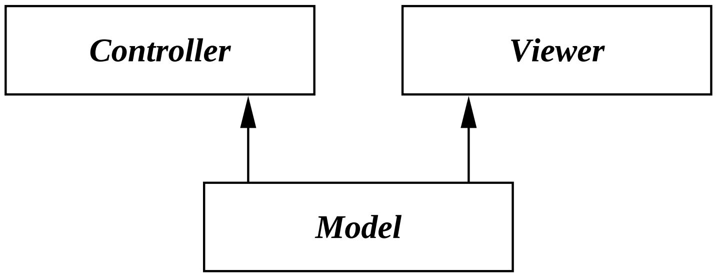
Models, Viewers und Controllers



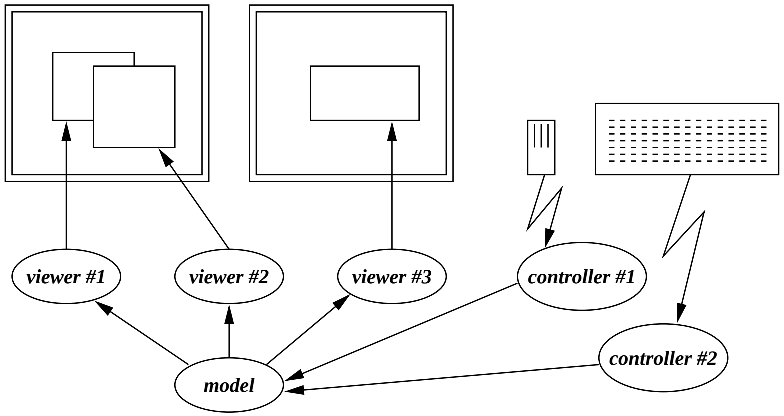
- Die Einteilung in Models, Viewers und Controllers im Zusammenhang mit graphischen Benutzeroberflächen wurde von Smalltalk-80 eingeführt. Die Vorteile dieses Ansatzes liegen in der damit gewonnen Flexibilität und der leichteren Wiederverwendbarkeit.

- Model-Klassen repräsentieren ein abstraktes Modell, das der jeweiligen Applikation zugrunde liegt. Das Modell ist im einfachen Fall ein reines Datenmodell oder bietet im allgemeinen Fall eine Reihe von Abfragemethoden und Operationen, die den Zustand verändern können.

- Viewer-Klassen sind in der Lage, bestimmte abstrakte Modelle graphisch in einer Benutzeroberfläche zu repräsentieren. Relevant ist hier, daß ein konkretes Modell gleichzeitig durch eine Vielzahl von konkreten Instanzen von Viewer-Klassen dargestellt werden kann.

- Controller-Klassen können Ereignisse von einer graphischen Benutzeroberfläche (Interaktionen) oder externen Quellen in Operationen auf dem Modell konvertieren.

Informationsflüsse

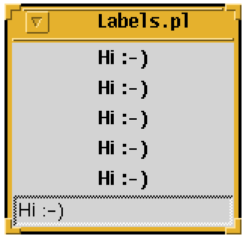


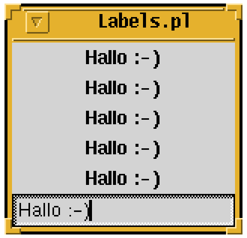
- Eingehende Ereignisse werden von den Controller-Objekten erfaßt und in Form von Operationen an das Modell-Objekt weitergeleitet.

- Alle Viewer-Objekte, die sich zuvor bei dem Modell-Objekt registriert haben, werden über alle Statusänderungen informiert.

- Bei einem verteilten System können nicht nur beliebig viele Viewer- und Controller-Objekte beteiligt sein, sondern auch mehr als ein Rechner und eine graphische Benutzeroberfläche partizipieren.

MVC-Modell bei Widgets





labels.pl

```
my $main = new MainWindow;
my $textvariable = "Hi :-)";
foreach (1 .. 5) {
   $main->Label('-textvariable' => \$textvariable)->pack;
}
$main->Entry('-textvariable' => \$textvariable)->pack;
MainLoop;
```

- Selbst bei den allereinfachsten Widgets gibt es ein dahinterliegendes abstraktes Modell.

