10 Software Engineering Techniques for Multimedia Software

10.1 Specific Design Patterns for Multimedia Software
10.2 Classical Design Patterns Applied to Multimedia
10.3 Modeling of Multimedia Applications
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 and operation names):
    » Role name, i.e. place holders for parts of implementation
    » Fixed parts of implementation
  – Collaboration (sequence of events, possibly diagrams)
• Discussion
  – Pros and cons
  – Dependencies, restrictions
  – Special cases
• Known uses
Patterns for Multimedia Software

• The following examples of patterns are not taken from literature, but derived from the material in this lecture
  – Based on various platforms, also older ones
• Types of patterns:
  – Cross-platform patterns
  – Patterns specific for a certain platform
Classification Space

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

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

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

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, JavaFX
  - EnterFrame-Events in Flash ActionScript
  - Ticking scripts in Squeak
  - Clock class in PyGame
  - CreateJS “Ticker”
  - PActivity in Piccolo

```
PActivity flash =
    new PActivity(-1, 500, currentTime + 5000) {
        protected void activityStep(long elapsedTime) {
            ...
        }
    }
```

University of Maryland “Piccolo” framework (see cs.umd.edu/hcil/piccolo)
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 transitions

```java
PActivity a1 = aNode.animateToPositionScaleRotation(0, 0, 0.5, 0, 5000);
```
Cross-Platform Multimedia Pattern: Scene Graph

- Graph structure for all represented objects
- 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

“SceneBeans”
Cross-Platform 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, synchronization 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 Syntactical Representations for a Single Concept

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

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

XML  
Java

Authoring Tool (Flash-like)
10 Software Engineering Techniques for Multimedia Software

10.1 Specific Design Patterns for Multimedia Software
10.2 Classical Design Patterns Applied to Multimedia
10.3 Modeling of Multimedia Applications

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

- Situation:
  - Complex tree-structured objects

- Motivation:
  - Use homogeneous interface for objects and compositions thereof

Exercise (JavaFX):
- Compare with classes
  - Node
  - Parent
  - Group
  - Rectangle
  - MediaView
GoF Behavioral Pattern: Mediator

- **Situation:**
  - Many objects interact in a complex way
  - Individual objects are not able to co-ordinate their communication

- **Motivation:**
  - Create single point of co-ordination
Mediator Example: Sprite Manager

public class SpriteManager {

    private List<Sprite> sprite_list;

    public SpriteManager(double scW, double scH) {
        sprite_list = new ArrayList<Sprite>();
    }

    public void addSprite(Sprite s) {
        sprite_list.add(s);
    }

    public void updateSprites() {
        for (Sprite s: sprite_list) {
            s.update();
        }
        for (Sprite s1: sprite_list) {
            for (Sprite s2: sprite_list) {
                if (s1 != s2) {
                    s1.handleCollision(s2);
                }
            }
        }
    }

}
Creation Pattern Example: Builder

• Situation:
  – Many parameters of a constructor have to be set
  – Leading to an explosion of constructor variants
    (telescoping constructor anti-pattern)

• Motivation:
  – Make code simpler
  – Separate between concrete (low-level) representation and creation process
  – Move default settings to appropriate location
  – Avoid typical copy-paste errors

• Idea:
  – Provide static class which first collects information on object to be built and
    finally constructs it
GoF Creation Pattern: Builder

- **Name:** Builder (dt.: Erbauer)
- **Problem:**
  - Step-by-step creation of complex objects
- **Solution:**

```plaintext
Director
construct(builder: Builder)

Builder
buildPartX()
getProduct(): Product

ConcreteBuilder
buildPartX()
getProduct(): Product

Product
<<constructs>>
```
Example: Series of Similar Buttons

- Do not repeat all detailed settings (e.g. styles) for all buttons
  - Avoid copy-paste problems
- Define standard combinations for settings to be applied easily in construction process
- Remove low-level information (like styles) from high-level program logic
Button Series without Builder

