
constructor tGatter.init(aOwner: tWinControl;
x,y,b,h: integer);
begin
inherited init(aOwner,x,y,b,h);
bild.onMouseDown := MouseDown;
bild.OnMouseUp := MouseUp;
bild.OnMouseMove := MouseMove;
uhr := tTimer.create(self);
uhr.OnTimer := schalte;
uhr.interval := 100;
end;
usw.
constructor tBaustein.init(aOwner: tWinControl;
x,y,b,h: integer);
begin
inherited init(aOwner,x,y,b,h);
wirdBewegt := false;
BTyp := unbekannt;
end;
constructor tTeil.init(aOwner: tWinControl;
x,y,b,h: integer);
var ok: boolean;
begin
inherited create(aOwner);
parent := aOwner;
setBounds(x,y,b,h);
bild := tImage.Create(self);
bild.Parent := self;
bild.Align := alClient;
xabs := left;
yabs := top - deltay;
end;
… und zurückspringen
Eigenschaften setzen …
zurück vom Konstruktor

5. Bausteine erzeugen und verschieben

Wir gehen von einem Typ aus, der angibt welche Art von Bauteilen gerade erzeugt werden soll: `tBauteil = (unbekannt,NAND2,SCHALTER);`

In einer Variablen wird der aktuelle Wert gespeichert: `var Bauteil : tBauteil = NAND2;`

Wird jetzt die Maus auf dem Formular losgelassen, dann wird ein Bauteil der gewünschten Art erzeugt.

```
procedure TDigitalsimulator.Image1MouseUp(Sender: TObject;
             Button: TMouseButton; Shift: TShiftState; X, Y: Integer);
var n: tGatter;
begin
case Bauteil of
   NAND2    : n := tNand2.init(self,x,y+deltay);
   SCHALTER: n := tSchalter.init(self,x,y+deltay);
   end;
end;
```

Bauteile werden verschoben, indem auf die entsprechenden Mausereignisse in der üblichen Weise reagiert wird.

```
procedure tGatter.MouseDown(Sender: TObject; Button: TMouseButton;
                                       Shift: TShiftState; X, Y: Integer);
begin
xOffset := x;
yOffset := y;
wirdBewegt := true;
end;

procedure tGatter.MouseMove(Sender: TObject; Shift: TShiftState;
                                                          X, Y: Integer);
begin
if wirdBewegt then
   setBounds(left+x-xOffset,top+y-yOffset,width,height);
end;

procedure tGatter.MouseUp(Sender: TObject; Button: TMouseButton;
                                       Shift: TShiftState; X, Y: Integer);
begin
xabs := left;
yabs := top-deltay;
aendereAbhaengigeWerte;
wirdBewegt := false;
With Digitalsimulator.Image1.Canvas do
  begin
  pen.Color := clWhite;
  brush.Color := clWhite;
  brush.style := bsSolid;
  rectangle(0,0,Digitalsimulator.width,Digitalsimulator.height);
  end;
end;
```

jetzt wahrscheinlich falsche Leitungen löschen

6. Bausteine verdrahten

Unsere Buchsen arbeiten etwas unterschiedlich:

- Ausgänge erhalten ihren *wert* vom Gatter, zu dem sie gehören. Dieses schreibt einfach den aktuellen Wert in ihr Attribut.
- Eingänge erhalten einen Zeiger *verbindung* auf einen Ausgang, also ein entsprechendes Objekt. Hat dieses keinen vernünftigen Wert, dann ist der Eingang nicht verbunden und liefert den Wert true als Ergebnis einer Anfrage. Ist der Eingang mit einem Ausgang verbunden, dann liefert er den Wert des Ausgangs:

```
function tEingang.wert: boolean;
begin
if verbindung = nil then result := true
  else result := verbindung.wert
end;
```

Wie werden Buchsen verbunden?

