8 Design Patterns for Multimedia Software

8.1 Design Patterns: The Idea

8.2 Classification Space for Multimedia Software

8.3 Patterns for Multimedia Software

8.4 Gang-of-Four Patterns Applied to Multimedia

- Factory Method
- Template Method
- State

Literature:

Gamma/Helm/Johnson/Vlissides: Design Patterns, Addison-Wesley 1994
(= „Gang of Four“, „GoF“)

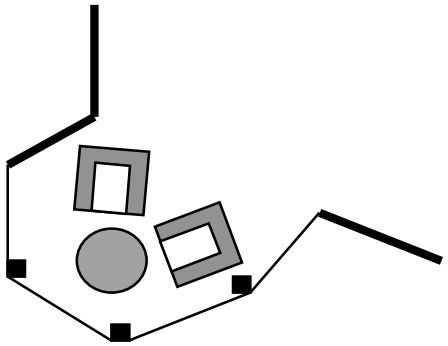
Design Patterns

- A *design pattern* is a generic solution for a class of recurring programming problems
  - Helpful idea for programming
  - No need to adopt literally when applied
- Origin:
  - Famous book by Gamma/Helm/Johnson/Vlissides (“Gang of Four”)
    - » List of standard design patterns for object-oriented programming
    - » Mainly oriented towards graphical user interface frameworks
    - » Examples: Observer, Composite, Abstract Factory
- Frequently used in all areas of software design
- Basic guidelines:
  - Patterns are not invented but recovered from existing code
  - Pattern description follows standard outline
    - » E.g.: Name, problem, solution, examples

Window Place: Architectural Pattern

Christopher Alexander et al., A Pattern Language, 1977
(quoted in Buschmann et al. 1996)

- **Problem:** In a room with a window and a sitting opportunity users have to decide whether to have a look or to sit.
- **Solution:**
  At least one window of the room shall provide a sitting place.
- **Structure:**



Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice.
Christopher Alexander et al., A Pattern Language

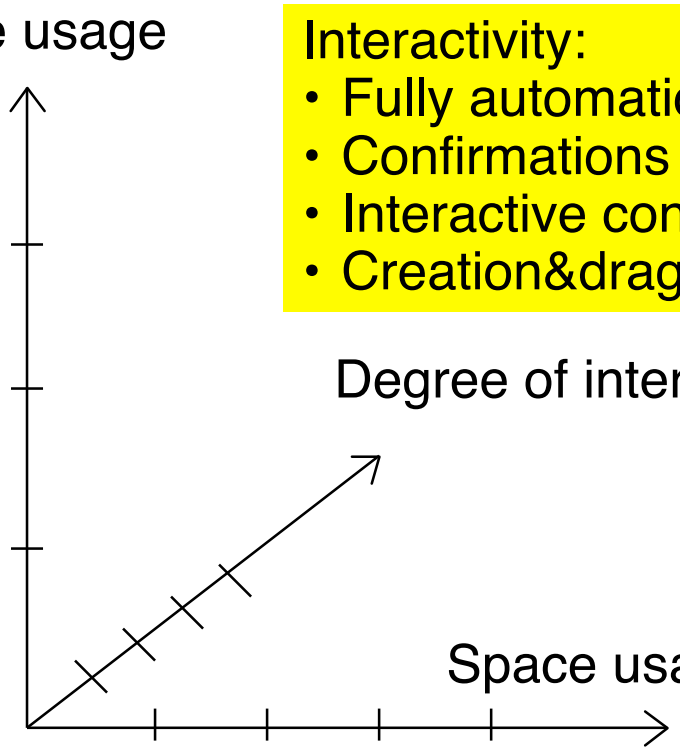
Description of a Design Pattern

- Name
- Problem
  - Motivation
  - Application area
- Solution
  - Structure (class diagram)
  - Participants (usually class, association und operation names):
    - » Role name, i.e. place holders for parts of implementation
    - » Fixed parts of implementaton
  - Collaboration (sequence of events, possibly diagrams)
- Discussion
  - Pros and cons
  - Dependencies, restrictions
  - Special cases
- Known uses

8 Design Patterns for Multimedia Software

Classification Space

Time usage:
- Still picture
- Linear progress
- Interaction dependent progress

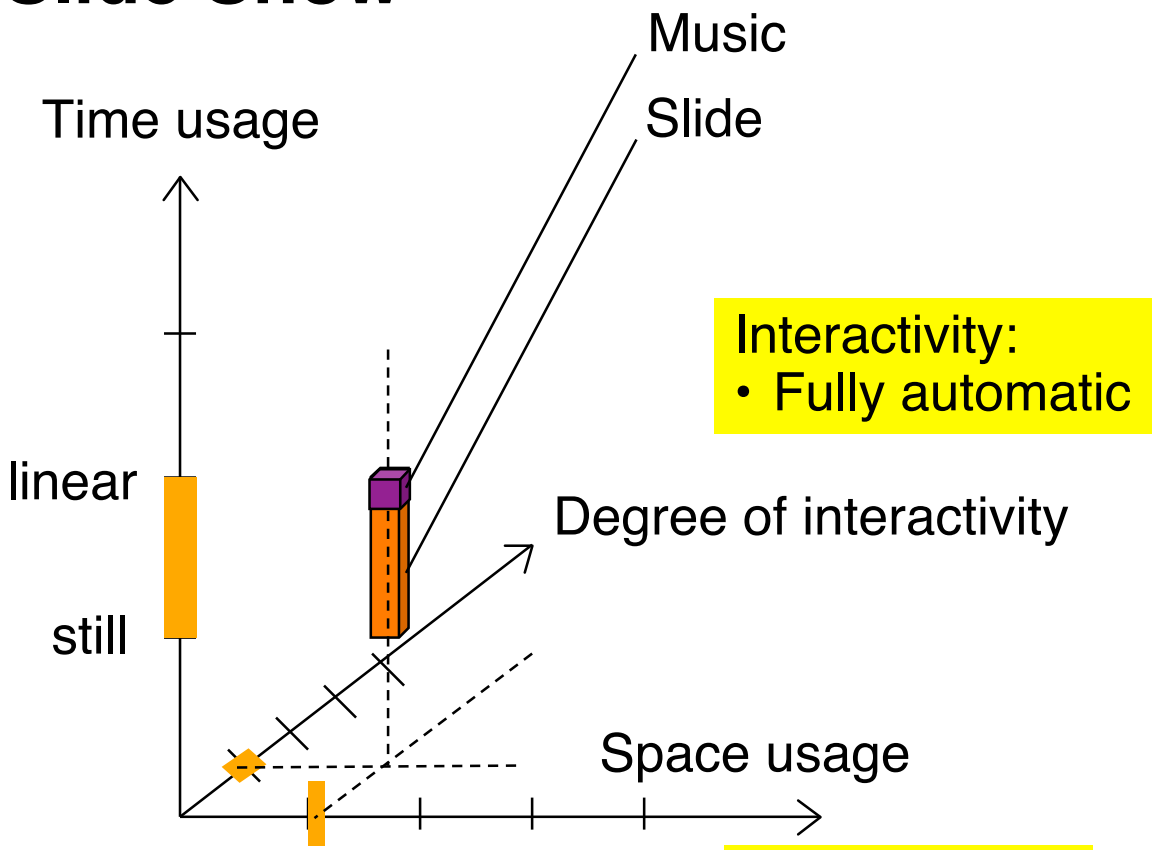


Interactivity:
- Fully automatic (passive)
- Confirmations & questions (reactive)
- Interactive controls (proactive)
- Creation&dragging of objects (directive)

