

## Objective-C: Geschichte

Entwickelt von Brad J. Cox in den frühen 80ern

- 1986: Cox: "Object-Oriented Programming, An Evolutionary Approach"
- 1988: NeXTStep
- 1994: OpenStep-API (Sun und NeXT), OPENSTEP als Implementierung, beides von NeXT
- 1994: GNUStep als freie Implementierung von OpenStep
- 1996: Appel kauft NeXT
- 2001: OS X mit Cocoa als NeXTSTEP-Nachfolger

## Philosophie der Sprache

Hauptfeind der Softwareentwicklung ist die Veränderung

- Anforderungen sind zu Beginn nicht klar und ändern sich
  - Software entwickelt sich über die Zeit
- ⇒ die Sprache muss Veränderung unterstützen

## Hybride Verteidigung

Die Sprache sollte Möglichkeiten bieten für:

- Klassische Antwort: OO unterstützt Kapselung und Vererbung
- Entkoppelung: dynamic binding, latent typing
- Formbarkeit: Verwendung von Laufzeittechniken anstelle statischer Techniken
- Wiederverwendung von Legacy-Code: Einbettung in bestehende Sprache

## Vorab-Bemerkungen zur Geschwindigkeit

- Objective-C ist latent getypt
- Klassen können zur Laufzeit manipuliert werden

⇒ Viel passiert erst zur Laufzeit

Aber: Immer noch schnell genug

- OS X baut komplett auf Objective-C auf
- NeXTStep lief auf einem 68030 mit 40MHz

## Objective-C und C

- Einbettung in C
- Erweiterungen um
  - Klassendeklaration und -definition
  - Methodenaufrufe
  - Typsystem für Objekte
- Außerdem
  - `#import`
  - Neue String-Literale: `@"..."` liefern String-Objekt

## Typen in Objective-C

- `BOOL`: Typ für Booleans ( `YES` und `NO` )
- `Class`: Typ für Klassenobjekte
- `SEL`: Typ für Methodennamen
- Klassendefinitionen führen neue Typen ein
- Spezieller Typ `id` umfasst alle Objekte

## Deklaration von Klassen

Syntax:

```
@interface ClassName : ItsSuperclass {  
    instance variable declarations  
}  
method declarations  
@end
```

- Keine Überladung
- In Header-Dateien ( `.h` )

## Beispiel

```
@interface Shape : NSObject {  
    int pos_x;  
    int pos_y;  
}  
@end
```

## Methodendeklarationen

Keine Parameter:

- (*RetType*) *Method*

Ein Parameter:

- (*RetType*) *Method*: (*ParaType*) *ParaName*;

Zwei Parameter:

- (*RetType*) *Method*: (*ParaType1*) *ParaName1*  
    *ParaLabel*: (*ParaType2*) *ParaName2*;

Methodenname ist die Konkatenation aus Methode und Labels inklusive :

## Beispiele:

```
-(NSString*)getName;  
-(void)setX:(int)x;  
-(void)setPosWithX:(int)x andY:(int)y;
```

Methodennamen sind: `getName`, `setX:` und `setPosWithX:andY:`

## Methodenaufruf

Objective-C-Terminologie: Einem Objekt (dem *Empfänger*) wird eine Nachricht (Message) gesendet, die besagt, dass es eine Methode ausführen soll

Syntax für Nachrichtenausdrücke: `[receiver message]`

- *receiver* ein Ausdruck, der zu einem Objekt auswertet
- *message* ist Name der auszuführenden Methode und Argumente
- Argumente folgen nach `:` und Label

## Beispiele

```
[s getName]; // -> "Shape"  
[x setX:20];  
[x setPosWithX:100 andY:150];
```

## Typchecks in Objective-C

- Ist der statische Typ des Empfänger-Objekts eine Klasse, so überprüft der Typchecker Methode und Argumente
- Ist der statische Typ aber `id` erfolgt keine Überprüfung

