Aspect-Oriented and Context-Oriented Programming

Modularization of cross-cutting concerns with AspectJ and JCop

Advanced Modularity, WS 12/13

Matthias Springer

Hasso Plattner Institute

January 8, 2013

Overview

Power Management for a Mobile Device

Classical Object-Oriented Programming

Aspect-Oriented Programming

Context-Oriented Programming

Comparison: AspectJ (AOP) and JCop (COP)

Overview

Power Management for a Mobile Device

- Basic components
- Power Management

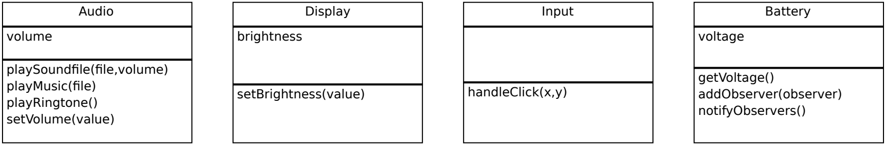
Classical Object-Oriented Programming

Aspect-Oriented Programming

Context-Oriented Programming

Comparison: AspectJ (AOP) and JCop (COP)

Basic components



Audio

volume

playSoundfile(file,volume)
playMusic(file)
playRingtone()
setVolume(value)

Display

brightness

setBrightness(value)

Input

handleClick(x,y)

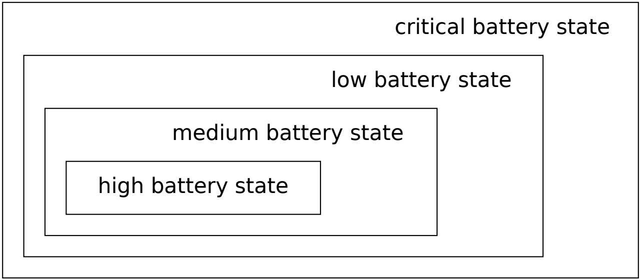
Battery

voltage

getVoltage()
addObserver(observer)
notifyObservers()

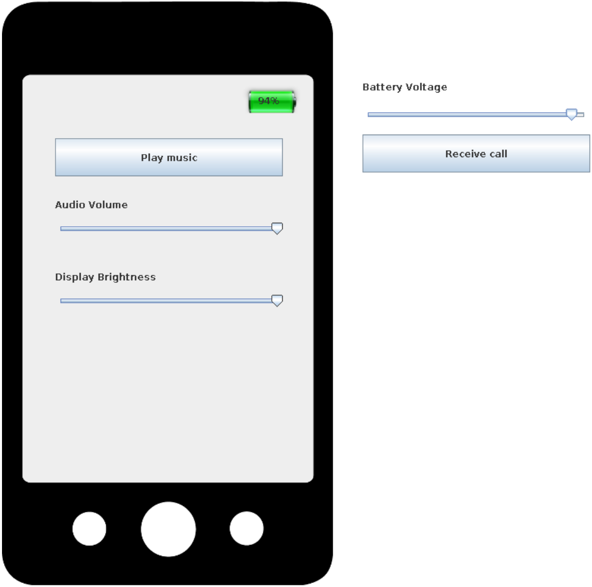
- **Audio:** plays ringtone and music
- **Display:** manages settings (brightness, contrast, . . .) and draws graphics, only brightness implemented in this example
- **Input:** handles input events
- **Battery:** notifies other components about battery changes

Power Management



- **Medium battery state:** reduce sound volume by 25%, reduce display brightness to 50% after 5 seconds of no interaction
- **Low battery state:** further reduce sound volume by 33%, turn off display after 10 seconds of no interaction
- Low battery state implies medium battery state

Demo


94%
Play music
Audio Volume
Display Brightness
Battery Voltage
Receive call

Overview

Power Management for a Mobile Device

Classical Object-Oriented Programming

- Implementation of Power Management
- Implementation drawbacks

Aspect-Oriented Programming

Context-Oriented Programming

Comparison: AspectJ (AOP) and JCop (COP)

Implementation of Power Management

```
class Audio {
  public void playSoundfile(String filename, float volume) {
    if (Battery.getVoltage() < 2.5) {
      volume *= 0.5;
    }
    else if (Battery.getVoltage() < 5.0) {
      volume *= 0.75;
    }

    // play soundfile
  }
}
```