Space usage:
- Static layout
- Scenes
- Scenes & objects
- Fully dynamic

Example 1: Slide Show

Time usage:
- Still picture & Linear progress

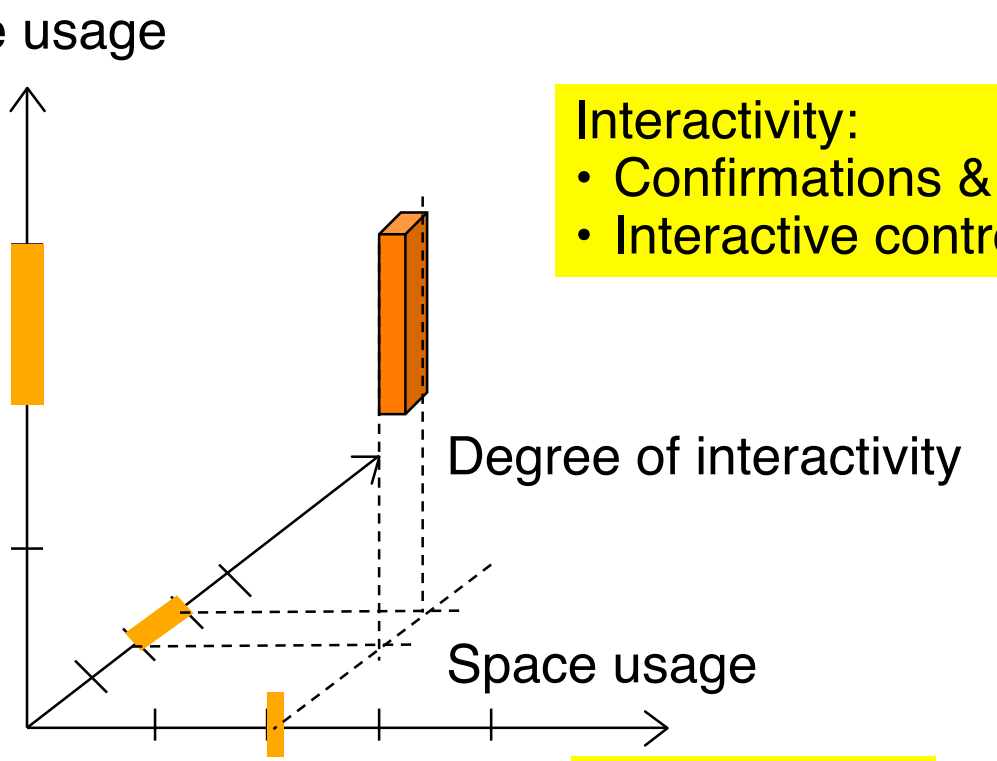


Interactivity:
- Fully automatic

Space usage:
- Static layout

Example 2: Animated Product Presentation

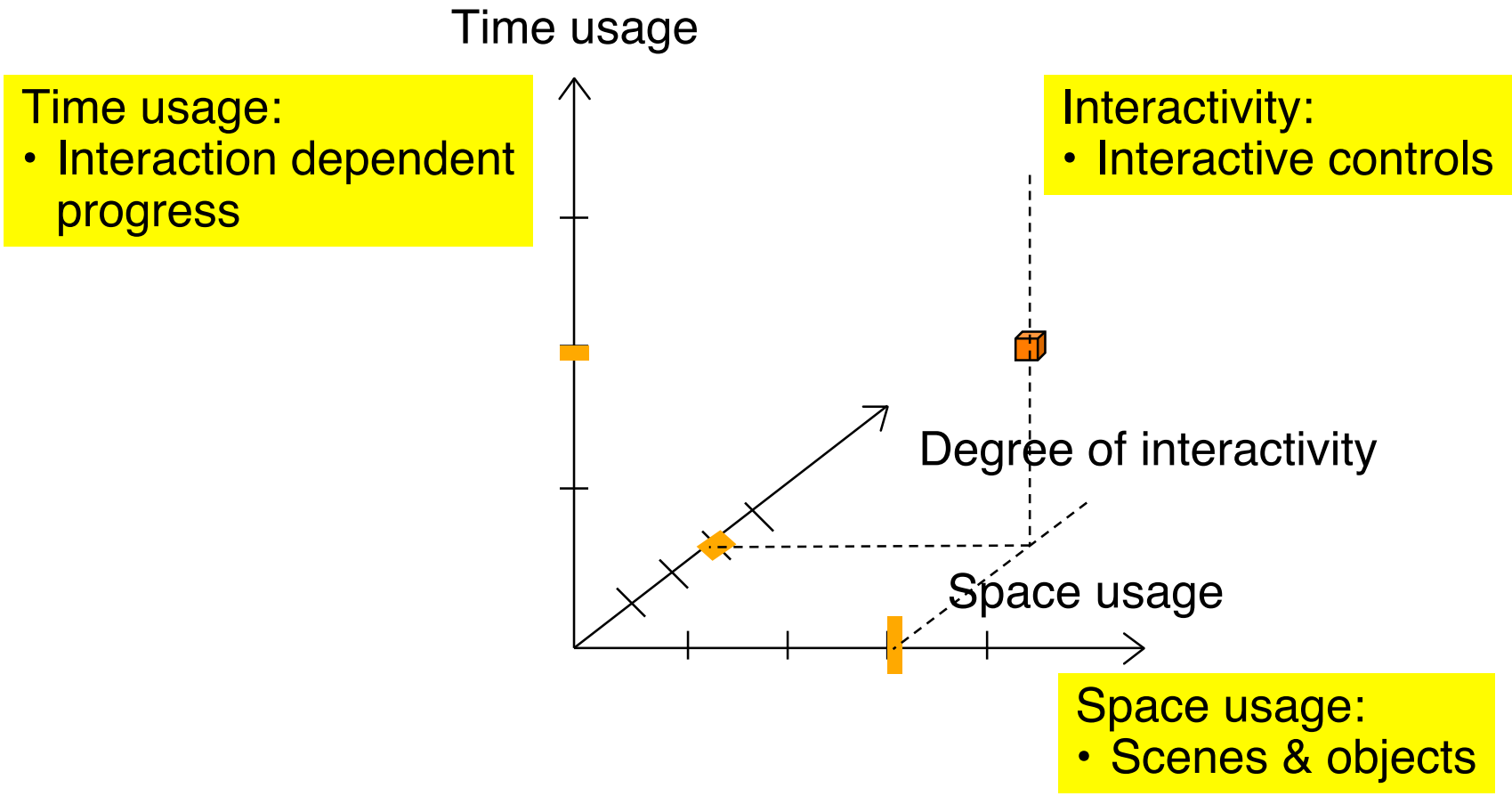
Time usage:
- Linear progress
- Interaction dependent progress