## Definition von Klassen

Syntax:

```
#import "ClassName.h"  
@implementation ClassName  
method definitions  
@end
```

Methodendefinitionen = Deklaration + { ... }

In Quell-Dateien ( `.m` )

## self

Innerhalb von Methoden steht der Empfänger in der impliziten Variable `self` zur Verfügung

Eine Anwendung:

Aufruf anderer Methoden des Empfängers via Nachricht an `self`

## Zugriff auf Instanzvariablen

Mittels Objektzeiger und `->`

Methoden auf Instanzvariablen des Empfängers: Name genügt (Compiler fügt Zugriff auf `self` hinzu)

## Beispiel

```
@implementation Shape
-(NSString*)getName{
    return @"Shape";
}
-(void)setX:(int)x{
    pos_x = x;
}
-(void)setPosWithX:(int)x andY:(int)y{
    pos_x = x;
    pos_y = y;
}
@end
```

## Klassenobjekte

Zu jeder Klasse gibt es ein Objekt, das die Informationen der Klasse zur Laufzeit enthält

- Typ der Klassenobjekte: **Class**
- Jedes Objekt enthält implizit eine Instanzvariable **isa**, die auf das Objekt der Klasse zeigt

## Klassenmethoden

Methoden, die auf Klassenobjekten operieren  
Deklaration und Definition mit **+** anstelle von **-** einleiten

## Klassenvariablen

... gibt es nicht

Simulation mit globalen **static**-Variablen (d.h. nur in der Übersetzungseinheit sichtbar)

Hilfreich: Klassenmethode **initialize**

## Schutzmechanismen

Instanzvariablen können mit `@public`, `@private` und `@protected` geschützt werden

Default ist `@protected`

Semantik analog zu C++

Aber: Methoden sind immer sichtbar

## Beispiel zu Schutzmechanismen

```
@interface Shape : NSObject {
    @private
        int hidden;
    @protected
        int pos_x;
        int pos_y;
    @public
        int readme;
}
-(int)getX;
@end
```

## Abstrakte Klassen

"Beitrag einer abstrakten Klasse sind nicht ihre Instanzen, sondern die Methoden und Instanzvariablen, die sie Subklassen zur Verfügung stellt"

Also kein sprachlicher Mechanismus, sondern nur Konvention

Mögliche Simulation: "abstrakte" Methoden werfen Exception

Beispiele: `NSObject`, `NSArray`

## Referenzsemantik

- In Objective-C sind Objekte stets Zeiger auf Heap-Daten
- `nil` zeigt auf leeres Objekt
- Speicherverwaltung semi-automatisch
- Wir ignorieren Speicherfreigabe hier völlig
- Für Wertsemantik entscheidet (per Konvention) `copy`-Methode, wie die Instanzvariablen kopiert werden

## Allokation und Initialisierung

- Objective-C kennt keine Konstruktoren
- Klassen legen mittels Klassenmethode `alloc` Speicherplatz für Objekte an
- Beliebige Methoden initialisieren Objekte (oft: `init` und `initWith...`)
- Achtung: `init` muss `super` immer mit initialisieren

## Beispiel

```
-(id)initWithX:(int)x andY:(int)y;

-(id)initWithX:(int)x andY:(int)y{
    [super init];

    pos_x = x;
    pos_y = y;

    return self;
}

Shape * s = [[Shape alloc] initWithX:100 andY:150];
```

## super

Keine Variable (wie `self`) sondern nur Flag für den Compiler  
Kann als Empfänger in Methodenaufrufen verwendet werden

Semantik wie in Java/C++: Beginne Suche nach  
Methodenimplementierung in der Superklasse

## Klassenobjekte im Einsatz

`NSMatrix` stellt Zellen als Gitter dar

Zellen sind Subklassen von `NSCell`

`initWithFrame:cellClass:` teilt `NSMatrix` Klasse der Zellen  
mit

`NSMatrix` sendet `alloc/init` an diese Klasse um neue Zellen zu  
erstellen

Vorteil: Benutzer muss nicht von `NSMatrix` erben, um Matrix für  
neue Zellenklasse zu erstellen