Button btn1 = new Button();
btn1("-fx-background-color: lightgrey;
       -fx-padding: 9; -fx-font: 20pt 'sans-serif';");
btn1((ActionEvent event) -> {
    System.out.println("Btn1 clicked");
});
Button btn2 = new Button();
btn2("-fx-background-color: lightgrey;
       -fx-padding: 9; -fx-font: 20pt 'sans-serif';
       -fx-border-color: black;-fx-border-width: 2px;");
btn2.setOnAction((ActionEvent event) -> {
    System.out.println("Btn2 clicked");
});
Button btn3 = new Button();
btn3.setStyle("-fx-background-color: lightgrey;
               -fx-padding: 9; -fx-font: 20pt 'sans-serif';");
btn3.setOnAction((ActionEvent event) -> {
    System.out.println("Btn3 clicked");
});
Please Note: Lambda Expressions in Java 8

btn1((ActionEvent event) -> {
    System.out.println("Btn1 clicked");
});
Example: Builder (1)

```java
public class MyButton extends Button {

    private MyButton (Builder builder) {
        super(builder.label);
        if (builder.hilited) {
            this.setStyle("-fx-background-color: lightgrey;
                -fx-padding: 9; -fx-font: 20pt 'sans-serif';
                -fx-border-color: black;-fx-border-width: 2px;";
        }
        else {
            this.setStyle("-fx-background-color: lightgrey;
                -fx-padding: 9; -fx-font: 20pt 'sans-serif';");
        }
        this.setOnAction((ActionEvent event) -> {
            System.out.println(this.getText() +" clicked");
        });
        builder.reset();
    }

    ...
```
Example: Builder (2)

```java
public static class Builder {
    private String label;
    private Boolean hilited;

    public Builder() {
        reset();
    }

    private void reset() {
        label = "";
        hilited = false;
    }

    public Builder label(String label) {
        this.label = label;
        return this;
    }

    public Builder hilited() {
        this.hilited = true;
        return this;
    }

    // ...
}
```

Additional pattern: **Method chaining**
Builder method returns builder object
Example: Builder (3)

```java
public MyButton build() {
    return new MyButton(this);
}
```

```java
class MyButton {
    private MyButton(Builder builder) {
        ...
    }
    
    public static class Builder {
        // collection of properties
        // construction methods
    }
}
```
Example: Builder (4)

```java
MyButton.Builder builder = new MyButton.Builder();
Button btn1 = builder.label("Btn 1").build();
Button btn2 = builder.label("Btn 2").hilited().build();
Button btn3 = builder.label("Btn 3").build();
```

QUIZ:
What are advantages and disadvantages of using this pattern?
**Builder in JavaFX**

**Richard Bair** richard.bair at oracle.com  
Mon Mar 25 10:35:51 PDT 2013

We made a mistake. When we released JavaFX 2.0 we included a (large) set of *Builder classes to provide a builder-pattern approach to building JavaFX UIs. The builder-pattern approach provides several very nice features:

- Ability to setup generic configuration once and "stamp out" multiple copies
- Structured code style that closely approximates the "container hierarchy" of the UI
- Strongly-typed "declarative" style programming

The Builders did come at a cost. [...] 

My proposal after having weighed the options is to phase out the Builders by deprecating them in 8 and removing them from the class path in 9.
GoF Structural Pattern: State

• Name: **State**

• Problem:
  
  – Flexible and extensible technique to change the behavior of an object when its state changes.

• Solution:

```
Context
  request()
  setState (s: State)

State
  {abstract}
  handle() {abstract}

State1
  handle()

State2
  handle()

...  
```

calls state.handle
Example for State (1)

```
State

startPlay()
stopPlay()

VideoPlayer

StopState

startPlay()
stopPlay()

PlayState

startPlay()
stopPlay()
```
abstract class State {
    abstract void startPlay();
    abstract void stopPlay();
}

class PlayState extends State {
    @Override
    void startPlay() {
        System.out.println("Already playing...");
    }

    @Override
    void stopPlay() {
        context.setState(context.getStopState());
    }
}

class StopState extends State {
    @Override
    void startPlay() {
        context.setStopState(context.getPlayState());
    }