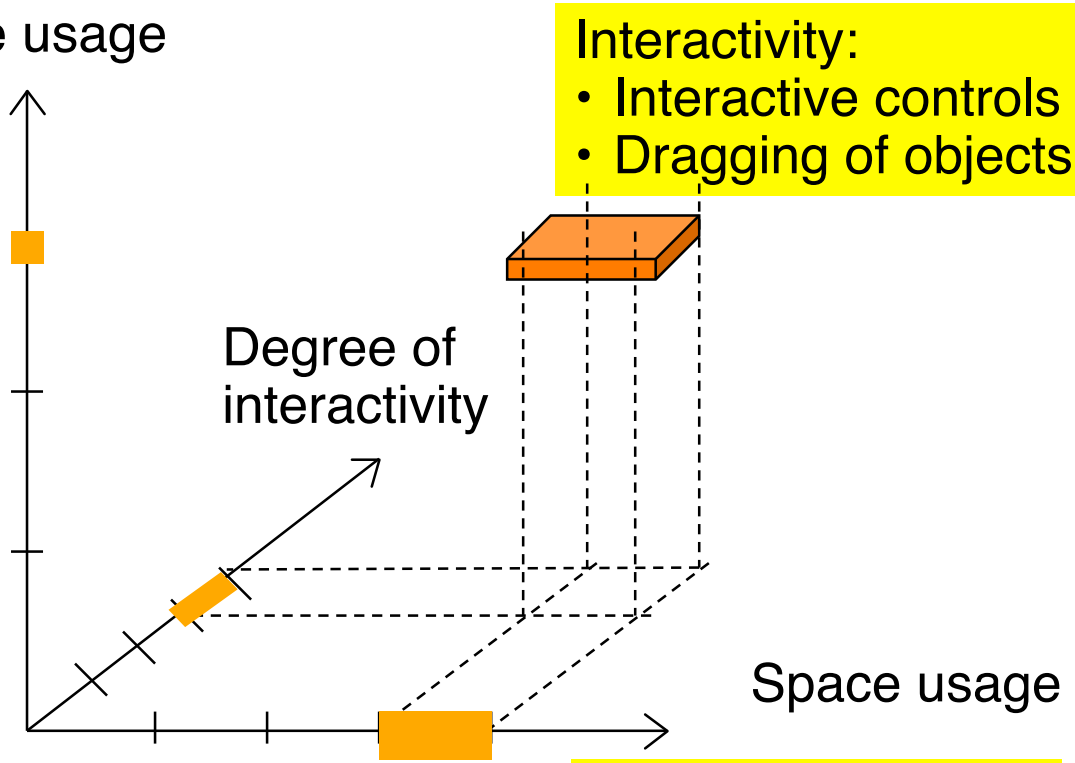
Interactivity:
- Confirmations & questions
- Interactive controls

Space usage:
- Scenes

Example 3: Game



Example 4: Virtual World



Time usage:
- Interaction dependent progress

Interactivity:
- Interactive controls
- Dragging of objects

Space usage:
- Scenes & objects
- Fully dynamic

8 Design Patterns for Multimedia Software

Patterns for Multimedia Software

- The following catalog of patterns is not taken from literature, but derived from the material in this lecture
  - Work in progress, needs to be revised/completed
- Types of patterns:
  - Cross-platform patterns
  - Patterns specific for a certain platform (e.g. Flash, Pygame, JavaFX)

Cross-Platform Multimedia Pattern: Event Handler

- Program code is not executed sequentially but triggered by events
- Space usage: any
- Time usage: Interaction dependent
- Interactivity: any
- Examples:
  - ActionScript event handlers
  - Lingo event handlers
  - JavaFX event handlers
  - Python event handlers
  - ...

Flash Pattern: Start Frame Code

- **Problem**: A Flash movie needs to carry out some ActionScript code which cannot be easily defined in a local, object-oriented style
  - Creation of objects on an application-global scale
  - Invocation of methods defined in external ".as" files
  - Assignment of methods to visible objects instantiated from the standard library (e.g. TextField)
- **Solution:**
  - Keep the "global code" in the main timeline.
  - Add a separate layer (e.g. "code" or "actions") to the main timeline.
  - Add all "global" code to frame 1 of the newly created layer of the main timeline.
  - Advantage: There is just one place where all global code can be found.
- **Examples**:
  - Plenty found in literature

Cross-Platform Multimedia Pattern: Clockwork

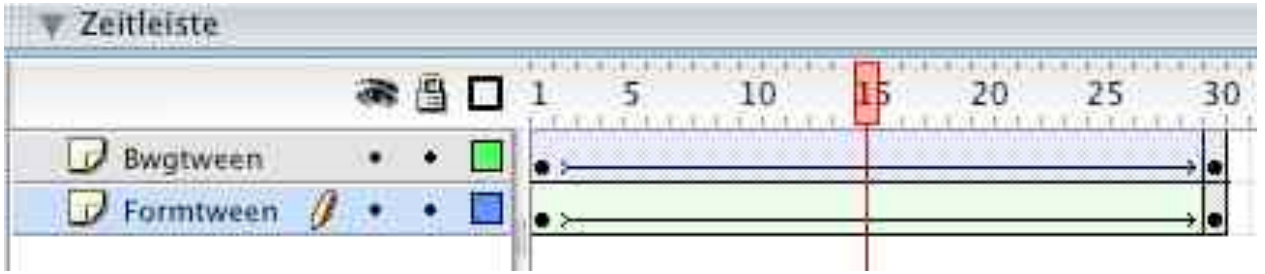
- The current properties of presentation elements are derived from the current value of a “clock” ticking at regular time intervals
- Time usage: Linear progress
- Limited interactivity: Automatic or confirmations&questions
- Usually combined with static layout or scenes and objects
- Examples:
  - Timeline in Flash, Director
  - EnterFrame-Events in Flash ActionScript
  - Ticking scripts in Squeak
  - PActivity in Piccolo


```
PActivity flash =
        new PActivity(-1, 500, currentTime + 5000) {

        protected void activityStep(long elapsedTime) {
        … }
```

Cross-Platform Multimedia Pattern: Interpolation

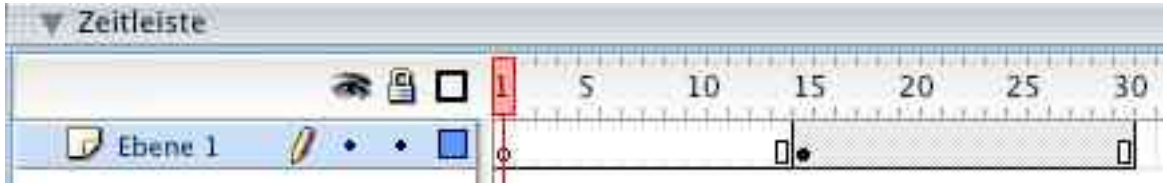
- A parameter (usually regarding a graphical property) is assumed to change its value continuously dependent of another parameter (e.g. time). The dependency can follow a linear or other rules of computation.
  - Fixed values for the dependent parameter are given for certain values of the base parameter.
  - Intermediate values of the dependent parameter are computed by interpolation.
- Space usage: scenes&objects mainly
- Time usage: Linear progress only
- Usually combined with low interactivity (on this level)
- Examples:
  - Tweening in Flash
  - Animation methods in Piccolo
  - JavaFX interpolators




```
PActivity a1 =
     aNode.animateToPositionScaleRotation(0, 0, 0.5, 0, 5000);
```

Cross-Platform Multimedia Pattern: Scheduled Time

- An activity is assumed to start at a given point in time. The start time is specified
  - in absolute terms, or
  - relatively to another activity
- Time usage: Mainly automatic
- Low interactivity
- Examples:
  - SMIL time specifications (begin attribute)
  - Placement of code or object in certain frame in Flash
  - setStartTime() and startAfter() methods in Piccolo




```
a1.setStartTime(currentTime);
a2.startAfter(a1);
a3.startAfter(a2);
```

Multimedia Development Pattern: Time Container Algebra

- Presentation is built from atomic parts (processes) each of which is executed in a *time container*.
- Time containers are composed by algebraic operations: sequential composition, parallel composition, repetition, mutual exclusion,synchronisation options
- Time usage: Linear progress
- Space usage: Scenes or scenes&objects
- Low interactivity
- Examples:
  - SMIL body: seq, par, excl
  - Animations class of “JGoodies” animation framework for Java
  - Sequence of frames and parallelism of layers in Flash