## Vererbung

- Nur Einfachvererbung
- Methoden können überschrieben werden
- Konvention: überschriebene Methoden im Interface deklarieren
- Oberste Superklasse: `NSObject`

## Beispiel

```
@interface Rectangle : Shape {
    int width;
    int height;
}
-(id)initWithX:(int)x andY:(int)y
        height:(int)h width:(int)w;
-(NSString*)getName;
@end
```

## Beispiel (Fortsetzung)

```
@implementation Rectangle
-(NSString*)getName{
    return @"Rectangle";
}
-(id)initWithX:(int)x andY:(int)y
        height:(int)h width:(int)w{
    [super initWithX:x andY:y];
    height = h
    width = w;
    return self;
}
```

## Vererbung und Initialisierung

Initialisierungsmethoden werden mitvererbt (keine Konstruktoren!)

Subklassen sollten sie überschreiben

Auch `init` von `NSObject`

`NSObject` vererbt außerdem `new = alloc + init`

## Beispiel

```
@implementation Shape
(id) init{
    return [self initWithX:0 andY:0];
}
...
@end

@implementation Rectangle
(id) initWithX:(int)x andY:(int)y{
    return [self initWithX:x andY:y height:0 width:0]
}
...
@end
```

## Konvention zu init-Methoden

Konvention: `init`-Methoden liefern `nil` wenn Initialisierung nicht möglich

Ausweg, wenn `init` nicht implementiert werden kann

⇒ Müssen Rückgabe von `[super init]` immer auf `nil` überprüfen

## Polymorphismus und dynamische Bindung

- Objekte können unterschiedlich auf die gleiche Nachricht (Message) reagieren
- Methodenname wählt zur Laufzeit Implementierung der Methode aus
- Mittels `isa`-Instanzvariable findet das RTS die Methode

## Beispiel

```
Shape *s = [[Rectangle alloc] init];
[s getName]; // -> @"Rectangle"
```

## Selektoren

Zur Methodenauswahl wird nicht der Methodenname sondern sog. *Selektor* verwendet

Selektor ist eindeutiger Bezeichner für Methode, der schnell verglichen werden kann

Implementierung: Selektor ist Index in Liste aller Methodennamen

Selektoren sind normale Werte:

- Typ `SEL` umfasst Selektoren
- Direktive `@selector` liefert Selektor zu Methodennamen
- `performSelector:` leistet Methodenaufruf mittels Selektor

## Beispiel

```
SEL one = @selector(first);
SEL two = @selector(second);

void do(id someObj, SEL s){
    ...
    [someObj performSelector:s];
    ...
}

do([SomeClass new], one);
do([SomeClass new], two);
```

## Das Target/Action-Paradigma

Normaler Methodenaufruf: Objekt variabel, Methodenname fest

⇒ eventuell störende Koppelung

Selektoren ermöglichen Abstraktion über Methodenname

Target/Action-Paradigma: Sowohl Empfänger (Target) als auch Methodenname (Action) sind Parameter

Praktisch für GUIs um View mit Controller oder Model mit Controller zu verbinden

## Beispiel

Callback für Button in Cocoa-GUI:

```
NSButton * acceptButton = [NSButton new];
[acceptButton setTarget:installWizard];
[acceptButton setAction:@selector(acceptLicence)];
```

Beobachtung: Weder Button noch Anwendung müssen angepasst werden

## Categories

- Für bestehende Klasse Methoden hinzufügen und überschreiben
- Alternative zur Vererbung
- Einschränkungen:
  - Kein Zugriff auf überschriebene Methoden
  - Hinzufügen von Instanzvariablen nicht möglich
- Anwendungen:
  - Klassen erweitern, auf deren Quellcode man keinen Zugriff hat
  - Implementierung von Methoden für verschiedene Klassen zusammenfassen
  - Aufteilung der Implementierung in mehrere Dateien