- So gehört zu einem Label das abstrakte Modell einer Zeichenkette, das von dem Label (im Rahmen seiner Möglichkeiten) dargestellt wird.

- Im Beispiel ist die Abstraktion einer skalaren Variablen von Perl das Modell, die Labels sind allesamt Viewer und der Entry operiert simultan als Viewer und Controller.

- Leider hört bei komplizierteren Widgets (Textfenster beispielsweise) die Unterstützung bei Tk auf.

MVC-Modell bei Applikationen

- Genauso wie sich die graphische Benutzeroberfläche einer Applikation aus vielen sich zusammensetzenden Einzelkomponenten besteht, kann das MVC-Modell beginnend mit den elementaren Widgets Ebene um Ebene bis hin zur Abstraktionsebene der Applikation hochgezogen werden.

- Im Gesamtbild gibt es dann drei Hierarchien (Models, Viewers und Controllers), die jeweils in ihrer Ebene zueinander in Beziehung stehen und bei denen von Ebene zu Ebene eine Aggregation stattfindet.

MVC-Klassen in Perl

texts.pl

```
#!/usr/local/bin/perl -w

use strict;
use Tk;
use MVC::Model::Text;
use MVC::Viewer::ROText;

my $main = new MainWindow;
my $model = new MVC::Model::Text "some\nlines\nof\ntext\n";

foreach (1..3) {
   my $viewer = $main->Scrolled('ROTextViewer',
      '-model' => $model,
      '-width' => 20,
      '-height' => 5);
   $viewer->pack;
}
my $line = "";
my $entry = $main->Entry('-textvariable' => \$line)->pack;
$entry->bind('<Return>',
   sub {
      $model->append($line . "\n");
      $line = "";
   });

MainLoop;
```

- Natürlich kann die MVC-Strukturierung auch in Perl/Tk übernommen werden, wenn einmal die Grundmechanismen dafür geschaffen worden sind.

MVC::Model

MVC/Model.pm

```
package MVC::Model;

require Exporter;
use strict;

@MVC::Model::ISA = qw(Exporter);


sub new {
   my ($package, @options) = @_;
   my ($self) = bless {}, $package;
   $self->init(@options);
   return $self;
}

sub init {
   my ($self) = @_;
   $self->{viewers} = {};
}
```

- `MVC::Model` dient als Basisabstraktion für alle Modelle.
- Typischerweise werden die Initialisierungsarbeiten außerhalb des Konstruktors von einer speziellen Methode durchgeführt. In diesem Fall von **`init`**. Von `MVC::Model` abgeleitete Klassen dürfen zwar **`init`** überdefinieren – müssen jedoch sicherstellen, daß auch **`init`** für die Basisklassen aufgerufen wird.
- In **`$self->{viewers}`** verwaltet `MVC::Model` die Menge der registrierten Viewer.

MVC::Model

MVC/Model.pm

```
sub register {
   my ($self, $viewer) = @_;
   $self->{viewers}->{$viewer} = $viewer;
}

sub deregister {
   my ($self, $viewer) = @_;
   delete $self->{viewers}->{$viewer};
}

sub update {
   my ($self) = @_;
   foreach my $viewer (values %{$self->{viewers}}) {
      $viewer->update($self);
   }
}

1;
```

- Neu hinzukommene Viewer können sich mit **`register`** registrieren und dies später mit **`deregister`** rückgängig machen.

- Klassen, die von `MVC::Model` abgeleitet sind, müssen **`update`** aufrufen, wenn sich der Zustand in einer von außen sichtbaren Weise verändert hat.

- Alle von `MVC::Viewer` abgeleiteten Klassen benötigen eine Methode **`update`**, die für die Synchronisation des aktuellen Zustands eines Modells mit der Präsentierung verantwortlich ist.

MVC::Viewer

MVC/Viewer.pm

```
package MVC::Viewer;

require Exporter;
@ISA = qw(Exporter);

use Carp;
use strict;

sub new {
   my ($package, @options) = @_;
   my ($self) = bless {}, $package;
   $self->init(@options);
   return $self;
}

sub init {
   my ($self, $model) = @_;
   $self->{model} = $model;
   $model->register($self) if defined $model;
}
```

- `MVC::Viewer` ist die Basisklasse für alle Klassen, die ein Modell präsentieren.