Implementation of Power Management

```java
class Input {
  private boolean isDimmed = false;

  // no valid Java, just a shortcut
  private Thread dimDisplay = {
    Thread.sleep(5000);
    brightness = Display.getBrightness();
    Display.setBrightness(0.5 * brightness);
    isDimmed = true;
  }

  private Thread turnOffDisplay = {
    Thread.sleep(10000);
    Display.setBrightness(0);
  }

  public void handleClick(int x, int y) {
    if (isDimmed) {
      Display.setBrightness(brightness);
      isDimmed = false;
    }

    dimDisplay.stop();
    turnOffDisplay.stop();

    if (Battery.getVoltage() < 2.5) {
      dimDisplay.start();
      turnOffDisplay.start();
    }
    else if (Battery.getVoltage() < 5.0) {
      dimDisplay.start();
    }

    // handle click event
  }
}
```

Implementation drawbacks

`http://deviq.com/spaghetti-code`

- **Scattering:** Power Management is scattered across multiple classes (`Display`, `Audio`).
- **Understandability:** Who would assume display dimming to be implemented in `Input`?
- **Code Duplication:** calculation of power state (`if` tests)
- Power Management is a *cross-cutting concern*.
- Multiple unmodularized cross-cutting concerns can lead to spaghetti code.

Overview

Power Management for a Mobile Device

Classical Object-Oriented Programming

Aspect-Oriented Programming
- Implementation (AspectJ)
- Benefits over previous approach
- Modularization of Power States
- Benefits over previous approach

Context-Oriented Programming

Comparison: AspectJ (AOP) and JCop (COP)

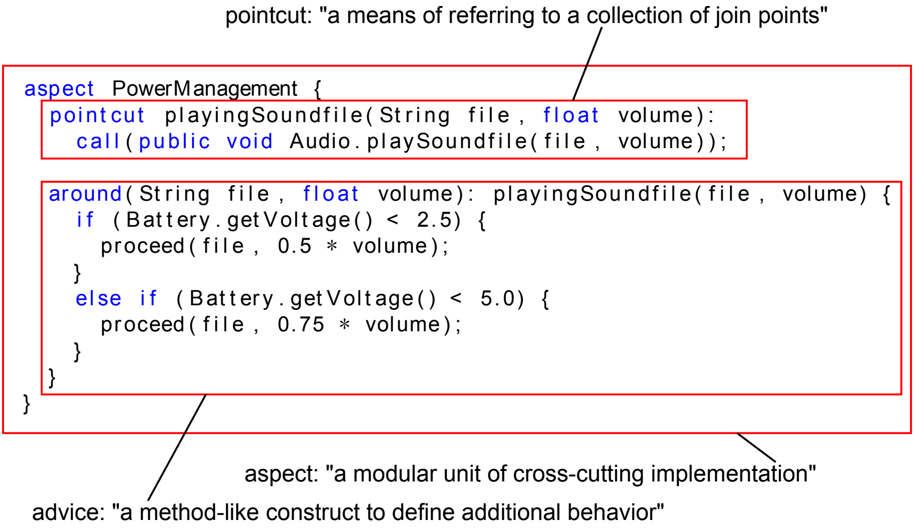
Implementation (AspectJ)

```
aspect PowerManagement {
  private float brightness;
  private boolean isDimmed = false;

  // no valid Java, just a shortcut
  private Thread dimDisplay = {
    // ...
  }

  private Thread turnOffDisplay = {
    // ...
  }

  pointcut receivingInput():
    call(public void Input.handle*(..));

  before(): receivingInput() {
    if (isDimmed) {
      Display.setBrightness(brightness);
      isDimmed = false;
    }

    dimDisplay.stop();
    turnOffDisplay.stop();

    if (Battery.getVoltage() < 2.5) {
      dimDisplay.start();
      turnOffDisplay.start();
    }
    else if (Battery.getVoltage() < 5.0) {
      dimDisplay.start();
    }
  }
}
```