## Syntax für Categories

- Interface:

```
#import "ClassName.h"
@interface ClassName (CategoryName)
method declarations
@end
```

- Implementierung:

```
#import "CategoryName.h"
@implementation ClassName (CategoryName)
method definitions
@end
```

## Beispiel

Die Category `FortyTwo` fügt allen Objekten die Methode `fortyTwo` hinzu.

```
@interface NSObject (FortyTwo)
-(int) fortyTwo;
@end
```

```
@implementation NSObject (FortyTwo)
-(int) fortyTwo{
    return 42;
}
@end

...
[[Rectangle new] fortyTwo]; // -> 42
[[NSButton new] fortyTwo]; // -> 42
```

## Protocols

Protocol: Liste von Methoden

Klassen können Protokolle implementieren

Zwei Arten:

- Informal Protocols: Methoden müssen nicht implementiert werden
- Formal Protocols: Entsprechen Interfaces in Java

## Anwendung von Protocols

- Anforderungen an andere Objekte formulieren
- Zu einem Objekt nur die Schnittstelle, nicht aber die Klasse bekanntgeben
- Gemeinsamkeiten zwischen Klassen festhalten, die nicht über Vererbung verbunden sind

## Informal Protocols

- Category für `NSObject`, aber nur Interface-Deklaration
- Klassen die dem Protokoll entsprechen wollen, implementieren die Methoden
- Keine weitere Unterstützung vom Compiler/RTS
- Es müssen nicht alle Methoden implementiert werden
  - ⇒ Aufrufer überprüft i.d.R. ob die Methoden implementiert sind
  - ⇒ Nützlich für Delegation

## Formal Protocols definieren

Syntax:

```
@protocol ProtocolName  
method declarations  
@end
```

Außerdem:

```
@protocol ProtocolName <protocol list>  
method declarations  
@end
```

Vorwärtsdeklarationen

```
@protocol ProtocolName;
```

## Beispiel

```
@protocol Printable  
-(void)print;  
@end
```

## Formal Protocols implementieren

Implementierung durch Klasse:

```
@interface Class : ItsSuperclass <protocol list> ...
```

Implementierung durch Category:

```
@interface Class (CategoryName) <protocol list> ...
```

## Typen mit formal Protocols

Protocols sind keine Typen

Stattdessen: Einschränkung bestehender Typen auf Protocol-Implementierer

Erweiterung der Typ-Syntax: `type <protocol list>`

```
id <Printable> p;  
Shape <Printable> * ps;
```

## Posing

Klasse gibt sich als ihre Superklasse aus

⇒ Ersetzt alle Methoden der Superklasse durch die Methoden der abgeleiteten Klasse

```
+ (void)poseAsClass:(Class)mySuperClass
```

Gefährlich, aber zwei Vorteile gegenüber Categories:

- Methoden aus Categories können überschrieben werden
- Auf überschriebene Methoden kann mittels `super` zugegriffen werden

## Laufzeitchecks

Eine Auswahl:

- - (BOOL)conformsToProtocol:(Protocol \*)aProtocol
- - (BOOL)respondToSelector:(SEL)aSelector
- - (BOOL)isKindOfClass:(Class)aClass
- - (BOOL)isMemberOfClass:(Class)aClass
- + (BOOL)instancesRespondToSelector:SEL

Populär in Objective-C

Wird oft verwendet, um flexibel auf andere Klassen reagieren zu können (Entkoppelung)

## Entwicklungsumgebung für Objective-C

OS X kommt mit kompletter IDE (Xcode) inklusive Interface Builder (IB)

Dynamik von Objective-C ermöglicht dabei:

- GUI kann im IB sofort ausprobiert werden
- IB erzeugt keinen Code sondern dumpst Objekte
- IB initialisiert Instanzvariablen, die auf andere Objekte verweisen (Outlets)