    @Override
    void stopPlay() {
        System.out.println("Already stopped...");
    }
}
Administrative Overhead for State

```java
interface Context {
    abstract public void setState (State s);
    abstract public State getStopState();
    abstract public State getPlayState();
}

public class JavaFXStatePattern extends Application implements Context {
    private State state;
    final private State stopState = new StopState();
    final private State playState = new PlayState();

    @Override
    public void setState (State s) {
        state = s;
    }

    @Override
    public State getStopState() {
        return stopState;
    }
    ...
```
Test for Extensibility

• Adding a “pause” state
• First step: Change the State interface
  
  abstract void pausePlay();
  
  -> Compiler checks completeness of transitions
• All additional code is concentrated in one class = one file
10 Software Engineering Techniques for Multimedia Software

10.1 Specific Design Patterns for Multimedia Software
10.2 Classical Design Patterns Applied to Multimedia
10.3 Modeling of Multimedia Applications

Literature:
Model-Driven Architecture

- Model-Driven Architecture (MDA): A concrete framework defined by the Object Management Group (OMG) for model-driven development
  - CIM: Computation independent model
  - PIM: Platform independent model
  - PSM: Platform specific model
Example Application: Break Out Game
Diagrams in MML

- Structure Diagram
  - Application Entities
  - Media Components

- Scene Diagram

- Abstract User Interface Diagram

- (Conventional) UI
  - Media UI

- Interaction Diagram

Software Design

Media Design

User Interface Design
Example: Application structure for Break Out Game Application

**Inner Structure**

- **Player**
  - lives : int
  - score : int
  +getLives() : int
  +decreaseLives()
  +increaseScore()

- **BlockOut**

- **Level**
  - number : int
  +countBricks() : int

- **Paddle**
  - leftRight
  +reboundBall()

- **Brick**
  +hit()

- **Ball**
  +startMoving()
  +move()
  +init()
  +rebound()

**Media Representation**

- **Media Component**

- **Animation** PaddleAnimation

- **Animation** BrickAnimation

- **Sound** BrickSound

- **Animation** BallAnimation

**Diagram Details**

- **Association**
  - BlockOut
  - Player
  - Level
  - Paddle
  - Brick
  - Ball
Example: Scenes for Break Out Game Application

- **Scene**: Game
  - **Entry Operation**: startGame(p:Player, hasSound:Boolean)
  - **Transitions**:
    - levelFinished(p:Player)
    - [p.lives > 0] -> nextLevel()

- **Scene**: Score
  - **Transitions**:
    - gameOver(p:Player)
    - <<history>> resumeMenu

- **Scene**: Menu
  - **Transitions**:
    - initialMenu
    - menuHelp
    - <<history>> resumeMenu

- **Scene**: Help
  - **Transitions**:
    - <<history>> resumeMenu

- **Scene**: Highscore
Example: Abstract User Interface for Scene ‘Game’

- LevelNo {Level.number}
- Score {Player.score}
- Lives {Player.lives}
- Ball {Ball}
- Bricks [0..n] {Brick}
- Paddle {Paddle.leftRight}
- Start {Ball.startMoving}
Example: Media User Interface for Scene ‘Game’

UI Realization

Sensor

AUls without specific realization
Example: Interaction diagram for Scene ‘Game’

- init (ball)
- startMoving (ball::)
- CallOperation Action
- Object (property of the scene)
- Ball.start Moving
- UIInputEvent
- Paddle. leftRight
- Sensor Event
- "Collision" Paddle
- "Collision" Wall
- "Collision" Brick
- "Collision" OffField

...
Pros and Cons of Model-Driven Development for Multimedia Applications

• Advantages:
  – Code generation achieves independence form multimedia platform
  – Higher level of abstraction leads to deeper analysis
  – Code generators can help to create well-structured code (e.g. modular Flash applications)

• Disadvantages:
  – Full code generation problematic, platform-specific additions needed
  – Round-trip engineering still needs to be developed
  – Writing abstract specifications is not attractive for multimedia developers

• Open issue:
  – What is the right language level for integrating the various design views/activities?