

# Objektorientiertes Programmieren in GAP

Max Neunhöffer

Lehrstuhl D für Mathematik  
RWTH Aachen

Aachen 30.10.2006

# Die Idee

GAP Objekte repräsentieren mathematische Objekte.

# Die Idee

GAP Objekte repräsentieren mathematische Objekte.

Es gibt „Operationen“ und „Methoden“.

GAP Objekte repräsentieren mathematische Objekte.

Es gibt „Operationen“ und „Methoden“.

Eigenschaften der Objekte (ihr **Typ**)



Auswahl der „richtigen“ Methode

# Die Idee

GAP Objekte repräsentieren mathematische Objekte.

Es gibt „Operationen“ und „Methoden“.

Eigenschaften der Objekte (ihr Typ)



Auswahl der „richtigen“ Methode

Objekte können während ihrer Lebenszeit „lernen“!  
(d.h. ihren Typ verändern)

# Die Idee

GAP Objekte repräsentieren mathematische Objekte.

Es gibt „Operationen“ und „Methoden“.

Eigenschaften der Objekte (ihr Typ)



Auswahl der „richtigen“ Methode

Objekte können während ihrer Lebenszeit „lernen“!  
(d.h. ihren Typ verändern)

Die verwendeten Methoden ändern sich dann!

# Die Idee

GAP Objekte repräsentieren mathematische Objekte.

Es gibt „Operationen“ und „Methoden“.

Eigenschaften der Objekte (ihr Typ)



Auswahl der „richtigen“ Methode

Objekte können während ihrer Lebenszeit „lernen“!  
(d.h. ihren Typ verändern)

**Die verwendeten Methoden ändern sich dann!**

GAP verwendet also:

- dynamische Typisierung zur Laufzeit

# Die Idee

GAP Objekte repräsentieren mathematische Objekte.

Es gibt „Operationen“ und „Methoden“.

Eigenschaften der Objekte (ihr Typ)



Auswahl der „richtigen“ Methode

Objekte können während ihrer Lebenszeit „lernen“!  
(d.h. ihren Typ verändern)

**Die verwendeten Methoden ändern sich dann!**

GAP verwendet also:

- dynamische Typisierung zur Laufzeit
- statische Datenbank von Methoden

# Die Idee

GAP Objekte repräsentieren mathematische Objekte.

Es gibt „Operationen“ und „Methoden“.

Eigenschaften der Objekte (ihr Typ)



Auswahl der „richtigen“ Methode

Objekte können während ihrer Lebenszeit „lernen“!  
(d.h. ihren Typ verändern)

**Die verwendeten Methoden ändern sich dann!**

GAP verwendet also:

- dynamische Typisierung zur Laufzeit
- statische Datenbank von Methoden
- Methodenselektion „just in time“

# Typen

Ein **Typ in GAP** ist ein Paar:

(eine „Familie“, eine Bit-Liste von „elementaren Filtern“)

# Typen

Ein **Typ in GAP** ist ein Paar:

(eine „Familie“, eine Bit-Liste von „elementaren Filtern“)

Die **Familien** bilden eine **Partition der Menge der Objekte**.  
ein Part ist z.B. die `PermutationsFamily`

# Typen

Ein **Typ in GAP** ist ein Paar:

(eine „Familie“, eine Bit-Liste von „elementaren Filtern“)

Die **Familien** bilden eine **Partition der Menge der Objekte**.  
ein Part ist z.B. die `PermutationsFamily`

Ein **elementarer Filter** ist **sowohl**

- ein Bit in der 2. Komponente vom Typ **als auch**
- die Menge aller Objekte, bei denen dieses Bit gesetzt ist

# Typen

Ein **Typ in GAP** ist ein Paar:

(eine „Familie“, eine Bit-Liste von „elementaren Filtern“)

Die **Familien** bilden eine **Partition der Menge der Objekte**.  
ein Part ist z.B. die `PermutationsFamily`

Ein **elementarer Filter** ist **sowohl**

- ein Bit in der 2. Komponente vom Typ **als auch**
- die Menge aller Objekte, bei denen dieses Bit gesetzt ist

Ein **Filter** ist **entweder**

- ein elementarer Filter **oder**
- eine Und-Verknüpfung elementarer Filter

# Typen

Ein **Typ in GAP** ist ein Paar:

(eine „Familie“, eine Bit-Liste von „elementaren Filtern“)

Die **Familien** bilden eine **Partition der Menge der Objekte**.  
ein Part ist z.B. die `PermutationsFamily`

Ein **elementarer Filter** ist **sowohl**

- ein Bit in der 2. Komponente vom Typ **als auch**
- die Menge aller Objekte, bei denen dieses Bit gesetzt ist

Ein **Filter** ist **entweder**

- ein elementarer Filter **oder**
- eine Und-Verknüpfung elementarer Filter

Jedes Objekt `o` „liegt“ entweder „in einem Filter“ oder nicht. Kann getestet werden mit `FILTERNAME(o)`.