Implementation (AspectJ)

```
aspect PowerManagement {
  pointcut playingSoundfile(String file, float volume):
    call(public void Audio.playSoundfile(file, volume));

  around(String file, float volume): playingSoundfile(file, volume) {
    if (Battery.getVoltage() < 2.5) {
      proceed(file, 0.5 * volume);
    }
    else if (Battery.getVoltage() < 5.0) {
      proceed(file, 0.75 * volume);
    }
  }
}
```

Terms and definitions [7]



pointcut: "a means of referring to a collection of join points"

```
aspect PowerManagement {
  pointcut playingSoundfile(String file, float volume):
    call(public void Audio.playSoundfile(file, volume));

  around(String file, float volume): playingSoundfile(file, volume) {
    if (Battery.getVoltage() < 2.5) {
      proceed(file, 0.5 * volume);
    }
    else if (Battery.getVoltage() < 5.0) {
      proceed(file, 0.75 * volume);
    }
  }
}
```

aspect: "a modular unit of cross-cutting implementation"

advice: "a method-like construct to define additional behavior"

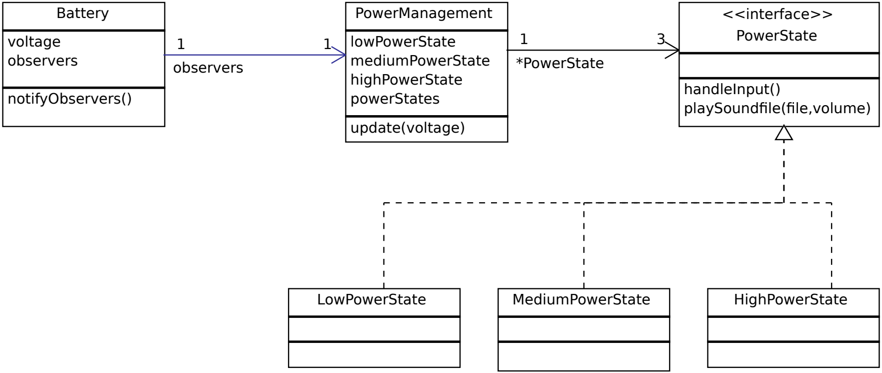
Benefits over previous approach

- **No scattering:** Power Management is encapsuled in a single aspect.
- **Improved understandability:** due to avoided scattering

But . . .

- **Code duplication**
- Further **modularization** desired: seperate power states

Modularization of Power States



Modularization of Power States: Context

```
aspect PowerManagement {
  private PowerState[] states;

  private PowerState lowPower = new LowPowerState();
  private PowerState mediumPower = new MediumPowerState();
  private PowerState highPower = new HighPowerState();

  public void update(float voltage) {
    if (voltage < 2.5) {
      states = [lowPower, mediumPower]; // no valid Java
    }
    else if (voltage < 5.0) {
      states = [mediumPower];
    }
    else {
      states = [highPower];
    }
  }
```

Modularization of Power States: Context

```
  pointcut receivingInput():
    call(public void Input.handle*(..));

  before(): receivingInput() {
    for (PowerState state : states) {
      state.handleInput();
    }
  }

  pointcut playingSoundfile(String file, float volume):
    call(public void Audio.playSoundfile(file, volume));

  around(String file, float volume): playingSoundfile(file, volume) {
    states[0].playSoundfile(file, volume);
  }
}
```

Modularization of Power States: State

```
class MediumPowerState implements PowerState {
  public void playSoundfile(String file, float volume) {
    Audio.playSoundfile(file, 0.75 * volume);
  }

  private boolean isDimmed = false;
  private float brightness;

  private Thread dimDisplay = { /* ... */ }

  public void handleInput() {
    if (isDimmed) {
      Display.setBrightness(brightness);
      isDimmed = false;
    }

    dimDisplay.stop();
    dimDisplay.start();
  }
}
```

Modularization of Power States: State

```
class LowPowerState implements PowerState {
  public void playSoundfile(String file, float volume) {
    Audio.playSoundfile(file, 0.5 * volume);
  }

  private Thread turnOffDisplay = { /* ... */ }

  public void handleInput() {
    turnOffDisplay.stop();
    turnOffDisplay.start();
  }
}

class HighPowerState implements PowerState {
  public void playSoundfile(String file, float volume) {
    Audio.playSoundfile(file, volume);
  }

  public void handleInput() {}
}
```

Benefits over previous approach

- **Improved understandability:** states are properly modularized
- No code duplication

But . . .

