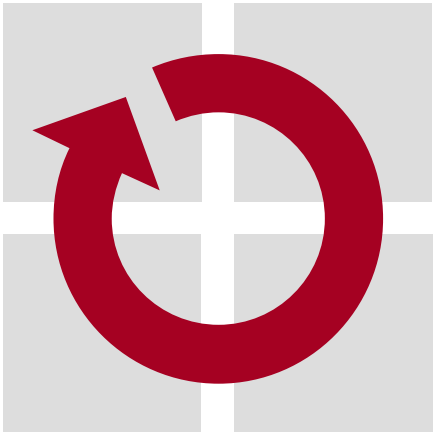
AspectC++

Benjamin Polak

7. Juli 2003

Hauptseminar:
Aspektorientierte Systemprogrammierung

Lehrstuhl für Informatik 4
Verteilte Systeme und Betriebssysteme
Universität Erlangen-Nürnberg



Gliederung

- Grundlegende Konzepte und Sprachkonstrukte von AspectC++
  - Join Points und Pointcuts
  - Advices
  - Aspects
- Fallstudien zum Einsatz von AspectC++ beim Betriebssystembau
  - Unterbrechungssynchronisation
  - Fadensynchronisation

AspectC++

- Eine Spracherweiterung für aspektorientierte Programmierung mit C++
- Syntaktisch angelehnt an AspectJ
- AC++ Compiler als Präprozessor realisiert
  - Transformiert AspectC++ Code in regulären C++ Code
  - Übersetzung des transformierten Codes mit gewöhnlichem C++-Compiler (g++, VisualC++, etc.)

Join Points und Pointcuts (1)

- Join Points
  - Stellen im Anwendungscode, an denen Aspekte greifen können:
    - Methoden
    - Attribute
    - Typen
    - Objekte
    - Stellen im Kontrollfluss
      (von denen aus auf andere Join Points zugegriffen wird)
- Pointcuts
  - Mengen von Join Points
  - Beschrieben durch Pointcut Expression:
    - Match Expressions
    - Pointcut Functions
    - Algebraische Operatoren

Join Points und Pointcuts (2)

- Match Expressions
  - Suchmuster, mit opionalen Wildcard-Operatoren:
    - `"%"` für beliebige Namensteile
    - `"..."` für beliebige Anzahl von Parametern bei Funktionen
  - Beispiele:
    - `int` wählt gleichnamigen primitiven Datentypen von C++ aus
    - `%List` wählt alle Klassen, struct‘s, oder union‘s aus, deren Name auf List endet
    - `% printf(...)` wählt alle Funktionen namens „printf“ mit beliebigen Rückgabetypen und Parametern aus

Join Points und Pointcuts (3)

- Pointcut Functions
  - Vordefinierte Funktionen zum Herausfiltern oder Abbilden von Join Points aus Pointcuts
  - Beispiele:
    - `derived("List")`
      wählt alle von „List“ abgeleiteten Klassen aus
    - `call("% printf(...)")`
      wählt alle Aufrufstellen von „printf“ aus
    - `execution("% printf(...)")`
      wählt die Implementierung von „printf“ aus

Join Points und Pointcuts (4)

- Mengenoperatoren
  - Zur Verknüpfung von Pointcuts
  - `&&` für Schnittmenge
  - `||` für Vereinigungsmenge
  - `!` für Komplementmenge
- Deklaration von Pointcuts
  - Schlüsselwort „pointcut“, z.B.
    - `pointcut lists() = derived("List");`
  - Auch virtuell:
    - `pointcut virtual methods() = 0;`

Advices

- Code, der an Join Points gebunden wird
- Aktivierung des Advice Code wenn Join Point bei Ausführung der Anwendung erreicht wird
- Aktivierungszeitpunkt:
  - Vor Join Point: `before()`
  - Nach Join Point: `after()`
  - Statt Join Point: `around()`
- Beispiel:
  - ```
    advice execution("void login(…)") : before() {
        cout << "Logging in." << endl;
    }
    ```
    ```

Introduction Advices

- Statt `before()`, `after()`, `around()` eine beliebige gültige C++ Deklaration ⇒ Introduction
- Deklaration wird beim Join Point eingebunden
- Beispiel:
  - ```
    pointcut classes() = "Circle" || "Rect";

    advice classes() : bool shaded;

    advice classes() : void set_shaded(bool s) {
        shaded = s;
    }
    ```
    ```

Aspects (1)

- Zur Modularisierung und Kapselung von logisch zusammengehörigen Advices, Introductions und deren Zustandsinformationen
- Eine Erweiterung des Klassenkonzeptes von C++
  - Aspekte können wie Klassen auch Attribute und Methoden enthalten
  - Aspekte können von anderen Aspekten und Klassen erben

Aspects (2)

- Beispiel:

```
aspect InstantiationCounter {
    static int count;
    InstantiationCounter() : count(0) { }
    pointcut counted() = "Circle" || "Rect";
    advice counted() : class Helper {
        Helper() { InstantiationCounter::count++; }
    } helper;
    advice execution("% main(...)") : after() {
        cout << "Final instantiation count: "
             << InstantiationCounter::count << endl;
    }
};
```