## Dynamik in Objective-C und Java

Größter Unterschied zwischen den Sprachen ist die Dynamik:

- ObjC hat Dynamik eingebaut, Javas Reflection wirkt aufgesetzt
- Was guter ObjC-Stil ist, ist in Java möglicherweise schlechtes Design

→ "gegen die Sprache programmieren"

### Beispiel: Shape.h

```
@interface Shape : NSObject {
    NSPoint origin;
}
-(id)initWithOrigin:(NSValue *)o;
@end
```

### Shape.m

```
#import "Shape.h"

@implementation Shape
-(id)initWithOrigin:(NSValue *)o{
    if (![super init])
        return nil;
    origin = [o pointValue];
    return self;
}
@end
```

## Line.h

```
#import "Shape.h"

@interface Line : Shape {
    NSPoint end;
}
-(id) initWithStart:(NSValue*) s andEnd:(NSValue*) e;
@end
```

## Line.m

```
#import "Line.h"

@implementation Line
-(id) initWithStart:(NSValue*) s andEnd:(NSValue*) e{
    if (![super initWithOrigin:s])
        return nil;
    end = [e pointValue];
    return self;
}
@end
```

## Rectangle.h

```
#import "Shape.h"

@interface Rectangle : Shape {
    NSSize s;
}
-(id) initWithUpperLeft:(NSValue *) s
    andLowerRight:(NSValue *) e;
@end
```

## Rectangle.m

```
#import "Rectangle.h"

@implementation Rectangle
-(id) initWithUpperLeft:(NSValue *) start
    andLowerRight:(NSValue *) end{
    if (![super initWithOrigin:start])
        return nil;
    NSPoint startp = [start pointValue];
    NSPoint endp = [end pointValue];
    s.width = endp.x - startp.x;
    s.height = endp.y - startp.y;
    return self;
}
@end
```

## Jetzt: Graphischer Editor

Abgespeckte Version des Java-Editors

Neue Formen können sich registrieren

Wiederverwendung der `shape`-Klassen

Verschiedene Formen können gemalt werden

Auswahl der momentanen Form über das Menü

Wieder ohne Speichermanagement...

## Die Mitspieler

- `Shape`, `Line`, `Rectangle`: repräsentieren die Formen
- `store`: Das Model, speichert die gemalten Formen
- `ShapeRegistry`: verwaltet die registrierten Formen und die momentane Form
- `Editor`: Delegate für `NSApp`
- `EditorViewController`: View und Controller

## Malen

Anwendung muss irgendwie malen können

Protocol `Paintable`

```
@protocol Paintable
- (void) paint;
@end
```

## Shapes malen

Neuer Aspekt für Shapes

→ Categories für Shape-Klassen in `PaintableShapes.h`

```
#import "Paintable.h"
#import "Line.h"
#import "Rectangle.h"

@interface Line (PaintableMGLLine) <Paintable>
- (void) paint;
@end

@interface Rectangle (PaintableRectangle) <Paintable>
- (void) paint;
@end
```

## PaintableShapes.m

```
#import "Paintable.h"
#import "Line.h"
#import "Rectangle.h"

@implementation Line (PaintableLine)
-(void) paint {
    [NSBezierPath strokeLineFromPoint:origin
                             toPoint:end];
}
@end
@implementation Rectangle (PaintableRectangle)
-(void) paint {
    NSRect r = {origin, s};
    [NSBezierPath strokeRect:r];
}
@end
```

## Beobachtung

Das war: Implementierung von Methoden für verschiedene Klassen zusammenfassen

## Das Model: Store speichern

store-Klasse: Wrapper um NSMutableArray

Zusätzlich: Controller informieren, wenn neues Objekt dazukommt

## Notifications

In OpenStep eingebautes Konzept zur Benachrichtigung anderer Objekte

Benannte Nachrichten können versendet und empfangen werden

## Store.h

```
@interface Store : NSObject {
    NSMutableArray* elements;
}
- (void)addShape:(id)s;
- (NSEnumerator *)shapeEnumerator;
@end
```