MVC::Viewer

MVC/Viewer.pm

```
sub attach {
   my ($self, $model) = @_;
   $self->{model}->deregister($self)
      if defined $self->{model};
   $self->{model} = $model;
   $model->register($self) if defined $model;
}

sub detach {
   my ($self) = @_;
   $self->{model}->deregister($self)
      if defined $self->{model};
   $self->{model} = undef;
}

sub update {
   croak("update has not been declared by derived class");
}

1;
```

- Mit `attach` und `detach` ist es möglich, die Verbindung zwischen einem Viewer und einem konkreten Model zu ändern bzw. aufzulösen.
- Virtuelle Methoden, die überdefiniert werden müssen, sollten in der Basisklasse mit der Generierung eines Fehlers (über `croak`) vordefiniert werden.

MVC::Model::Text

MVC/Model/Text.pm

```
package MVC::Model::Text;

use MVC::Model;
use strict;

@MVC::Model::Text::ISA = qw(MVC::Model);

sub init {
   my ($self, $init) = @_;
   $self->SUPER::init();
   $init = "" unless defined $init;
   $self->{text} = $init;
}

sub get {
   my ($self) = @_;
   return $self->{text};
}
```

- `MVC::Model::Text` ist ein (sehr einfaches) Modell für Text (Zeichenfolge, die über beliebig viele Zeilen gehen kann).
- Das Durchlaufen der Initialisierungskette für ein neues Objekt durch alle beteiligten Klassen wird durch `$self->SUPER::init()` sichergestellt. `SUPER` ist ein Verweis auf die übergeordnete Klasse.

MVC::Model::Text

MVC/Model/Text.pm

```
sub set {
   my ($self, $text) = @_;
   $text = "" unless defined $text;
   if ($text ne $self->{text}) {
      $self->{text} = $text;
      $self->update();
   }
}

sub append {
   my ($self, $text) = @_;
   if (defined ($text) && $text ne "") {
      $self->{text} .= $text;
      $self->update();
   }
}

1;
```

- Mit `set` oder `append` ist es möglich, den aktuellen Zustand des Text-Modells sichtbar zu verändern. Entsprechend ist es notwendig, die Methode `update` jeweils anschließend aufzurufen, um Änderungen an die Viewer weiterzugeben.

MVC::Viewer::ROText

MVC::Viewer::ROText.pm

```
package MVC::Viewer::ROText;

use Carp;
use Tk;
use Tk::ROText;
use MVC::Viewer;
use strict;

@MVC::Viewer::ROText::ISA =
   qw(Tk::ROText MVC::Viewer);

Construct Tk::Widget 'ROTextViewer';

sub init {
   my ($self, @options) = @_;
   $self->SUPER::init(@options);
}
```

- Die Klasse `MVC::Viewer::ROText` ist eine Ableitung sowohl von dem Widget `Tk::ROText` (repräsentierendes Widget) als auch der Basisklasse für Viewer `MVC::Viewer`.
- Mit `Construct Tk::Widget 'ROTextViewer'` wird dafür gesorgt, daß `ROTextViewer` als Konstruktor-Methode für alle Widgets eingetragen wird, die andere aufnehmen können (`MainWindow`, `TopLevel`, `Frame` usw).
- Da `Tk::ROText` innerhalb `@ISA` zuerst genannt wird, hat der Konstruktor (Methode `new`) von `Tk::Widget` Priorität vor dem von `MVC::Viewer`.

MVC::Viewer::ROText

MVC::Viewer::ROText.pm

```
sub InitObject {
   my ($self, $args) = @_;
   my $model = undef;
   if (defined($args->{'-model'})) {
      $model = $args->{'-model'};
      delete $args->{'-model'};
   }
   $self->SUPER::InitObject($args);
   $self->init($model);
   $self->update($model);
}

sub update {
   my ($self, $model) = @_;
   $self->delete('1.0', 'end');
   $self->insert('end', $model->get());
   $self->see('end');
}

1;
```

- Bei den von `Tk::Widget` abgeleiteten Modulen geht die Initialisierungskette nicht über die Methode `init`, sondern über `InitObject`. Dies ist ein glücklicher Umstand, da es sonst bei *multiple inheritance* nicht ganz einfach ist, alle beteiligten Klassen an der Initialisierung teilhaben zu lassen.