Aspects (3)

- Besser, weil wiederverwendbar:

```
aspect InstantiationCounter {
    // ...
    pointcut virtual counted() = 0;
    // ...
};

aspect ShapesCounter : public InstantiationCounter {
    pointcut counted() = derived("Shape");
};
```

Aspekte in PURE

- Fallstudien an der existierenden Betriebssytemfamilie PURE, um die Eignung von AOP/AspectC++ beim Betriebssystembau zu untersuchen
- Hier vorgestellt:
  - Unterbrechungssynchronisation
  - Fadensynchronisation

Unterbrechungssynchronisation (1)

- Kritische Bereiche in vielen Teilen des Betriebssystems vorhanden (*MM*, *I/O*, *Scheduling*, etc.)
- Schutz vor überlappter Ausführung notwendig

$\Rightarrow$ *cross-cutting concern*

- In PURE: Prolog/Epilog-Modell
  - Einrahmung von kritischen Bereichen mit „`lock.enter()`“ und „`lock.leave()`“
  - Kritische Operationen der Unterbrechungsbehandlung werden ggf. verzögert

Unterbrechungssynchronisation (2)

- Probleme der herkömmlichen Implementierung:
  - Synchronisationsaufrufe sehr zahlreich und quer durch das System verstreut
  - Von Unterbrechungssynchronisation abhängige Module schlecht wiederverwendbar, falls kein Synchronisationsbedarf oder anderes Synchronisationsverfahren
  - Strategie der Synchronisation (z.B. bzgl. Granularität) schlecht änderbar, da aufwendige und fehleranfällige Anpassung der zahlreichen Synchronisationsaufrufe notwendig

Unterbrechungssynchronisation (3)

- Fallstudie: Reimplementierung der Unterbrechungssynchronisation mittels AOP
  - Verstreute Synchronisationsaufrufe aus allen Modulen entfernt $\Rightarrow$ jetzt bessere Wiederverwendbarkeit
  - Unterbrechungssynchronisation vollständig in einem Aspekt modularisiert $\Rightarrow$ jetzt Änderungen einfach zentral vornehmbar
  - Quellcode-Umfang und Speicherplatzverbrauch des Kompilats reduziert, da einige Wrapper-Klassen, Schnittstellen-Klassen entfernt

Unterbrechungssynchronisation (4)

- Realisierung (eine Variante):

```
// Beschreibung der PURE Subsysteme
pointcut osek()   = derived("osekBase") || "osekOS";
pointcut thread() = derived("Schemer") && !osek();
pointcut case()   = derived("Counter");
// ...

// Aspekt zur grobgranularen Unterbrechungssynchronisation
aspect IntSync {
    // Zu synchronisierende Region
    pointcut kernel() = osek() || thread() || case() /* || ... */;
    // Eintritte in die zu synchronisierende Region
    pointcut locked() = call(kernel()) && !within(kernel());
    // Austritte aus der zu synchronisierenden Region
    pointcut unlocked() = within(kernel()) &&
            (call("void Triplet::action()") /* || ... */);
    // Advice-Code zum Sperren und Erlauben von Epilogen
    advice unlocked() : before() { lock.leave(); }
    advice unlocked() : after()  { lock.enter(); }
    advice locked()   : before() { lock.enter(); }
    advice locked()   : after()  { lock.leave(); }
}
```

Fadensynchronisation (1)

- Synchronisation nebenläufiger Fäden notwendig, falls gemeinsamer Betriebsmittelzugriff (z.B. auf Gerätetreiber)
- In PURE Gerätetreiber unsynchronisiert, Fäden selbst für Synchronisation verantwortlich
- Fallstudie: Fadensynchronisation mittels AOP am Beispiel Floppy-Treiber

Fadensynchronisation (2)

- Variante Gegenseitiger Ausschluss:

```
aspect Mutex {
    Semaphore mutex;
    pointcut virtual funcs() = 0;
    Mutex() : mutex(1) {} // Semaphor-Initialisierung
    // Abfangen und Schutz aller Aufrufe
    advice funcs() : before() { mutex.wait(); }
    advice funcs() : after()  { mutex.signal(); }
};

aspect FloppyAndDMAMutex : public Mutex {
    pointcut inst() = "DMA8237A" || "PCFloppyController";
    pointcut floppy() = derived("PCFloppyController") &&
                        base("PCFloppyDriver");
    pointcut funcs() = (call("DMA8237A") && !within("DMA8237A")) ||
                       (call(floppy()) && !within(floppy()));
    advice inst() : FloppyAndDMAMutex __inst;
    static FloppyAndDMAMutex *aspectOf() {
        return &thisJoinPoint->target()->__inst;
    }
};
```

Fazit

- Umsetzung der AOP-Konzepte in AspectC++ gelungen
- Vorteile aus AOP tatsächlich erreichbar, wie durch Fallstudien belegt