# Typen

Ein **Typ in GAP** ist ein Paar:

(eine „Familie“, eine Bit-Liste von „elementaren Filtern“)

Die **Familien** bilden eine **Partition der Menge der Objekte**.  
ein Part ist z.B. die `PermutationsFamily`

Ein **elementarer Filter** ist **sowohl**

- ein Bit in der 2. Komponente vom Typ **als auch**
- die Menge aller Objekte, bei denen dieses Bit gesetzt ist

Ein **Filter** ist **entweder**

- ein elementarer Filter **oder**
- eine Und-Verknüpfung elementarer Filter

Jedes Objekt `o` „liegt“ entweder „in einem Filter“ oder nicht. Kann getestet werden mit `FILTERNAME(o)`.

**Beispiele:** `IsSolvable`, `IsNilpotent`, `IsAbelian`

# Operationen und Methoden

Eine **Operation** ist eine Sammlung von Methoden.

Man deklariert

- den Namen,
- die Anzahl der Argumente, und
- einen Filter für jedes der Argumente.

# Operationen und Methoden

Eine **Operation** ist eine Sammlung von Methoden.

Man deklariert

- den Namen,
- die Anzahl der Argumente, und
- einen Filter für jedes der Argumente.

```
DeclareOperation("Size", [IsGroup]);
```

# Operationen und Methoden

Eine **Operation** ist eine Sammlung von Methoden.

Man deklariert

- den Namen,
- die Anzahl der Argumente, und
- einen Filter für jedes der Argumente.

```
DeclareOperation("Size", [IsGroup]);
```

Man installiert eine oder mehrere **Methoden**:

- Das sind Funktionen mit dieser Anzahl Argumente.
- Man kann weitere Einschränkungen machen:

# Operationen und Methoden

Eine **Operation** ist eine Sammlung von Methoden.

Man deklariert

- den Namen,
- die Anzahl der Argumente, und
- einen Filter für jedes der Argumente.

```
DeclareOperation("Size", [IsGroup]);
```

Man installiert eine oder mehrere **Methoden**:

- Das sind Funktionen mit dieser Anzahl Argumente.
- Man kann weitere Einschränkungen machen:

```
InstallMethod(Size,  
  [IsGroup and IsPermGroup],  
  function(p) ... return ...; end);
```

# Operationen und Methoden

Eine **Operation** ist eine Sammlung von Methoden.

Man deklariert

- den Namen,
- die Anzahl der Argumente, und
- einen Filter für jedes der Argumente.

```
DeclareOperation("Size", [IsGroup]);
```

Man installiert eine oder mehrere **Methoden**:

- Das sind Funktionen mit dieser Anzahl Argumente.
- Man kann weitere Einschränkungen machen:

```
InstallMethod(Size,  
  [IsGroup and IsPermGroup],  
  function(p) ... return ...; end);
```

Wir nennen diese Einschränkungen „**Anforderungen**“.

# Die Methodenselektion

Wenn nun jemand `size(g)` für ein Objekt `g` aufruft,

# Die Methodenselektion

Wenn nun jemand `size(g)` für ein Objekt `g` aufruft,

- wird der **Typ** von `g` bestimmt,

# Die Methodenselektion

Wenn nun jemand `Size(g)` für ein Objekt `g` aufruft,

- wird der **Typ** von `g` bestimmt,
- **alle** Methoden für `Size` angeschaut,

# Die Methodenselektion

Wenn nun jemand `size(g)` für ein Objekt `g` aufruft,

- wird der **Typ** von `g` bestimmt,
- **alle** Methoden für `size` angeschaut,
- ermittelt, welche passen (**applicable methods**),

# Die Methodenselektion

Wenn nun jemand `size(g)` für ein Objekt `g` aufruft,

- wird der **Typ** von `g` bestimmt,
- **alle** Methoden für `size` angeschaut,
- ermittelt, welche passen (**applicable methods**),
- und **diejenige** aufgerufen, die
  - **passt**, und
  - **am meisten Anforderungen hat**  
(bei gleichen Anforderungen die später installierte).