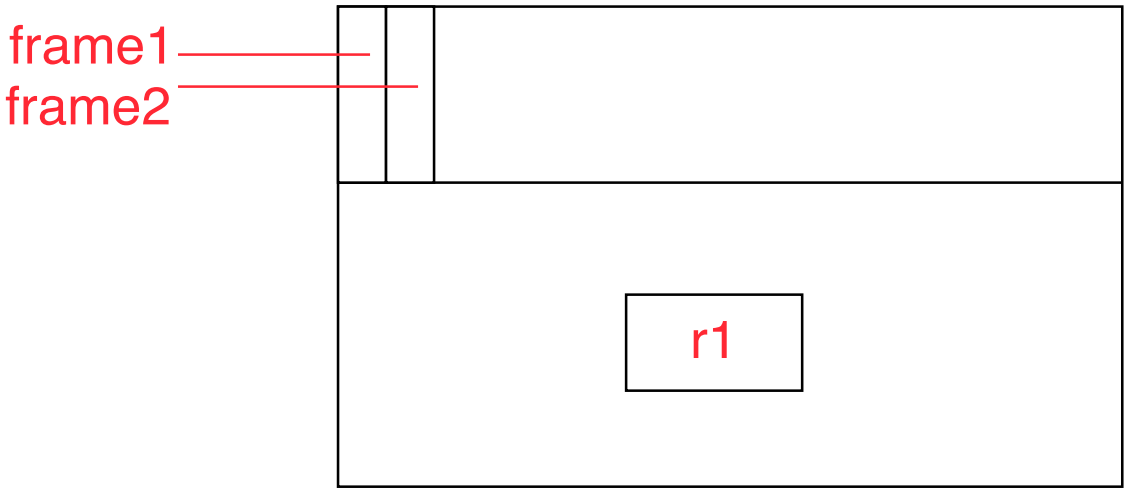
Various Representations of a Single Concept

```
<layout>
  <region id="r1" ...>
</layout>
<body>
  <seq>
      ...frame1
      ...frame2
  </seq>
</body>
```

XML

```
Component r1 = ...;
Animation frame1 = ...;
Animation frame2 = ...;
Animation all =
  Animations.sequential(
    new Animation[]{
      frame1, frame2});
```

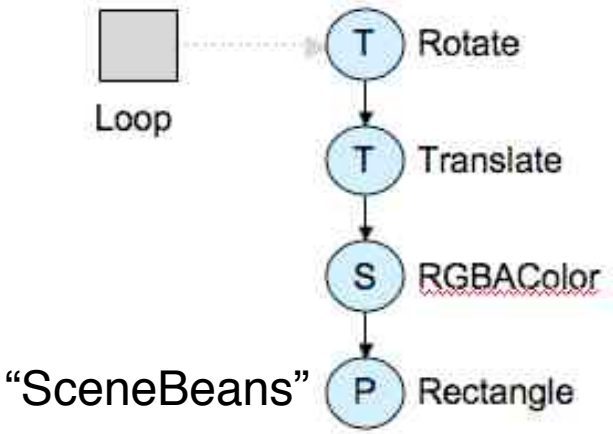
Java

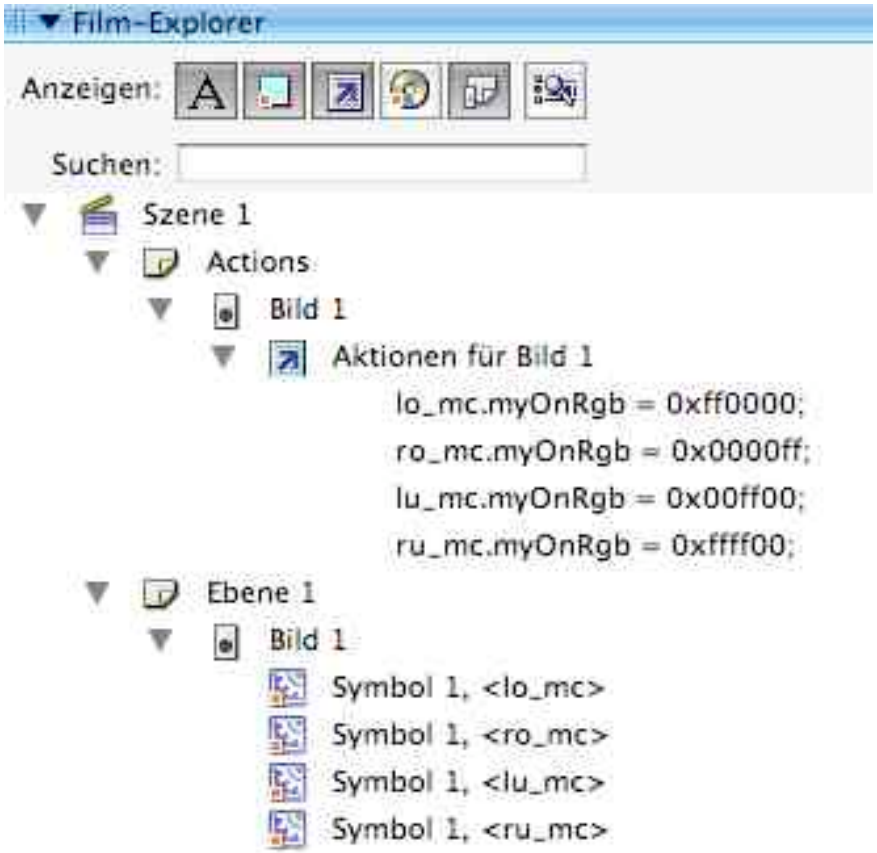


Authoring Tool (Flash-like)

Cross-Platform Multimedia Pattern: Scene Graph

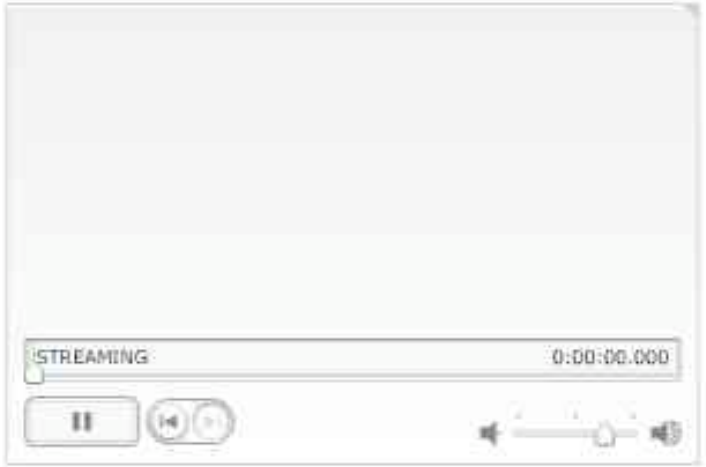
- Graph structure comprises all represented objects together with the operations (transformations) applied to them
- Space usage: Scenes&objects or fully dynamic
- Time usage: Linear progress or interaction dependent
- Examples:
  - Scene graph of JavaFX
  - Scene graph of Piccolo
  - Implicit: Film Explorer view in Flash





Multimedia Pattern for Selected Platforms: Player Component

- For standardized time-dependent media types, a pre-fabricated component is made available which provides
  - Playback of associated media files
  - Standard VCR-style controls (play, pause, stop, rewind)
- Space usage: any
- Time usage: Linear progress
- Interactivity: Interactive controls
- Examples:
  - Flash FLVPlayer component
  - JMF Player component
  - QuickTime player in QT4Java




```
try {
      p = Manager.createPlayer(new MediaLocator("file:"+file));
      p.addControllerListener(new ContrEventHandler());
      p.realize();
    }
```