## Erzeugen neuer Shape-Objekte

Controller muss neue Objekte erzeugen, wenn Benutzer Maus zieht

Klasse der Objekte ist jedoch unbekannt

⇒ Neue Klassen registrieren ihr Klassenobjekt

Initialisierungsmethode ebenfalls unbekannt

⇒ Neue Klassen registrieren Selektor, der initialisiert

Zusätzlich:

- Eintrag im Menü
- Shortcut im Menü

## Store.m

```
@implementation Store
-(id)init {
    if (![super init])
        return nil;
    elements = [NSMutableArray new];
    return self;
}
- (void)addShape:(id)s{
    [elements addObject:s];
    [[NSNotificationCenter defaultCenter]
        postNotificationName:@"ShapeAdded" object:self];
}
- (NSEnumerator *)shapeEnumerator{
    return [elements objectEnumerator];
}
@end
```

## ShapeRegistry

Erlaubt das Registrieren neuer Klassen

Hilfsklasse `ShapeDesc` speichert Klasse und Selektor

Verwaltet momentan ausgewählte Shape

Außerdem: Benutzer wählt Shape über Menü aus

- ⇒ Brauchen Abbildung Menüeintrag->Shape

## ShapeRegistry.h

```
@interface ShapeRegistry : NSObject
{
    NSMutableDictionary * shapes;
    Class currentShapeClass;
    SEL initShape;
}
-(id)init;
-(void)registerShape:(Class)shapeClass
    init:(SEL)init name:(NSString*)name
    key:(NSString*)key;
-(void)setCurrentShape:(Class)shapeClass
    init:(SEL)init;
-(Class)getCurrentClass;
-(SEL)getCurrentSEL;
-(NSArray *)allKeys;
-(Class)getClassForMenuItem:(NSMenuItem*)mi;
-(SEL)getSELForMenuItem:(NSMenuItem*)mi;
@end
```

## ShapeRegistry.m

```
@implementation ShapeRegistry

-(id)init{
    if (![super init])
        return nil;
    shapes = [NSMutableDictionary new];
    currentShapeClass = [Line class];
    initShape = @selector(initWithStart:andEnd:);
    [self registerShape:[Line class]
        init:@selector(initWithStart:andEnd:)
        name:@"Line" key:@"l"];
    // dto für Rectangle
    return self;
}
```

## ShapeRegistry.m (Fortsetzung)

```
-(void)setCurrentShape:(Class)shapeClass
    init:(SEL)init{
    currentShapeClass = shapeClass;
    initShape = init;
}
-(void)registerShape:(Class)shapeClass
    init:(SEL)init
    name:(NSString *)name
    key:(NSString *)key{
    NSMenuItem *m = [[NSMenuItem alloc]
        initWithTitle:name
        action:
            @selector(setCurrentShapeFromMenu:) }
        keyEquivalent:key];
    [shapes setObject:[[ShapeDesc alloc]
        newClass: shapeClass
        init:init] forKey:m];
}
```

## ShapeRegistry.m (Fortsetzung)

```
-(Class)getClassForMenuItem:(NSMenuItem*)mi{
    ShapeDesc *desc = [shapes objectForKey:mi];
    return [desc class];
}
-(SEL)getSELForMenuItem:(NSMenuItem*)mi{
    ShapeDesc *desc = [shapes objectForKey:mi];
    return [desc initSelector];
}
```