# Die Methodenselektion

Wenn nun jemand `size(g)` für ein Objekt `g` aufruft,

- wird der **Typ** von `g` bestimmt,
- **alle** Methoden für `size` angeschaut,
- ermittelt, welche passen (**applicable methods**),
- und **diejenige** aufgerufen, die
  - **passt**, und
  - **am meisten Anforderungen hat**  
(bei gleichen Anforderungen die später installierte).

Geht überhaupt nur effizient durch trickreiches **Caching**!

# Die Methodenselektion

Wenn nun jemand `size(g)` für ein Objekt `g` aufruft,

- wird der **Typ** von `g` bestimmt,
- **alle** Methoden für `size` angeschaut,
- ermittelt, welche passen (**applicable methods**),
- und **diejenige** aufgerufen, die
  - **passt**, und
  - **am meisten Anforderungen hat**  
(bei gleichen Anforderungen die später installierte).

Geht überhaupt nur effizient durch trickreiches **Caching!**

**Genauer:** Jeder elementare Filter hat einen „Rang“.  
Es wird die Methode genommen, für die die Rangsumme der Anforderungen maximal ist.

# Die Idee an Familien

Die Familien **partitionieren** die Menge der Objekte.

Dagegen bilden die **Filter** eine **Hierarchie** von Mengen.

# Die Idee an Familien

Die Familien **partitionieren** die Menge der Objekte.

Dagegen bilden die **Filter** eine **Hierarchie** von Mengen.

z.B.: `PermutationsFamily`, `CyclotomicsFamily`.

# Die Idee an Familien

Die Familien **partitionieren** die Menge der Objekte.

Dagegen bilden die **Filter** eine **Hierarchie** von Mengen.

z.B.: `PermutationsFamily`, `CyclotomicsFamily`.

Bei endlich präsentierten Gruppen bilden alle Elemente einer solchen Gruppe eine Familie.

# Die Idee an Familien

Die **Familien** **partitionieren** die Menge der Objekte.

Dagegen bilden die **Filter** eine **Hierarchie** von Mengen.

z.B.: `PermutationsFamily`, `CyclotomicsFamily`.

Bei endlich präsentierten Gruppen bilden alle Elemente einer solchen Gruppe eine Familie.

Eine **Collection** besteht aus Objekten **derselben Familie**.

# Die Idee an Familien

Die **Familien partitionieren** die Menge der Objekte.

Dagegen bilden die **Filter** eine **Hierarchie** von Mengen.

z.B.: `PermutationsFamily`, `CyclotomicsFamily`.

Bei endlich präsentierten Gruppen bilden alle Elemente einer solchen Gruppe eine Familie.

Eine **Collection** besteht aus Objekten **derselben Familie**.

Zu jeder Familie kann man die „`CollectionsFamily`“ bilden.

# Die Idee an Familien

Die **Familien partitionieren** die Menge der Objekte.

Dagegen bilden die **Filter** eine **Hierarchie** von Mengen.

z.B.: `PermutationsFamily`, `CyclotomicsFamily`.

Bei endlich präsentierten Gruppen bilden alle Elemente einer solchen Gruppe eine Familie.

Eine **Collection** besteht aus Objekten **derselben Familie**.

Zu jeder Familie kann man die „`CollectionsFamily`“ bilden.  
und zu jeder `Collectionsfamily` ihre „`ElementsFamily`“:

# Die Idee an Familien

Die **Familien partitionieren** die Menge der Objekte.

Dagegen bilden die **Filter** eine **Hierarchie** von Mengen.

z.B.: `PermutationsFamily`, `CyclotomicsFamily`.

Bei endlich präsentierten Gruppen bilden alle Elemente einer solchen Gruppe eine Familie.

Eine **Collection** besteht aus Objekten **derselben Familie**.