- Manual **iteration over all active states** necessary
- No real support for **around advice**
  (states are no aspects, thus no support for `proceed`)
- `HighBatteryState` adds **no new behavior**
- Advice code is merely **wrapper code**

Overview

Power Management for a Mobile Device

Classical Object-Oriented Programming

Aspect-Oriented Programming

Context-Oriented Programming

Comparison: AspectJ (AOP) and JCop (COP)

Implementation (JCop): Context

```
contextclass PowerManagement {
  private PowerState[] states;

  private PowerState lowPower = new LowPowerState();
  private PowerState mediumPower = new MediumPowerState();

  public void update(float voltage) {
    if (voltage < 2.5) {
      states = [lowPower, mediumPower];
    }
    else if (voltage < 5.0) {
      states = [mediumPower];
    }
    else {
      states = [];
    }
  }

  when(true): with(states);
}
```

Terms and definitions [3]

context evaluation: "everything that is computationally accessible"

```
contextclass PowerManagement {
  private PowerState[] states;

  private PowerState lowPower = new LowPowerState();
  private PowerState mediumPower = new MediumPowerState();

  public void update(float voltage) {
    if (voltage < 2.5) {
      states = [lowPower, mediumPower];
    }
    else if (voltage < 5.0) {
      states = [mediumPower];
    }
    else {
      states = [];
    }
  }

  when(true): with(states);
}
```

dynamic, declarative layer activation

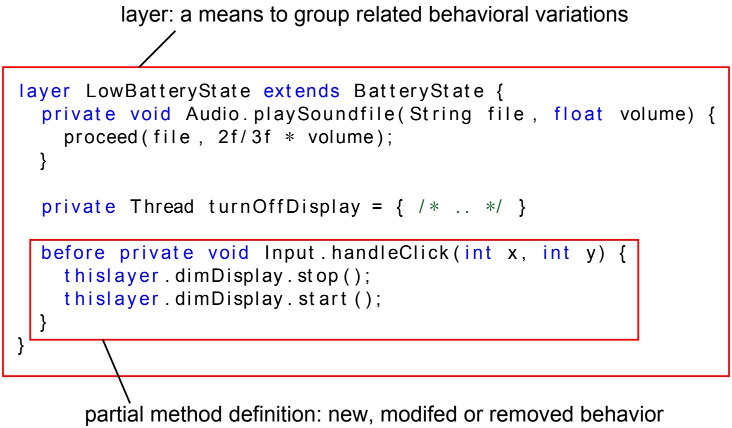
Implementation (JCop): Layers

```
layer MediumBatteryState extends BatteryState {
  private void Audio.playSoundfile(String file, float volume) {
    proceed(file, 0.75f * volume);
  }

  private Thread dimDisplay = { /* .. */ }
  private boolean isDimmed = false;

  before private void Input.handleClick(int x, int y) {
    if (isDimmed) {
      Display.setBrightness(brightness);
      thislayer.isDimmed = false;
    }

    thislayer.dimDisplay.stop();
    thislayer.dimDisplay.start();
  }
}
```

Implementation (JCop): Layers

```
layer LowBatteryState extends BatteryState {
  private void Audio.playSoundfile(String file, float volume) {
    proceed(file, 2f/3f * volume);
  }

  private Thread turnOffDisplay = { /* .. */ }

  before private void Input.handleClick(int x, int y) {
    thislayer.dimDisplay.stop();
    thislayer.dimDisplay.start();
  }
}
```

Terms and definitions [6]



layer: a means to group related behavioral variations

```
layer LowBatteryState extends BatteryState {
  private void Audio.playSoundfile(String file, float volume) {
    proceed(file, 2f/3f * volume);
  }

  private Thread turnOffDisplay = { /* .. */ }

  before private void Input.handleClick(int x, int y) {
    thislayer.dimDisplay.stop();
    thislayer.dimDisplay.start();
  }
}
```

partial method definition: new, modifed or removed behavior

Overview

Power Management for a Mobile Device

Classical Object-Oriented Programming

Aspect-Oriented Programming

Context-Oriented Programming

Comparison: AspectJ (AOP) and JCop (COP)