8 Design Patterns for Multimedia Software

8.1 Design Patterns: The Idea

8.2 Classification Space for Multimedia Software

8.3 Patterns for Multimedia Software

8.4 Gang-of-Four Patterns Applied to Multimedia

- Factory Method
- Template Method
- State

Literature:

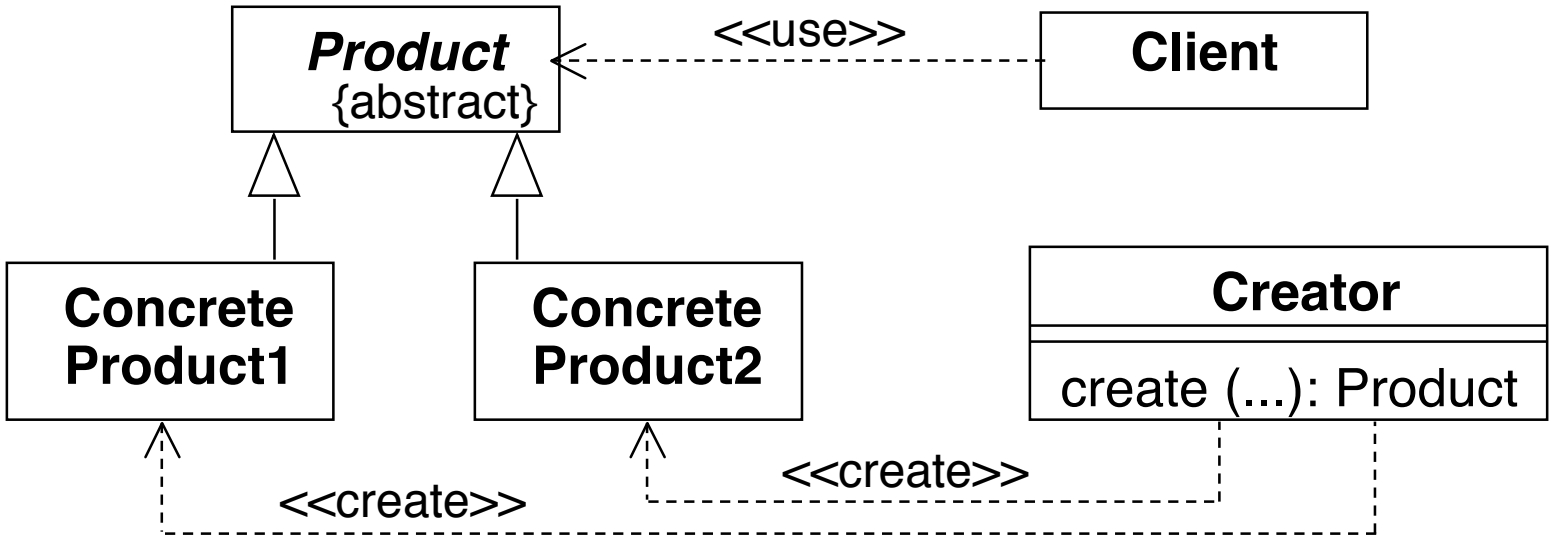
W. Sanders, C. Cumaranatunge: ActionScript 3.0 Design Patterns, O'Reilly 2007

Creation Pattern Example: Factory Method

- Situation:
  - Families of products which behave similarly
    - » Same interface
  - Example: Different kinds of players, weapons etc. in a game
- Motivation:
  - Keep code easy to change
    - » Typical change: Adding a new member of the family
  - Decouple *using* the products from *creating* the products
  - Code creating a product shall not know about the range of possible products
    - » Shall not have access to the product subclasses
- Idea:
  - Provide method with the only purpose of creating products (factory method)

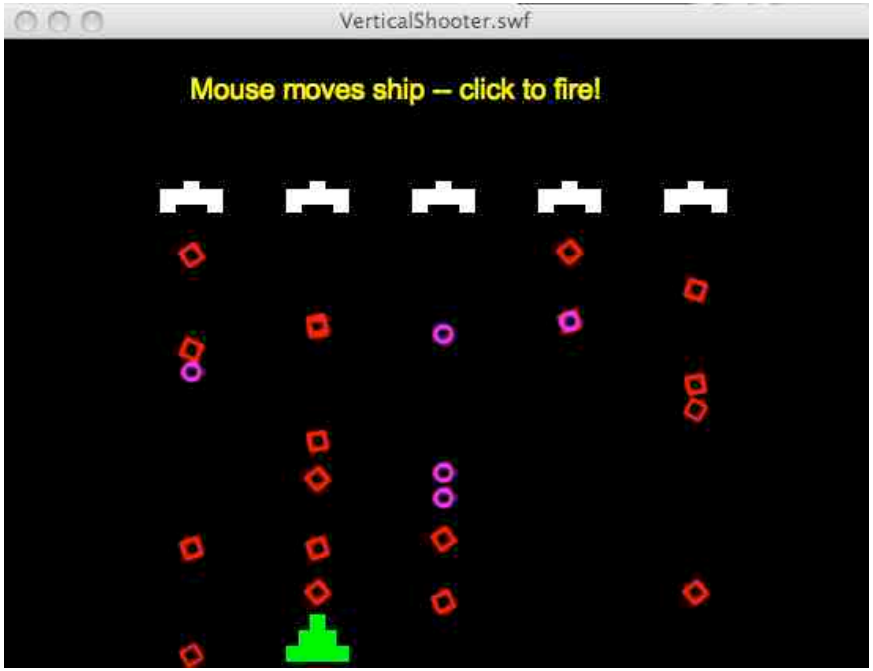
GoF Creation Pattern: Factory Method

- Name: **Factory Method**
  (dt.: Fabrikmethode, auch: Virtueller Konstruktor)
- Problem:
  - Choose at creation time between variants of a product
- Solution:



Example for Factory Method (1)

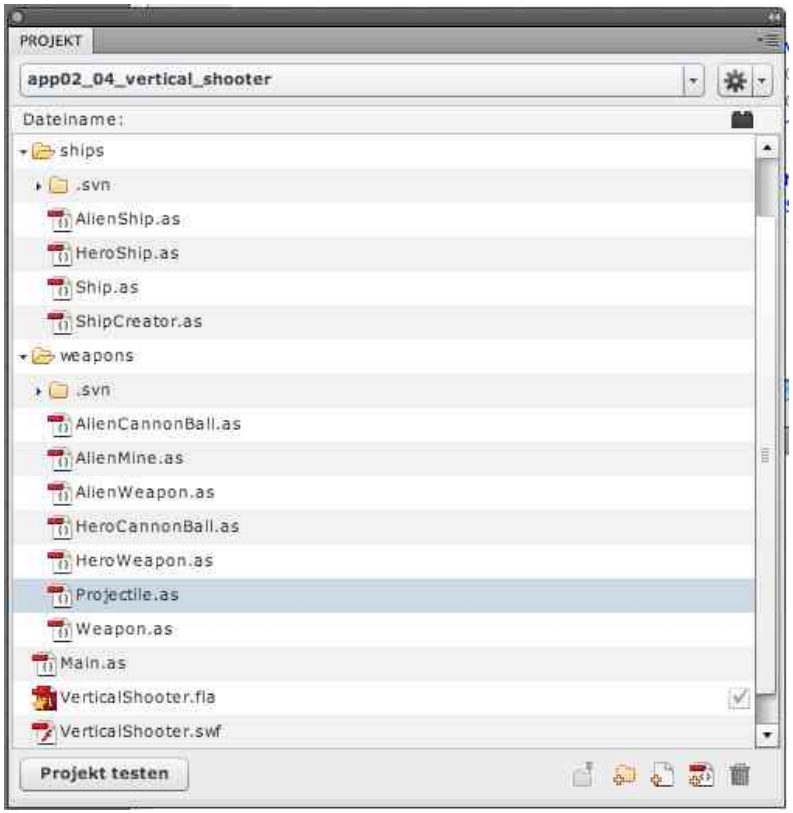
- Variants of products:
  - Ships:
    - » Hero ship
    - » Alien ship
  - Weapons:
    - » Hero weapon
      - Cannon
    - » Alien weapon
      - Cannon
      - Mine