Zu jeder Familie kann man die „`CollectionsFamily`“ bilden.  
und zu jeder `Collectionsfamily` ihre „`ElementsFamily`“:

```
gap> f:=CollectionsFamily(CyclotomicsFamily);;
gap> CyclotomicsFamily=ElementsFamily(f);
true
gap> FamilyObj((1,2,3))=PermutationsFamily;
true
```

# Kategorien und Repräsentationen

„Kategorien“ und „Repräsentationen“ sind nichts anderes als **elementare Filter** mit ein bisschen **Philosophie** dahinter!

# Kategorien und Repräsentationen

„Kategorien“ und „Repräsentationen“ sind nichts anderes als **elementare Filter** mit ein bisschen **Philosophie** dahinter!

Objekte in derselben **Kategorie** sollen (**mathematisch**) gleichartige Objekte sein.

Objekte ändern ihre Kategorie **nie!**

# Kategorien und Repräsentationen

„Kategorien“ und „Repräsentationen“ sind nichts anderes als **elementare Filter** mit ein bisschen **Philosophie** dahinter!

Objekte in derselben **Kategorie** sollen (**mathematisch**) gleichartige Objekte sein.

Objekte ändern ihre Kategorie **nie!**

Mathematisch gleiche oder gleichartige Objekte können **unterschiedlich repräsentiert** sein, dann sollten sie in unterschiedlichen **Repräsentationen** liegen.

# Kategorien und Repräsentationen

„Kategorien“ und „Repräsentationen“ sind nichts anderes als **elementare Filter** mit ein bisschen **Philosophie** dahinter!

Objekte in derselben **Kategorie** sollen (**mathematisch**) gleichartige Objekte sein.

Objekte ändern ihre Kategorie **nie!**

Mathematisch gleiche oder gleichartige Objekte können **unterschiedlich repräsentiert** sein, dann sollten sie in unterschiedlichen **Repräsentationen** liegen.

`IsPerm` ist eine Kategorie.

`IsPerm2Rep` und `IsPerm4Rep` sind Repräsentationen.

# Kategorien und Repräsentationen

„Kategorien“ und „Repräsentationen“ sind nichts anderes als **elementare Filter** mit ein bisschen **Philosophie** dahinter!

Objekte in derselben **Kategorie** sollen (**mathematisch**) gleichartige Objekte sein.

Objekte ändern ihre Kategorie **nie!**

Mathematisch gleiche oder gleichartige Objekte können **unterschiedlich repräsentiert** sein, dann sollten sie in unterschiedlichen **Repräsentationen** liegen.

`IsPerm` ist eine Kategorie.

`IsPerm2Rep` und `IsPerm4Rep` sind Repräsentationen.

**Kategorien** kommen in **Operationsdeklarationen** vor, während **Repräsentationen** in Anforderungen für **Methodeninstallationen** benutzt werden.

# Vererbung in GAP

Vererbung funktioniert über **Subfilter**.

# Vererbung in GAP

Vererbung funktioniert über **Subfilter**.

Man deklariert Subfilter und konstruiert Objekte, die in diesen zusätzlichen Subfiltern liegen.

# Vererbung in GAP

Vererbung funktioniert über **Subfilter**.

Man deklariert Subfilter und konstruiert Objekte, die in diesen zusätzlichen Subfiltern liegen.

Braucht man speziellere Methoden, werden diese mit den Subfiltern als zusätzliche Anforderungen installiert.

# Vererbung in GAP

Vererbung funktioniert über **Subfilter**.

Man deklariert Subfilter und konstruiert Objekte, die in diesen zusätzlichen Subfiltern liegen.

Braucht man speziellere Methoden, werden diese mit den Subfiltern als zusätzliche Anforderungen installiert.

## Hypothetisches Beispiel:

```
DeclareCategory("IsGroup", IsObject);
DeclareCategory("IsAbelianGroup", IsGroup);
DeclareOperation("Size", [IsGroup]);
InstallMethod(Size, "for arbitrary groups",
               [IsGroup],
               function(g) ... end);
InstallMethod(Size, "for abelian groups",
               [IsAbelianGroup],
               function(a) ... end);
```

# Die Deklarationen

```
BindGlobal ("BlubbsFamily",  
           NewFamily ("BlubbsFamily"));  
DeclareCategory ("IsBlubb",  
                IsComponentObjectRep);  
DeclareRepresentation ("IsBlubbDenseRep",  
                       IsBlubb, ["wo", "p"]);  
BindGlobal ("BlubbDenseType",  
           NewType (BlubbsFamily, IsBlubbDenseRep));
```