Zuerst klickt man auf einen Eingang. Dieser kopiert seinen Wert in eine globale Variable namens *Eingang* vom Typ *tEingang*. Zusätzlich ändert er kurz seine Farbe und speichert seine absoluten Koordinaten auf dem Formular, damit später eine Leitung gezeichnet werden kann.

```
procedure tEingang.MouseUp(Sender: TObject; Button: TMouseButton;
                                 Shift: TShiftState; X, Y: Integer);
begin
farbe := alteFarbe;
ZeigeDich;
Eingang := self;
xanf := xabs;
yanf := yabs;
end;
```

da passiert es! (→ Eingang := self;)

Wird jetzt ein Ausgang angeklickt, dann teilt dieser dem Eingang seine Adresse mit und zeichnet die Leitung.

```
procedure tAusgang.MouseUp(Sender: TObject; Button: TMouseButton;
                                 Shift: TShiftState; X, Y: Integer);
begin
farbe := alteFarbe;
if Eingang = nil then exit;
Eingang.Verbindung := self;
with Digitalsimulator.Image1.Canvas do
  begin
  pen.Color := clBlue;
  pen.Width := 3;
  moveto(xanf,yanf);
  lineto(xabs,yabs);
  end;
Eingang := nil;
ZeigeDich;
end;
```

da auch! (→ Eingang.Verbindung := self;)

7. Bausteine arbeiten lassen

Für nicht zu große Schaltungen kann eine sehr einfache und auch nahe liegende Methode gewählt werden: Jeder Baustein erhält eine eigene Uhr, die in regelmäßigen Abständen an den Eingängen nachsieht, welche Werte dort anliegen, und daraus die Ausgangswerte berechnet.

In unserem Fall startet ein *Timer* beim *onTime*-Ereignis einen Eventhandler *schalte*, der nichts anderes zu tun hat, als eine virtuelle Methode *arbeite* aufzurufen, die für jeden Baustein spezifisch ist.

```
constructor tGatter.init(aOwner: tWinControl; x,y,b,h: integer);
begin
inherited init(aOwner,x,y,b,h);
bild.onMouseDown := MouseDown;
bild.OnMouseUp   := MouseUp;
bild.OnMouseMove := MouseMove;
uhr := tTimer.create(self);
uhr.OnTimer := schalte;
uhr.interval := 100;
end;
```

Timer einrichten

```
procedure tGatter.schalte(Sender: tObject);
begin
arbeite
end;
```

virtuelle Methode aufrufen

```
procedure tNand2.arbeite;
var h: boolean;
begin
h := a.wert;
a.wert := not(e1.wert and e2.wert);
if a.wert <> h then a.zeigeDich;
end;
```

da isse für ein NAND

Fertig! - Der Rest ist Feinarbeit.

8. Aufgaben

Ergänzen Sie den Hardwaresimulator wie folgt:

1. Fügen Sie weitere Grundgatter mit zwei Eingängen und einem Ausgang ein: *UND2*, *ODER2*, *EXOR*
2. Führen Sie einen *NICHT*-Baustein ein, der direkt aus der Gatter-Klasse abgeleitet wird.
3. Führen Sie die Klasse *Gatter4E* der Gatter mit vier Eingängen ein. Leiten Sie aus dieser entsprechende Grundgatter ab.
4. Führen Sie Rechenschaltungen ein: Halbaddierer *HA* und Volladdierer *VA.*
5. Entwickeln Sie ein *Auffang-Flipflop*, also eine Schaltung, die ein Bit speichern kann.
6. Leiten Sie aus dem Auffangflipflop ein *JK-MS-FF* ab.
7. Entwickeln Sie *Binärzähler*, *Register* und ein *RAM*.
8. a: Versuchen Sie, die Probleme beim Einsatz etwas übersichtlicher *Leitungen* im Simulator einzuschätzen. Diskutieren Sie verschiedene Möglichkeiten und den zu deren Lösung erforderlichen Aufwand.

   b: Leiten Sie aus der Bausteinklasse eine Klasse *Leitungsknoten* ab, zwischen denen Leitungsstücke verlaufen. Machen Sie die Knoten bei Bedarf verschiebbar.

   c: Führen Sie neue Knotentypen *Verzweigung* ein, von dem aus Leitungen in zwei bzw. drei Zweige aufgespaltet werden können.
9. Diskutieren Sie den Aufwand, der bei der Umstellung des Simulators auf eine *Time-Priority-Queue* anfällt. Diese Schlange verwaltet den einzigen Timer des Systems, in dessen Warteschlange sich Bausteine eintragen und von dem nach Verlauf eines entsprechenden Zeitintervalls die *arbeite*-Methode der Bausteine aufgerufen werden.