- We want to keep the code extensible for new ship and weapon types
  - "Open-closed principle": Open for extensions, closed for code modification

Example: Sanders/Cumaranatunge

Example for Factory Method (2)



Example for Factory Method (3)

```
package ships {

   import flash.display.Sprite;
   import flash.events.*;

   // ABSTRACT Class (should not be instantiated)
   internal class Ship extends Sprite {

       internal function setLoc(xLoc:int, yLoc:int):void {
             this.x = xLoc;
             this.y = yLoc;
       }

       // ABSTRACT Method (must be overridden in a subclass)
       internal function drawShip():void {
       }

       // ABSTRACT Method (must be overridden in a subclass)
       internal function initShip():void {
       }
   }
}
```

Example for Factory Method (4a)

```
package ships {

   import flash.display.*;
   import weapons.HeroWeapon;
   import flash.events.*;

   internal class HeroShip extends Ship {

      private var weapon:HeroWeapon;

      override internal function drawShip():void {
            graphics.beginFill(0x00FF00); // green color
            graphics.drawRect(-5, -15, 10, 10);
            graphics.drawRect(-12, -5, 24, 10);
            graphics.drawRect(-20, 5, 40, 10);
            graphics.endFill();
      }

   ...
```

Example for Factory Method (4b)

```
   ...

      override internal function initShip():void {
         weapon = new HeroWeapon();
         this.stage.addEventListener(MouseEvent.MOUSE_MOVE,
            this.doMoveShip);
         this.stage.addEventListener(MouseEvent.MOUSE_DOWN,
            this.doFire);
      }

      protected function doMoveShip(event:MouseEvent):void {
         this.x = event.stageX;
         event.updateAfterEvent(); // process this event first
      }

      protected function doFire(event:MouseEvent):void {
         weapon.fire(HeroWeapon.CANNON,
            this.stage, this.x, this.y - 25);
         event.updateAfterEvent(); // process this event first
      }
   }
}
```

Example for Factory Method (5)

```
package {

   import flash.display.*;
   import flash.text.*;
   import ships.*;

   public class Main extends MovieClip {

       public function Main() {
             // show instructions
             ...
             var shipFactory:ShipCreator = new ShipCreator();
             shipFactory.addShip
               (ShipCreator.HERO, stage,
                  stage.stageWidth/2, stage.stageHeight-20);
             for (var i:Number = 0; i < 5; i++) {
                shipFactory.addShip(ShipCreator.ALIEN,
                  stage, 120 + 80 * i, 100);
             }
       }
   }
}
```

Example for Factory Method (6a)

```
package ships {

   import flash.display.Stage;

   public class ShipCreator {

      public static const HERO          :uint =  0;
      public static const ALIEN         :uint =  1;

      public function addShip(cShipType:uint,
            target:Stage, xLoc:int, yLoc:int):void {

         var ship:Ship = this.createShip(cShipType);
         ship.drawShip();
         ship.setLoc(xLoc, yLoc);
         target.addChild(ship);
         ship.initShip();
      }

...
```

Example for Factory Method (6b)

```
...

      // concrete factory method
      private function createShip(cShipType:uint):Ship {
         if (cShipType == HERO) {
            trace("Creating new hero ship");
            return new HeroShip();
         }
         else if (cShipType == ALIEN) {
            trace("Creating new alien ship");
            return new AlienShip();
         }
         else {
            throw new Error("Invalid kind of ship specified");
            return null;
         }
      }
   }
}
```

Test for Encapsulation

```
public function Main() {
      ...
      var testShip = new HeroShip();
      ...
}
```

Compiler-Fehler:

1180: Aufruf einer möglicherweise undefinierten Methode HeroShip.

Test for Extensibility (1)

- How to add a new weapon?
- HeroShip.as:

```
override internal function initShip():void {
    weapon = new HeroWeapon();
    this.stage.addEventListener
        (MouseEvent.MOUSE_MOVE, this.doMoveShip);
    this.stage.addEventListener(MouseEvent.MOUSE_DOWN, this.doFire);
    var newweapon = new NewWeapon();
    newweapon.fire(NewWeapon.NEW, this.stage, this.x, this.y - 50);
}
```

- New classes added *(without modification of existing code!)*
  - NewWeapon.as
    - » The new kind of weapon
    - » Concrete creator for bullets, derived from abstract creator *Weapon*
  - NewBullet.as
    - » The bullet fired by the new kind of weapon
    - » Concrete product, derived from abstract product *Projectile*

Test for Extensibility (2)

```
package weapons {

  public class NewWeapon extends Weapon {

     public static const NEW            :uint =  3;

     override protected function
          createProjectile(cWeapon:uint):Projectile {
        if (cWeapon == NEW) {
           trace("Creating new bullet");
           return new NewBullet();
        } else {
           throw new Error("Invalid kind of projectile");
           return null;
        }
      }
  }
}
```

NewWeapon.as

Test for Extensibility (3)

```
package weapons {

   internal class NewBullet extends Projectile {

      override internal function drawProjectile():void
      {
            graphics.beginFill(0xFF0000);
            graphics.drawCircle(0, 0, 15);
            graphics.endFill();
      }

      override internal function arm():void {
            nSpeed = -15; // set the speed
      }
   }
}
```

NewBullet.as

- Methods **`drawProjectile()`** and **`arm()`** are called in method **`fire()`** of abstract class Weapon
  - Idea of *Template Method* pattern