# Die Deklarationen

```
BindGlobal ("BlubbsFamily",  
           NewFamily ("BlubbsFamily"));  
DeclareCategory ("IsBlubb",  
               IsComponentObjectRep);  
DeclareRepresentation ("IsBlubbDenseRep",  
                      IsBlubb, ["wo", "p"]);  
BindGlobal ("BlubbDenseType",  
           NewType (BlubbsFamily, IsBlubbDenseRep));  
  
DeclareOperation ("Blubb", [IsString, IsInt]);  
DeclareOperation ("IsShort", [IsBlubb]);  
DeclareOperation ("NrLetters", [IsBlubb]);
```

# Die Deklarationen

```
BindGlobal("BlubbsFamily",
           NewFamily("BlubbsFamily"));
DeclareCategory("IsBlubb",
               IsComponentObjectRep);
DeclareRepresentation("IsBlubbDenseRep",
                     IsBlubb, ["wo", "p"]);
BindGlobal("BlubbDenseType",
           NewType(BlubbsFamily, IsBlubbDenseRep));

DeclareOperation("Blubb", [IsString, IsInt]);
DeclareOperation("IsShort", [IsBlubb]);
DeclareOperation("NrLetters", [IsBlubb]);

InstallMethod(Blubb, "constructor",
              [IsString, IsInt], function(s, i)
                local r;
                r := rec(wo:=s, p:=i);
                return Objectify(BlubbDenseType, r);
              end);
```

# Die Implementation

```
InstallMethod(IsShort, "for dense Blubbs",  
  [IsBlubbDenseRep],  
  function(bl)  
    return Length(bl!.wo) <= 5;  
  end);
```

# Die Implementation

```
InstallMethod(IsShort, "for dense Blubbs",  
  [IsBlubbDenseRep],  
  function(bl)  
    return Length(bl!.wo) <= 5;  
  end);
```

```
InstallMethod(NrLetters, "for dense Blubbs",  
  [IsBlubbDenseRep],  
  function(bl)  
    return Length(Set(bl!.wo));  
  end);
```

# Die Implementation

```
InstallMethod(IsShort,"for dense Blubbs",  
  [IsBlubbDenseRep],  
  function(bl)  
    return Length(bl!.wo) <= 5;  
  end);
```

```
InstallMethod(NrLetters,"for dense Blubbs",  
  [IsBlubbDenseRep],  
  function(bl)  
    return Length(Set(bl!.wo));  
  end);
```

```
InstallMethod(ViewObj,"for dense Blubbs",  
  [IsBlubbDenseRep],  
  function(bl)  
    Print("<a dense blubb wo=",bl!.wo,  
          " p=",bl!.p,">");  
  end);
```

# Benutzung

Man kann nun Blubb-Objekte wie folgt nutzen:

```
gap> b := Blubb("abac",17);  
<a dense blubb wo=abac p=17>  
gap> NrLetters(b);  
3  
gap> IsShort(b);  
true  
gap> b!.wo;  
"abac"  
gap> b!.p;  
17
```

# Benutzung

Man kann nun `Blubb`-Objekte wie folgt nutzen:

```
gap> b := Blubb("abac",17);  
<a dense blubb wo=abac p=17>  
gap> NrLetters(b);  
3  
gap> IsShort(b);  
true  
gap> b!.wo;  
"abac"  
gap> b!.p;  
17
```

Man sollte Methoden für

- `ViewObj` (zum Anschauen für den Benutzer: kurz!),
- `PrintObj` (wenn möglich GAP-lesbar),
- und ev. `Display` (hübsch formatiert)

installieren.

# Properties

Eine **Property** „XYZ“ ist realisiert durch:

- einen **elementaren Filter** `HasXYZ` und
- einen **elementaren Filter** `XYZ`.

# Properties

Eine **Property** „XYZ“ ist realisiert durch:

- einen **elementaren Filter** `HasXYZ` und
- einen **elementaren Filter** `XYZ`.

Deklariert durch:

```
DeclareProperty("IsShort", IsBlubb);
```

# Properties

Eine **Property** „XYZ“ ist realisiert durch:

- einen **elementaren Filter** `HasXYZ` und
- einen **elementaren Filter** `XYZ`.