- Static Aspect Weaving vs. Dynamic Layer Activation
- Aspect-scoped Advice vs. Base-class-scoped Partial Methods
- Scope of Context/Aspect and Layer Activation
- Join Point Model vs. Partial Method Definitions
- Summary

Static Aspect Weaving vs. Dynamic Layer Activation

AspectJ

```
pointcut receivingInput():
  call(Input.handle*(..));

before(): receivingInput() {
  // ...
}
```

- Static aspect weaving at compile time
- **Use case:** modularization of cross-cutting concerns without changing behavior (e.g. logging, security checks)

JCop

```
when(true): with(states);
when(Battery.getVoltage() < 5.0):
  with (new MediumBatteryState());
```

- Language support for further modularization of aspects
- Dynamic layer activation at runtime
- **Use case:** modularization of cross-cutting concerns with changing behavior (e.g. power management, location-dependent behavior)

Aspect-scoped Advice vs. Base-class-scoped Partial Methods

AspectJ

- Advice code is always executed in the scope of the aspect.
- Calling private method of receiver/sender is not allowed.
- **Concept:** cross-cutting concern operates on the object externally

JCop

- Partial methods may access internal object state and behavior.
- Layer-scoped methods for shared behavior (access via `thislayer`)
- **Concept:** cross-cutting concern changes internal behavior

Scope of Context/Aspect and Layer Activation

AspectJ

- Aspects are globally enabled at compile time.

JCop

```
without(LowBatteryState) {
  // ...
}
```

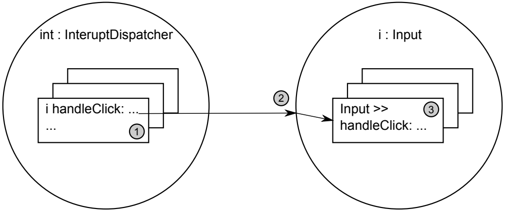
JCop

- Contexts are programmatically activated at runtime.
- Global and thread-local activation [2]
- Layer activation preserved on method calls
- Explicit layer deactivation supported

Join Point Model vs. Partial Method Definitions [7]

AspectJ

> *Join point: a well-defined point in the execution of the program*



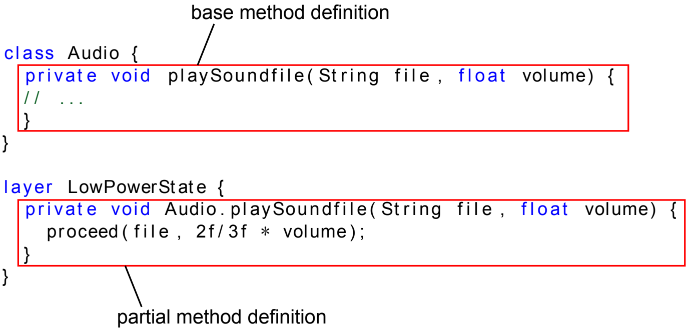
```
pointcut receivingInput():
  call(public void Input.handle*(..));
```

- More, e.g. constructor call, field get/set, exception handler execution
- Multiple methods may be affected by one piece of advice.

Join Point Model vs. Partial Method Definitions

JCop

- Supports method execution only.
- Partial methods are bound to one base method.
- Therefore no pointcuts are necessary.



base method definition

```
class Audio {
  private void playSoundfile(String file, float volume) {
  // ...
  }
}

layer LowPowerState {
  private void Audio.playSoundfile(String file, float volume) {
    proceed(file, 2f/3f * volume);
  }
}
```

partial method definition

More differences

AspectJ

- Declarative pointcut definitions and advice activation
- Advice always defined in aspects

JCop

- Declarative and imperative layer (de-)activation
- Layers defined standalone or classes [1]
- JCop potentially slower than AspectJ (dynamic method lookup)