## View und Controller

Hier zusammengelegt

Normalerweise erzeugt IB das View

View verwaltet Bildschirmausschnitt und erhält Mausereignisse

Außerdem Nachrichten vom Model

## EditorViewController.h

```
@interface EditorViewController : NSView
{
    NSPoint mouseDown;
    Store* model;
    ShapeRegistry* shapeReg;
}
- (id)initWithFrame:(NSRect)frameRect model:(Store*)s
    shapeReg:(ShapeRegistry*) sr;
- (void)drawRect:(NSRect)rect;
- (void)shapeAdded:(NSNotification *)notification;
- (void)mouseDown:(NSEvent *)theEvent;
- (void)mouseUp:(NSEvent *)theEvent;
@end
```

## EditorViewController.m

```
@implementation EditorViewController
-(id)initWithFrame:(NSRect)frameRect model:(Store*) s
    shapeReg:(ShapeRegistry*) sr{
    if (![super initWithFrame:frameRect])
        return nil;

    model = s;
    shapeReg = sr;

    mouseDown = NSMakePoint (10, 20);;
    [[NSNotificationCenter defaultCenter]
    addObserver:self
    selector:@selector(shapeAdded:)
    name:@"ShapeAdded" object:model];
    return self;
}
```

## Zeichnen in EditorViewController.m

```
- (void)drawRect:(NSRect)rect
{
    [[NSColor whiteColor] set];
    NSRectFill([self bounds]);
    [[NSColor blackColor] set];
    id <Paintable> shape;
    NSEnumerator *enumerator = [model shapeEnumerator];
    while ((shape = [enumerator nextObject]) != nil){
        [shape paint];
    }
}
```

## Ereignisse in EditorViewController.m

```
- (void)shapeAdded:(NSNotification *)notification{
    [self setNeedsDisplay:YES];
}
- (void)mouseDown:(NSEvent *)theEvent
{
    mouseDown = [theEvent locationInWindow];
}
```

## Maus hoch!

```
- (void)mouseUp:(NSEvent *)theEvent
{
    NSPoint mouseUp;

    mouseUp = [theEvent locationInWindow];

    NSValue* mouseUpValue =
        [NSValue valueWithPoint: mouseUp];
    NSValue* mouseDownValue =
        [NSValue valueWithPoint: mouseDown];
    id s = [[[shapeReg getClass] alloc]
            performSelector:[shapeReg getCurrentSEL]
            withObject:mouseDownValue
            withObject:mouseUpValue];
    [model addShape:s];
}
```

## Delegate für NSApp

Initialisierung der Applikation  
Fenster und Menü erzeugen  
M,V,C erzeugen  
Action für Menüeintrag implementieren

## Editor.m

```
- (void) createMenu
{
    NSMenu *menu;
    menu = ([NSMenu new]);
    NSArray * keys = [shapeReg allKeys];
    int count = [keys count];
    int i;
    for (i = 0; i < count; i++){
        NSMenuItem* mi =
            (NSMenuItem *)[keys objectAtIndex:i];
        [menu addItem:mi];
    }

    [menu addItemWithTitle: @"Quit"
        action: @selector (terminate:)
        keyEquivalent: @"q"];
    [NSApp setMainMenu: menu];
}
```

## Action implementieren

Für das Auswählen von Menüeinträgen

```
-(void)setCurrentShapeFromMenu:(id)menuItem{
    [shapeReg
     setCurrentShape:[shapeReg
                      getClassForMenuItem:menuItem]
     initWith:[shapeReg getSELForMenuItem:menuItem]];
}
```

## Zusammenfassung Editor

Entkoppelung von Shapes und Editor durch:

- `id` für Speicherung
- Protocol für Deklaration von `paint`
- Kategorie für Implementierung von `paint`

## Zusammenfassung

Objective-C macht Spaß!

Wenige Sprachmittel reichen aus, um ein OO-C zu machen

Basissprache bleibt erhalten

Laufzeitmechanismen ermöglichen Entkoppelung

Gut für GUI-Programmierung geeignet

Nachteile:

Wenig statische Sicherheit

- Keine automatische Speicherverwaltung
- Unsicherheit von C bleibt (Casts, Zeiger)