Deklariert durch:

```
DeclareProperty("IsShort", IsBlubb);
```

Dies definiert **automatisch**

- einen **elementaren Filter** `HasIsShort`,

# Properties

Eine **Property** „XYZ“ ist realisiert durch:

- einen **elementaren Filter** `HasXYZ` und
- einen **elementaren Filter** `XYZ`.

Deklariert durch:

```
DeclareProperty("IsShort", IsBlubb);
```

Dies definiert **automatisch**

- einen **elementaren Filter** `HasIsShort`,
- einen **elementaren Filter** `IsShort`,

# Properties

Eine **Property** „XYZ“ ist realisiert durch:

- einen **elementaren Filter** `HasXYZ` und
- einen **elementaren Filter** `XYZ`.

Deklariert durch:

```
DeclareProperty("IsShort", IsBlubb);
```

Dies definiert **automatisch**

- einen **elementaren Filter** `HasIsShort`,
- einen **elementaren Filter** `IsShort`,
- eine **Operation** `IsShort`,

# Properties

Eine **Property** „XYZ“ ist realisiert durch:

- einen **elementaren Filter** `HasXYZ` und
- einen **elementaren Filter** `XYZ`.

Deklariert durch:

```
DeclareProperty("IsShort", IsBlubb);
```

Dies definiert **automatisch**

- einen **elementaren Filter** `HasIsShort`,
- einen **elementaren Filter** `IsShort`,
- eine **Operation** `IsShort`,
- eine **Methode** für `IsShort` für Objekte im Filter `IsBlubb` and `HasIsShort`, die im Typ nachschaut, und

# Properties

Eine **Property** „XYZ“ ist realisiert durch:

- einen **elementaren Filter** `HasXYZ` und
- einen **elementaren Filter** `XYZ`.

Deklariert durch:

```
DeclareProperty("IsShort", IsBlubb);
```

Dies definiert **automatisch**

- einen **elementaren Filter** `HasIsShort`,
- einen **elementaren Filter** `IsShort`,
- eine **Operation** `IsShort`,
- eine **Methode** für `IsShort` für Objekte im Filter `IsBlubb` and `HasIsShort`, die im Typ nachschaut, und
- eine **Operation mit Methode** `SetIsShort`.

Max Neunhöffer

```
DeclareAttribute ("NrLetters", IsBlubb);
```

Objekte,  
Operationen und  
Methodenselektion

Die Idee

Typen

Operationen und Methoden

Konventionen

Familien

Kategorien und  
Repräsentationen

Vererbung

Ein Beispiel

Die Deklarationen

Die Implementation

Benutzungsbeispiel

Weitere  
Bequemlichkeiten

Properties

**Attribute**

Debugging

Max Neunhöffer

```
DeclareAttribute("NrLetters", IsBlubb);
```

Objekte,  
Operationen und  
Methodenselektion

Die Idee

Typen

Operationen und Methoden

Konventionen

Familien

Kategorien und  
Repräsentationen

Vererbung

Ein Beispiel

Die Deklarationen

Die Implementation

Benutzungsbeispiel

Weitere  
Bequemlichkeiten

Properties

**Attribute**

Debugging

definiert **automatisch**

- einen elementaren Filter `HasXYZ`,

```
DeclareAttribute ("NrLetters", IsBlubb);
```

definiert **automatisch**

- einen elementaren Filter `HasXYZ`,
- eine Operation `XYZ`.

# Attribute

```
DeclareAttribute("NrLetters", IsBlubb);
```

definiert **automatisch**

- einen elementaren Filter `HasXYZ`,
- eine Operation `XYZ`.

**Vererbt** man von `IsComponentObjectRep` and `IsAttributeStoringRep`, dann sogar noch:

- Eine Operation `SetXYZ` für `[IsBlubb, IsObject]`, die das 2. Argument in die `!.XYZ`-Komponente speichert und `HasXYZ` setzt.

# Attribute

```
DeclareAttribute("NrLetters", IsBlubb);
```

definiert **automatisch**

- einen elementaren Filter `HasXYZ`,
- eine Operation `XYZ`.

**Vererbt** man von `IsComponentObjectRep` and `IsAttributeStoringRep`, dann sogar noch:

- Eine Operation `SetXYZ` für `[IsBlubb, IsObject]`, die das 2. Argument in die `!.XYZ`-Komponente speichert und `HasXYZ` setzt.
- Jede Methode für `XYZ` speichert ihr Ergebnis automatisch dort und setzt `HasXYZ`.