8 Design Patterns for Multimedia Software

8.1 Design Patterns: The Idea

8.2 Classification Space for Multimedia Software

8.3 Patterns for Multimedia Software

8.4 Gang-of-Four Patterns Applied to Multimedia

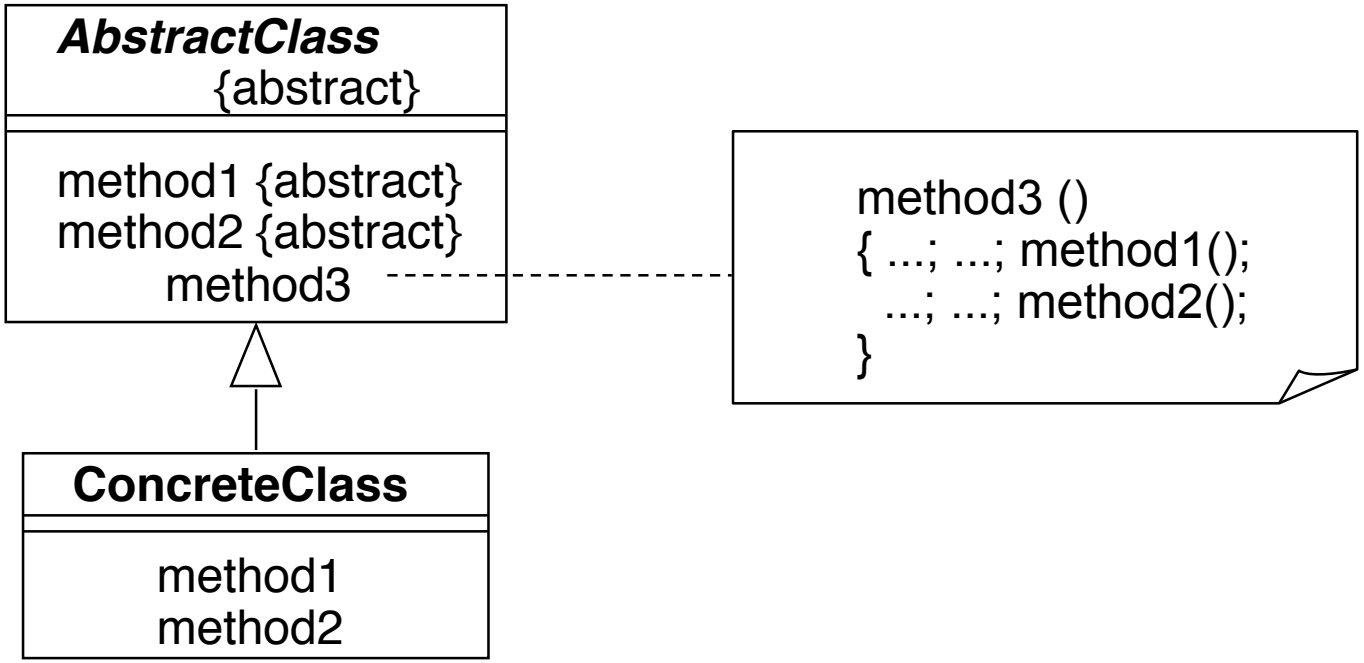
- Factory Method
- Template Method 
- State

Literature:

W. Sanders, C. Cumaranatunge: ActionScript 3.0 Design Patterns, O'Reilly 2007

GoF Behovioral Pattern: Template Method

- **Problem:** Operation consists of fixed and variable code parts
- **Solution:** *Template method* in superclass calls abstract *methods*, which are defined in subclasses (one subclass per variant).



Example for Template Method (1)

- Multimedia jukebox for video and audio files
  - Same mechanisms for selecting titles
  - Different mechanisms for playing back
- Very simplified example:
  - Two buttons for playing a fixed audio resp. video file

Example: Sanders/Cumaranatunge

Example for Template Method (2)

```
private function doButton():void {
    tuneButton=new TuneButton();
    videoButton=new VideoButton();
    addChild (tuneButton);
    addChild (videoButton);
    tuneButton.x=
      ((stage.stageWidth/2)-(1.5*tuneButton.width)),
    tuneButton.y=30;
    videoButton.x=((stage.stageWidth/2)+5), videoButton.y=30;
    tuneButton.addEventListener
      (MouseEvent.CLICK,getMedia);
    videoButton.addEventListener
      (MouseEvent.CLICK,getMedia);
}
private function getMedia(va:VidAudio):void {
    var va: VidAudio;
    if (e.target == tuneButton)
       va = new Audio();
    else
       va = new Vid();
    va.mediaProducer();
    addChild (va);
}
```

Example for Template Method (3)

```
package { ...

   //Abstract Class
   class VidAudio extends Sprite {
       //Template method
       public final function mediaProducer():void {
             selectMedia ();
             playNow ();
             fromMediaDesign ();
       }
       protected function selectMedia ():void {
             //Awaiting instructions
       }
       protected function playNow():void {
             //Awaiting instructions
       }
       private final function fromMediaDesign():void {
             mText=new TextField();
             mText.text="Welcome to Template Media!";
             ... // Show text field
       }
   }
}
```

Example for Template Method (4)

```
package {
   //A concrete class
   //Vid class

   public class Vid extends VidAudio {

      private var vidName:String;

      override protected function selectMedia():void {
            vidName="media";
      }

      override protected function playNow():void {
            var playVideo=new PlayVideo(vidName);
            addChild (playVideo);
      }
   }
}
```

PlayVideo can be a rather complex class...

8 Design Patterns for Multimedia Software

8.1 Design Patterns: The Idea

8.2 Classification Space for Multimedia Software

8.3 Patterns for Multimedia Software

8.4 Gang-of-Four Patterns Applied to Multimedia

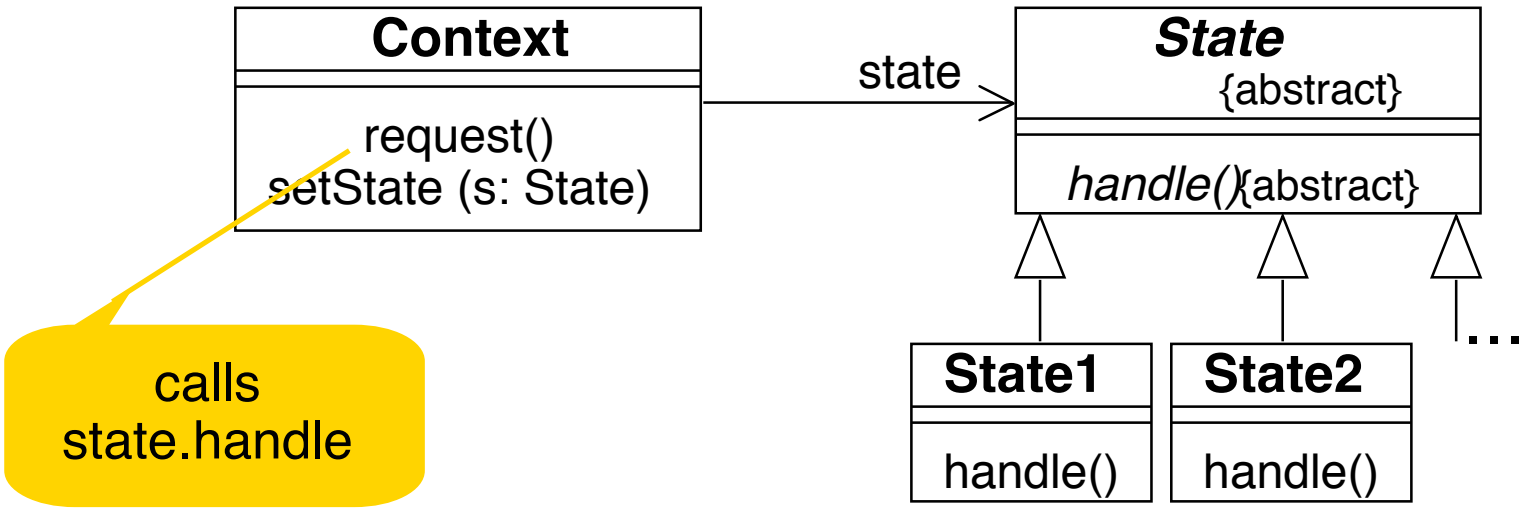
- Factory Method
- Template Method
- State 

Literature:

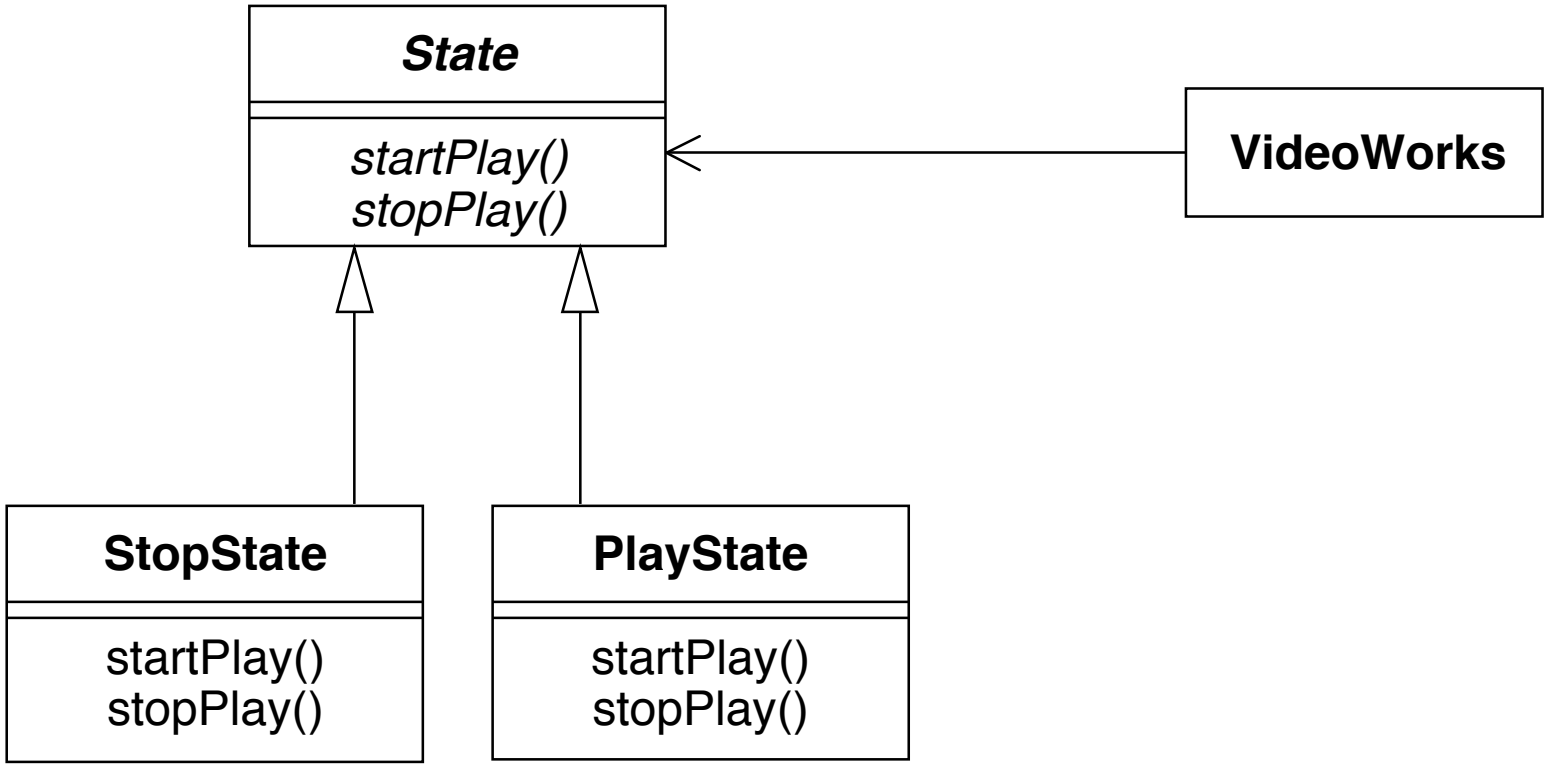
W. Sanders, C. Cumaranatunge: ActionScript 3.0 Design Patterns, O'Reilly 2007

GoF Structural Pattern: State

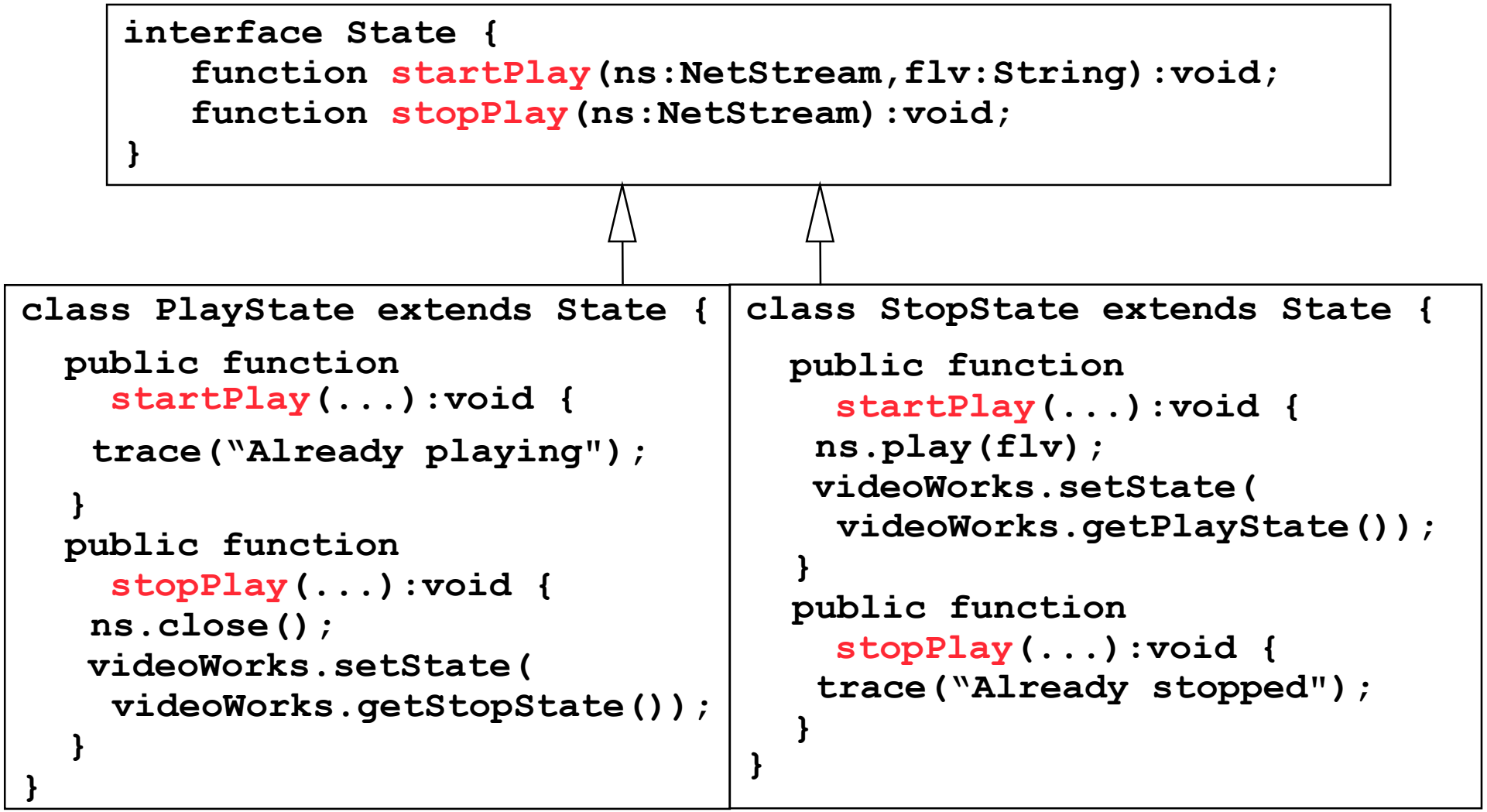
- Name: **State**
- Problem:
  - Flexible and extensibe technique to change the behaviour of an object when its state changes.
- Solution :



Example for State (1)



Example for State (2)



Test for Extensibility

- Adding a “pause” state
- First step: Change the state interface

```
function doPause(ns:NetStream):void;
```

  – Compiler checks completeness of transitions

  (1044: Schnittstellenmethode doPause in Namespace State nicht durch Klasse PlayState implementiert.)

- Second step: Extend existing concrete state classes
  – React to “pause” request in all existing states
  – Transition to “pause state” from play state
- Third step: Add a new concrete state class **`PauseState`**
  – Implements state interface