Summary

- **Aspect-Oriented Programming:** Modularization of static cross-cutting concerns
- **Context-Oriented Programming:** Modularization of dynamic cross-cutting concerns with behavioral changes

Appendix

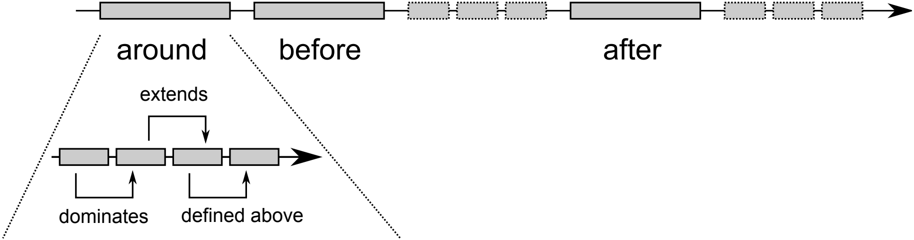
Aspect inheritance [4]

```
import org.aspectj.lang.JoinPoint;

abstract aspect SimpleTracing {

  abstract pointcut tracePoints();

  protected abstract void trace(JoinPoint jp);

  before(): tracePoints() {
    trace(thisJoinPoint);
  }

}
```

- Abstract aspect: may have abstract pointcuts, advice on the pointcuts
- Use cases: overriding pointcut definitions, using pointcut definitions for other aspects [5]

Advice execution sequence

- Multiple advice per join point allowed
- Advice sequence: around, before, . . . , after, . . .
- Sequence for same *type* of advice
  - Same aspect: definition of advice in the source code
  - A_1 extends A_2: A_1 (more specific one) first
  - A_1 dominates A_2: A_1 first
  - Other cases: undefined



Implementation of AspectJ

- Compiler-based implementation
- Aspect transformation: compile advice to methods, insert method call at join points (e.g. after constructor call)
- Code not affected by aspects is compiled to ordinary Java bytecode
- *No observable performance overhead* [7] (static or final method calls)

Terms and definitions [6]

- **Behavioral variation:** new, modified or removed behavior
- **Layer:** a means to group related behavioral variations
- **Layer activation:** activation or deactivation of layers at runtime
- **Layer scoping:** a means to control the scope for layer activation or deactivation
- **Context:** information accessible at runtime

Behavioral variations (examples)

new, modified or removed behavior

- **Actor-dependent behavior variations:** visualize data differently (e.g. *file not found* for normal users, *404 error code with additional information* for developers)
- **Environment-dependent behavior variations:** on shutdown, install updates only if not running on battery mode
- **System-dependent behavior variations:** use SQL database or XML file storage

References I

Appeltauer, M., and Hirschfeld, R.
The jcop language specification.
Technische Berichte Nr. 59 (2012).

Appeltauer, M., Hirschfeld, R., Haupt, M., Lincke, J., and Perscheid, M.
A comparison of context-oriented programming languages.
In *International Workshop on Context-Oriented Programming* (New York, NY, USA, 2009), COP '09, ACM, pp. 6:1–6:6.

Appeltauer, M., Hirschfeld, R., Masuhara, H., Haupt, M., and Kawauchi, K.
Event-specific software composition in context-oriented programming.
In *Proceedings of the 9th international conference on Software composition* (Berlin, Heidelberg, 2010), SC'10, Springer-Verlag, pp. 50–65.

References II

ERNST, E., AND HOC, A.
Aspects and polymorphism in aspectj.
In *In Proceedings of the 2nd International Conference on Aspect-Oriented Software Development* (2003), ACM Press, pp. 150–157.

HANENBERG, S., AND UNLAND, R.
Using and reusing aspects in aspectj.
In *In OOPSLA Workshop on Advanced Separation of Concerns in Object-Oriented Systems* (2001).

HIRSCHFELD, R., COSTANZA, P., AND NIERSTRASZ, O.
Context-oriented programming.
Journal of Object Technology, March-April 2008, ETH Zurich 7, 3 (2008), 125–151.

References III

Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., and Griswold, W. G.
An overview of aspectj.
In *Proceedings of the 15th European Conference on Object-Oriented Programming* (London, UK, UK, 2001), ECOOP '01, Springer-Verlag, pp. 327–353.