# Attribute

```
DeclareAttribute ("NrLetters", IsBlubb);
```

definiert **automatisch**

- einen elementaren Filter `HasXYZ`,
- eine Operation `XYZ`.

**Vererbt** man von `IsComponentObjectRep` and `IsAttributeStoringRep`, dann sogar noch:

- Eine Operation `SetXYZ` für `[IsBlubb, IsObject]`, die das 2. Argument in die `!.XYZ`-Komponente speichert und `HasXYZ` setzt.
- Jede Methode für `XYZ` speichert ihr Ergebnis automatisch dort und setzt `HasXYZ`.
- Eine hochrangige Methode für `XYZ` für Objekte im Filter `IsBlubb` and `HasXYZ`, die einfach nur `!.XYZ` liefert.

## Beispiel — revisited

Wir können in unserem Beispiel einfach

```
DeclareCategory ("IsBlubb",  
                IsComponentObjectRep);  
DeclareOperation ("IsShort", [IsBlubb]);  
DeclareOperation ("NrLetters", [IsBlubb]);
```

durch

```
DeclareCategory ("IsBlubb",  
                IsAttributeStoringRep);  
DeclareProperty ("IsShort", IsBlubb);  
DeclareAttribute ("NrLetters", IsBlubb);
```

ersetzen und dadurch automatisch Caching erhalten:

## Beispiel — revisited

Wir können in unserem Beispiel einfach

```
DeclareCategory ("IsBlubb",  
                IsComponentObjectRep);  
DeclareOperation ("IsShort", [IsBlubb]);  
DeclareOperation ("NrLetters", [IsBlubb]);
```

durch

```
DeclareCategory ("IsBlubb",  
                IsAttributeStoringRep);  
DeclareProperty ("IsShort", IsBlubb);  
DeclareAttribute ("NrLetters", IsBlubb);
```

**ersetzen und dadurch automatisch Caching erhalten:**

```
gap> b := Blubb ("abac", 17);  
<a dense blubb wo=abac p=17>  
gap> HasNrLetters (b);  
false  
gap> NrLetters (b);;  
gap> HasNrLetters (b);  
true
```

Wenn man sehen will, welche Methoden verfügbar sind:

```
gap> ApplicableMethod(NrLetters, [b], 3, "all");
#I   Searching Method for NrLetters with 1 \
                                     arguments:
#I   Total: 2 entries
#I   Method 1: ``NrLetters: system getter'', \
                                     value: 2*SUM_FLAGS+4
#I   - 1st argument needs \
                                     [ "IsAttributeStoringRep", \
                                     "Tester(NrLetters)" ]
#I   Method 2: ``NrLetters: for dense \
                                     Blubbs'', value: 3
#I   Skipped:
[ function( bl ) ... end ]
```

## Das Beispiel nochmal komplett

```
BindGlobal("BlubbsFamily",
           NewFamily("BlubbsFamily"));
DeclareCategory("IsBlubb",
               IsAttributeStoringRep);
DeclareRepresentation("IsBlubbDenseRep",
                    IsBlubb, ["wo", "p"]);
BindGlobal("BlubbDenseType",
           NewType(BlubbsFamily, IsBlubbDenseRep));

DeclareOperation("Blubb", [IsString, IsInt]);
DeclareProperty("IsShort", IsBlubb);
DeclareAttribute("NrLetters", IsBlubb);

InstallMethod(Blubb, "constructor",
              [IsString, IsInt], function(s, i)
                local r;
                r := rec(wo:=s, p:=i);
                return Objectify(BlubbDenseType, r);
              end);
```

# Beispiel komplett, Fortsetzung

```
InstallMethod(IsShort,"for dense Blubbs",  
  [IsBlubbDenseRep],  
  function(bl)  
    return Length(bl!.wo) <= 5;  
  end);
```

```
InstallMethod(NrLetters,"for dense Blubbs",  
  [IsBlubbDenseRep],  
  function(bl)  
    return Length(Set(bl!.wo));  
  end);
```

```
InstallMethod(ViewObj,"for dense Blubbs",  
  [IsBlubbDenseRep],  
  function(bl)  
    Print("<a dense blubb wo=",bl!.wo,  
          " p=",bl!.p,">");  
  end);
```